Contents

Next >

Tutorial: Create an EJB 3.0 application

This tutorial describes how to create an EJB 3.0 application that contains an EJB 3.0 project and a web project to display a counter program.

Learning objectives

This tutorial enables you to create:

- An EJB 3.0 project, called `EJBCounterSample`, with an EJB 3.0 stateless session bean (with both interface and implementation classes) and a JPA 1.0 entity class
- A web project (`EJBCounterWeb`), and with a Java™ Server Page (JSP) page and a utility Java class
- A Java EE 5 application (`EJBCounterSampleEAR`) with the EJB and Web projects

Time required

30 minutes

Related information:

Sample: EJB 3.0 Counter

Next >

Create an EJB 3.0 application

This tutorial describes how to use the EJB 3.0 specification to create, deploy, and run a simple EJB 3.0 application that increments a counter.
This tutorial might require some optionally installable components. If you encounter errors or cannot find user interface options when you run the tutorial, ensure that you installed the appropriate optional components:
- IBM® WebSphere® Application Server, version 7.0 installed
  To use this tutorial, you must have an application server installed and configured. To verify that a server runtime environment is available, click **Window** > **Preferences**, expand Server, and then click **Installed Runtimes**. You can use this pane to add, remove, or edit installed server runtime definitions.

This tutorial is divided into several exercises that must be completed in sequence for the tutorial to work properly. This tutorial teaches you how to use the EJB 3.0 specification to create an EJB project that includes an EJB bean and remote and local interfaces; you also create a web project that includes a Servlet and a Faces JSP to deploy the application on the server.

While completing the exercises, you will:
- Use the EJB 3.0 specification to create, deploy, and run a simple EJB 3.0 application that increments a counter.
- Create a Java™ project, `EJBCounterSample` with an EJB 3.0 stateless session bean (with both interface and implementation classes), `StatelessCounterBean.java`, and a JPA 1.0 entity class `JPACounterEntity.java`.
- Create web project (EJBCounterWeb), and with a Java Server Page (JSP) page and a utility Java class
- Create EAR application, `EJBCounterSampleEAR`, containing the EJB and web projects.

Time required

This tutorial takes approximately 30 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete.

Skill level

Experienced

Audience

This tutorial is intended for users who are familiar with Enterprise JavaBeans, EJB 3.0 in particular, and Java EE 5 technology.

System requirements

To complete this tutorial, you need to have the following tools and components installed:
- **The WebSphere Application Server v. 7.0 installed**.
- **A clean workspace**.

Prerequisites

In order to complete this tutorial end to end, you should be familiar with:

- Java EE 5 and Java programming
- Basic Enterprise JavaBeans (EJB) and EJB 3.0 concepts

Lessons in this tutorial

- **Lesson 1.1: Create an EJB 3.0 project**
  This lesson leads you through the detailed steps to create an EJB 3.0 project used to contain your EJB session bean.
- **Lesson 1.2: Create the required classes and interfaces for the StatelessCounterBean.java class**
  Lesson 1.2 steps you through the creation of required classes and interfaces for the `StatelessCounterBean.java` class.
- **Lesson 1.3: Create an Entity class and a data source for data persistence**
  Lesson 1.3 leads you through the creation of an entity class and a database for data persistence.
- **Lesson 1.4: Create a web project to test your application**
  Lesson 1.4 leads you through the creation of a web project to test your application.

< Previous | Next >

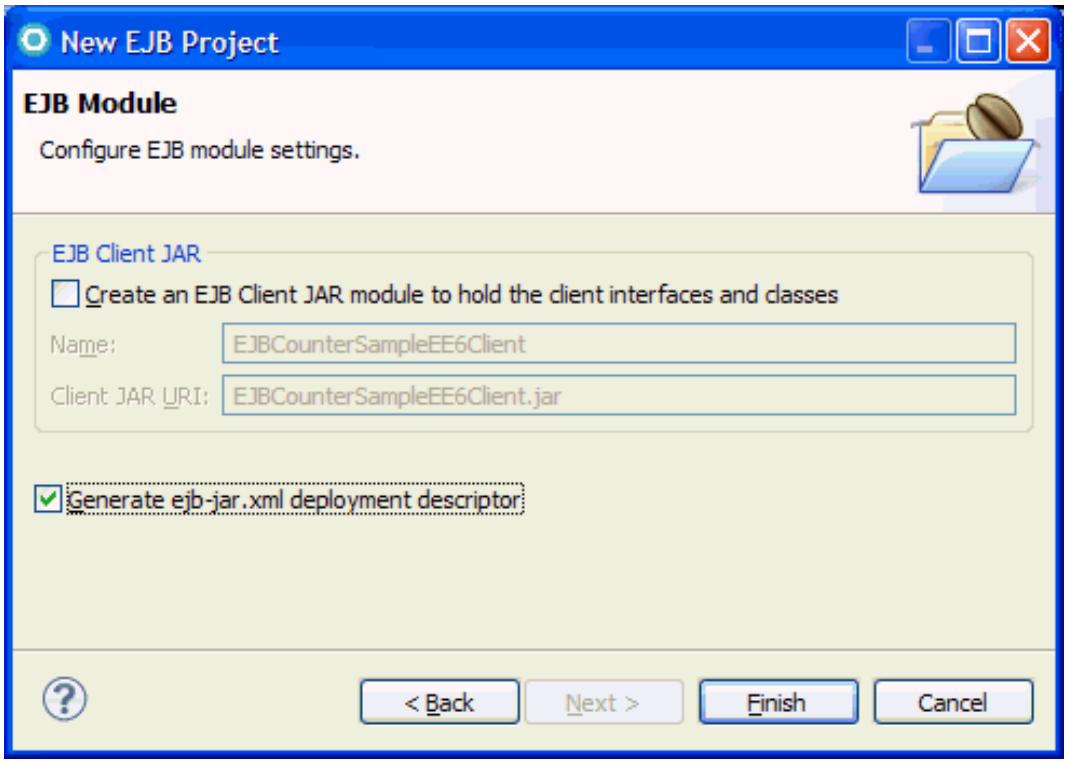
Lesson 1.1: Create an EJB 3.0 project

This lesson leads you through the detailed steps to create an EJB 3.0 project used to contain your EJB session bean.

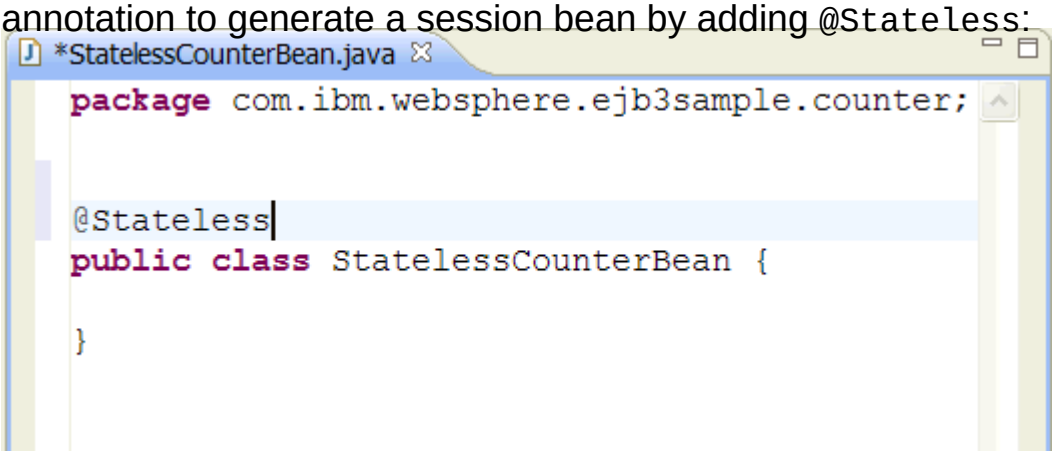
In order for you to create an EJB 3.0 project, you need to have the WebSphere® Application Server version 6.1 with the EJB 3.0 Feature Pack installed OR WebSphere Application Server version 7.0, and to have a profile created for this server.

To create an EJB 3.0 counter project:

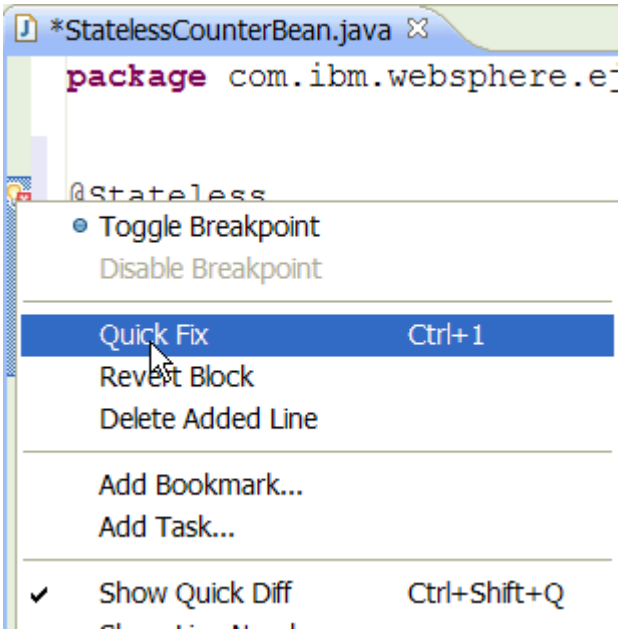
1. **Create a server configuration for the WebSphere Application Server version7.0**:
   A. Open the Servers view by selecting **Window** > **Show View** > **Servers**.
   B. Define a new WebSphere Application Server by right clicking the Servers view and selecting **New** > **Server**. Follow the instructions in the **New Server** wizard, ensuring that you select the WebSphere Application Server version 7.0.
2. Create an EJB 3.0 project:
   A. If the Java™ EE icon, , does not appear in the top right tab of the workspace, you need to switch to the Java EE perspective. From the menu bar, select **Window > Open Perspective > Other**. The Select Perspective window opens. Select **Java EE**. Click **OK**. The Java EE perspective opens.
   B. In the Java EE perspective, select **File** > **New** > **Other** > **EJB** > **EJB project**.
   C. In the **Project name** field, type `EJBCounterSample`. Select **Add project to EAR**, and click **Next**. On the Java page, accept the defaults and click **Next**.
   D. On the EJB Module page, clear **Create an EJB Client JAR module to hold the client interfaces and classes** and select **Generate ejb-jar.xml deployment descriptor:**, and click **Finish**.



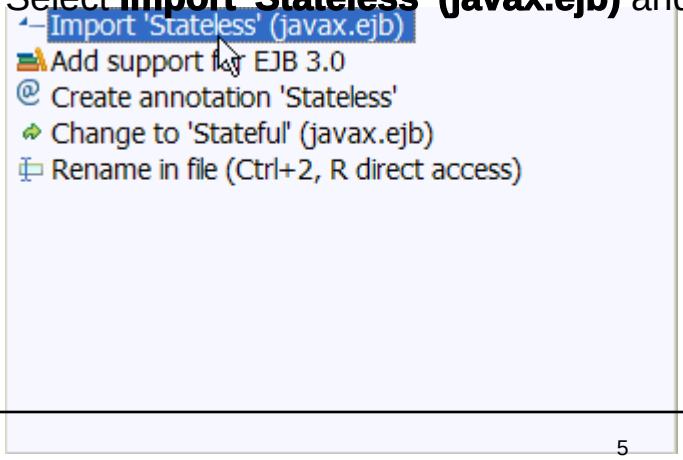
3. Add a Java class, @Stateless annotation:
   A. In the Enterprise explorer view, right click the EJBCounterSample project and select **New** > **Class**.
   B. Accept the default **Source folder** (EJBCounterSample/ejbModule). In the **Package** field, type `com.ibm.example.websphere.ejb3sample.counter`, and in the **Name** field, type `StatelessCounterBean`.
   C. Click **Finish**.
   D. Your `StatelessCounterBean` class opens in the Java Editor. Add the EJB 3.0 annotation to generate a session bean by adding `@Stateless`:



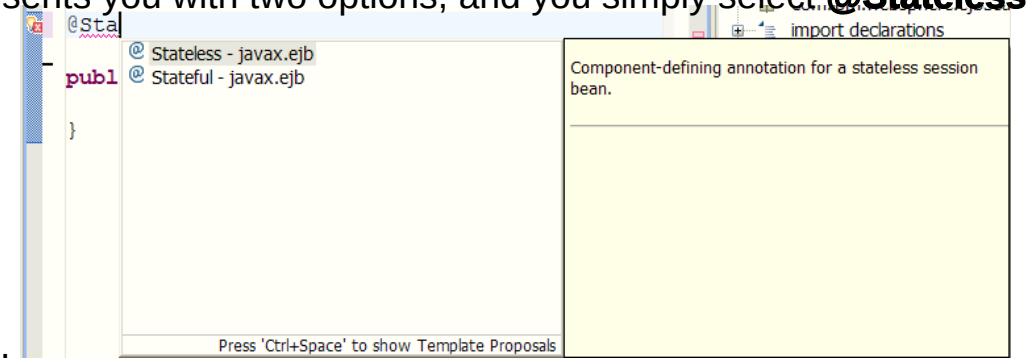
   E. When you press CTRL+S to save, you can see a quick-fix icon beside the @Stateless line.
   F. Right click the quick-fix icon and select **Quick Fix**:



   G. Select **Import 'Stateless' (javax.ejb)** and press CTRL+S to save:



The required dependencies are automatically added to the source code.**Tip:** A shortcut to using the Quick Fix is to type `@Sta`, and press CTRL+Spacebar, which presents you with two options, and you simply select **@Stateless - javax.ejb**.



The required dependencies are automatically added to the source code. You now are ready to move on to Exercise 1.2, Create required classes and interfaces for the `StatelessCounterBean`.

< Previous | Next >

Lesson 1.2: Create the required classes and interfaces for the StatelessCounterBean.java class

Lesson 1.2 steps you through the creation of required classes and interfaces for the `StatelessCounterBean.java` class.
Before you begin, you must complete Lesson 1.1.
In this lesson, you
- Add @Interceptors annotation and required dependencies.
- Create an `Audit.java` Java™ class.
- Add code to your `StatelessCounterBean.java` class
- Create a `LocalCounter.java` interface and a `RemoteCounter.java` interface

1. Open your `StatelessCounterBean.java` class in the Java Editor.
2. Under the `@Stateless` annotation, type `@Interceptors`.
3. When you press CTRL+S to save, you can see a quick-fix icon beside the `@Interceptors` line. Right-click the quick-fix icon and select **Quick Fix**:
4. Select **import javax.interceptor.Interceptors**, and press CTRL+S to save.
5. Right-click the quick-fix icon and select **Quick Fix**. Select **Add missing attributes** and replace **(value={null})** with `(Audit.class)`, press CTRL+S to save.
6. Right-click the quick-fix icon and select **Quick Fix**. Select **Create class 'Audit'**. In the Create a new Java class page, accept the default values and click **Finish** .
7. In the Java editor for `Audit.java`, replace the default code with this code, and press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;

import java.io.Serializable;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class Audit implements Serializable {

   private static final long serialVersionUID = 4267181799103606230L;

   @AroundInvoke

   public Object methodChecker (InvocationContext ic)
   throws Exception
   {
       System.out.println("Audit:methodChecker - About to execute method: " + ic.getMethod());
```

```
        Object result = ic.proceed();
        return result;
    }
}
```

8. Open your `StatelessCounterBean` class in the Java Editor and **replace this code**, and press CTRL+S to save:`public class StatelessCounterBean {`

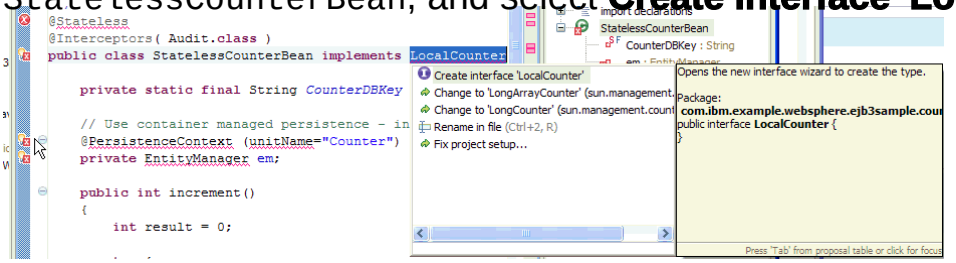
```
}
```

with this code: `public class StatelessCounterBean implements LocalCounter, RemoteCounter {`

```
    private static final String CounterDBKey = "PRIMARYKEY";

    // Use container managed persistence - inject the EntityManager
    @PersistenceContext (unitName="Counter")
    private EntityManager em;

    public int increment()
    {
        int result = 0;

        try {

            JPACounterEntity counter = em.find(JPACounterEntity.class, CounterDBKey);

            if ( counter == null ) {
                counter = new JPACounterEntity();
                counter.setPrimaryKey(CounterDBKey);
                em.persist( counter );
            }

            counter.setValue( counter.getValue() + 1 );
            em.flush();
            em.clear();

            result = counter.getValue();

        } catch (Throwable t) {
            System.out.println("StatelessCounterBean:increment - caught unexpected exception: " + t);
            t.printStackTrace();
        }

        return result;
    }



    public int getTheValue()
    {
```

```
        int result = 0;

        try {
            JPACounterEntity counter = em.find(JPACounterEntity.class, CounterDBKey);

            if ( counter == null ) {
                counter = new JPACounterEntity();
                em.persist( counter );
                em.flush();
            }

            em.clear();

            result = counter.getValue();
        } catch (Throwable t) {
            System.out.println("StatelessCounterBean:increment - caught unexpected exception: " + t);
            t.printStackTrace();
        }

        return result;
    }
}
```

9. Right-click the quick-fix icon beside the line `public class StatelessCounterBean`, and select **Create Interface 'LocalCounter'**:



10. Your `LocalCounter.java` class opens in the Java Editor. Replace all the code with this code, and press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;

import javax.ejb.Local;

@Local
public interface LocalCounter {
   public int increment();
   public int getTheValue();
}
```

11. Return to the `StatelessCounterBean.java` class in the Java editor.
12. Follow the steps for creating a Local Counter to create a Remote Counter interface, and replace all the code in RemoteCounter.java with this code, and

press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;

import javax.ejb.Remote;

@Remote
public interface RemoteCounter {
   public int increment();
   public int getTheValue();
}
```

13. Return to the `StatelessCounterBean.java` class in the Java editor.
14. Right-click the quick fix icon beside `@PersistenceContext`, and select **Quick Fix**. Select **Import 'PersistenceContext' (javax.persistence)**, and press CTRL+S to save.
15. Right-click the quick-fix icon beside `@EntityManager`, and select **quick-fix**. Select **Import 'EntityManager' (javax.persistence)**, and press CTRL+S to save.
16. Right-click the quick-fix icon beside `JPACounterEntity`, and select **Quick Fix**. Select **Create class 'JPACounterEntity.java'**. In the New Class wizard page, accept the default values and click **Finish**

You now are ready to move on to Exercise 1.3, Create an Entity class and a database for data persistence.

< Previous | Next >

Lesson 1.3: Create an Entity class and a data source for data persistence

Lesson 1.3 leads you through the creation of an entity class and a database for data persistence.

Before you begin, you must complete Lesson 1.2.

In this lesson you will

- Add code to the entity class, `JPACounterEntity.java`.
- Create a database, `EJB3SampleDB`, to persist the counter data.

1. **Add code to the entity class**:
   A. Open `JPACounterEntity.java` in the Java™ editor, replace all the code with this code, and press CTRL+S to save:

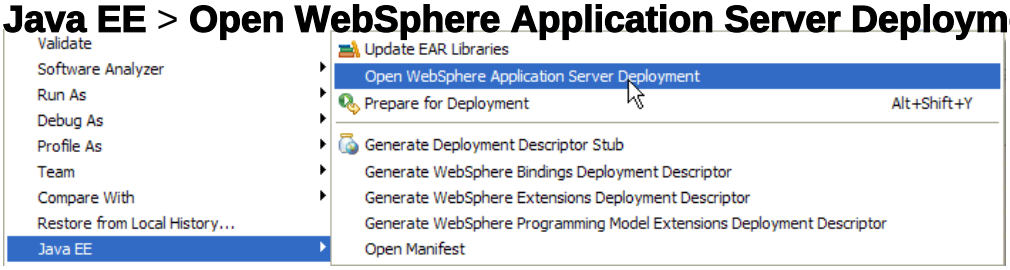
```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;



@Entity
@Table(name="EJB3COUNTERTABLE")

public class JPACounterEntity {

   @Id
   private String primarykey = "PRIMARYKEY";

   private int value = 0;

   public void setValue( int newValue )
   {
       System.out.println ("JPACounterEntity:setValue = " + newValue);
       value = newValue;
   }

   public int getValue()
   {
       System.out.println ("JPACounterEntity:getValue = " + value);
       return value;
   }

   public void setPrimaryKey( String newKey )
```

```
    {
        System.out.println ("JPACounterEntity:setPrimaryKey = '" + newKey + "'");
        primarykey = newKey;
    }

    public String getPrimaryKey()
    {
        System.out.println ("JPACounterEntity:getPrimaryKey = '" + primarykey + "'");
        return primarykey;
    }
}
```

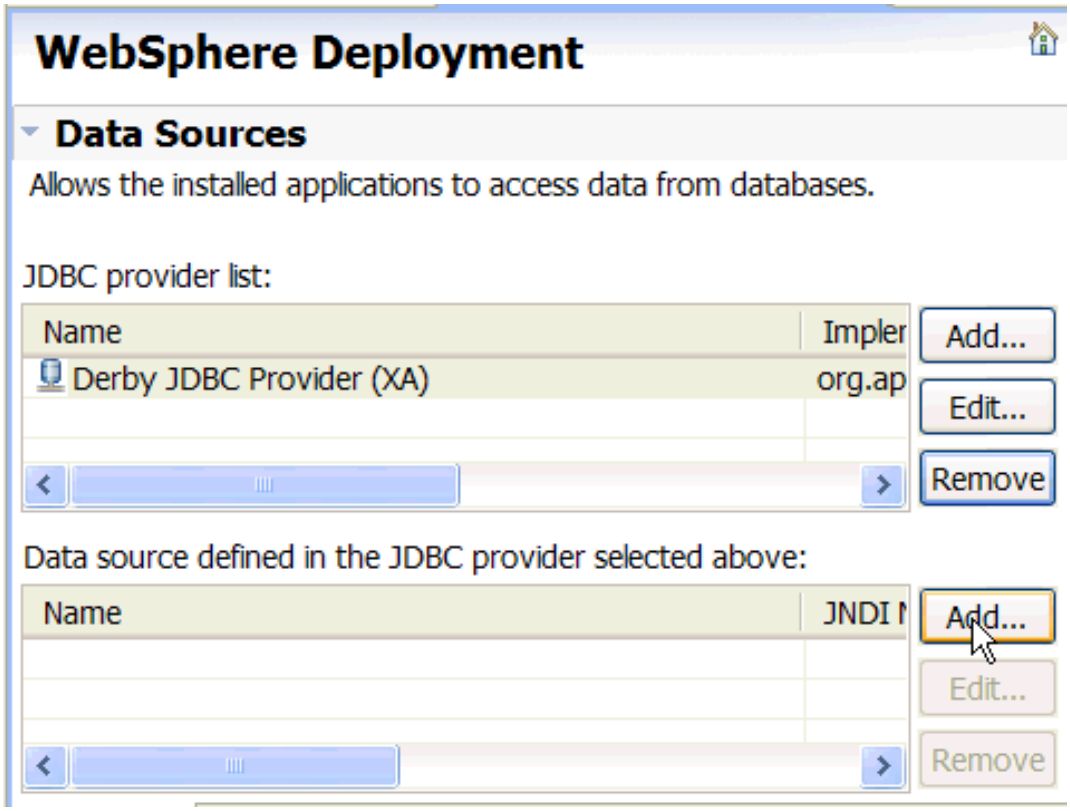
B. In the Enterprise explorer view, navigate to `EJBCounterSample/ejbModule/META-INF`. Right-click on `META-INF` and select **File** > **New** > **File**. Type `persistence.xml` in the **File name** field, and click **Finish**. The `persistence.xml` file opens in the Editor. Select source, and copy and paste this code into the source window:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
        <persistence-unit name="Counter">
                <jta-data-source>jdbc/EJB3SampleDatasource</jta-data-source>
                <class>com.ibm.example.websphere.ejb3sample.counter.JPACounterEntity</class>
                <exclude-unlisted-classes>true</exclude-unlisted-classes>
        </persistence-unit>
</persistence>
```

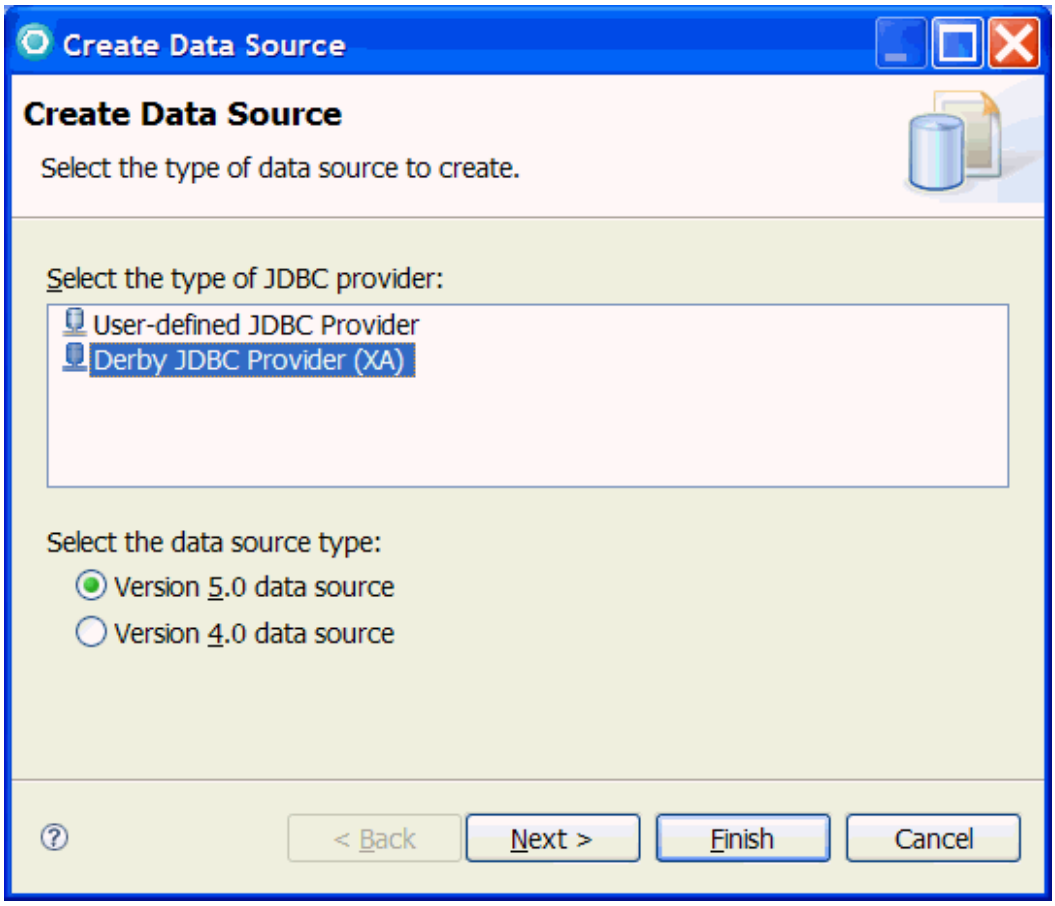
2. Define a data source
   A. In the Enterprise explorer view, right click `EJBCounterSampleEAR`, and select **Java EE** > **Open WebSphere Application Server Deployment**:



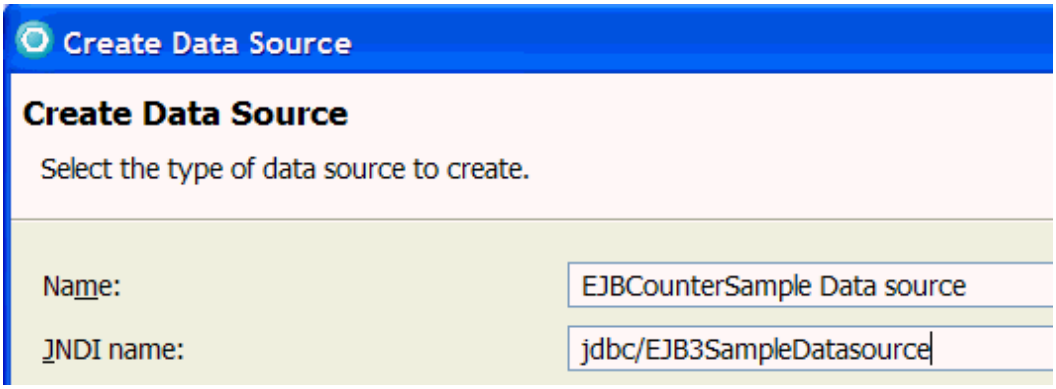
   B. In the WebSphere® Deployment editor, select **Derby JDBC Provider (XA)**, and in the **Data source defined in the JDBC provider selected above:** field, click **Add**:



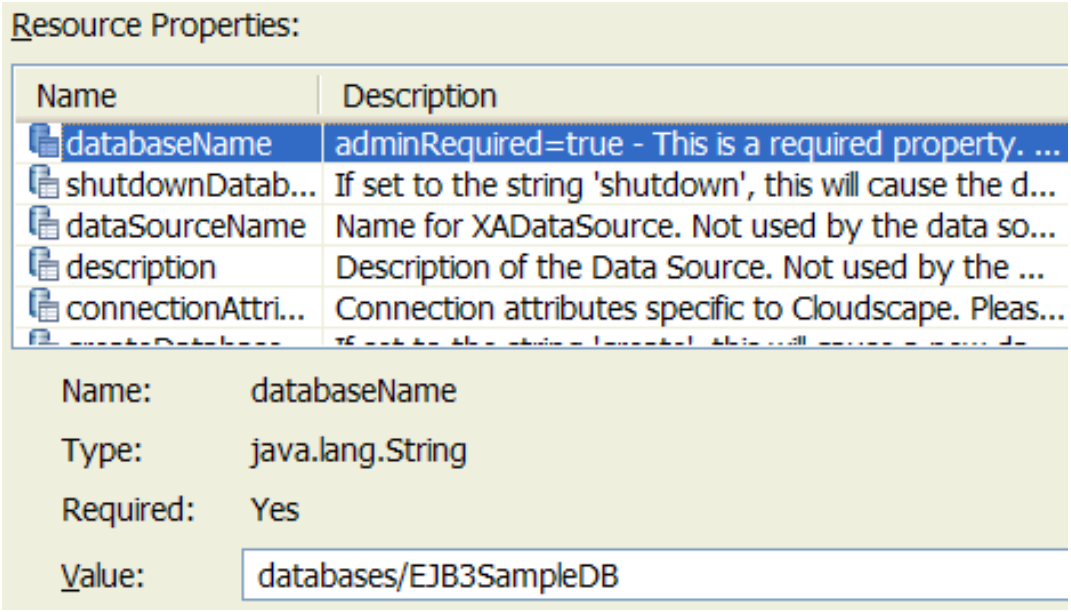
C. In the **Create Data Source** page, select `Derby JDBC Provider (XA)`, and click **Next**:



D. On the Select type of JDBC provider to create page, in the **Name** field, type `EJBCounterSample Data source`. In the **JNDI name** field, type `jdbc/EJB3SampleDatasource`, and click **Next**:



E. In the **Create Resource Properties** page, highlight **databaseName property** field, and in the **Value** field, type `databases/EJB3SampleDB`, and click **Finish**:



You now are ready to move on to Exercise 1.4, Create a web project to test your application.

< Previous | Next >

Lesson 1.4: Create a web project to test your application

Lesson 1.4 leads you through the creation of a web project to test your application. Before you begin, you must complete Lesson 1.3.

In this lesson you will

- Create a web project, `EJBCounterWeb`
- Create a web page, `EJBCount.jsp`
- Create a Servlet, `EJBCount.java`
- Run the Servlet to test your application

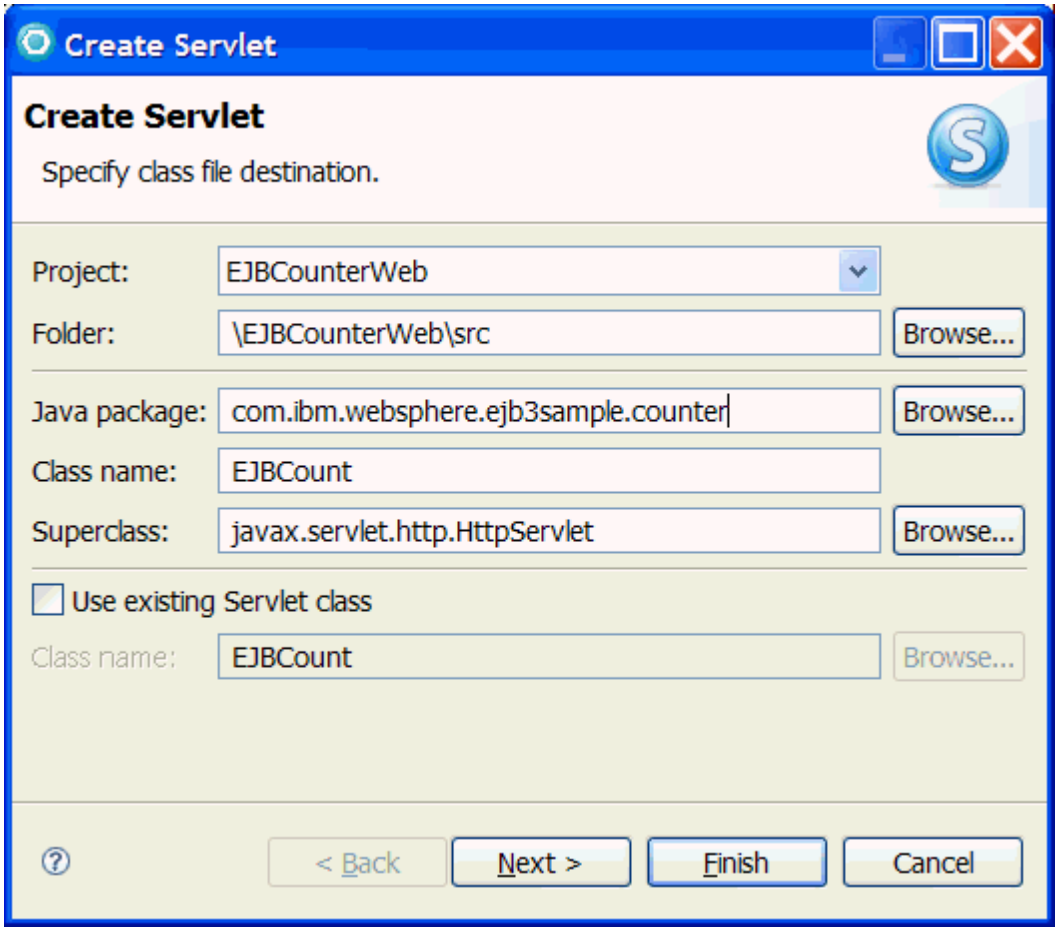
1. Extract the EJBCounterDB
   A. Import the required database: The import wizard imports a database to provide persistence for the EJB 3.0 Counter project.Import the EJB 3.0 tutorial resources
   B. Expand **EJBCounterDB** > **EJBCounterDB.zip** and double click `EJBCounterDB.zip`.
   - Windows: Extract the database into your `/derby/databases` folder of your WebSphere® Application Server install folder:
   - Linux: Extract the database into your `/derby/databases` folder of your WebSphere Application Server install folder.
     - Give your non-root user access to the databases directory. (The easiest way is to give everyone access: `chmod ugo+x databases`.
     - Give your non-root user write-access to the extracted database. (For example, you can extract as the non-root user, provided you have access to the databases directory).

   Important: Depending on the type of WebSphere Application Server, the default location of your `/derby/databases` may differ. For information about default installation directories, see Creating a WebSphere Application Server.
2. In the Java™ EE perspective, right-click your enterprise application project and select **New** > **Web Project** to open the web project wizard.
3. In the Web Project page, in the **Project name** field, type `EJBCounterWeb`.
4. In the **Project Templates** field, select **Simple**.
5. In the **Programming Model** field, select **Java EE**.
6. On the deployment page, from the list of available configuration options, click **Deployment** to open the Deployment configuration page.
   - In the **Target runtime** select WebSphere Application Developer v7 or v8 from the drop-down box.
   - Clear **Add support for WebSphere bindings and extensions**.
   - In the **Web module version** field, select **3.0**
   - In the **EAR membership** field, click **Add project to an EAR**..
   - In the **EAR project name** field, ensure that **EJBCounterWebEAR** appears.
7. Accept the other defaults and click **Finish**. If asked to **Open associated perspective?**, click **No**.
8. Right click the `EJBCounterWeb` project, and select **New** > **Web page**.

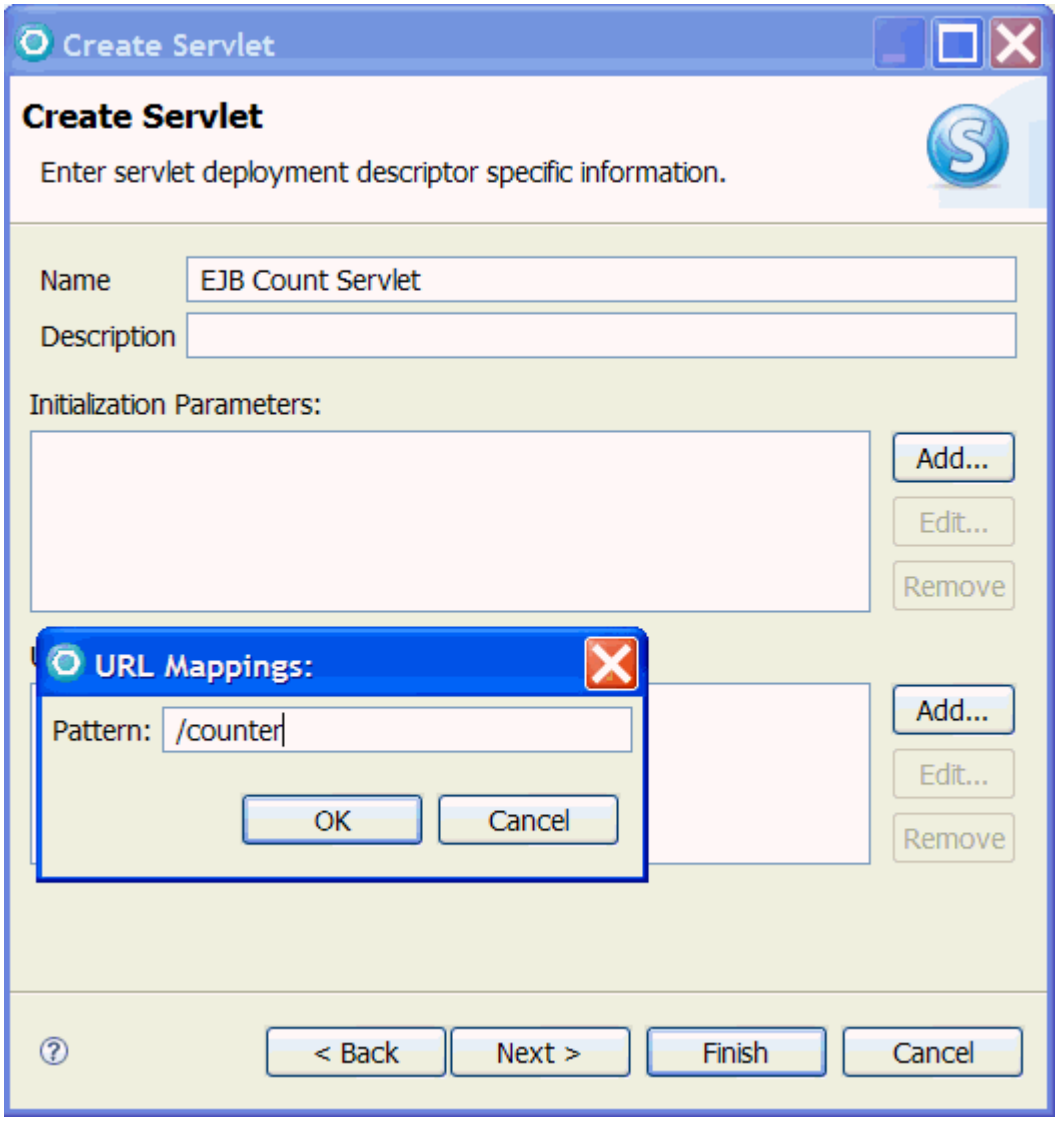
9. On the **New Web page**, in the **File name** field, type `EJBCount.jsp`.
10. In the Source view of the Web page editor, replace all the existing code with this code, and press CTRL+S to save:

```
<%@page session="false"%>
<HTML>
<HEAD>
<TITLE>IBM WebSphere EJB3 and JPA1 Counter Sample</TITLE>
<BODY bgcolor="cornsilk">
<H1>EJB 3.0 and JPA 1.0 Counter Sample</H1>
<P>
<B>
This application communicates with the WebSphere Application Server using http requests to increment a stateless
EJB 3.0 counter bean which is using a JPA 1.0 entity (ie. keeps a persistent counter in a Derby database table).
</B>
<FORM METHOD=POST ACTION="counter">
<BR/>
<%
   response.setHeader("Pragma", "No-cache");
   response.setHeader("Cache-Control", "no-cache");
   response.setDateHeader("Expires",0);
   String msg = (String) request.getAttribute("msg");
   if (msg == null) msg = "";
%>
<B>Click on the Increment button to increment the count</B>
<BR/><BR/>
<INPUT TYPE=SUBMIT VALUE="Increment">
</FORM>
<H3><%=msg%></H3>
</BODY>
</HTML>
```

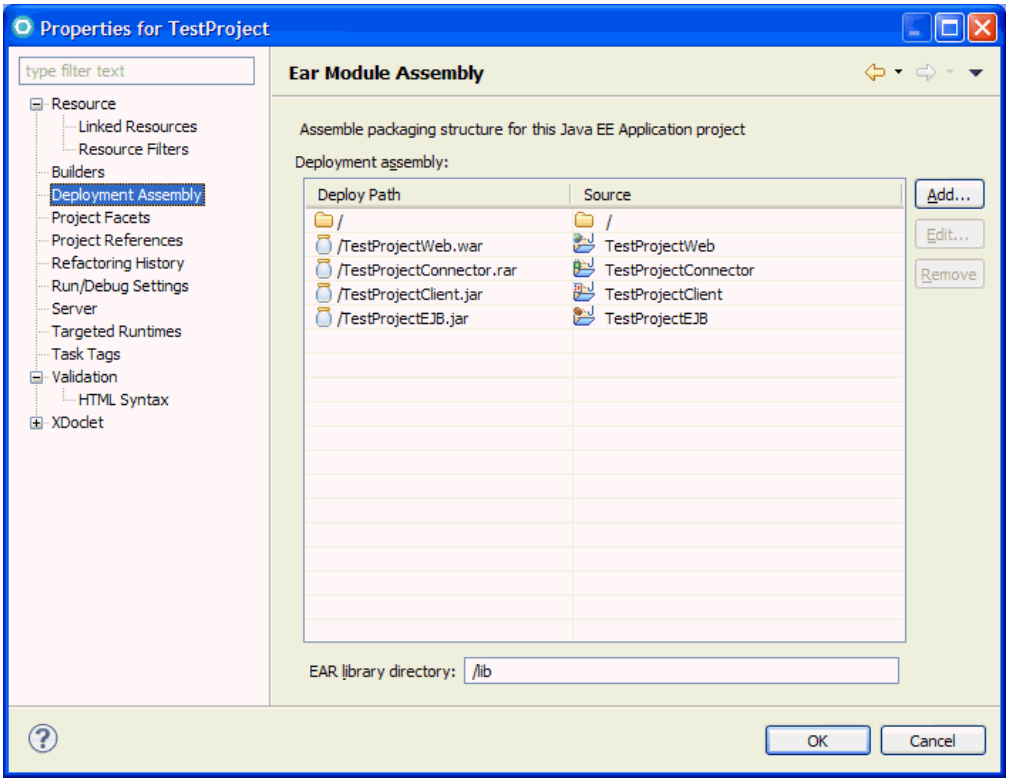
11. Right click the `EJBCounterWeb` project, and select **New** > **servlet**.
12. On the **New Servlet** page, in the **Java package** field, type `com.ibm.example.websphere.ejb3sample.counter`.
13. In the **Class name** field, type `EJBCount`, and Click **Next**:



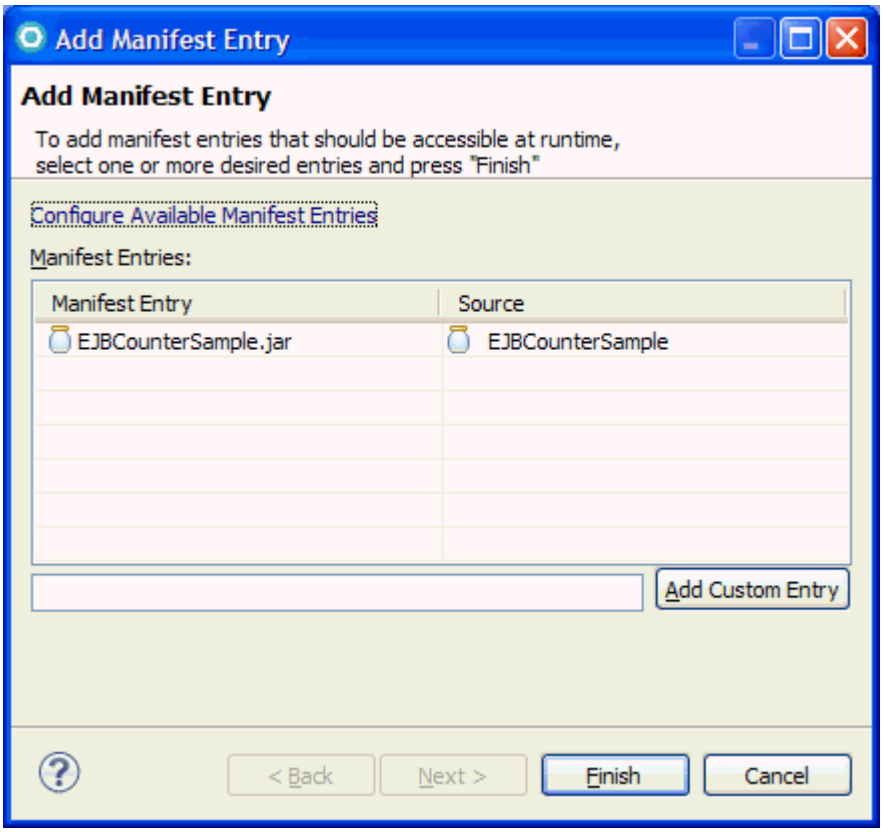
14. In the **Name** field, type `EJB Count Servlet`. In the **URL mappings** field, edit the existing mapping, highlight /EJB Count Servlet and click **Edit**. Replace the pattern it with `/counter`, and click **Finish**:



15. Right click the `EJBCounterWeb` project, and select **Properties**.
16. Select **Deployment Assembly**, select **Manifest Entries**, and click **Add**:



17. Select `EJBCounterSampleEE6.jar`, and click **Finish**, and then click **OK**



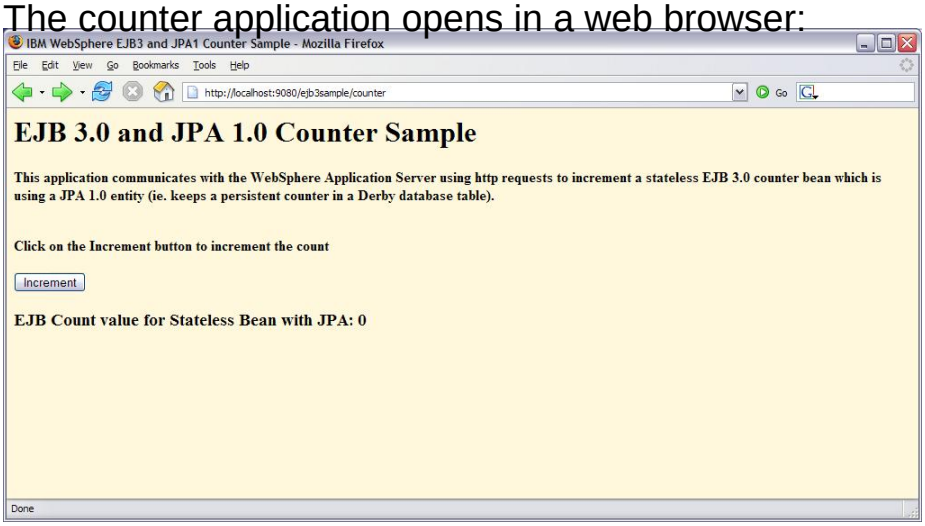
18. Expand **EJBCounterWeb** > **Java Resources** > **src** > **com.ibm.example.websphere.ejb3sample.counter**, and double click the `EBJCount.java` file. It opens in the Java editor.
19. Replace the existing code with the following code, and press CTRL+S to save:

```
package com.ibm.websphere.ejb3sample.counter;
```

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

import java.io.IOException;

import javax.ejb.EJB;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
* This servlet demonstrates an EJB3 counter bean with JPA.
*/

public class EJBCount extends HttpServlet {

   private static final long serialVersionUID = -5983708570653958619L;

   // Use injection to get the ejb
   @EJB private LocalCounter statelessCounter;

   public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
  String msg = null;
  int ejbCount = 0;

  try {
   ejbCount = statelessCounter.getTheValue();
  }
  catch (RuntimeException e) {
   msg = "Error - getTheValue() method on EJB failed!";
        e.printStackTrace();
  }
  msg = "EJB Count value for Stateless Bean with JPA: " + ejbCount;

  // Set attributes and dispatch the JSP.
       req.setAttribute("msg", msg);
       RequestDispatcher rd = getServletContext().getRequestDispatcher("/EJBCount.jsp");
       rd.forward(req, res);
 }

   public void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
  String msg = null;
  int ejbCount = 0;

  try {
   ejbCount = statelessCounter.increment();
  }
```

```
 catch (RuntimeException e) {
  msg = "Error - increment() method on EJB failed!";
       e.printStackTrace();
 }
 msg = "EJB Count value for Stateless Bean with JPA: " + ejbCount;

 // Set attibutes and dispatch the JSP.
      req.setAttribute("msg", msg);
      RequestDispatcher rd = getServletContext().getRequestDispatcher("/EJBCount.jsp");
      rd.forward(req, res);
}

}
```

20. In the Enterprise Explorer view, expand the **EJBCounterWeb** > **Deployment Descriptor** > **Java Resources/src** > **com.ibm.example.websphere.ejb3sample.counter**, and right-click the EJBCount.java file, and select **Run** > **Run on Server**
21. The counter application opens in a web browser:



You have now completed the EJB 3.0 Counter tutorial.

< Previous | Next >

< Previous

Create an EJB 3.0 application summary

This tutorial guided you through the detailed steps to create an EJB 3.0 application that contains an EJB 3.0 project and a web project to display a counter program. From this tutorial, you have learned how to create an EJB project and to use the EJB 3.0 annotation support for creating EJB 3.0 session beans. You also created an Enterprise Application project and a web project to deploy your application on a WebSphere® application server.

Lessons learned

While completing the exercises, you learned how to:

- Create an EJB 3.0 project, called EJBCounterSample, with an EJB 3.0 stateless session bean (with both interface and implementation classes) and a JPA 1.0 entity class
- Create a web project (EJBCounterWeb), and with a Java™ ServerPages (JSP) page and a utility Java class
- Create a Java EE 5 application (EJBCounterSampleEAR) with the EJB and Web projects

< Previous

Next >

Tutorial: Create an EJB 3.1 application

This tutorial describes how to create an EJB 3.1 application that contains an EJB 3.1 project, a web project to display a counter program, and a Web fragment project.

Learning objectives

This tutorial enables you to create:

- An EJB 3.1 project, called EJBCounterSampleEE6, with an EJB 3.1 singleton bean (with both interface and implementation classes) and a JPA 2.0 entity class
- A web project (EJBCounterWebEE6), and with a Java™ ServerPages (JSP) page and a utility Java class
- A web fragment project (EJBCounterWebFragEE6).
- A Java EE 6 application (EJBCounterEAR6) with the EJB and web projects

Time required

30 minutes

Related information:

EJB 3.1 Counter sample

Next >

Create an EJB 3.1 application

This tutorial describes how to use the EJB 3.1 specification to create, deploy, and run a simple EJB 3.1 application that increments a counter.

This tutorial might require some optionally installable components. If you encounter errors or cannot find user interface options when you run the sample, ensure that you installed the appropriate optional components:

- IBM® WebSphere® Application Server, version 8.0 installed

To use this sample, you must have an application server installed and configured. To verify that a server runtime environment is available, click **Window** > **Preferences** > **Server** > **Installed Runtimes**. You can use this pane to add, remove, or edit installed server runtime definitions.

This tutorial is divided into several exercises that must be completed in sequence for the tutorial to work properly. This tutorial teaches you how to use the EJB 3.1 specification to create an EJB project that includes an EJB bean and remote and local interfaces; you also create a web project that includes a Servlet and a Faces JSP to deploy the application on the server.

While completing the exercises, you will:

- Use the EJB 3.1 specification to create, deploy, and run a simple EJB 3.1 application that increments a counter.
- Create a Java™ project, `EJBCounterSampleEE6` with an EJB 3.1 singleton bean (with both interface and implementation classes), `SingletonCounterBean.java`, and a JPA 2.0 entity class `JPACounterEntity.java`.
- Create web project (EJBCounterWeb), and with a Java ServerPages (JSP) page and a utility Java class
- Create EAR application, `EJBCounterEAR6`, containing the EJB and web projects.

Time required

This tutorial takes approximately 30 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete.

Skill level

Experienced

Audience

This tutorial is intended for users who are familiar with Enterprise JavaBeans, EJB 3.1 in particular, and Java EE 6 technology.

System requirements

To complete this tutorial, you need to have the following tools and components installed:

- **The WebSphere Application Server v. 8.0 installed**.
- **A clean workspace**.

Prerequisites

In order to complete this tutorial end to end, you should be familiar with:

- Java EE 6 and Java programming
- Basic Enterprise JavaBeans (EJB) and EJB 3.1 concepts**Note:** For more information about EJB 3.1, see Developing enterprise applications and Developing EJB 3.1 Applications

Lessons in this tutorial

- **Lesson 1.1: Create an EJB 3.1 project**

  This lesson leads you through the detailed steps to create an EJB 3.1 project used to contain your EJB session bean.

- **Lesson 1.2: Create required classes and interfaces for the SingletonCounterBean.java class**

  Lesson 1.2 steps you through the creation of required classes and interfaces for the `SingletonCounterBean.java` class.
- **Lesson 1.3: Create an entity class and a data source for data persistence**

  Lesson 1.3 leads you through the creation of an entity class and a database for data persistence.
- **Lesson 1.4: Create a web project to test your application**

  Lesson 1.4 leads you through the creation of a web project to test your application.
- **Lesson 1.5: Create a web fragment project**

  Lesson 1.5 leads you through the creation of a web fragment project to test your application.

< Previous | Next >

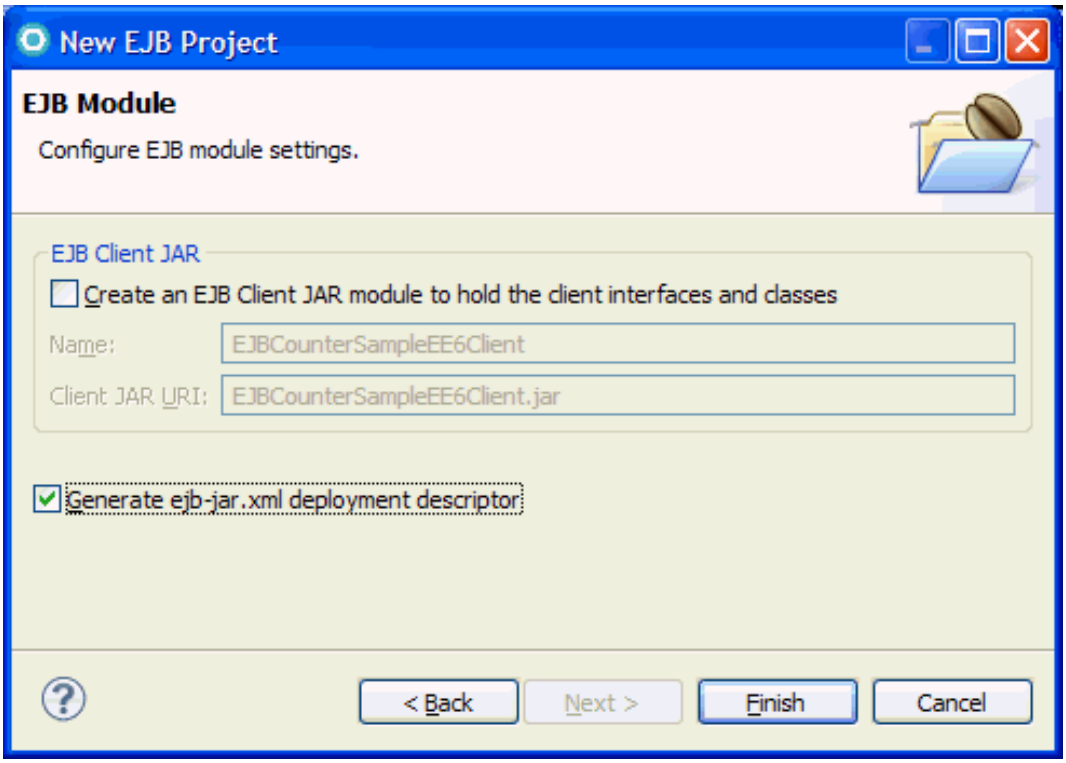
Lesson 1.1: Create an EJB 3.1 project

This lesson leads you through the detailed steps to create an EJB 3.1 project used to contain your EJB session bean.

In order for you to create an EJB 3.1 project, you need to have the WebSphere® Application Server version v8.0 installed, and you need to have a profile created for this server.

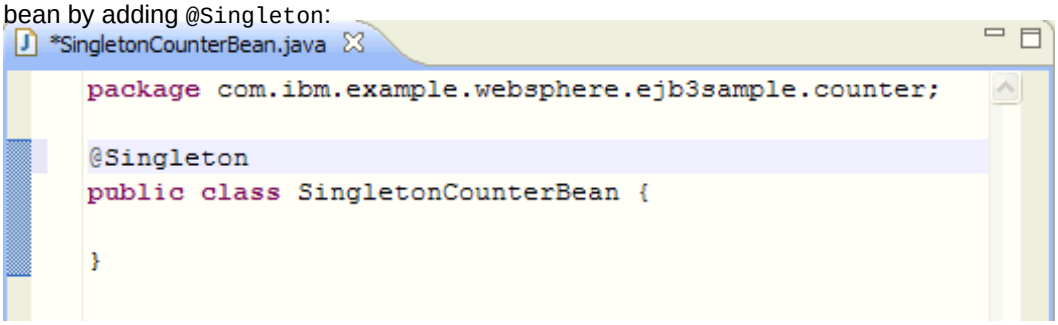
To create an EJB 3.1 counter project:

1. **Create a server configuration for the WebSphere Application Server v8.0**:
   A. Open the Servers view by selecting **Window** > **Show View** > **Servers**.
   B. Define a new WebSphere Application Server by right clicking the Servers view and selecting **New** > **Server**. Follow the instructions in the **New Server** wizard, ensuring that you select the WebSphere Application Server v8.0.
2. **Create an EJB 3.1 project**:
   A. If the Java™ EE icon,  , does not appear in the top right tab of the workspace, you need to switch to the Java EE perspective. From the menu bar, select **Window > Open Perspective > Other**. The Select Perspective window opens.
   B. Select **Java EE**. Click **OK**. The Java EE perspective opens.
   C. In the Java EE perspective, select **File** > **New** > **Other** > **EJB** > **EJB project**, and click **Next**.
   D. On the New EJB Project page,
      - in the **Project name** field, type `EJBCounterSampleEE6`,
      - in the **Project location** field, select `Use default location` or click **Browse** to select a different location.
      - in the **Target runtime** field, select `WebSphere Application Server v8.0`
      - in the **EJB module version** field, select `3.1`
      - in the **Configuration** field, select `Default configuration for WebSphere Application Server v8.0`.
      - Select **Add project to EAR**, and click **Next**.
      - In the **Working Sets** section, leave the **Add project to working sets** field clear, and click **Next**.
   E. On the Java page, accept the defaults and click **Next**.
   F. On the EJB Module page, clear **Create an EJB Client JAR module to hold the client interfaces and classes**, and select **Generate an ejb-jar.xml deployment descriptor**:

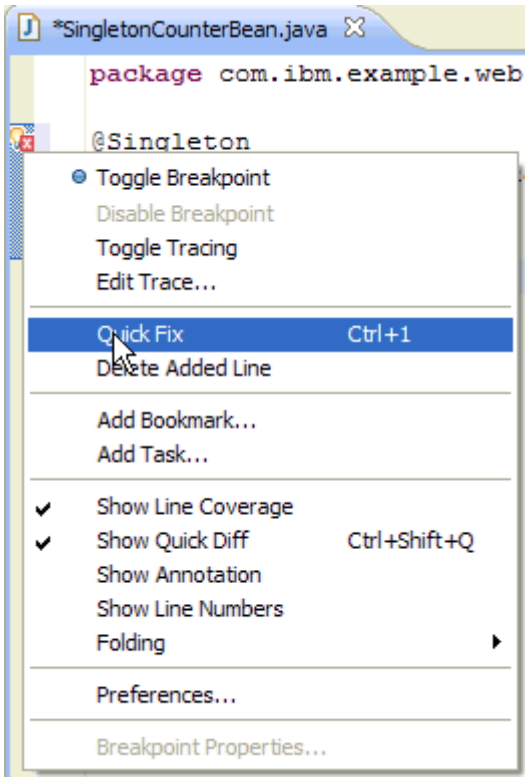


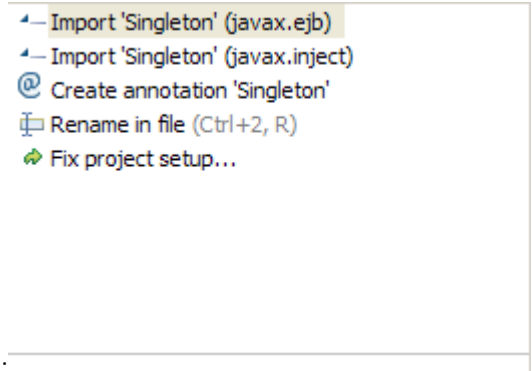
Accept the other default values and click **Finish**.

3. Add a Java class, @Singleton annotation:
   A. In the Enterprise Explorer view, right click the `EJBCounterSampleEE6` project and select **New** > **Class**.
   B. Accept the default **Source folder** (`EJBCounterSampleEE6/ejbModule`). In the **Package** field, type `com.ibm.example.websphere.ejb3sample.counter`, and in the **Name** field, type `SingletonCounterBean`.
   C. Click **Finish**.
   D. Your `SingletonCounterBean` class opens in the Java editor. Add the EJB 3.1 annotation to generate a singleton bean by adding `@Singleton`:



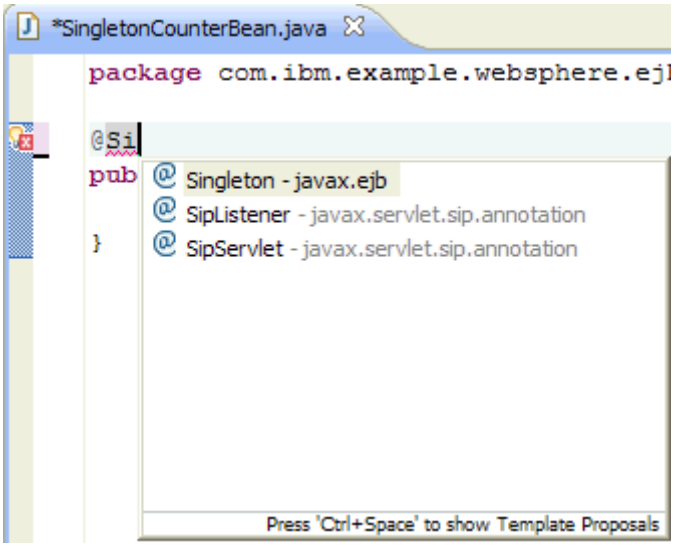
   E. When you press CTRL+S to save, you can see a quick fix icon beside the `@Singleton` line.
   F. Right click the quick fix icon and select **Quick Fix**:





   G. Select **Import 'Singleton' (javax.ejb)** and press CTRL+S to save:

   The required dependencies are automatically added to the source code.**Tip:** A shortcut to using the Quick Fix is to type `@Si`, press CTRL+Spacebar, and select **@Singleton - javax.ejb**.



The required dependencies are automatically added to the source code.

You now are ready to move on to Exercise 1.2, Create required classes and interfaces for the `SingletonCounterBean.java` class.

< Previous | Next >

Lesson 1.2: Create required classes and interfaces for the SingletonCounterBean.java class

Lesson 1.2 steps you through the creation of required classes and interfaces for the `SingletonCounterBean.java` class.

Before you begin, you must complete Lesson 1.1.

In this lesson, you

- Add `@Interceptors` annotation and required dependencies.
- Create an `Audit.java` Java™ class.
- Add code to your `SingletonCounterBean.java` class
- Create a `LocalCounter.java` interface and a `RemoteCounter.java` interface

1. Open your `SingletonCounterBean.java` class in the Java editor.
2. Under the `@Singleton` annotation, type `@Interceptors`.
3. When you press CTRL+S to save, you can see a quick-fix icon beside the `@Interceptors` line. Right click the quick-fix icon and select **Quick Fix**:
4. Select **import 'Interceptors' (javax.interceptor)**, and press CTRL+S to save.
5. Right click the quick-fix icon and select **Quick Fix**. Select **Add missing attributes** and replace **(value={null})** with `(Audit.class)`, press CTRL+S to save.
6. Right click the quick-fix icon and select **Quick Fix**. Select **Create class 'Audit'**. In the Java Class page, accept the defaults and click **Finish**.
7. In the Java editor for `Audit.java`, replace the default code with this code, and press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;


import java.io.Serializable;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;


public class Audit implements Serializable {

   private static final long serialVersionUID = 4267181799103606230L;

   @AroundInvoke

   public Object methodChecker (InvocationContext ic)
   throws Exception
   {
       System.out.println("Audit:methodChecker - About to execute method: " + ic.getMethod());
       Object result = ic.proceed();
       return result;
   }
```

```
}
```

8. Open your `SingletonCounterBean` class in the Java Editor and replace this code,`public class SingletonCounterBean {`

```
}
```

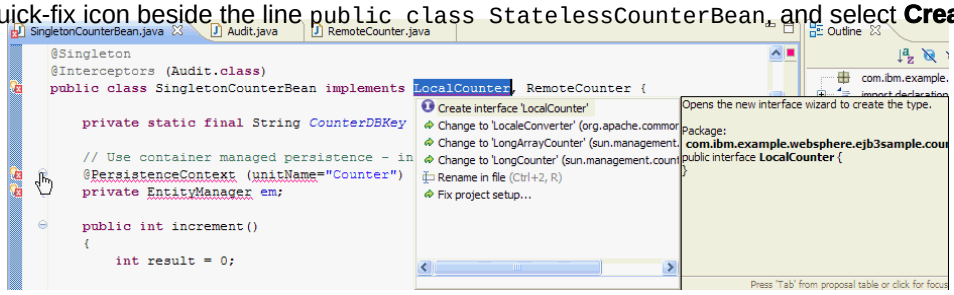
with this code:`public class SingletonCounterBean implements LocalCounter, RemoteCounter {`

```
    private static final String CounterDBKey = "PRIMARYKEY";

    // Use container managed persistence - inject the EntityManager
    @PersistenceContext (unitName="Counter")
    private EntityManager em;

    public int increment()
    {
        int result = 0;

        try {

            JPACounterEntity counter = em.find(JPACounterEntity.class, CounterDBKey);

            if ( counter == null ) {
                counter = new JPACounterEntity();
                counter.setPrimaryKey(CounterDBKey);
                em.persist( counter );
            }

            counter.setValue( counter.getValue() + 1 );
            em.flush();
            em.clear();

            result = counter.getValue();

        } catch (Throwable t) {
            System.out.println("SingletonCounterBean:increment - caught unexpected exception: " + t);
            t.printStackTrace();
        }

        return result;
    }


    public int getTheValue()
    {
        int result = 0;

        try {
            JPACounterEntity counter = em.find(JPACounterEntity.class, CounterDBKey);
```

```
            if ( counter == null ) {
                counter = new JPACounterEntity();
                em.persist( counter );
                em.flush();
            }

            em.clear();

            result = counter.getValue();
        } catch (Throwable t) {
            System.out.println("SingletonCounterBean:increment - caught unexpected exception: " + t);
            t.printStackTrace();
        }

        return result;
    }
}
```

Press CTRL+S to save.

9. Right click the quick-fix icon beside the line `public class StatelessCounterBean`, and select **Create Interface 'LocalCounter'**:



10. In the Java Interface page, accept the defaults and click **Finish**.
11. Your `LocalCounter.java` class opens in the Java Editor. Replace all the code with this code, and press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;

import javax.ejb.Local;

@Local
public interface LocalCounter {
   public int increment();
   public int getTheValue();
}
```

12. Return to the `SingletonCounterBean.java` class in the Java editor.
13. Follow the steps for creating a Local Counter to create a Remote Counter interface, and replace all the code in `RemoteCounter.java` with this code, and press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;
```

```
import javax.ejb.Remote;


@Remote
public interface RemoteCounter {
   public int increment();
   public int getTheValue();
}
```

14. Return to the `SingletonCounterBean.java` class in the Java editor.
15. Right click the quick fix icon beside `@PersistenceContext`, and select **Quick Fix**. Select **Import 'PersistenceContext' (javax.persistence)**, and press CTRL+S to save.
16. Right click the quick-fix icon beside `@EntityManager`, and select **Quick-Fix**. Select **Import 'EntityManager' (javax.persistence)**, and press CTRL+S to save.
17. Right click the quick-fix icon beside `JPACounterEntity`, and select **Quick Fix**. Select **Create class 'JPACounterEntity'**.
18. In the Java Class page, accept the defaults and click **Finish**.

You now are ready to move on to Exercise 1.3, Create an entity class and a database for data persistence.

< Previous | Next >

Lesson 1.3: Create an entity class and a data source for data persistence

Lesson 1.3 leads you through the creation of an entity class and a database for data persistence.

Before you begin, you must complete Lesson 1.2.

In this lesson you will

- Add code to the entity class, `JPACounterEntity.java`.
- Create a database, EJB3SampleDB, to persist the counter data.

1. **Add code to the entity class**:

   A. Open `JPACounterEntity.java` in the Java™ editor, replace all the code with this code, and press CTRL+S to save:

```
// This program may be used, executed, copied, modified and distributed
// without royalty for the purpose of developing, using, marketing, or distributing.

package com.ibm.example.websphere.ejb3sample.counter;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;


@Entity
@Table(name="EJB3COUNTERTABLE")

public class JPACounterEntity {

   @Id
   private String primarykey = "PRIMARYKEY";

   private int value = 0;

   public void setValue( int newValue )
   {
       System.out.println ("JPACounterEntity:setValue = " + newValue);
       value = newValue;
   }

   public int getValue()
   {
       System.out.println ("JPACounterEntity:getValue = " + value);
       return value;
   }

   public void setPrimaryKey( String newKey )
   {
       System.out.println ("JPACounterEntity:setPrimaryKey = '" + newKey + "'");
```

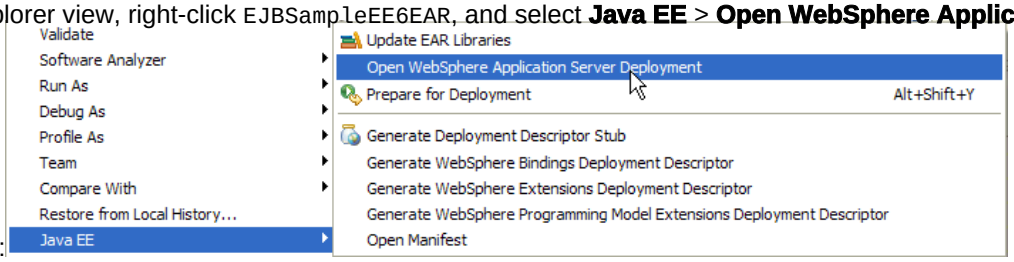
```
        primarykey = newKey;
    }

    public String getPrimaryKey()
    {
        System.out.println ("JPACounterEntity:getPrimaryKey = '" + primarykey + "'");
        return primarykey;
    }
}
```

B. In the Enterprise explorer view, navigate to `EJBCounterSampleEE6/ejbModule/META-INF`. Right-click on *META-INF* and select **New** > **File**. Type `persistence.xml` in the **File name** field, and click **Finish**. The `persistence.xml` file opens in the XML editor. Select Source, and copy and paste this code into the Source window:

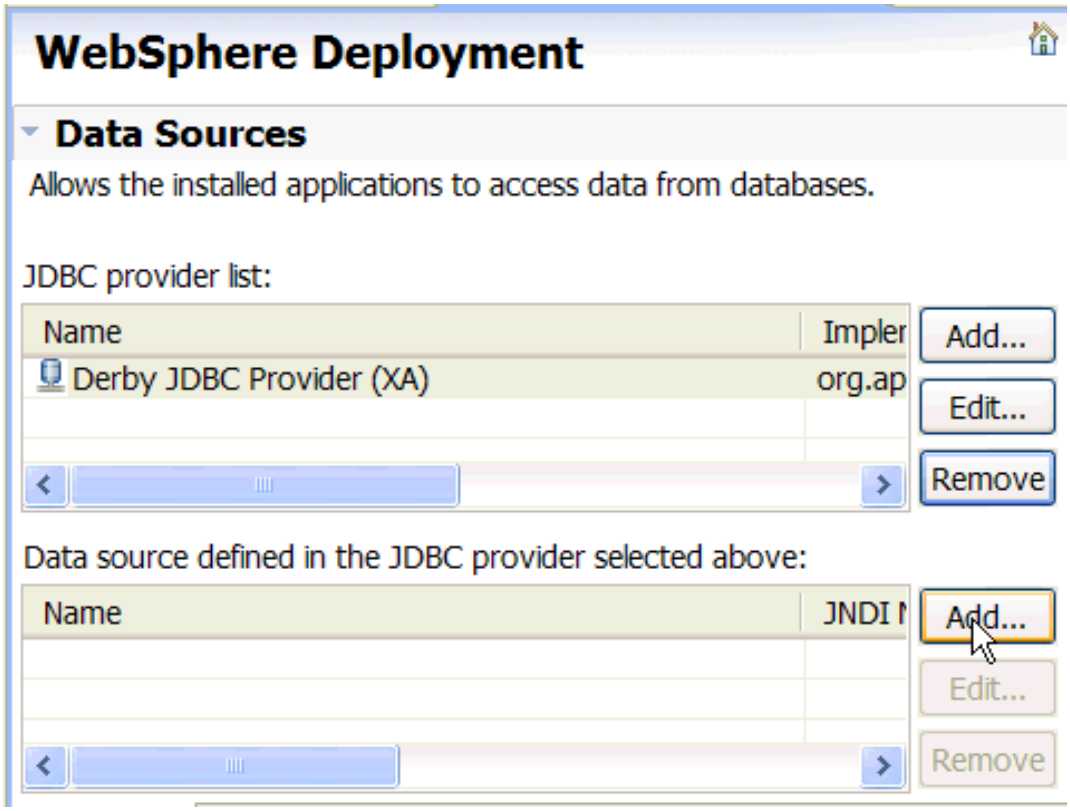
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
       <persistence-unit name="Counter">
              <jta-data-source>jdbc/EJB3SampleDatasource</jta-data-source>
              <class>com.ibm.example.websphere.ejb3sample.counter.JPACounterEntity</class>
              <exclude-unlisted-classes>true</exclude-unlisted-classes>
       </persistence-unit>
</persistence>
```

2. Define a data source

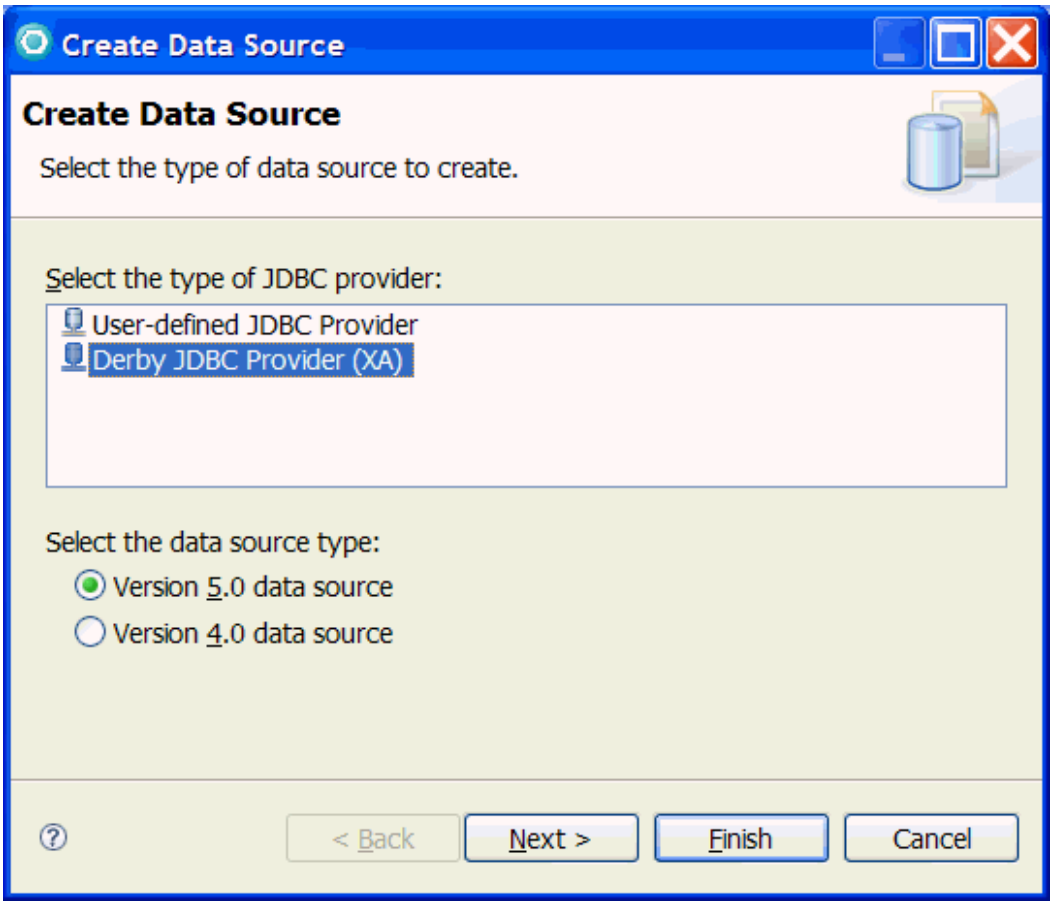
A. In the Enterprise Explorer view, right-click `EJBSampleEE6EAR`, and select **Java EE** > **Open WebSphere Application Server Deployment**:



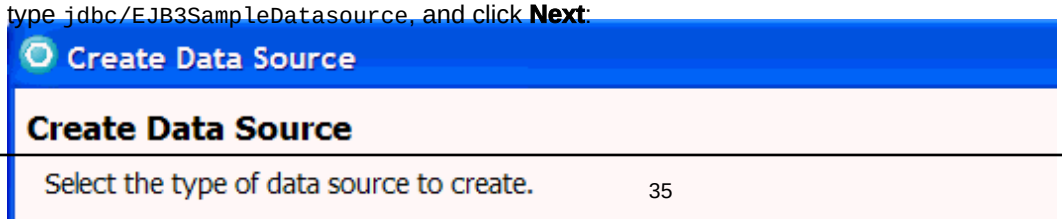
B. In the WebSphere® Deployment editor, select `Derby JDBC Provider (XA)`, and in the **Data source defined in the JDBC provider selected above:** field, click **Add**:



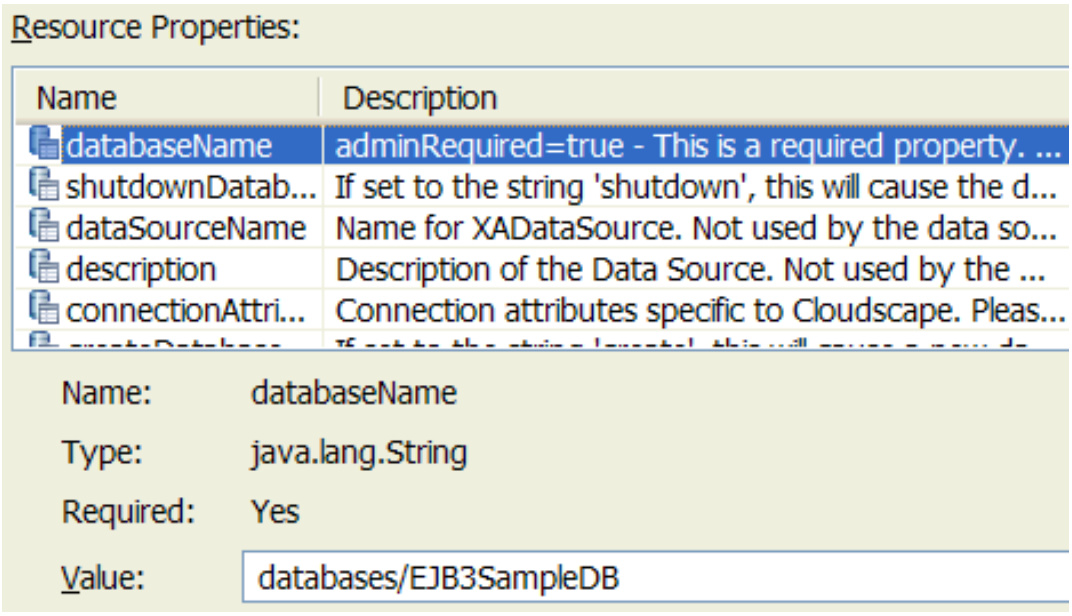
C. In the **Create Data Source** page, select `Derby JDBC Provider (XA)`, and click **Next**:



D. On the Create Data Source page, in the **Name** field, type `EJBCounterSample Data source`. In the **JNDI name** field, type `jdbc/EJB3SampleDatasource`, and click **Next**:



E. In the **Create Resource Properties** page, highlight **databaseName** property field, and in the **Value** field, type `databases/EJB3SampleDB`, and click **Finish**:



F. Press CTRL+S to save changes to the WebSphere Application server deployment page.

You now are ready to move on to Exercise 1.4, Create a Web project to test your application.

< Previous | Next >

Lesson 1.4: Create a web project to test your application

Lesson 1.4 leads you through the creation of a web project to test your application.

Before you begin, you must complete Lesson 1.3.

In this lesson you will

- Create a web project, EJBCounterWebEE6
- Create a web fragment project, EJBCounterWebFragEE6
- Create a web page, EJBCount.jsp
- Create a Servlet, EJBCount.java
- Run the Servlet to test your application

1. Extract the EJB3CounterDB.zip
   A. Import the required database: The import wizard imports a database to provide persistence for the EJB 3.1 Counter project.Import the EJB 3.1 tutorial resources
   B. After you have imported the EJB3CounterDB, expand **EJB3CounterDB** > **EJB3CounterDB.zip** and double click EJB3CounterDB.zip. A file extract utility opens in an external window.
   - Windows: Extract the database into your `/derby/databases` folder of your WebSphere® Application Server installation folder:
   - Linux: Extract the database into your `/derby/databases` folder of your WebSphere Application Server installation folder.
     - Give your non-root user access to the databases directory. (The easiest way is to give everyone access: `chmod ugo+x databases`.
     - Give your non-root user write-access to the extracted database. (For example, you can extract as the non-root user, provided you have access to the `databases` directory).

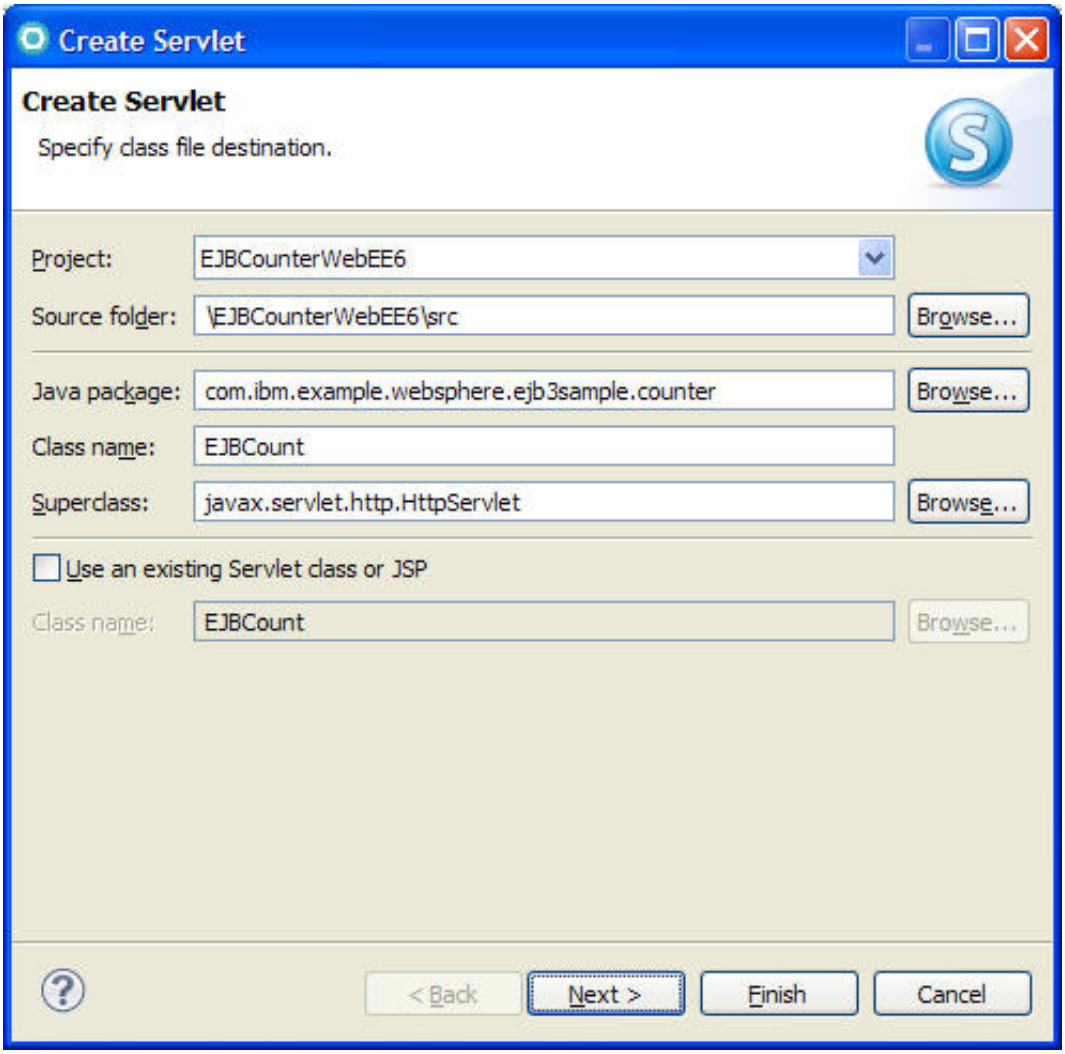
   Important: Depending on the type of WebSphere Application Server, the default location of your `/derby/databases` may differ. For information about default installation directories, see Creating a WebSphere Application Server.
2. In the Java™ EE perspective, right-click your enterprise application project and select **New** > **Web Project** to open the web project wizard.
3. On the Web Project page,
   A. In the **Project name** field, type `EJBCounterWebEE6`.
   B. In the **Project Templates** field, select **Simple**.
   C. In the **Programming Model** field, select **Java EE**.
   D. On the deployment page, from the list of available configuration options, click **Deployment** to open the Deployment configuration page.
   - In the **Target runtime** select WebSphere Application Developer v8 from the drop-down box.
   - Clear **Add support for WebSphere bindings and extensions**.
   - In the **Web module version** field, select **3.0**
   - In the **EAR membership** field, click **Add project to an EAR**..
   - In the **EAR project name** field, ensure that **EJBCounterSampleEE6EAR** appears.

   E. Accept the other defaults and click **Finish**. If asked to **Open Associated Perspective?**, click **No**.
4. Right-click the `EJBCounterWebEE6` project, and select **New** > **Web Page**.
5. On the **New Web Page** page, in the **File name** field, type `EJBCount.jsp`, click **Finish**.
6. In the Source view of the Web page editor, replace all the existing code with this code, and press CTRL+S to save:

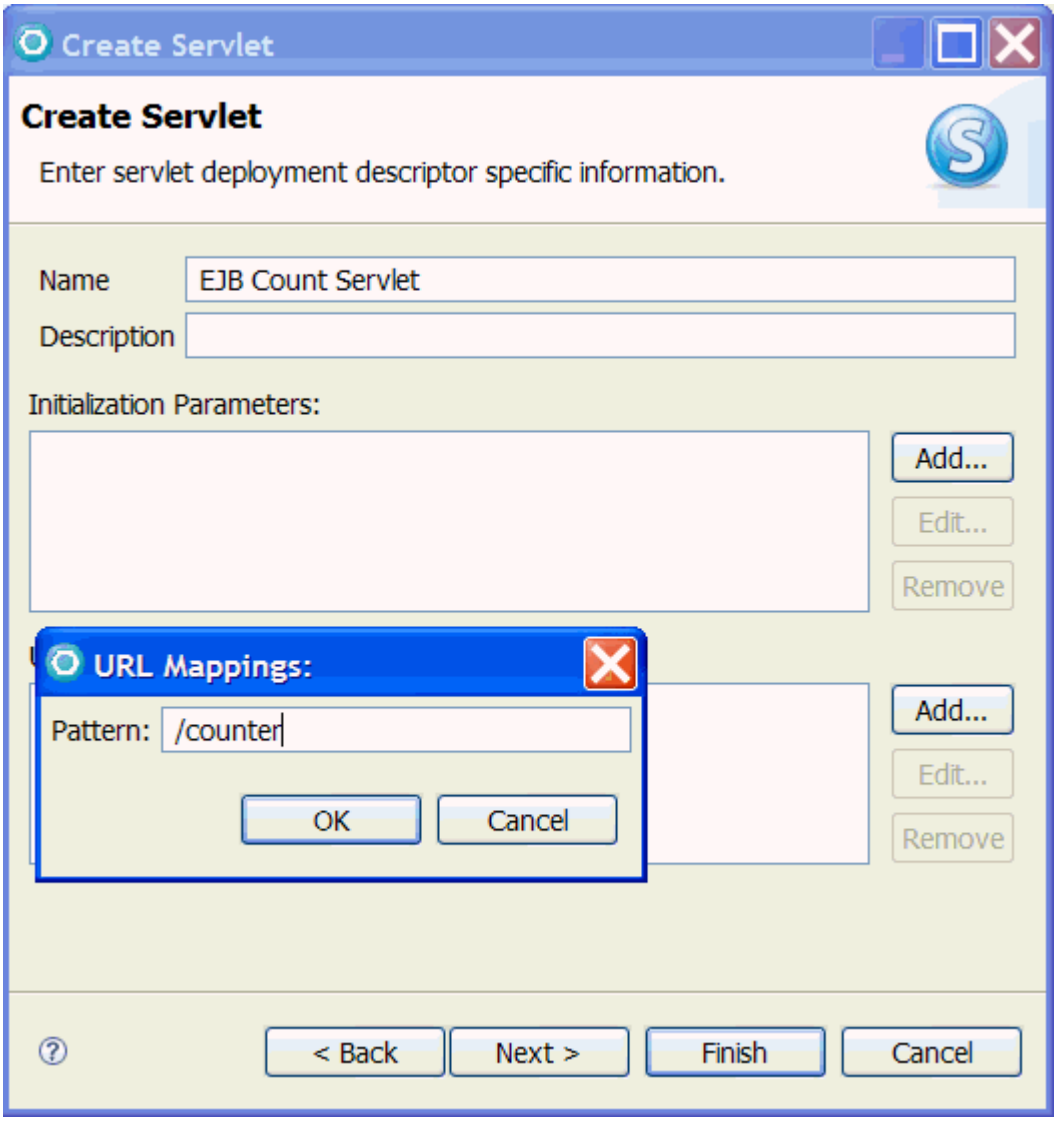
```
<%@page session="false"%>
```

```
<HTML>
<HEAD>
<TITLE>IBM WebSphere EJB 3.1 and JPA 2.0 Counter Sample</TITLE>
<BODY bgcolor="cornsilk">
<H1>EJB 3.1 and JPA 2.0 Counter Sample</H1>
<P>
<B>
This application communicates with the WebSphere Application Server using http requests to increment a singleton EJB
3.1 counter bean which is using a JPA 2.0 entity (ie. keeps a persistent counter in a Derby database table).
</B>
<FORM METHOD=POST ACTION="counter">
<BR/>
<%
   response.setHeader("Pragma", "No-cache");
   response.setHeader("Cache-Control", "no-cache");
   response.setDateHeader("Expires",0);
   String msg = (String) request.getAttribute("msg");
   if (msg == null) msg = "";
%>
<B>Click on the Increment button to increment the count</B>
<BR/><BR/>
<INPUT TYPE=SUBMIT VALUE="Increment">
</FORM>
<H3><%=msg%></H3>
</BODY>
</HTML>
```

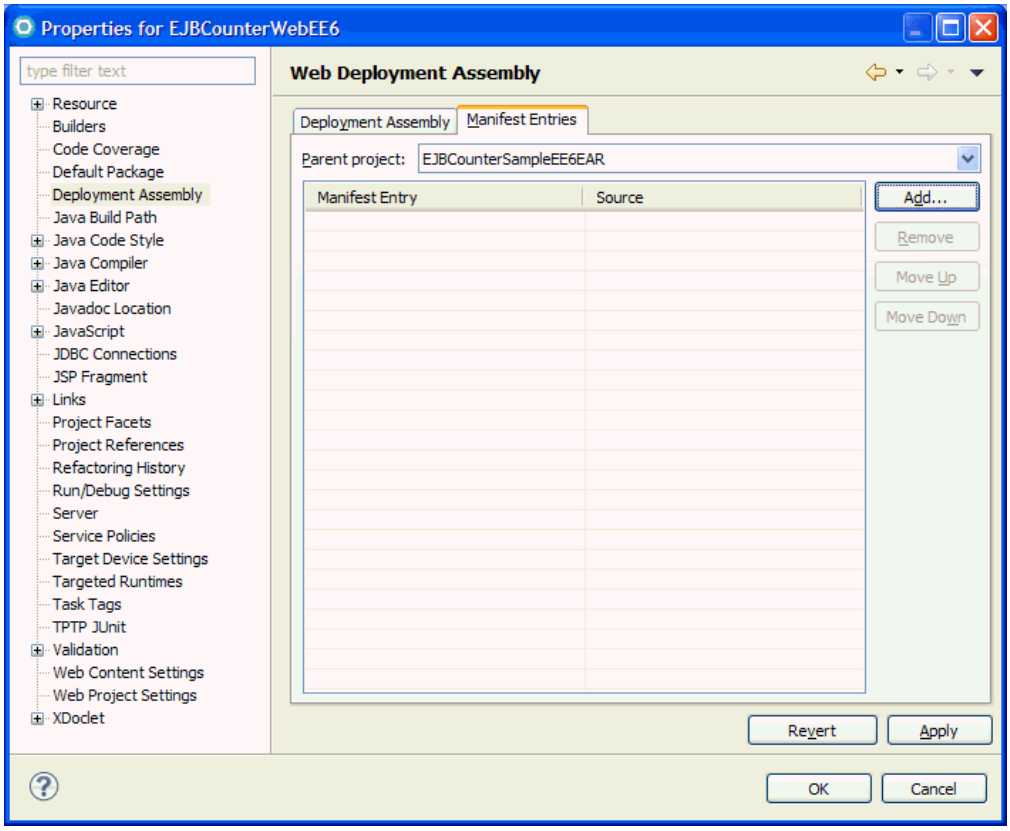
7. Right click the `EJBCounterWebEE6` project, and select **New** > **Servlet**.
8. On the **New Servlet** page, in the **Java package** field, type `com.ibm.example.websphere.ejb3sample.counter`.
9. In the **Class name** field, type `EJBCount`, and click **Next**:



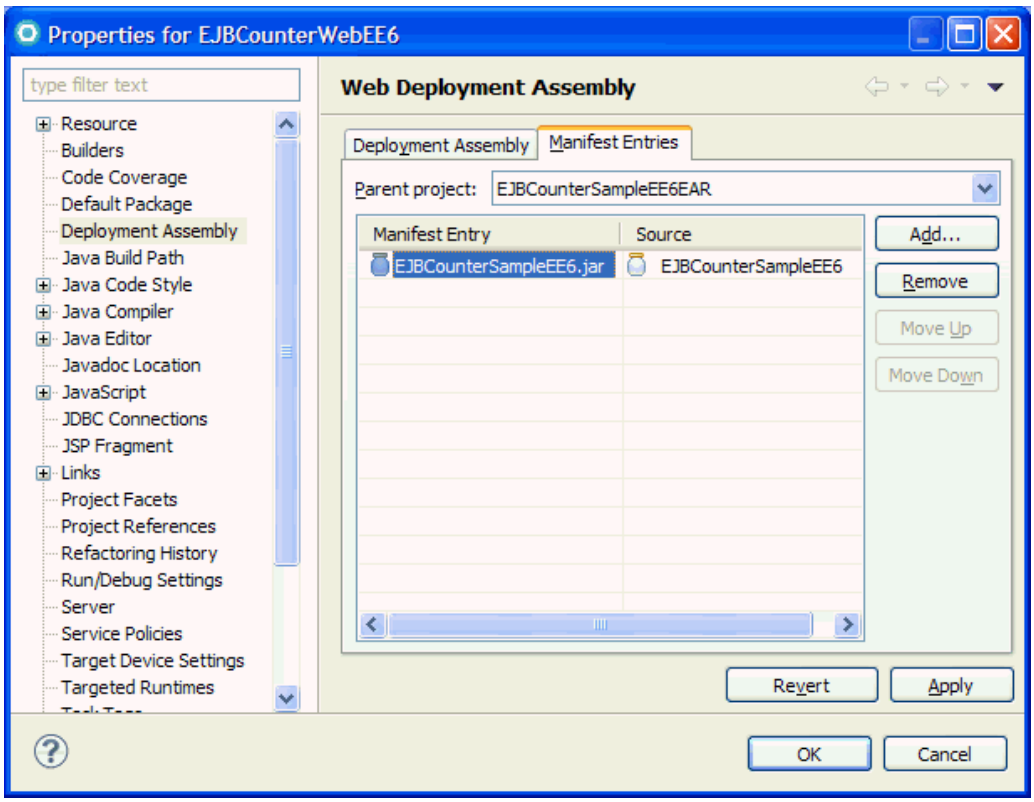
10. In the **Name** field, type `EJB Count Servlet`. In the **URL mappings** field, edit the existing mapping, highlight /EJB Count Servlet and click **Edit**. Replace the pattern with `/counter`, and click **Finish**:



11. Add deployment assembly entries:
    A. Right click the `EJBCounterWebEE6` project, and select **Properties**.
    B. Select **Deployment Assembly**, select **Manifest Entries**, and click **Add**.



C. Select `EJBCounterSampleEE6.jar`, and click **Finish**, and then click **OK**.



12. Expand **EJBCounterWebEE6** > **Java Resources** > **src** > **com.ibm.example.websphere.ejb3sample.counter**, and double click the `EBJCount.java` file. It opens in the Java editor.
13. Replace the existing code with the following code, and press CTRL+S to save: `package com.ibm.example.websphere.ejb3sample.counter;`

```
// This program may be used, executed, copied, modified and distributed
```

```
// without royalty for the purpose of developing, using, marketing, or distributing.

import java.io.IOException;

import javax.ejb.EJB;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
* This servlet demonstrates an EJB3 counter bean with JPA.
*/

@WebServlet(
  description="This servlet demonstrates the various ways to increment EJB 3.1 counter beans.",
  name="EJB Count Servlet",
  displayName="EJB Count Servlet",
  urlPatterns="/counter"
  )
public class EJBCount extends HttpServlet {

   private static final long serialVersionUID = -5983708570653958619L;

   // Use injection to get the ejb
   @EJB private LocalCounter singletonCounter;

   public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
  String msg = null;
  int ejbCount = 0;

  try {
   ejbCount = singletonCounter.getTheValue();
  }
  catch (RuntimeException e) {
   msg = "Error - getTheValue() method on EJB failed!";
        e.printStackTrace();
  }
  msg = "EJB Count value for Singleton Bean with JPA: " + ejbCount;

  // Set attributes and dispatch the JSP.
       req.setAttribute("msg", msg);
       RequestDispatcher rd = getServletContext().getRequestDispatcher("/EJBCount.jsp");
       rd.forward(req, res);
 }

   public void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
```

```
    String msg = null;
    int ejbCount = 0;

    try {
     ejbCount = singletonCounter.increment();
    }
    catch (RuntimeException e) {
     msg = "Error - increment() method on EJB failed!";
          e.printStackTrace();
    }
    msg = "EJB Count value for Singleton Bean with JPA: " + ejbCount;

    // Set attibutes and dispatch the JSP.
          req.setAttribute("msg", msg);
          RequestDispatcher rd = getServletContext().getRequestDispatcher("/EJBCount.jsp");
          rd.forward(req, res);
   }


  }
```

You now are ready to move on to Exercise 1.5, Create a web fragment project to test your application.

< Previous | Next >

Lesson 1.5: Create a web fragment project

Lesson 1.5 leads you through the creation of a web fragment project to test your application.

Before you begin, you must complete Lesson 1.4.

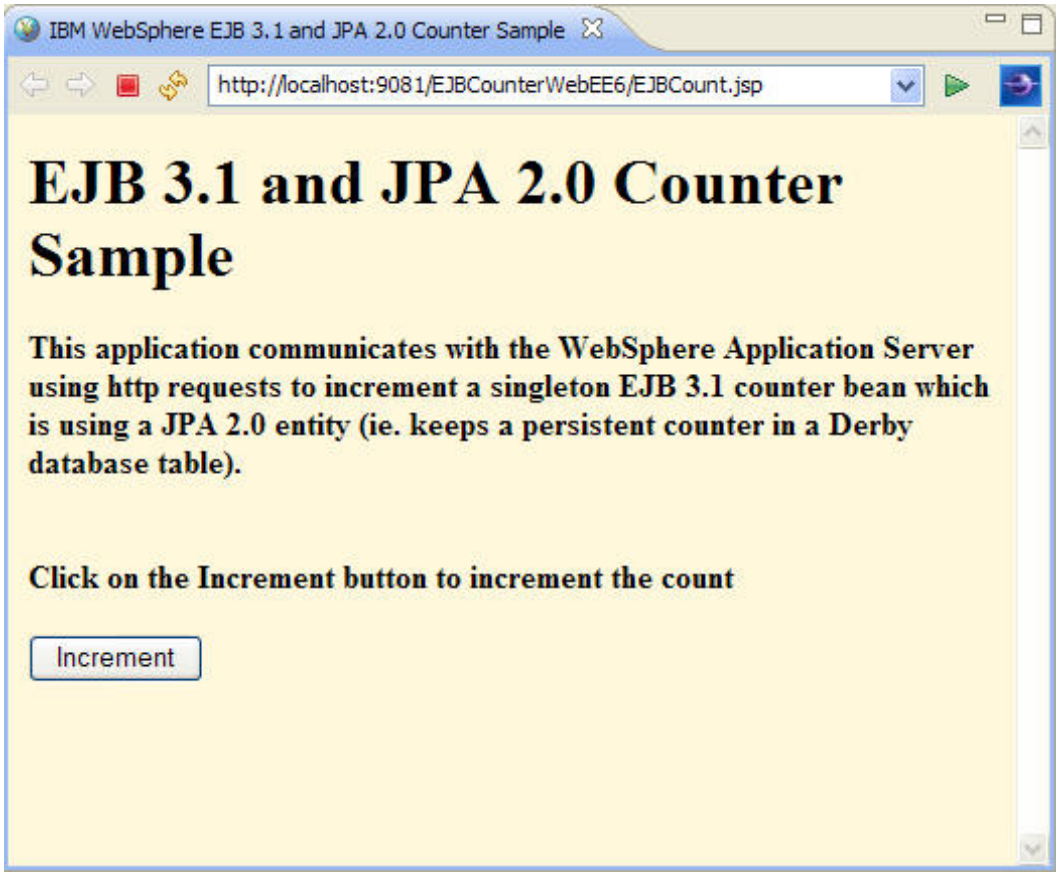
In this lesson you will

- Create a web fragment project, `EJBCounterWebFragEE6`

1. In the Java™ EE perspective, select **File** > **New** > **Other** > **Web Fragment Project**, and click **Next**.
2. On the Web Fragment Project page,
   A. In the **Project name** field, type `EJBCounterWebFragEE6`.
   B. In the **Project location** field, select `Use default location` or click **Browse** to select a different location.
   C. In the **Target runtime** field, accept `WebSphere Application Server v8.0`.
   D. In the **Configuration** field, select `Default configuration for WebSphere Application Server v8.0`.
   E. In the **Dynamic Web project membership** field, select **Add project to a Dynamic Web project**, and in the **Dynamic Web project name** field, select `EJBCounterWebEE6`.
   F. Accept the other defaults and click **Finish**. If asked to **Open associated perspective?**, click **No**
3. In the Enterprise Explorer view, expand your web fragment project to find the web-fragment.xml file. Select **EJBCounterWebFragEE6** > **src** > **META-INF** > **web-fragment.xml**, and right-click and select **Open With** > **Web Fragment 3.0 Deployment Descriptor Editor**. Select **Source**, and replace all the code in the file with this code:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-fragment id="WebFragment_ID" version="3.0"
 xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-fragment_3_0.xsd">
 <display-name>EJBCounterWebFragEE6</display-name>
 <name>EJBCounterWebFragEE6</name>
</web-fragment>
```

   Press CTRL+S to save.
4. To test the web application,expand the **EJBCounterWebEE6** > **Web Content**, and right-click the `EJBCount.jsp` file, and select **Run** > **Run on Server**
5. On the Run On Server page, select `Choose an existing server` and select `WebSphere Application Server v8.0`. Click **Finish**.
6. The counter application opens in a web browser:



You have now completed the EJB 3.1 Counter tutorial.

< Previous | Next >

< Previous

Create an EJB 3.1 application summary

This tutorial guided you through the detailed steps to create an EJB 3.1 application that contains an EJB 3.1 project and a web project to display a counter program.

From this tutorial, you have learned how to create an EJB project and to use the EJB 3.1 annotation support for creating EJB 3.1 session beans. You also created an enterprise application project and a web project to deploy your application on a WebSphere® application server.

Lessons learned

While completing the exercises, you learned how to:

- Create an EJB 3.1 project, called `EJBCounterSample`, with an EJB 3.1 Singleton bean (with both interface and implementation classes) and a JPA 2.0 entity class
- Create a web project (`EJBCounterWebEE6`), and with a Java™ Server Page (JSP) page and a utility Java class
- Create a web fragment project (`EJBCounterWebFragEE6`).
- Create a Java EE 6 application (`EJBCounterSampleEAREE6`) with the EJB and web projects

< Previous

Next >

OSGi EJB tutorial

Learning objectives

In this tutorial you will learn how to create an OSGi application that exposes an EJB as a service. It demonstrates how to create OSGi bundles with EJB support, use the OSGi tools to manage EJB exports, and create a servlet that accesses the EJB as an OSGi service. This tutorial shows how to create the same application that is demonstrated in the OSGi EJB temperature converter sample.

The application developed in this tutorial is a simple temperature conversion application, that requires four projects:

- An OSGi application ConverterApp to include the bundles developed.
- An OSGi bundle project called EJB that has EJB support. The EJB will be exposed as a service using the OSGi header Export-EJB.
- An EJB client project EJBClient to contain the interface code for the EJB. EJBClient will have OSGi bundle support.
- An OSGi bundle project called Web that has Web 3.0 support. This project will include a servlet that will be configured to access the EJB that is exposed as an OSGi service.

To complete this tutorial, you will need approximately 60 minutes.

When you are ready, begin Lesson 1: Create an OSGi project with EJB support

Related information:

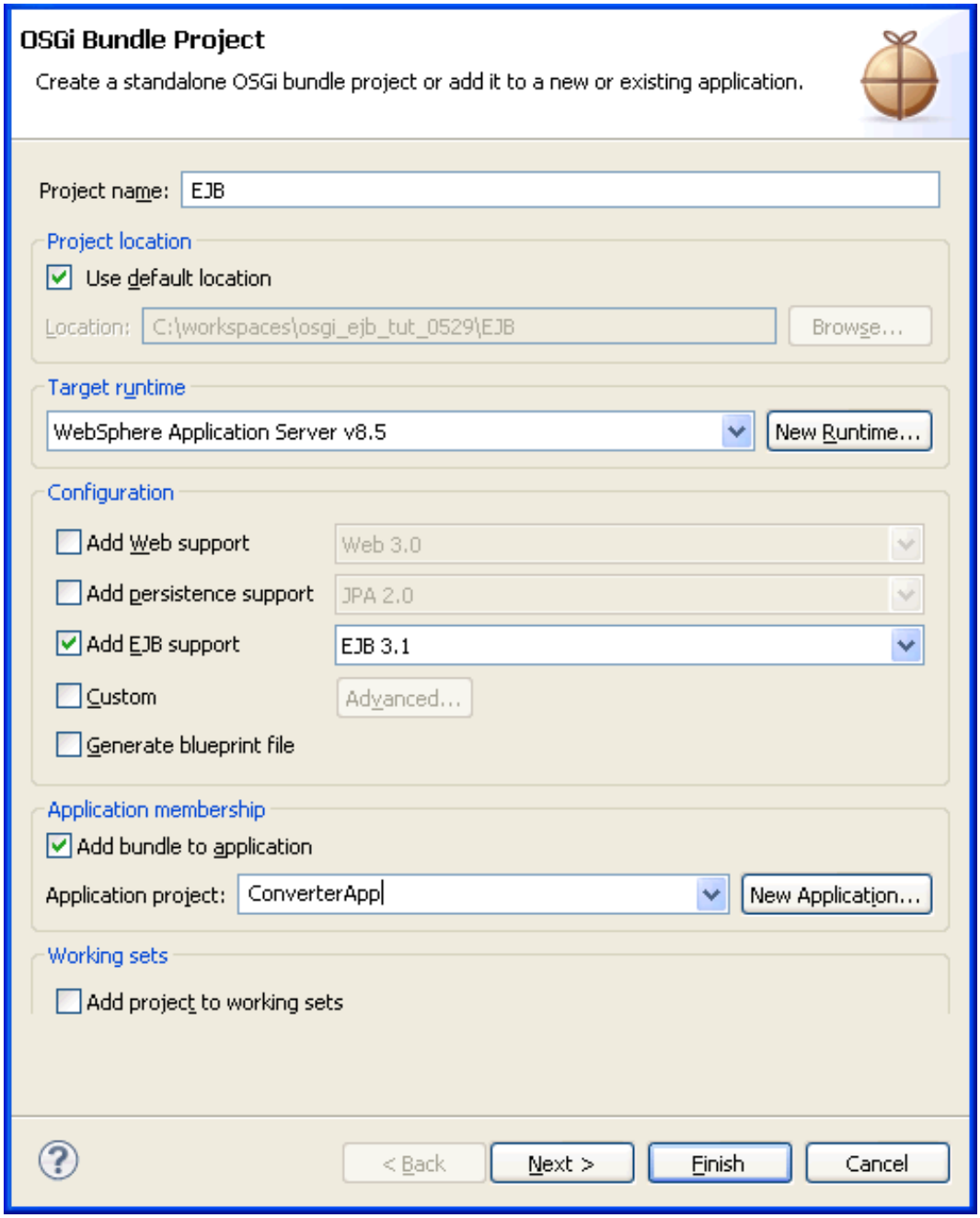
OSGi EJB temperature converter sample

Next >

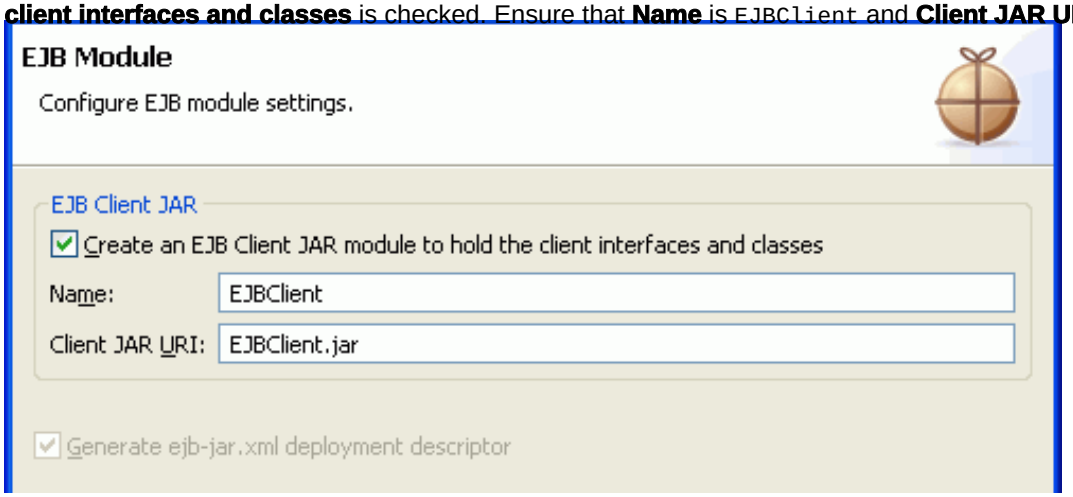
Lesson 1: Create an OSGi project with EJB support

In this lesson you will create the initial projects required for the application. You will use the OSGi bundle project wizard to create: a bundle with EJB support, an EJB client project to contain client interfaces, and an OSGi application that includes the bundles.

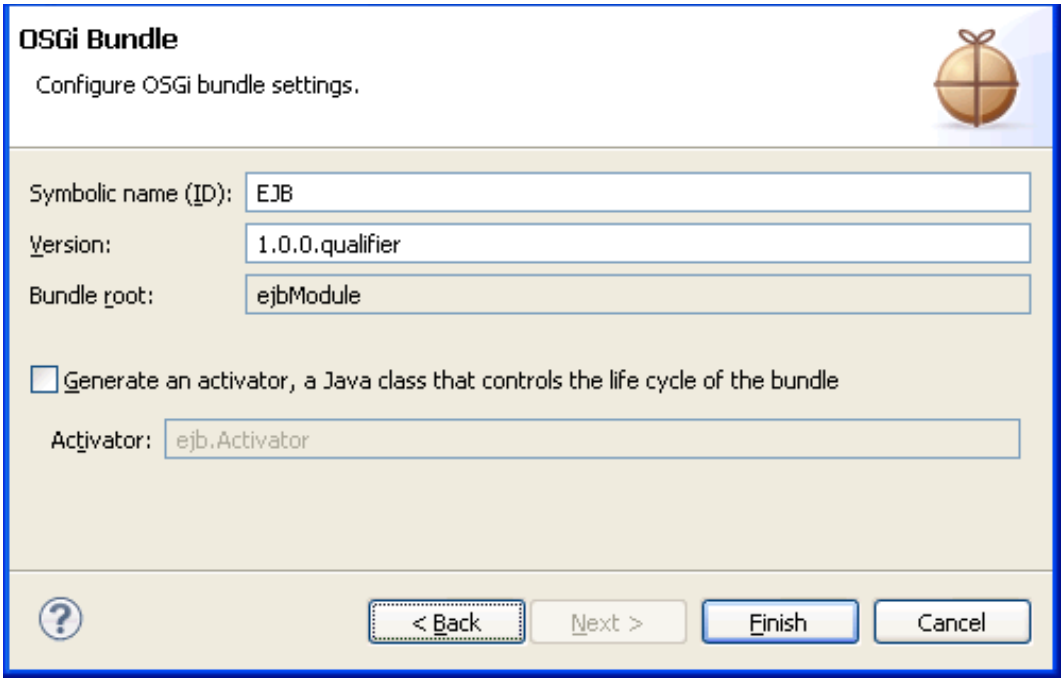
1. Launch the OSGi bundle project wizard. Click **File** > **New** > **OSGi Bundle Project**.
2. Create a new project with EJB support. In the OSGi bundle project wizard, enter the following values:
   - **Project name**
     - EJB
   - **Target runtime**
     - WebSphere Application Server v8.5
   - **Add EJB support**
     - Check **Add EJB support** and for this tutorial select **EJB 3.1**
   - **Add bundle to application > Application project**
     - Check **Add bundle to application** and for **Application project** enter `ConverterApp`



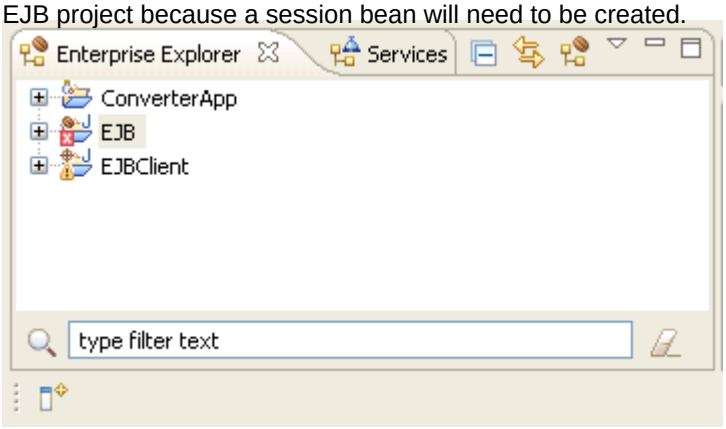
3. Click **Next** until you are at the EJB Module screen. Ensure that the box **Create an EJB Client JAR module to hold the client interfaces and classes** is checked. Ensure that **Name** is `EJBClient` and **Client JAR URI** is `EJBClient.jar`.



4. Click **Next** to go to the OSGi Bundle screen. Accept the default values.



5. Click **Finish**. The ConverterApp, EJB, and EJBClient projects are created. At this point, there may be an error in the EJB project because a session bean will need to be created.



< Previous | Next >

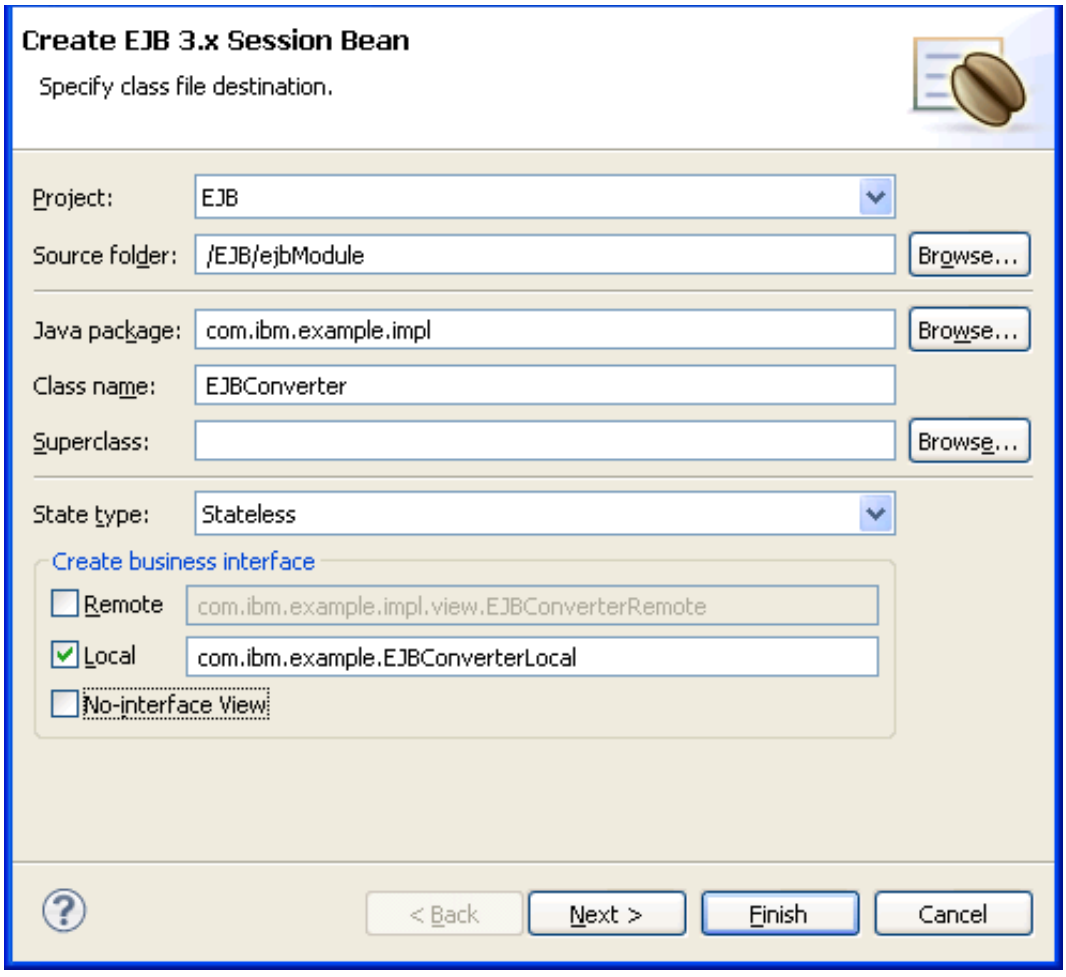
Lesson 2: Develop an EJB

In this lesson you will create a session bean, develop a local interface, and develop the implementation class for the bean. You will also use the manage EJB exports dialog to expose the EJB as an OSGi service.

Create a session bean

In this section you will create a session bean and specify the local interface.

1. Open the session bean wizard. In the EJB project, right-click ejbModule and select **New** > **Session Bean (EJB 3.x)**.
2. Create the bean. Enter the following values in the wizard:
   - **Project**
     - EJB
   - **Java package**
     - com.ibm.example.impl
   - **Class name**
     - EJBConverter
   - **State type**
     - Stateless
   - **Local**
     - Check the **Local** box and enter the following value: `com.ibm.example.EJBConverterLocal`
   - **No-interface View**
     - Uncheck **No-interface View**



Note: The application developed in this tutorial uses a local interface. If you want to use remote interfaces, select the **Remote** box in the wizard and then develop the corresponding interface. JNDI lookups for remote interfaces work differently, though. With remote interfaces you will need to adjust your lookup code. The next lesson will give an example.

3. Click **Finish**. The session bean is created and the class opens in the editor.

Develop the local interface

When you created the session bean, the EJBConverterLocal interface was created in the EJBClient project. In this section you will add two methods to the interface. The methods describe basic temperature conversion operations.

1. Open the EJBConverterLocal interface. In the EJBClient project, expand the ejbModule folder and the com.ibm.example package. Double-click the EJBConverterLocal interace. The interface opens in the editor.
2. Add methods to the interface. Add the following methods to the interface:
```
public double convertToFahrenheit(double celsius);
public double convertToCelsius(double fahrenheit);
```
The completed interface code should look like the following:
```
package com.ibm.example;

public interface EJBConverterLocal {

  public double convertToFahrenheit(double celsius);

  public double convertToCelsius(double fahrenheit);

}
```
3. Save your changes. Note that when you save the interface file, errors will be displayed for the EJBConverter class because it does not yet implement the methods that you added to the interface.

Develop the implementation class

In this section you will implement the temperature conversion methods required by the local interface.

1. Open the EJBConverter class. In the EJB project, expand the ejbModule folder and the com.ibm.example.impl package. Double-click the EJBConverter class. The class opens in the editor.
2. Remove the default constructor.
3. Add the implementations for the two methods you added to the EJBConverterRemote interface. Add the following method implementations to the class:
```
public double convertToCelsius(double fahrenheit) {
  return (fahrenheit - 32) * (5 / 9d);
}

public double convertToFahrenheit(double celsius) {
  return celsius * (9 / 5d) + 32;
}
```
The completed class should look like the following. Note that the original code generated by the wizard contains an annotation @Stateless and an annotation @Local that specifies the local interface that you entered in the session bean wizard.
```
package com.ibm.example.impl;

import com.ibm.example.EJBConverterLocal;
import javax.ejb.Local;
import javax.ejb.Stateless;
```

```
@Stateless
@Local(EJBConverterLocal.class)
public class EJBConverter implements EJBConverterLocal {

  public double convertToCelsius(double fahrenheit) {
     return (fahrenheit - 32) * (5 / 9d);
  }

  public double convertToFahrenheit(double celsius) {
     return celsius * (9 / 5d) + 32;
  }
}
```

4. Save your changes.

Exposing an EJB as an OSGi service

By default, EJBs that you add to an OSGi project that has EJB support are automatically exposed as OSGi services. This is controlled through the Export-EJB header in the OSGi manifest. In this section you will check the manifest and use the manage EJB exports dialog to confirm that EJBConverter is exposed as an OSGi service.

1. Open the manifest. In the EJB project, double-click **Manifest:EJB**. The manifest opens. Click the MANIFEST.MF tab to view the file in text format. Note that there is an **Export-EJB** header with an entry for the EJBConverter EJB. Note that you can manage EJB exports by manually adding entries to the Export-EJB header. EJBs can be added as a comma separated list. **Note:** There are two special case uses of the Export-EJB header to be aware of:

   - NONE

   - If you specify `NONE` as an entry for the Export-EJB header, no EJBs will be exposed as services. If you specify `NONE` but then also add an EJB to the list, the tools will display a warning.

   - BLANK

   - If you have an Export-EJB header in the manifest but there are no entries, by default all EJBs in the project will be exposed as services.

2. Open the manage EJB export dialog. Right-click the EJB project and select **OSGi** > **Expose EJBs as OSGi services**. The manage EJB exports dialog opens. In the dialog, confirm that the box next to the EJBConverter EJB is selected and click **OK**. You can use this dialog in your projects to add and remove EJBs that are exposed as services. Changing the EJBs selected in this dialog changes the entries to the Export-EJB header in the manifest.

Note: If you create new EJBs to the project, they will automatically be exposed as OSGi services. You can use the

manage EJB exports dialog to deselect any EJBs that you do not want to expose.

< Previous | Next >

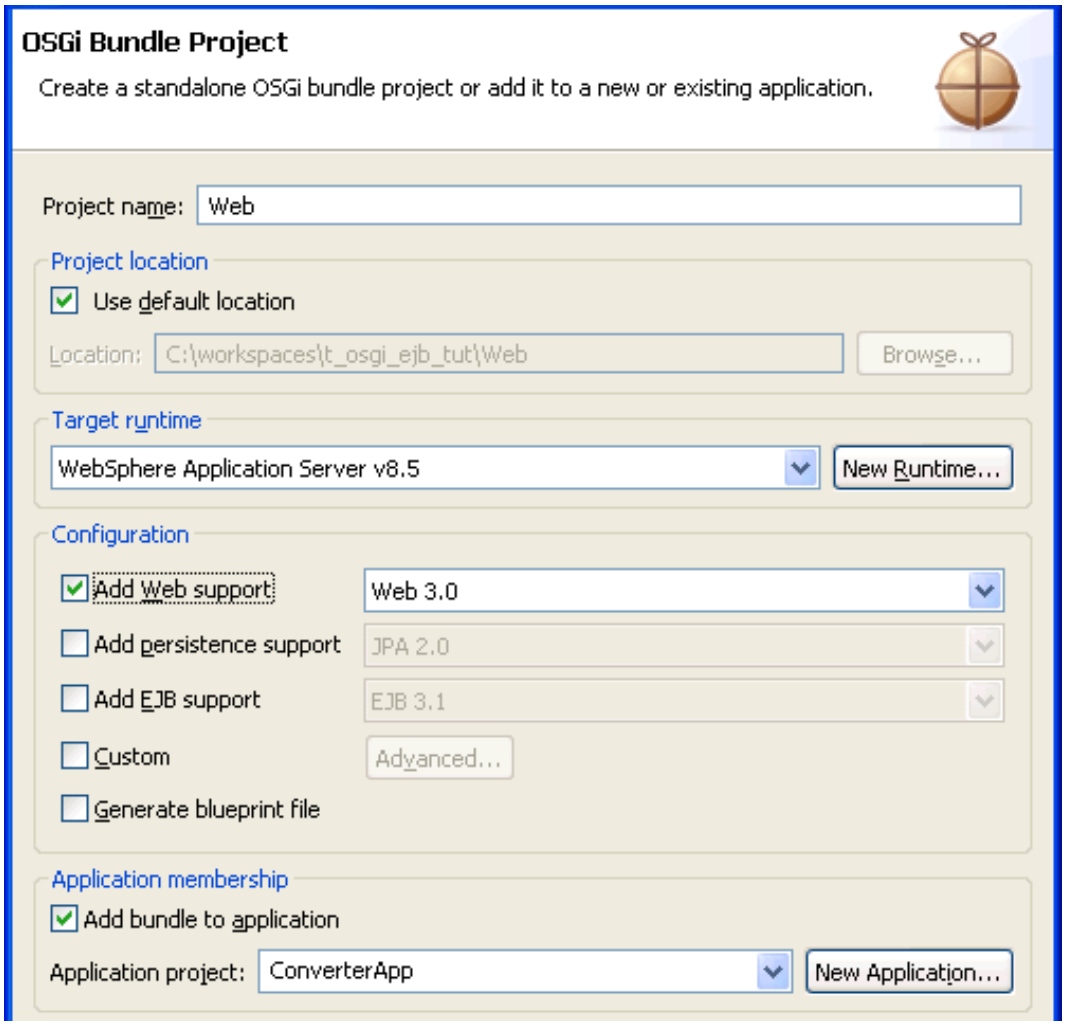
Lesson 3: Develop a Web bundle that accesses the EJB

In the previous lesson, you exposed the EJB as an OSGi service. In this lesson you will develop a Web bundle that accesses the service. You will create the bundle, configure the manifest to import required packages, develop a servlet that will access the EJB as an OSGi service, and create an HTML page to access the application through a browser.

Create a Web bundle project

In this section you will create an OSGi bundle with Web 3.0 support and add the bundle to the ConverterApp OSGi application.

1. Open the OSGi bundle project wizard. Click **File** > **New** > **OSGi Bundle Project**. The wizard opens.
2. Create the project. Enter or select the following values in the wizard:
   - **Project name**
     - Web
   - **Target runtime**
     - WebSphere Application Server v8.5
   - **Add Web support (check this box and select the following value)**
     - Web 3.0
   - **Add bundle to application (check this box and select the following application)**
     - ConverterApp

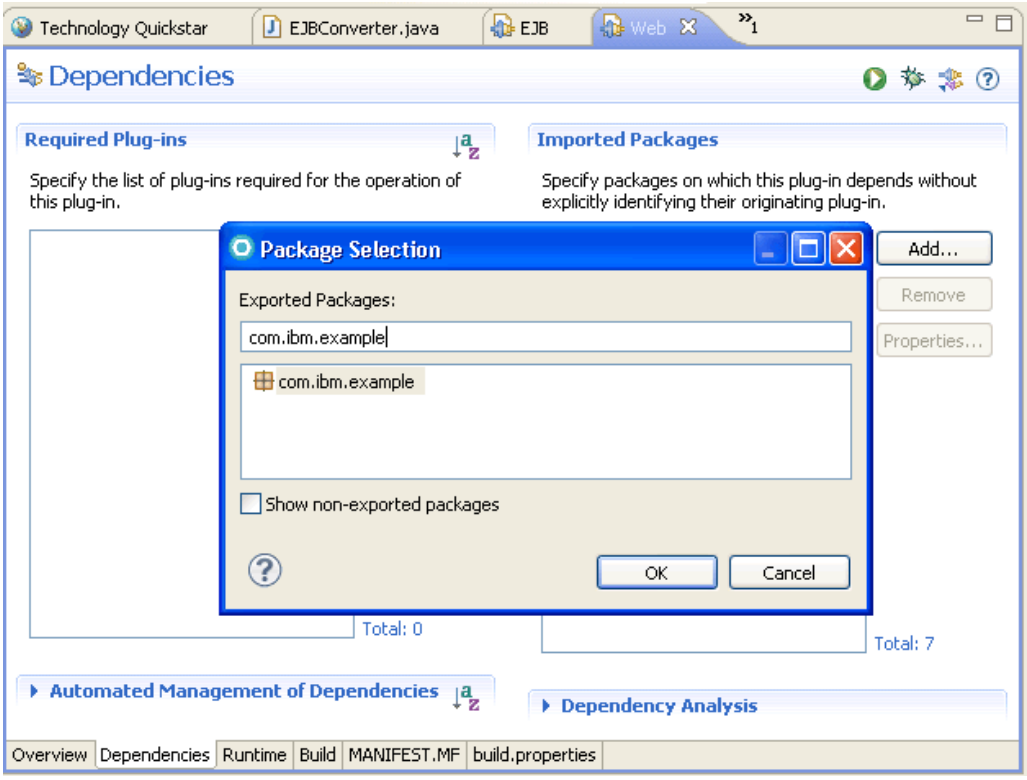


3. Click **Finish**. The project is created.

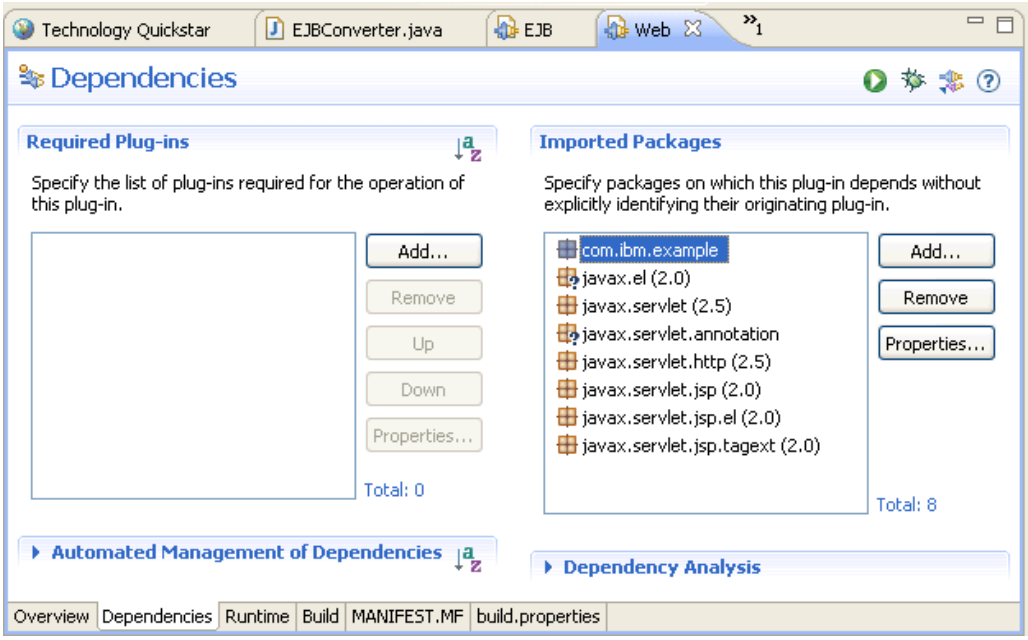
Configure the bundle manifest in the Web project to import the package from the EJBClient project

In this section, you will add an entry for the com.ibm.example package in the EJBClient project to the Import-Package header in the manifest. The com.ibm.example package contains the interface you developed for the EJB.

1. Open the manifest. In the Web project, double-click **Manifest: Web**. The manifest editor opens.
2. Add a dependency to the com.ibm.example package. Click the **Dependencies** tab. In the **Imported Packages** section, click **Add**. The package selection dialog opens. In the **Exported Packages** field, enter `com.ibm.example`. Select com.ibm.example from the package list and click **OK**.



The dependency to com.ibm.example is added to the Imported Packages section. Note that because the bundle project is configured with Web 3.0 support, additional import entries for servlet packages such as javax.servlet have already been added.



3. Save your changes.

Create a servlet

In this section you will create an initial servlet with the servlet wizard.

1. Open the servlet wizard. Right-click the Web project and select **New** > **Servlet**. The servlet wizard opens.
2. Create the servlet. Enter or select the following values in the wizard:

   - Project

   - Web

   - Java package

   - com.ibm.example.servlets

   - Class name

   - ConverterServlet

   Click **Finish**. The ConverterServlet is created in the Web project.

Develop the servlet

In this section you will develop the servlet and include code to access the EJB as an OSGi service.

1. Open the servlet in the Java™ editor. If the servlet is not already open in the editor, in the Web project, expand the Java Resources/src folder. In the com.ibm.example.servlets package, double-click the ConverterServlet.
2. Add new import statements. Add the following import statements to the import section at the top of the source file:

```
import com.ibm.example.EJBConverterLocal;
import java.io.PrintWriter;
import java.text.NumberFormat;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

3. Develop the doGet() method. The servlet contains a basic doGet() method. Replace the auto-generated method with the following:

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {

  PrintWriter w = response.getWriter();
```

```
  NumberFormat nf = NumberFormat.getInstance();
  nf.setMaximumFractionDigits(2);

  try {
     InitialContext context = new InitialContext();
     EJBConverterLocal converter = (EJBConverterLocal) context
           .lookup("osgi:service/" + EJBConverterLocal.class.getName());

     String temperature = request.getParameter("temperature");
     if (temperature == null) {
        w.println("<p>A temperature value was not specified.</p>");
     } else if (!temperature.matches("[-+]?[0-9]*\\.?[0-9]+")) {
        w.println("Invalid temperature specified.");
     } else {
        double degrees = Double.parseDouble(temperature);

        double celsius = converter.convertToCelsius(degrees);
        w.println("<p>" + temperature + " degrees fahrenheit is "
              + nf.format(celsius) + " degrees celsius</p>");

        double fahrenheit = converter.convertToFahrenheit(degrees);
        w.println("<p>" + temperature + " degrees celsius is "
              + nf.format(fahrenheit) + " degrees fahrenheit</p>");
     }

     w.println("<a href='index.html'>Back</a>");
  } catch (NamingException e) {
     w.println(e.getMessage());
  } catch (NumberFormatException e) {
     w.println("An incorrect temperature was specified");
  }
}
```

Note: Items to note about this code:The servlet creates an EJBConverterLocal object called converter. Recall that EJBConverterLocal was the interface you created that contains two temperature conversion methods convertToCelsius() and convertToFahrenheit(). The object creation is handled in the following line that accesses the EJB as an OSGi service:`EJBConverterLocal converter = (EJBConverterLocal) context.lookup("osgi:service/" + EJBConverterLocal.class.getName());`

The temperature value to convert is passed to the servlet by the *temperature* parameter. This value is converted to a Double called *degrees* than can processed by the convertToCelsius() and convertToFahrenheit() methods on the converter object.

Note: If you want to use remote interfaces, you will need to adjust your JNDI lookup code. For example, the context.lookup section above could be written in this format:`EJBConverterRemote converter = (EJBConverterRemote) context.lookup("osgi:service/" + EJBConverterRemote.class.getName() + "/(service.exported.interfaces=*)");`

4. Develop the doPost() method. Replace the auto-generated doPost() method with the code below. This ensures that the servlet can be called by both post and get requests from a browser. Calls to doPost() will automatically execute the

doGet() method.
```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
 doGet(request, response);
}
```

5. Check your work. The final code for the servlet should look like the following:

```
package com.ibm.example.servlets;

import java.io.IOException;
import java.io.PrintWriter;
import java.text.NumberFormat;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.example.EJBConverterLocal;

@WebServlet("/ConverterServlet")
public class ConverterServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  public ConverterServlet() {
     super();
  }

  protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {

     PrintWriter w = response.getWriter();

     NumberFormat nf = NumberFormat.getInstance();
     nf.setMaximumFractionDigits(2);

     try {
        InitialContext context = new InitialContext();
        EJBConverterLocal converter = (EJBConverterLocal) context
              .lookup("osgi:service/" + EJBConverterLocal.class.getName());

        String temperature = request.getParameter("temperature");
        if (temperature == null) {
           w.println("<p>A temperature value was not specified.</p>");
        } else if (!temperature.matches("[-+]?[0-9]*\\.?[0-9]+")) {
           w.println("Invalid temperature specified.");
        } else {
           double degrees = Double.parseDouble(temperature);
```

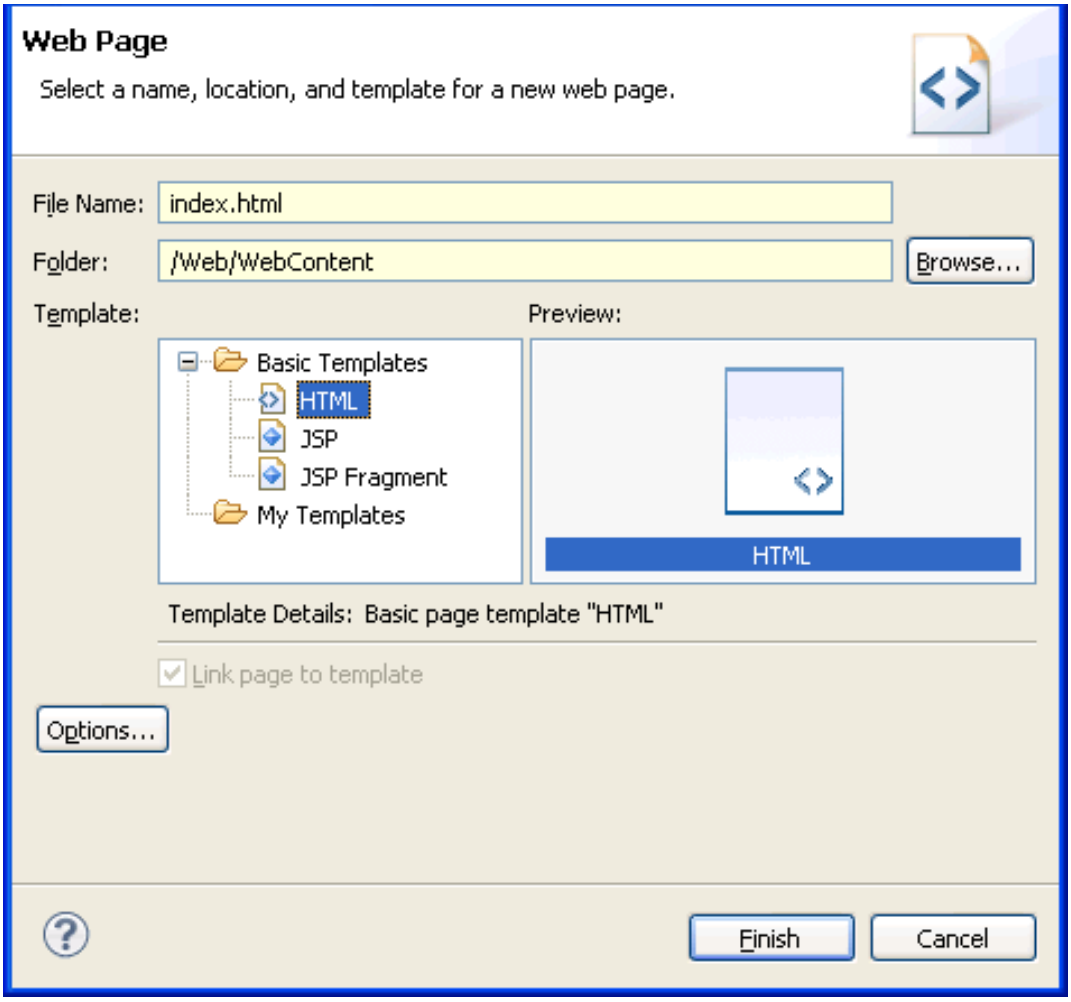
```
            double celsius = converter.convertToCelsius(degrees);
            w.println("<p>" + temperature + " degrees fahrenheit is "
                  + nf.format(celsius) + " degrees celsius</p>");

            double fahrenheit = converter.convertToFahrenheit(degrees);
            w.println("<p>" + temperature + " degrees celsius is "
                  + nf.format(fahrenheit) + " degrees fahrenheit</p>");
         }

         w.println("<a href='index.html'>Back</a>");
      } catch (NamingException e) {
         w.println(e.getMessage());
      } catch (NumberFormatException e) {
         w.println("An incorrect temperature was specified");
      }
   }

   protected void doPost(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException {
   }

}
```

6. Save your changes.

Create an HTML file to access the application

In this section you will create an HTML file index.html that will contain a form for entering values to submit to the servlet for conversion.

1. Open the new Web page wizard. Right-click the Web project and select **New** > **Web Page**. The Web page wizard opens.
2. Create the Web page. In the wizard, in the **File Name** field, enter `index.html`. In the **Template** section, select **HTML**. Click **Finish**. The page is created and opens in the editor.



3. Select the **Source** tab of the Web page editor.
4. Update the source. Replace the default source with the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>OSGi EJB Temperature Converter</title>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<script type="text/javascript">

function validate(form) {
 var temperature = new Number(form.temperature.value);
 if (isNaN(temperature)) {
  alert("Please enter a valid number.");
  return false;
 }
 return true;
}

</script>
</head>
<body>
<form action="ConverterServlet" method="post" onsubmit="return validate(this);">
 Enter a temperature value:
```

```
 <br/>
 <input type="text" id="temperature" name="temperature"/>
 <br/>
 <br/>
 <input type="submit" value="Submit"/>
</form>
</body>
</html>
```

This HTML contains a form that submits a *temperature* parameter value to the ConverterServlet. Before it is sent to the servlet, the value submitted by the user is validated by the validate() function to ensure that a numerical value was entered.

< Previous | Next >

< Previous

Lesson 4: Test the application

In this lesson you will test the application that you developed in the proceeding lessons. You will ensure that you have an application server instance set up to run the application, run the application on the server, and submit test values for temperature conversion.

Check servers view for the application server

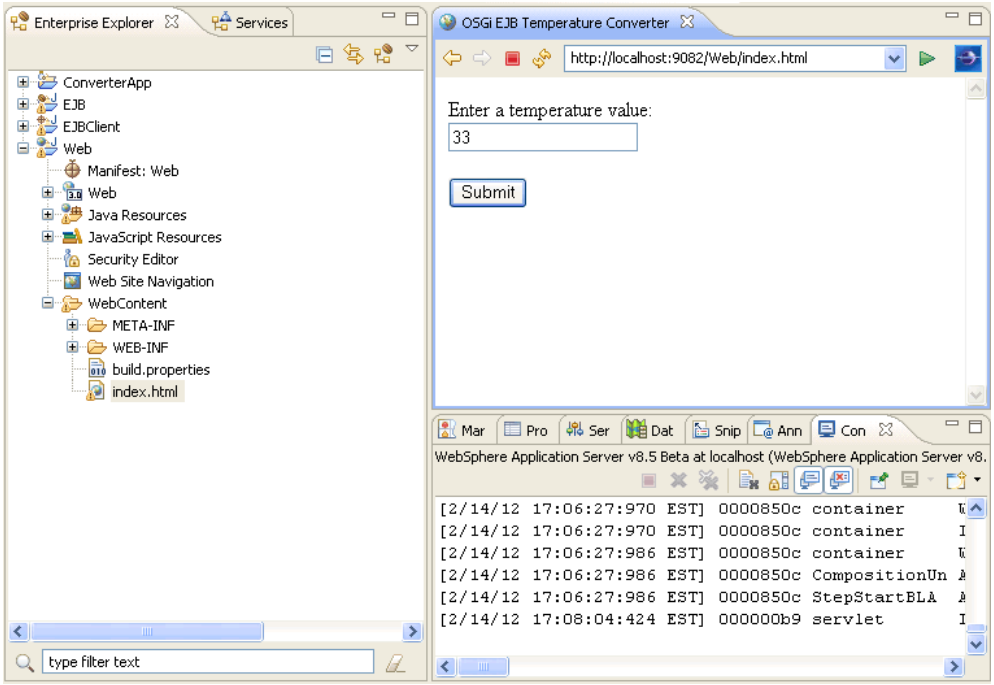
In this section you will ensure that you have a WebSphere® Application Server v8.5 instance set up to run the OSGi application.

1. Click the **Servers** tab to access the servers view.
2. Check that you have a WebSphere Application Server v8.5 server. If you do not have a server, right-click in the **Servers** view and select **New** > **Server**. Choose **WebSphere Application Server v8.5** from the list and follow the rest of the steps of the wizard.
3. Check that the server is running. If the server is running, the status information next to the server should say **Started**. If it is not running, to start it, right-click the server and select **Start**.

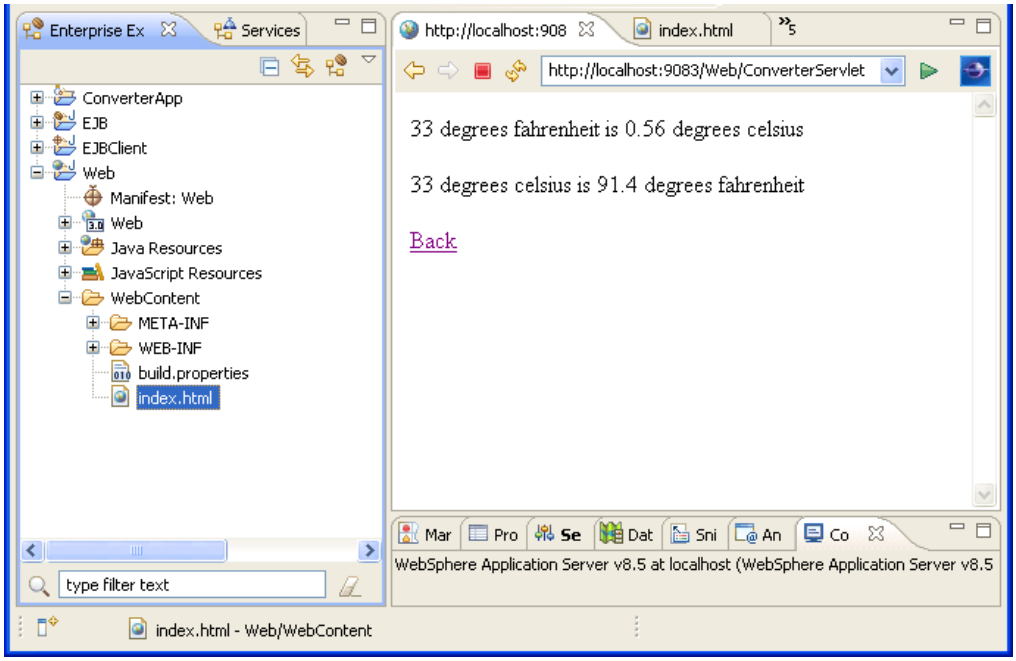
Run the application on the server

In this section you will run the application and submit test values.

1. Launch the application via the index.html file. In the Web project, right-click the index.html file that you created and select **Run As** > **Run on Server**. Choose the WebSphere Application Server v8.5 server from the server list and click **Finish**. The application is deployed to the server and the deployed index.html file opens in the embedded browser.
2. In the field on the page, enter a temperature value that you want to convert.



3. Click **Submit**. The value is converted. The processing is handled by the EJBConverter EJB that is available as an OSGi service.


Enterprise Ex
Services
ConverterApp
EJB
EJBClient
Web
Manifest: Web
Web
Java Resources
JavaScript Resources
WebContent
META-INF
WEB-INF
build.properties
index.html
type filter text
index.html - Web/WebContent
http://localhost:908
index.html
http://localhost:9083/Web/ConverterServlet
33 degrees fahrenheit is 0.56 degrees celsius
33 degrees celsius is 91.4 degrees fahrenheit
Back
Mar
Pro
Se
Dat
Sni
An
Co
WebSphere Application Server v8.5 at localhost (WebSphere Application Server v8.5

< Previous

Next >

Tutorial: Create a loan payment calculator with Dojo

Learn how to create a Dojo-based loan payment calculator.

Learning objectives

The calculator accepts three pieces of input: loan amount, interest rate, and term. It outputs the monthly payment, a pie chart displaying the percentage of loan costs going towards principal and interest, and an amortization table. There is no submit button because the output fields are updated in real time as users enter or change input. In this tutorial you will learn how to:

- Create a Dojo-enabled web application to contain all of the resources required by your application.
- Create a user interface with these Dojo widgets:
  - A Dojo layout widget to define the placement of objects on the web page
  - A Dojo custom widget to calculate and display monthly payments
  - A pie chart to display a breakdown of loan costs
  - A Dojo data grid to display the loan amortization data
- Use content assist, templates, and wizards to write Dojo code and HTML
- Deploy your project to a server
- Create a custom Dojo build to optimize your application
- Debug your web application using Firebug and the Dojo Firebug extension

Time required

This tutorial should take approximately 30 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete. You can also import the completed solution for the tutorial found at the end of lesson 7 .

Next >

Introduction: Create a loan payment calculator with Dojo

Learn how to create a Dojo-based loan payment calculator.

Learning objectives

In this tutorial you will:

- Create a Dojo-enabled web application to contain all of the resources required by your application.
- Create a user interface with these Dojo widgets:
  - A Dojo layout widget to define the placement of objects on the web page
  - A Dojo custom widget to calculate and display monthly payments
  - A pie chart to display a breakdown of loan costs
  - A Dojo data grid to display the loan amortization data
- Use content assist, templates, and wizards to write Dojo code and HTML
- Deploy your project to a server
- Create a custom Dojo build to optimize your application
- Debug your web application using Firebug and the Dojo Firebug extension

Time required

This tutorial should take approximately 30 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete. You can also import the completed solution for the tutorial found at the end of lesson 7 .

Prerequisites

1. Install the Ajax, Dojo toolkit, and HTML development tools.
2. Install Mozilla Firefox for lessons 4, 5, and 7.

Lessons in this tutorial

- **Lesson 1: Create a Dojo-enabled web project**
  In this lesson, you learn how to create a web project that is configured for Dojo application development by enabling specific project features.
- **Lesson 2: Create a custom Dojo widget**
  The Dojo toolkit includes dozens of standard widgets, including input fields, combo boxes and radio buttons. You can also create custom widgets to encapsulate reusable UI elements or a specific piece of functionality. In this lesson you will create a new custom Dojo widget.
- **Lesson 3: Add the custom Dojo widget to a web page**
  In this lesson you will insert your custom Dojo widget into a web page that will be laid out using a Dojo layout widget. You will use Dojo APIs to connect your widget to the output field and display the results.
- **Lesson 4: Add a pie chart using the Dojo charting API (optional)**
  In this optional lesson, you use content assist and the Dojo charting API to add a pie chart to your results page. The pie chart displays the percentage of total loan costs going towards interest and principal.

- **Lesson 5: Add a data grid (optional)**
  In this optional lesson you learn how to add an Enhanced Dojo Grid to your web page. The DataGrid widget creates a table that looks like a spreadsheet.
- **Lesson 6: Create a Dojo custom build (optional)**
  This optional lesson highlights the steps required to create a Dojo custom build. The purpose of a custom Dojo build is to create an efficient version of Dojo, and of your code, that is suitable for deployment.
- **Lesson 7: Debugging your Dojo application with Firebug (optional)**
  This optional lesson highlights how to use Firebug to debug your web application. This product includes a Dojo Firebug extension that includes additional tools for Dojo debugging.

< Previous | Next >

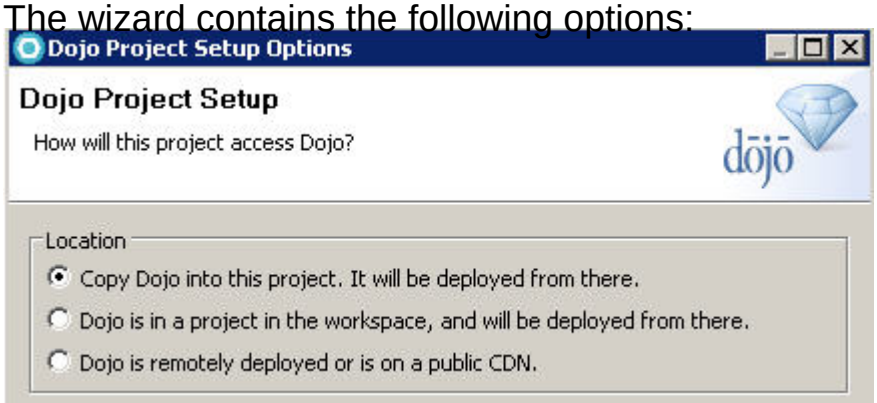
Lesson 1: Create a Dojo-enabled web project

In this lesson, you learn how to create a web project that is configured for Dojo application development by enabling specific project features.

This product provides a project setup wizard where you can customize how your application accesses Dojo during development and at runtime. The simplest option (and the one used in this tutorial) is to have a copy of Dojo in your project and deploy it along with the rest of your project resources. Another option is to use a public Content Delivery Network (CDN), a remote server with a copy of Dojo. The use of CDNs are not covered by this tutorial, but they are simple to use. For more information about CDNs, see www.dojotoolkit.org/download/.

To create a web project that is configured for Dojo application development:

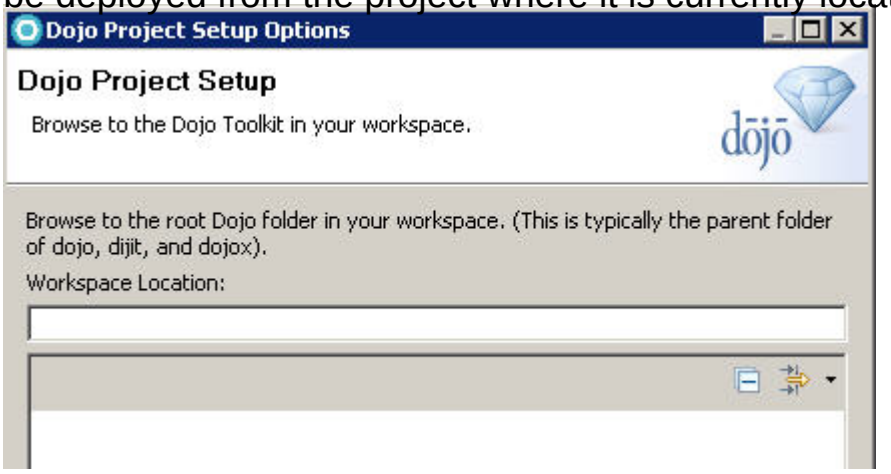
1. In the main menu, click **File** > **New** > **Web Project**.
2. In the **Name** field, type `LoanPaymentCalculator`.
3. From the list of project templates, click **Dojo Toolkit** to use the Dojo Toolkit project template to create your web project.
4. In the Programming Model section, click **Client-side only (HTML, JavaScript,...)** to use the Client-side only programming model when creating your project.
5. Click **Next** to configure your new web project.
6. From the list of available configuration options, click **Dojo Toolkit** to open the Dojo Toolkit page. The Dojo Toolkit included in this product includes additional IBM® extensions to the base Dojo Toolkit, including libraries for ATOM (ATOM Syndication Format) data access, analog and bar gauges, and simplified access for SOAP Web services. By default, the latest Dojo Toolkit supported by IBM is copied into your web project.
7. To determine how you want to use the Dojo Toolkit in your web project, click **Change these setup options** to open the Dojo Project Setup Options wizard. The wizard contains the following options:



- Copy Dojo into this project. It will be deployed from there.

- The Dojo toolkit is included inside your web project. You can specify the name of the Dojo folder and whether to use the default Dojo distribution included in the runtime environment or to use a Dojo distribution from disk.

- **Dojo is in a project in the workspace and will be deployed from there.**
  - On this page you can browse to the root Dojo folder in another project in your workspace. The copy of Dojo will not be copied into your project. It will be deployed from the project where it is currently located.



- **Dojo is remotely deployed or is on a public CDN.**
  - Use this option if your application uses a remotely hosted public CDN (content delivery network) or an existing copy of Dojo already deployed on your network.
    - **Public CDN**: You can enter the URL of a publicly available content delivery network. Content delivery networks provide geographically distributed hosting for open source JavaScript libraries. When a browser resolves the URL in your web application, the browser downloads the file from the closest available server.
    - **Remote URI**: You can enter the URI of the remote location to the root Dojo folder.

    In the Corresponding Dojo section, you can choose a Dojo source distribution that is the closest match to your remote Dojo Toolkit. If Dojo is not contained in your project the tools must reference a corresponding copy of Dojo in order to provide content assist and validation. You can choose the default Dojo Toolkit provided with this product, or browse to a Dojo folder in your workspace or file system. This option does not copy Dojo into your project or workspace.

8. Click **Copy Dojo into this project. It will be deployed from there.** to include the Dojo Toolkit inside your web project for this tutorial.
9. Click **Finish** to close the wizard.
10. Click **Finish** to create your web project.

Your project is now created and appears in the Enterprise Explorer view. Expand the WebContent folder to show the dojo folder that contains all of the Dojo resources. If asked to switch to the Web Perspective, click **Yes**.

Lesson checkpoint

You created the Dojo-enabled web project.
You learned :
- How to create a Dojo-enabled web project.
- How to change the setup options for a Dojo-enabled web project.

< Previous | Next >

Lesson 2: Create a custom Dojo widget

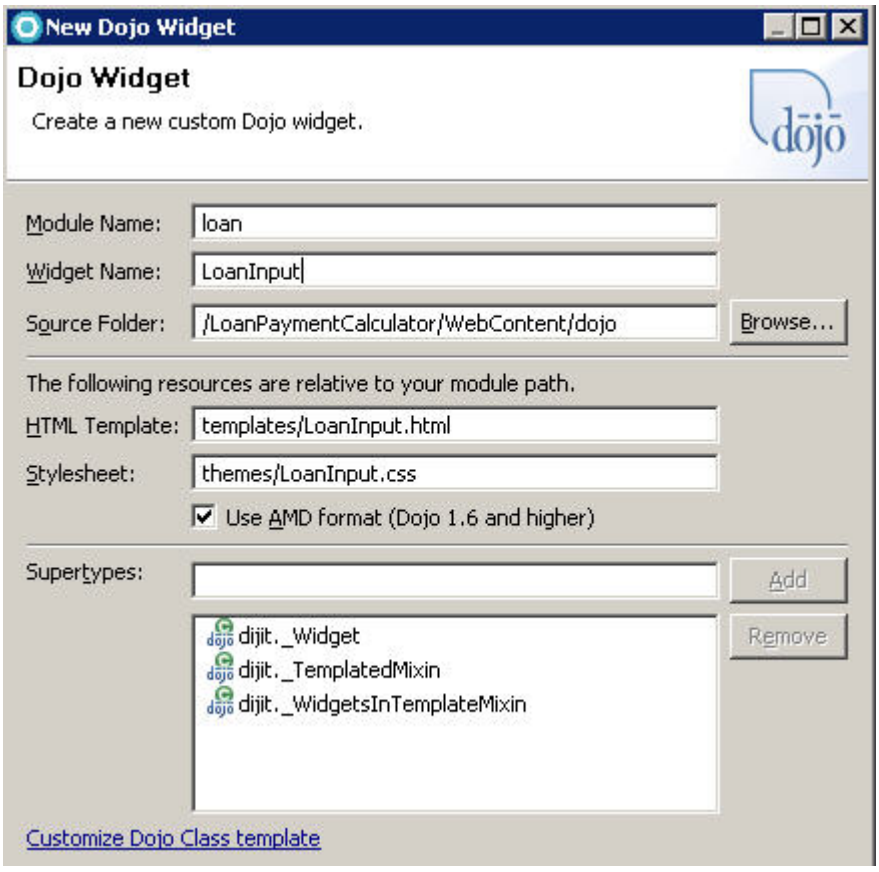
The Dojo toolkit includes dozens of standard widgets, including input fields, combo boxes and radio buttons. You can also create custom widgets to encapsulate reusable UI elements or a specific piece of functionality. In this lesson you will create a new custom Dojo widget.

Dojo widgets are composed of three files that the New Dojo Widget wizard creates for you:

- An HTML file
- A JavaScript file
- A CSS file

You then edit the HTML template and the JavaScript file.

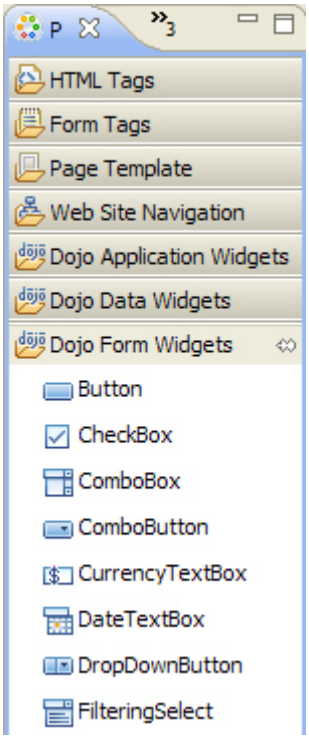
1. In the Enterprise Explorer view, right-click the WebContent/dojo folder and select **New** > **Dojo Widget**. The New Dojo Widget wizard appears.
2. In the **Module Name** field, enter `loan`.
3. In the **Widget Name** field, enter `LoanInput`. The HTML template and style sheet relative for the widget paths are populated automatically.



4. Click **Finish**. Three files are created under a folder named dojo/loan:
   - **`templates/LoanInput.html`**
     - The UI template for the widget.
   - **`themes/LoanInput.css`**
     - Provides the styling for the widget.
   - **`LoanInput.js`**

- Provides the JavaScript backend and business logic portion of the widget.

5. The LoanInput JavaScript source file for the widget opens in the editor.
6. Directly below the `templateString` field, add three additional fields that will be used to hold the results of our calculation – `principal`, `interestPaid`, and `monthlyPayment` - they should all have default values of `0`. Ensure that you add a comma after each field.
```
templateString : dojo.cache("loan", "templates/LoanInput.html"),

principal: 0,

interestPaid: 0,

monthlyPayment: 0,
```
7. Directly below the `postCreate` function, add a new function named `calculate`. Ensure that you add a comma after the `postCreate` function. Leave the new `calculate` function empty for now.
```
postCreate : function() {

},

// this function will perform the calculations for our loan payment

calculate: function() {

}
```
8. In the Enterprise Explorer view, double-click **templates/LoanInput.html** to open the HTML template for the widget.
9. At the bottom of the editor click the Source tab to display the page in the source pane.
10. Within the existing div create three additional child div tags. You can use content assist by typing `<d` and then pressing Ctrl + Space. In the pop-up window, select **<> div** to insert the tag.
11. Within each new div tag add a text label – `Loan Amount:`, `Interest Rate:`, and `Term (years):`. When complete your code should look like this:
```
<div class="LoanInput">

<div>Loan Amount: </div>

<div>Interest Rate: </div>

<div>Term (years): </div>

</div>
```
12. Now add Dojo widgets for each of the input fields:
    A. In the right-hand side of the workspace, surface the palette by clicking the appropriate tab. You should see several drawers containing Dojo widgets.
    B. Expand the **Dojo Form Widgets** drawer by clicking on it.



C. Select the **CurrencyTextBox** and drop it next to the Loan Amount label inside your div tag.
D. Within the newly created input element, type `data-` and use content assist (Ctrl + Space) to show a list of attributes. Click **data-dojo-props** to insert it. To follow HTML5 standards, beginning in Dojo Toolkit SDK 1.7 attributes are set by default through data-dojo-props.
E. Inside the data-dojo-props attribute, type `cu` and use content assist (Ctrl + Space) to show a list of attributes. Click **currency** to insert it.
F. Inside the currency attribute, type `USD`. Your code should look like this:

```
<div>Loan Amount: <input type="text"
    data-dojo-type="dijit.form.CurrencyTextBox"
    data-dojo-props="currency: 'USD'">
</div>
```

G. Next, insert the Dojo widget markup for the Interest Rate field. Place your cursor inside the second div tag after the label, type `<input d>` and with your cursor directly to the right of the d, use content assist (Ctrl + Space) to show a list of attributes. Click **data-dojo-type** to insert it.
H. Inside the data-dojo-type attribute, invoke content assist again to show a list of available dojo widgets. Begin typing `dijit.form.N` until you see `NumberSpinner`. Click **NumberSpinner** to insert it into your page.
I. Add the data-dojo-props attribute. Inside the attribute, set the following properties, separated by commas:
   1. value : 5
   2. smallDelta : .05
   3. intermediateChanges : true
   4. constraints : {min: 0}

   You can use content assist to insert these properties the same way you added the currency attribute above. When you are finished setting the properties for the attribute, your code should look like this:

```
<div>Interest Rate: <input data-dojo-type="dijit.form.NumberSpinner"
```

```
data-dojo-props="value: 5, smallDelta:= 0.05, intermediateChanges: true, constraints: {min: 0}"> </input></div>
```

J. From the palette drop a **ComboBox** widget into the Term (years) div.
K. Additional configuration for some widgets is available when you drop them from the palette, such as the ComboBox. In the Insert Combo Box dialog box you can add values for your ComboBox. Add values for 1, 2, 3, 4, 5, 10, 15, 30.
L. Set 15 as the default value by setting it to **True**.

13. Next, add the `data-dojo-attach-point` and `data-dojo-attach-event` attributes to each of your input widgets. The value specified for the `data-dojo-attach-point` attribute is the name that the widget instance can be referenced by from the `LoanInput.js` file. The `data-dojo-attach-event` attribute adds event handling to the widgets.
    A. Use content assist to add a `data-dojo-attach-point` attribute to each widget. Name them `amount`, `rate`, and `term` respectively.
    B. Add `data-dojo-attach-event="onChange: calculate"` for each widget. Each time that an onChange event takes place on this widget it calls the calculate function that you added to the LoanInput.js file. The final result of your LoanInput.html file should look like this:

```
<div class="LoanInput">
<div>Loan Amount: <input type="text"
data-dojo-type="dijit.form.CurrencyTextBox"
data-dojo-props="currency: 'USD'"
data-dojo-attach-point="amount"
data-dojo-attach-event="onChange: calculate"></div>
<div>Interest Rate: <input data-dojo-type="dijit.form.NumberSpinner"
data-dojo-props="value:5,
smallDelta:0.05,
intermediateChanges:true,
constraints:{min: 0}"
data-dojo-attach-point="rate"
data-dojo-attach-event="onChange: calculate">
</div>
<div>Term (years): <select name="select"
data-dojo-type="dijit.form.ComboBox"
data-dojo-props="autocomplete:false"
data-dojo-attach-point ="term"
data-dojo-attach-event="onChange: calculate">
<option>1</option>
<option>2</option>
<option>3</option>
<option>4</option>
<option>5</option>
<option>10</option>
<option selected="selected">15</option>
<option>30</option>
</select>
</div>
</div>
```

14. Save and close **LoanInput.html** and reopen **LoanInput.js**.
15. Add Dojo module dependencies for the three widgets used in the html file. These dependencies will load the necessary resources to create the widgets when the page is run. The second argument of the `define` function is an array of dependencies. Directly after the `"dijit/_WidgetsInTemplateMixin",` dependency, add the following module paths separated by commas:
    A. "dijit/form/CurrencyTextBox"
    B. "dijit/form/NumberSpinner"
    C. "dijit/form/ComboBox"
    Your code should look like this:
```
define("loan/LoanInput",
[ "dojo", "dijit", "dijit/_Widget",
"dijit/_TemplatedMixin",
"dijit/_WidgetsInTemplateMixin",
"dijit/form/CurrencyTextBox",
"dijit/form/NumberSpinner",
"dijit/form/ComboBox",
"dojo/text!loan/templates/LoanInput.html"
],
```

16. Now add the following code for the calculate function that you created earlier in Step 7. You can experiment with content assist if you want. Note that standard JavaScript objects such as the Math object are available in content assist. The variables we defined earlier, `principal`, `interestPaid`, and `monthlyPayment` are all available as well.
```
// this function will perform the calculations for our loan repayment
 calculate: function() {
  this.principal = this.amount.attr('value');
  if(this.principal == NaN) {
   this.monthlyPayment = 0;
   this.principal = 0;
   this.interestPaid = 0;
  } else {
   var interestRate = this.rate.attr('value') / 1200;
   var termInMonths = this.term.attr('value') * 12;

   this.monthlyPayment = Math.pow(1 + interestRate, termInMonths) - 1;
   this.monthlyPayment = interestRate + (interestRate / this.monthlyPayment);
   this.monthlyPayment = this.monthlyPayment * this.principal;

   this.interestPaid = (this.monthlyPayment * termInMonths) - this.principal;
  }
}
```
    The calculate function stores the principal of the loan, computes the monthly payment, and the amount of interest paid.
17. Save and close all of the files that make up the custom widget.

LoanInput.js should now look like this:
```
define("loan/LoanInput", [ "dojo", "dijit", "dijit/_Widget",
  "dijit/_TemplatedMixin", "dijit/_WidgetsInTemplateMixin", "dijit/form/CurrencyTextBox", "dijit/form/NumberSpinner",
"dijit/form/ComboBox",
  "dojo/text!loan/templates/LoanInput.html" ], function(dojo, dijit,
```

```
 _Widget, _TemplatedMixin, _WidgetsInTemplateMixin, CurrencyTextBox, NumberSpinner, ComboBox) {
dojo.declare("loan.LoanInput", [ dijit._Widget, dijit._TemplatedMixin,
  dijit._WidgetsInTemplateMixin ], {
 // Path to the template
 templateString : dojo.cache("loan", "templates/LoanInput.html"),
 principal: 0,
 interestPaid: 0,
 monthlyPayment: 0,

 // Override this method to perform custom behavior during dijit construction.
 // Common operations for constructor:
 // 1) Initialize non-primitive types (i.e. objects and arrays)
 // 2) Add additional properties needed by succeeding lifecycle methods
 constructor : function() {

 },

 // When this method is called, all variables inherited from superclasses are 'mixed in'.
 // Common operations for postMixInProperties
 // 1) Modify or assign values for widget property variables defined in the template HTML file
 postMixInProperties : function() {
 },

 // postCreate() is called after buildRendering().  This is useful to override when
 // you need to access and/or manipulate DOM nodes included with your widget.
 // DOM nodes and widgets with the dojoAttachPoint attribute specified can now be directly
 // accessed as fields on "this".
 // Common operations for postCreate
 // 1) Access and manipulate DOM nodes created in buildRendering()
 // 2) Add new DOM nodes or widgets
 postCreate : function() {
 },
 //this function will perform the calculations for our loan payment
 calculate: function() {
  this.principal = this.amount.attr('value');
  if(this.principal == NaN) {
   this.monthlyPayment = 0;
   this.principal = 0;
   this.interestPaid = 0;
  } else {
   var interestRate = this.rate.attr('value') / 1200;
   var termInMonths = this.term.attr('value') * 12;

   this.monthlyPayment = Math.pow(1 + interestRate, termInMonths) - 1;
   this.monthlyPayment = interestRate + (interestRate / this.monthlyPayment);
   this.monthlyPayment = this.monthlyPayment * this.principal;

   this.interestPaid = (this.monthlyPayment * termInMonths) - this.principal;
```

```
   }
}
 });
});
```

Lesson checkpoint

You created a custom Dojo widget.
You learned:
- How to use content assist and templates to rapidly write Dojo code
- How to modify the HTML template for a custom Dojo widget
- How to modify the JavaScript file for a custom Dojo widget

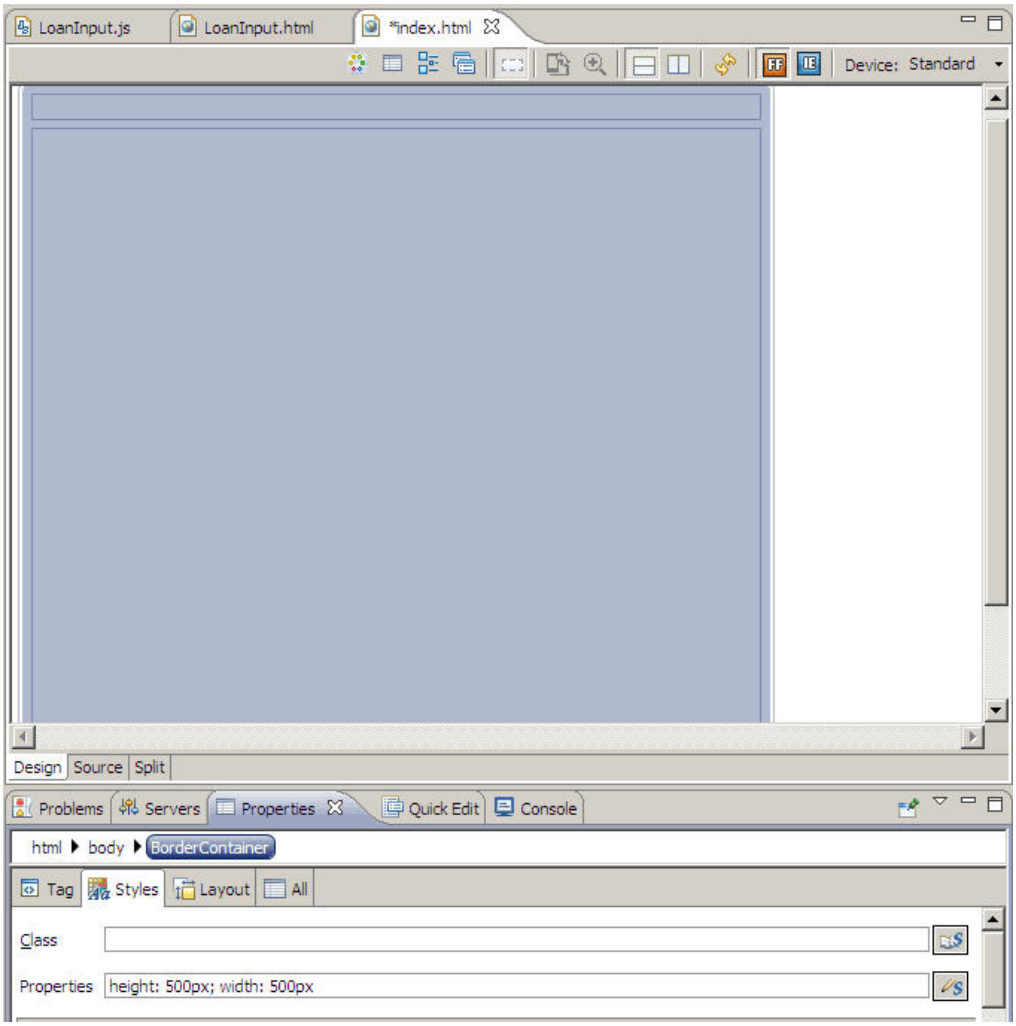
< Previous | Next >

Lesson 3: Add the custom Dojo widget to a web page

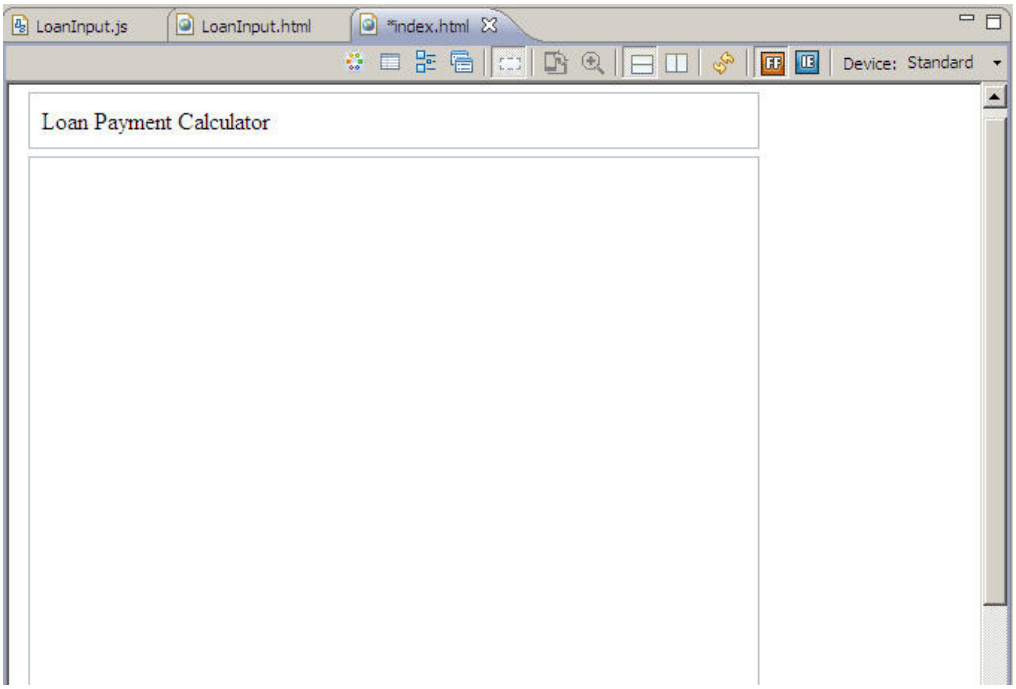
In this lesson you will insert your custom Dojo widget into a web page that will be laid out using a Dojo layout widget. You will use Dojo APIs to connect your widget to the output field and display the results.

After you create a custom Dojo widget, the widget is added to the Other Dojo Widgets drawer of the Palette, making it easy to add the widget to the web page.

1. Right-click the `WebContent` folder and select **New** > **Web Page**.
2. Name the page `index.html` and click **Finish**.
3. At the bottom of the editor click the **Design** tab to display the page in the Design view.
4. In the palette, open the **Dojo Layout Widgets** drawer and drop a **BorderContainer** onto the page. The Insert Border Container dialog opens to allow you to customize the `BorderContainer` widget.
   A. Select the **Top** and **Center** check boxes.
   B. Click **OK**.
5. A visual view of the `BorderContainer` widget is now displayed in the Design view. Click the `BorderContainer` visualization and then click the **Properties** tab to open the Properties view.



6. Click the **Styles** tab and in the **Properties** field, update the following values for the border container:
   - Change the height from 500px to 700px.
   - Change the width to 600px.
7. Double-click the top region to open a text box. Type `Loan Payment Calculator` and then click outside the text box to apply your change.



8. In the palette, expand the **Other Dojo Widgets** drawer.
9. Drop the **LoanInput** Dojo widget into the center region of the border container.
10. Click the **Source** tab to switch to the Source view.
11. In the `LoanInput` widget `div` tag, set the id attribute to `"LoanInput"`:
```
<div id="LoanInput" data-dojo-type="loan.LoanInput"></div>
```
12. Add a new `div` tag below the `LoanInput` widget to show the results. You can copy the text from below.
```
<div>Monthly Payment: <span id="monthlyPayment"></span></div>
```
13. Add a new module require for `dijit/registry` immediately after the existing dojo require.
```
[ "dojo", "dijit/registry", "dojo/parser", "dijit/layout/BorderContainer",
"dijit/layout/ContentPane", "loan/LoanInput" ],
```
14. Add a registry parameter to the require callback function.
```
function(dojo, registry) {
 dojo.ready(function() {
 });
});
```
15. Inside the `dojo.ready` function, perform the following steps:
    A. Type `var loanWidget = registry.b` and invoke content assist. Select the **byId(id)** suggestion so that it inserts into your page.
    B. Type `"LoanInput"` as the parameter for the `byId` function.
    C. Type a semicolon after the closing parenthesis for the `LoanInput` parameter.
    D. On the line directly below your call to `registry.byId`, type `dojo.c` and invoke content assist. Here you can select another default template for `dojo.connect`. There are two versions of this template. Choose the version that uses `dijit.registry.byId` and insert it into your page.

E. Set the first parameter as `LoanInput` and the second as `calculate`.
F. Inside the function parameter of the connect function add the following code:

```
var payment = loanWidget.monthlyPayment;
if (payment == NaN) {
 payment = 0;
}

// update the result field
var formattedValue = dojo.currency.format(payment, {currency: "USD"});
dojo.attr("monthlyPayment", "innerHTML", formattedValue);
```

G. Required modules are added to the `require` function as an array of strings, where each module is a slash separated string of module segments. Inside the `require` function, add `"dojo/currency"` to the require dependencies array.
H. Your final code for the `dojo.ready` function should look like this:

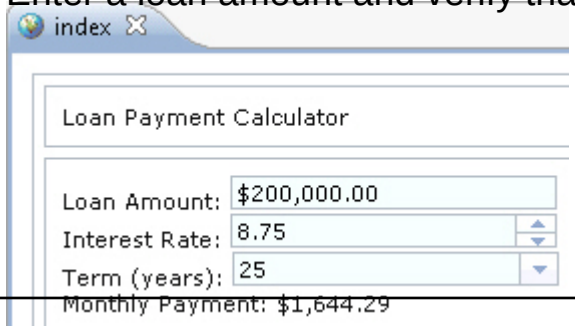
```
dojo.ready(function() {
  // get the LoanInput widget
  loanWidget = registry.byId('LoanInput');

  // handle "calculate" from widget "LoanInput"
  dojo.connect(dijit.registry.byId("LoanInput"), "calculate", function(event) {
   var payment = loanWidget.monthlyPayment;
   if (payment == NaN) {
    payment = 0;
   }

   // update the result field
   var formattedValue = dojo.currency.format(payment, {currency: "USD"});
   dojo.attr("monthlyPayment", "innerHTML", formattedValue);
  });
 });
```

16. Save the page. The source for the `index.html` file looks like the following:

```
<!DOCTYPE HTML>
<html>
<head>
<link rel="stylesheet" type="text/css"
 href="dojo/dijit/themes/dijit.css">
<link rel="stylesheet" type="text/css"
 href="dojo/dijit/themes/claro/claro.css">
<title>t1</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript"
 data-dojo-config="isDebug: false, async: true, parseOnLoad: true"
 src="dojo/dojo/dojo.js"></script>
<script type="text/javascript">
require(
// Set of module identifiers
[ "dojo", "dijit/registry", "dojo/parser", "dijit/layout/BorderContainer", "dijit/layout/ContentPane",
```

```
"loan/LoanInput", "dojo/currency" ],
// Callback function, invoked on dependencies evaluation results
function(dojo, registry) {
 dojo.ready(function() {
  // get the LoanInput widget
  loanWidget = registry.byId('LoanInput');

  // handle "calculate" from widget "LoanInput"
  dojo.connect(dijit.registry.byId("LoanInput"), "calculate", function(event) {
   var payment = loanWidget.monthlyPayment;
   if (payment == NaN) {
    payment = 0;
   }

   // update the result field
   var formattedValue = dojo.currency.format(payment, {currency: "USD"});
   dojo.attr("monthlyPayment", "innerHTML", formattedValue);
  });
 });
});
</script>
</head>
<body class="claro">
 <div id="BorderContainer" style="height: 500px; width: 500px"
  data-dojo-type="dijit.layout.BorderContainer"
  data-dojo-props="design:'headline'">
  <div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-props="region:'top'">Loan Payment Calculator</div>
  <div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-props="region:'center'">
   <div id="LoanInput" data-dojo-type="loan.LoanInput"></div>
   <div>Monthly Payment: <span id="monthlyPayment"></span></div>
  </div>
 </div>

 </body>
</html>
```

17. Now it is time to test your application on the server. In the Enterprise Explore view, right-click **index.html** and select **Run As** > **Run on Server**.
18. Select the **Web Preview Server** and click **Finish**. Your page opens in a web browser.
19. Enter a loan amount and verify that the results field updates.



Lesson checkpoint

You added your custom Dojo widget to the page and tested it on a server.
You learned:
- How to add a custom widget to a web page from the palette
- How to run a Dojo application on a server
- How to test the Dojo widgets that you created

< Previous | Next >

Lesson 4: Add a pie chart using the Dojo charting API (optional)

In this optional lesson, you use content assist and the Dojo charting API to add a pie chart to your results page. The pie chart displays the percentage of total loan costs going towards interest and principal.

Before you begin this lesson, install Mozilla Firefox.

1. Configure the web application to run on Firefox by clicking **Window** > **Web Browser** > **Firefox**.
2. In the `index.html` file add the following `require` statements to the existing script tag:
```
"dojox/charting/Chart",
"dojox/charting/plot2d/Pie",
"dojox/charting/action2d/Highlight",
"dojox/charting/action2d/MoveSlice",
"dojox/charting/action2d/Tooltip",
"dojox/charting/themes/Dollar",
```
3. Add the following div directly below the `MonthlyPayment` div you created to display the result:
```
<div id="chart" style="width: 350px; height: 350px"></div>
```
4. Inside the existing `dojo.ready` function, type `var chart = new dojox` above the connect function and invoke content assist. You should see a list of all available Dojo types in the `dojox` namespace.
5. Continue typing `.charting.C` and you should see the list begin to filter down. Select **dojox.charting.Chart2D** and insert it on your page.
6. Add two parameters: `chart` and `null` , and the id for the div node where it will be located on your page:
```
var chart = new dojox.charting.Chart("chart", null);
```
7. On the next line type `chart.set` and invoke content assist. You should see the `setTheme` method. Select this method and insert it into the page.
8. Enter `dojox.charting.themes.Dollar` as the parameter:
```
chart.setTheme(dojox.charting.themes.Dollar);
```
9. Copy the following code on the next line. You can invoke content assist on the various methods and types if you want.
```
chart.addPlot("default", {
    type: "Pie",
    labelOffset: -30,
    radius: 90
});
chart.addSeries("paymentSeries", []);

new dojox.charting.action2d.MoveSlice(chart, "default");
new dojox.charting.action2d.Highlight(chart, "default");
new dojox.charting.action2d.Tooltip(chart, "default");
```
This code adds plotting information to your chart and sets up highlighting and tooltips for when users hover over the pie chart slices. For more information on Dojo charting APIs see: www.dojotoolkit.org/reference-guide/dojox/charting.html#dojox-charting.
10. Inside the dojo.connect function, below the existing code, add the following code:
```
// add the data series to the chart and render
```

```
 chart.updateSeries("paymentSeries", [
      {
          y: loanWidget.principal,
          stroke: "black",
          tooltip: "Principle"
      },
      {
          y: loanWidget.interestPaid,
          stroke: "black",
          tooltip: "Interest"
      }]);
      chart.render()
```

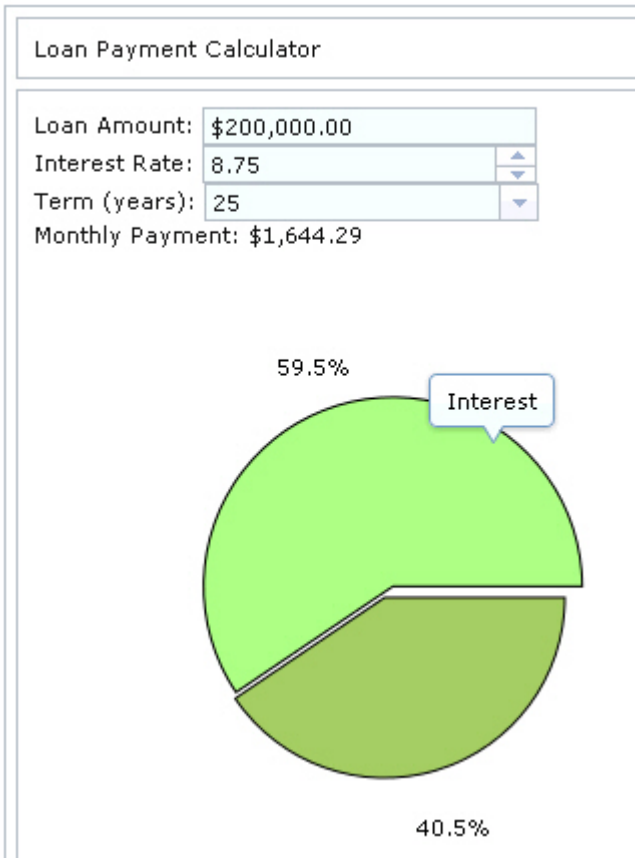
This code adds a new series of data to the chart and renders it each time that users change an input value. The resulting source is similar to the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<link rel="stylesheet" type="text/css"
 href="dojo/dijit/themes/dijit.css">
<link rel="stylesheet" type="text/css"
 href="dojo/dijit/themes/claro/claro.css">
<title>index</title>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<script type="text/javascript"
 data-dojo-config="isDebug: false, async: true, parseOnLoad: true" src="dojo/dojo/dojo.js"></script>
<script type="text/javascript">

require(
 // Set of module identifiers
 [ "dojo", "dojo/parser", "dijit/layout/BorderContainer",
   "dijit/layout/ContentPane", "loan/LoanInput", "dojo/currency",
   "dojox/charting/Chart", "dojox/charting/plot2d/Pie",
   "dojox/charting/action2d/Highlight",
   "dojox/charting/action2d/MoveSlice",
   "dojox/charting/action2d/Tooltip", "dojox/charting/themes/Dollar"],

 // Callback function, invoked on dependencies evaluation results
 function(dojo) {
  dojo.ready(function() {
   var chart = new dojox.charting.Chart("chart", null);
   chart.setTheme(dojox.charting.themes.Dollar);
   chart.addPlot("default", {
    type : "Pie",
    labelOffset : -30,
    radius : 90
   });
   chart.addSeries("paymentSeries", []);
```

```
   new dojox.charting.action2d.MoveSlice(chart, "default");
   new dojox.charting.action2d.Highlight(chart, "default");
   new dojox.charting.action2d.Tooltip(chart, "default");

   var loanWidget = dijit.byId("LoanInput");
   dojo.connect(dijit.byId("LoanInput"), "calculate", function(event) {
    // handle "calculate" from widget "LoanInput"
    var payment = loanWidget.monthlyPayment;
    if (payment == NaN) {
     payment = 0;
    }

    var formattedValue = dojo.currency.format(payment, {
     currency : 'USD'
    });
    dojo.attr("monthlyPayment", "innerHTML", formattedValue);

    // update the chart
    chart.updateSeries("paymentSeries", [ {
     y : loanWidget.principal,
     stroke : "black",
     tooltip : "Principal"
    }, {
     y : loanWidget.interestPaid,
     stroke : "black",
     tooltip : "Interest"
    } ]);
    chart.render()
    });
 });

</script>
</head>
<body class="claro">

 <div id="BorderContainer" style="height: 700px; width: 600px"
  data-dojo-type="dijit.layout.BorderContainer"
  data-dojo-props="design:'headline'">
  <div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-props="region:'top'">Loan Payment Calculator</div>
  <div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-props="region:'center'">
   <div data-dojo-type="loan.LoanInput" id="LoanInput"></div>
<div>Monthly Payment: <span id="monthlyPayment"></span></div>

<div id="chart" style="width: 350px; height: 350px"> </div>
 </div>
</div>
```

```
</body>
</html>
```

11. Save the page and run the page on server.
12. Enter a loan amount and view the pie chart.



Lesson checkpoint

You added a pie chart to your page and tested it on a server.
You learned:
- How to use content assist and the Dojo charting API to add a pie chart to a web page

< Previous | Next >

Lesson 5: Add a data grid (optional)

In this optional lesson you learn how to add an Enhanced Dojo Grid to your web page. The DataGrid widget creates a table that looks like a spreadsheet.

Before you begin this lesson, install Mozilla Firefox.

1. Configure the web application to run on Firefox by clicking **Window** > **Web Browser** > **Firefox**.
2. In the palette, click the **Dojo Data Widgets** drawer to open it.
3. Drag a **DataGrid** and drop it directly below the existing div tags created in Lesson 3 or 4 of this tutorial:

```
<body class="claro">
    <div id="BorderContainer" style="height: 700px; width: 600px"
        data-dojo-type="dijit.layout.BorderContainer"
        data-dojo-props="design:'headline'">
        <div data-dojo-type="dijit.layout.ContentPane"
            data-dojo-props="region:'top'">Loan Payment Calculator</div>
        <div data-dojo-type="dijit.layout.ContentPane"
            data-dojo-props="region:'center'">
            <div data-dojo-type="loan.LoanInput" id="LoanInput"></div>
            <div>
                Monthly Payment:<span id="monthlyPayment"></span>
            </div>
            <div id="chart" style="width: 350px; height: 350px"></div>
```

4. Clear the **Generate JavaScript to populate the grid** check box.
5. In the Columns section, specify the column heading label and JavaScript property for each column:
   A. In the **Heading Label** field, enter `Payment Number`.
   B. In the **JavaScript Property** field, enter `paymentNum`.
   C. Click **Add** to add the Heading Label - JavaScript Property pair to the table below.
   D. Repeat the previous steps to add the following pairs of heading labels and JavaScript properties:

Heading label	JavaScript property
Principal Paid	principal
Interest Paid	interest
Balance	balance

6. Click **Finish**. In the Source pane, you can see that the HTML markup for the data grid is added with the JavaScript properties populating the *field* attribute and the corresponding heading labels as column names.

```
<table id="gridId" autowidth="true"
    data-dojo-type="dojox.grid.DataGrid"
    data-dojo-props="rowSelector:'20px'">
    <thead>
        <tr>
            <th field="paymentNum">Payment Number</th>
            <th field="principal">Principal Paid</th>
            <th field="interest">Interest Paid</th>
            <th field="balance">Balance</th>
        </tr>
    </thead>
</table>
```

A set of Dojo require statements for the `DataGrid` and `ItemFileReadStore` modules are added automatically. These `require` statements load all of the Dojo packages that are required by the widgets on the web page. If needed, move them to the top of the script tag with the other `require` statements:`require(`

```
// Set of module identifiers
[ "dojo", "dojo/parser", "dijit/layout/BorderContainer",
  "dijit/layout/ContentPane", "loan/LoanInput", "dojo/currency",
  "dojox/charting/Chart", "dojox/charting/plot2d/Pie",
  "dojox/charting/action2d/Highlight",
  "dojox/charting/action2d/MoveSlice",
  "dojox/charting/action2d/Tooltip", "dojox/charting/themes/Dollar",
  "dojox/grid/DataGrid", "dojo/data/ItemFileReadStore",
  "dojo/data/ItemFileWriteStore" ],
```

Cascading style sheet (CSS) links are also added to the source code of the web page. Grid.css and claroGrid.css are stylesheets that are specific to data grid display:

```
<link rel="stylesheet" type="text/css" title="Style"
    href="dojo/dojox/grid/resources/Grid.css">
<link rel="stylesheet" type="text/css" title="Style"
    href="dojo/dojox/grid/resources/claroGrid.css">
```

7. Save the page.
8. Now you can populate data in the data grid using `ItemFileWriteStore`. Open the `LoanInput.js` file.
9. Add the following code below the `monthlyPayment` attribute.`amortizationSchedule: {},`
10. Add the following code to the `calculate()` function at the bottom of the function, inside the else statement. This code generates the amortization data.

```
//generate the Amortization Data
 this.generateAmortizationSchedule(this.principal, interestRate, termInMonths, this.monthlyPayment);
```

11. Add the following function after the `calculate()` function. Ensure that you add a comma after the calculate function before you add the `generateAmortizationSchedule` function.`generateAmortizationSchedule: function(principal,`

```
interestRate, termInMonths, monthlyPayment) {
  this.amortizationSchedule = [];
  var currentPrincipal = principal;
  for(var i = 0; i < termInMonths; i++) {
   var paymentData = {};
   paymentData.paymentNum = i + 1;
   paymentData.interest = (currentPrincipal * interestRate).toFixed(2);
   paymentData.principal = (monthlyPayment - paymentData.interest).toFixed(2);
   currentPrincipal = currentPrincipal - paymentData.principal;
   paymentData.balance = currentPrincipal.toFixed(2);
   this.amortizationSchedule.push(paymentData);
  }
 }
```

12. Save and close `LoanInput.js` and open `index.html`.
13. Add `dojo/data/ItemFileWriteStore` to the list of `require` statements:`require(`

```
// Set of module identifiers
```

```
 [ "dojo", "dojo/parser", "dijit/layout/BorderContainer",
   "dijit/layout/ContentPane", "loan/LoanInput", "dojo/currency",
   "dojox/charting/Chart", "dojox/charting/plot2d/Pie",
   "dojox/charting/action2d/Highlight",
   "dojox/charting/action2d/MoveSlice",
   "dojox/charting/action2d/Tooltip", "dojox/charting/themes/Dollar",
   "dojox/grid/DataGrid", "dojo/data/ItemFileReadStore",
   "dojo/data/ItemFileWriteStore" ],
```

14. Add the following code at the bottom of the `dojo.connect` function. `//create Store with Data`

```
  var gridData = {};
  gridData.items = loanWidget.amortizationSchedule;
  var gridStore = new dojo.data.ItemFileWriteStore({
   identifier:'mortgageData',
   label:"mortgageData",
   data:gridData});

  //set the store on grid
  var grid = dijit.byId("gridId");
  grid.setStore(gridStore);
```

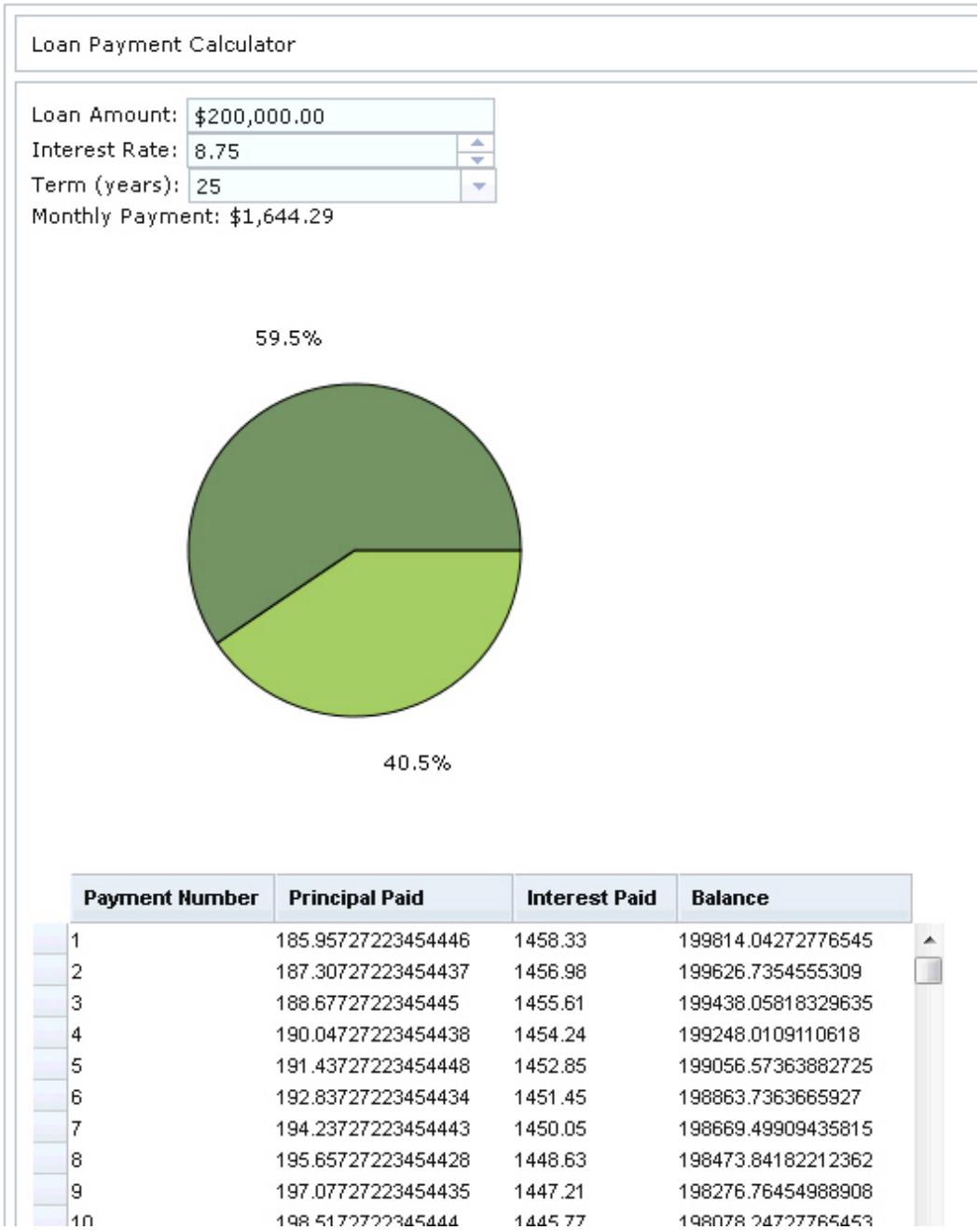
15. Run the page on server. The resulting `index.html` source looks like the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<link rel="stylesheet" type="text/css"
 href="dojo/dijit/themes/dijit.css">
<link rel="stylesheet" type="text/css"
 href="dojo/dijit/themes/claro/claro.css">
<title>index</title>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<script type="text/javascript"
 data-dojo-config="isDebug: false, async: true, parseOnLoad: true" src="dojo/dojo/dojo.js"></script>
<script type="text/javascript">

require(
 // Set of module identifiers
 [ "dojo", "dojo/parser", "dijit/layout/BorderContainer",
   "dijit/layout/ContentPane", "loan/LoanInput", "dojo/currency",
   "dojox/charting/Chart", "dojox/charting/plot2d/Pie",
   "dojox/charting/action2d/Highlight",
   "dojox/charting/action2d/MoveSlice",
   "dojox/charting/action2d/Tooltip", "dojox/charting/themes/Dollar",
   "dojox/grid/DataGrid", "dojo/data/ItemFileReadStore",
   "dojo/data/ItemFileWriteStore" ],

 // Callback function, invoked on dependencies evaluation results
 function(dojo) {
```

```
dojo.ready(function() {
 var chart = new dojox.charting.Chart("chart", null);
 chart.setTheme(dojox.charting.themes.Dollar);
 chart.addPlot("default", {
  type : "Pie",
  labelOffset : -30,
  radius : 90
 });
 chart.addSeries("paymentSeries", []);

 new dojox.charting.action2d.MoveSlice(chart, "default");
 new dojox.charting.action2d.Highlight(chart, "default");
 new dojox.charting.action2d.Tooltip(chart, "default");

 var loanWidget = dijit.byId("LoanInput");
 dojo.connect(dijit.byId("LoanInput"), "calculate", function(event) {
  // handle "calculate" from widget "LoanInput"
  var payment = loanWidget.monthlyPayment;
  if (payment == NaN) {
   payment = 0;
  }

  var formattedValue = dojo.currency.format(payment, {
   currency : 'USD'
  });
  dojo.attr("monthlyPayment", "innerHTML", formattedValue);

  // update the chart
  chart.updateSeries("paymentSeries", [ {
   y : loanWidget.principal,
   stroke : "black",
   tooltip : "Principal"
  }, {
   y : loanWidget.interestPaid,
   stroke : "black",
   tooltip : "Interest"
  } ]);
  chart.render()

  // create the store with data
  var gridData = {};
  gridData.items = loanWidget.amortizationSchedule;
  var gridStore = new dojo.data.ItemFileWriteStore({
   identifier: "mortgageData",
   label: "mortgageData",
   data: gridData
  });
```

```
    // set the store on the grid
    var grid = dijit.byId("gridId");
    grid.setStore(gridStore);
   });
  });
 });

</script>
<link rel="stylesheet" type="text/css" title =Style"
 href="dojo/dojox/grid/resources/Grid.css">
<link rel="stylesheet" type="text/css" title =Style"
 href="dojo/dojox/grid/resources/claroGrid.css">
</head>
<body class="claro">

 <div id="BorderContainer" style="height: 700px; width: 600px"
  data-dojo-type="dijit.layout.BorderContainer"
  data-dojo-props="design:'headline'">
  <div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-props="region:'top'">Loan Payment Calculator</div>
  <div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-props="region:'center'">
   <div data-dojo-type="loan.LoanInput" id="LoanInput"></div>
<div>Monthly Payment: <span id="monthlyPayment"></span></div>

<div id="chart" style="width: 350px; height: 350px"> </div>
 <table id="gridId" autowidth="true"
    data-dojo-type="dojox.grid.DataGrid"
    data-dojo-props="rowSelector:'20px'">
    <thead>
    <tr>
       <th field="paymentNum">Payment Number</th>
      <th field="principal">Principal Paid</th>
      <th field="interest">Interest Paid</th>
      <th field="balance">Balance</th>
    </tr>
    </thead>
</table>
</div>
</div>
</body>
</html>
```

The page should look like this with the data grid added to the bottom of the page:



Lesson checkpoint

You created and populated a Dojo data grid.
You learned:
- How to add a data grid to a web page.
- How to populate a data grid

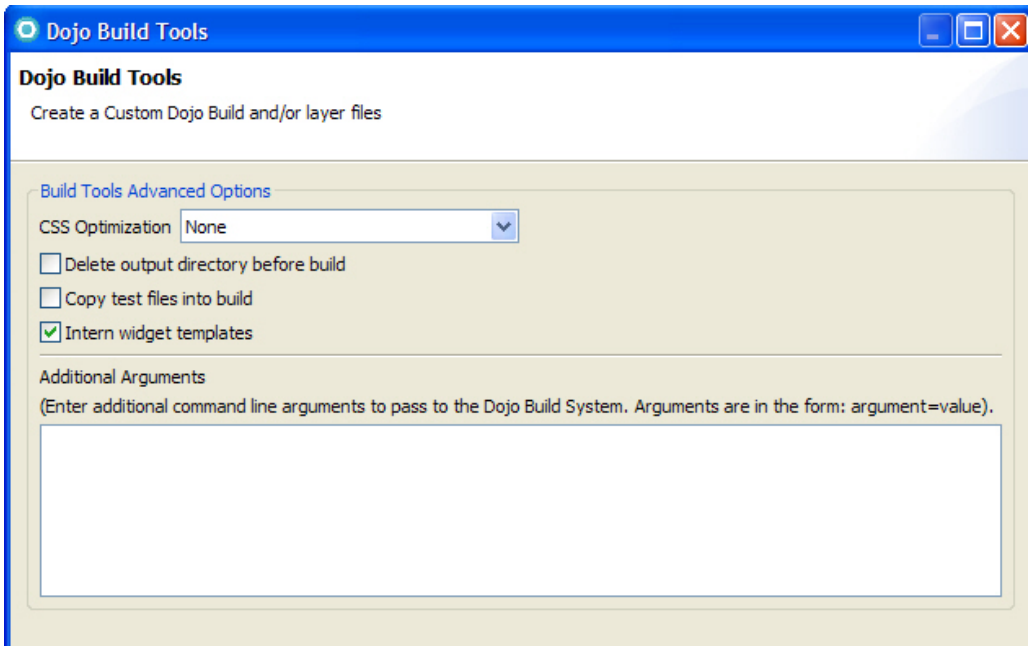
< Previous | Next >

Lesson 6: Create a Dojo custom build (optional)

This optional lesson highlights the steps required to create a Dojo custom build. The purpose of a custom Dojo build is to create an efficient version of Dojo, and of your code, that is suitable for deployment.

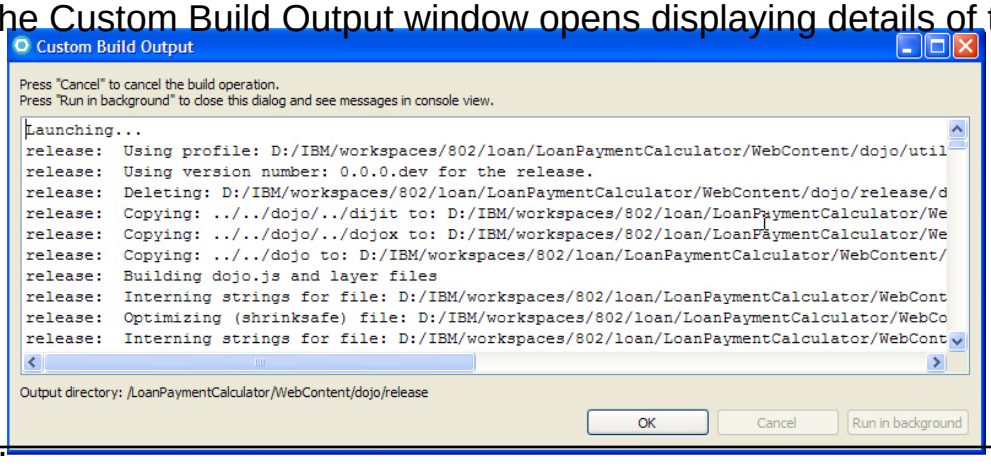
Learn more about the Dojo Build System here: www.dojotoolkit.org/reference-guide/build/index.html#build-index

1. In the Enterprise Explorer view, right-click the **LoanPaymentCalculator** project and select **New** > **Dojo Custom Build**. The Dojo Build Tools wizard opens.
2. Accept the default **Profile** location. Ensure that you complete a full build of the profile before you build individual layers. To complete a full build, clear the **Only Build Selected Layers** check box.
3. Specify the build scripts and output directories. You can accept the default values.
4. Click **Next** to specify advanced options.



   A. From the **CSS Optimization** list, select whether to remove comments and line returns.
   B. You can specify whether to delete output directories before building, copy test files into the build, or intern widget templates. When you intern a template, the HTML or CSS file is brought into the JavaScript file and assigned to a string. You can also specify other command line arguments.

5. Click **Finish**. The Custom Build Output window opens displaying details of the build operation.



The Dojo custom build built the entire Dojo distribution and the Dojo layer files that you selected into the output folder that you specified in the Custom Build wizard.

Lesson checkpoint

You ran a custom build on your Dojo project.
You learned:
- How to optimize your Dojo code for deployment.

< Previous | Next >

Lesson 7: Debugging your Dojo application with Firebug (optional)

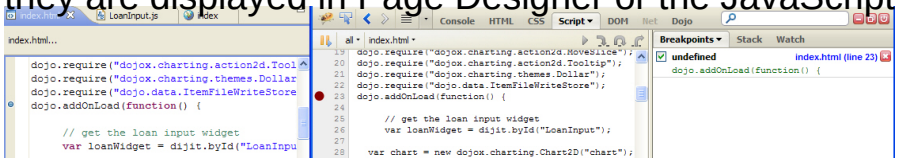
This optional lesson highlights how to use Firebug to debug your web application. This product includes a Dojo Firebug extension that includes additional tools for Dojo debugging.

Before you begin this lesson, install Mozilla Firefox.

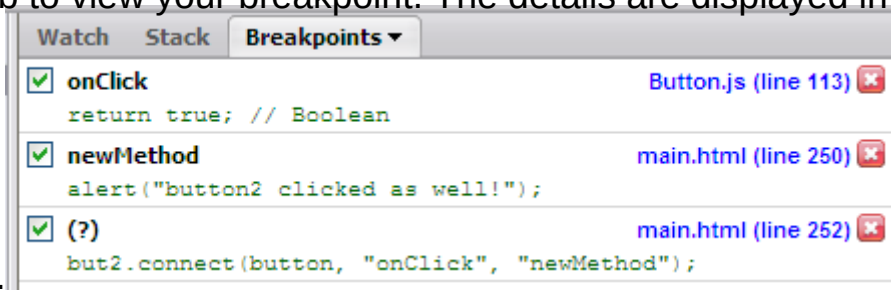
The Dojo Firebug extension helps you to :

- Inspect and highlight Dojo widgets listed on a web page
- Inspect connections and subscriptions information
- Inspect HTML elements

1. Configure the web application to run on Firefox with Firebug by clicking **Window** > **Web Browser** > **Firefox with Firebug**.
2. You can set JavaScript breakpoints inside script tags of a web page or in JavaScript files. The breakpoints are automatically transferred to Firebug when you debug on the server. Conversely, you can also set breakpoints in Firebug and they are displayed in Page Designer or the JavaScript editor.



3. In the Enterprise Explorer view, right-click `index.html` and select **Debug As** > **Debug on Server**.
4. Click **Finish**. Firebug is automatically installed, if required. The Dojo Firebug extension is also automatically installed into a new Firefox profile. Your new web page opens in Firefox.
5. Refresh the page in Firefox to ensure that the Dojo Firebug extension is loaded. A Dojo tab is displayed on the Firebug toolbar.
6. Enable the Console, Script, and Dojo Firebug panels by clicking the tab menu in each panel and selecting **Enabled**.
7. You can add breakpoints to connections by right-clicking the connection and selecting one of the following options:
   - Break on Target - sets a breakpoint at the beginning of a registered callback
   - Break on Event - sets a breakpoint at the beginning of the function which fires the event
   - Break on 'connect place' - sets a breakpoint where `dojo.connect()` is called to register this callback
8. Switch to the Script tab to view your breakpoint. The details are displayed in the Breakpoint side panel.



9. Continue to use the Firebug debugging tools to evaluate your code. For more information about debugging JavaScript with Firebug, see getfirebug.com/wiki/index.php/Main_Page.

Lesson checkpoint

You debugged your Dojo files with Firebug.
You learned:
- How to open Firebug
- How to set breakpoints on connections

Import the solution

Note: You can also import the completed web project solution into your workspace and run the project on the server to see the results of this tutorial.

< Previous | Next >

< Previous

Summary: Create a loan payment calculator with Dojo

You learned how to create a Dojo-based loan payment calculator

Lessons learned

By completing this tutorial you learned about the following concepts and tasks:

- How to create a Dojo-enabled web application to contain all of the resources required by your application.
- How to create a custom Dojo widget.
- How to write JavaScript and HTML code to create the Dojo user interface for your application.
- How to use content assistance, templates and wizards to write Dojo code and HTML.
- How to deploy your project to a server.
- How to create a Dojo custom build.
- How to debug your web application using Firebug and the Dojo Firebug extension.
- About the Dojo Toolkit.

Additional resources

For more information about the Dojo toolkit, see dojotoolkit.org. Documentation for the Dojo widgets is available at docs.dojocampus.org/.

See the Rational® Application Developer wiki for videos and articles to help you get started quickly with developing Dojo applications.

< Previous

Next >

Tutorial: Create and configure a JPA project

This tutorial shows you how to create and configure a Java™ Persistence API (JPA) project.

Learning objectives

In this tutorial, you will learn how to do the following things:

- Add JPA support to a web project
- Create a connection to a database
- Create JPA entities from database tables
- Add primary keys and query methods to entities
- Configure a runtime database connection
- Create JPA Manager beans for JPA entities

Restriction: The capabilities used in this tutorial are not available in IBM® WebSphere® Application Server Developer Tools for Eclipse.

Time required

30 minutes

Next >

Create and configure a JPA 2.0 project

This tutorial shows you how to add JPA support to a wweb project and then use JPA tools to create and configure entity beans from database tables.

This tutorial might require some optionally installable components. To ensure that you installed the appropriate optional components, see the System requirements list.

This tutorial is divided into several exercises that must be completed in sequence for the tutorial to work properly.

Learning objectives

In this tutorial, you will learn how to do the following things:

- Add JPA support to a web project
- Create a connection to a database
- Create JPA entity beans from database tables
- Add primary keys to entity beans
- Configure a runtime database connection
- Create JPA manager beans for entity beans

Time required

This tutorial should take approximately 30 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete.

Skill level

Intermediate

Audience

This tutorial is intended for users who want to become familiar with the some of the JPA tools provided in this product.

System requirements

To complete this tutorial, you need to have the following things:

- To use this tutorial, you must have one of the following application servers installed and configured.
  - **WebSphere® Application Server, Version 8.5**
  - **WebSphere Application Server, Version 8.0**
  - **WebSphere Application Server, Version 7.0, including the Java Persistence API 2.0 feature.**
- **A clean workspace**.

Lessons in this tutorial

- **Lesson 1.1: Import the required resources**
  Before you begin, you need to import the required resources for this tutorial: a sample Derby database and a set of web pages.
- **Lesson 1.2: Add JPA support to the project**
  In this lesson you will learn how to add JPA support to the web project that you imported in the previous lesson.
- **Lesson 1.3: Create a connection to a database**
  In this lesson you will learn how to add a connection to a sample Derby database.
- **Lesson 1.4: Create JPA entities from the database tables**
  In this lesson you will learn how generate JPA entities from tables in the sample Derby database that you connected to in the previous lesson.
- **Lesson 1.5: Add primary keys and named queries to the entities and configure the runtime database connection**
  In this lesson you will learn how to add primary keys and named queries to the JPA entities that you generated in the previous lesson. You will also learn how to configure a runtime connection to the Derby database.

- **Lesson 1.6: Create JPA manager beans**

  In this lesson you will learn how to generate JPA manager beans for the JPA entities that you created in the previous lessons. JPA Manager beans are used to manage all of the data access related to your JPA entities, such as retrieving data through queries and updating your entities. They fill the role that would normally be filled by a session bean in an EJB environment. All of the business logic related to an entity is performed by the JPA Manager bean.

- **Lesson 1.7: Run the Web application to test the entities**

  In this lesson you will learn how to run the Web application on the server.

< Previous | Next >

Lesson 1.1: Import the required resources

Before you begin, you need to import the required resources for this tutorial: a sample Derby database and a set of web pages.

The main purpose of this tutorial is to teach you how to add Java™ Persistence API (JPA) support to a project and to how to use the JPA tools create and configure JPA entity beans and JPA manager beans. To complete this tutorial, you will need to access a sample Derby database. To test your work, you will run a sample web application that uses JPA data bound to Java Server Faces (JSF) JSP files to access and manipulate data in the database. The tutorial does not go into the detail of designing the look and feel of a web site or working with JSF. Accordingly, the web site design has been prepared for you already.

The sample database and the web pages needed to complete this tutorial are included in a ZIP file. This lesson will take you through the steps of importing the ZIP file so that you can use the database and web pages. You will also set the target server for the web project.

To import the content of the ZIP file:

Import the sample project file

Import the project. Switch to the Web perspective (**Window** > **Open Perspective** > **Web**).

Migrate the imported resources

After you import the tutorial resources, you must migrate the resources to the current workspace version.

1. Click **Next** on the first Workspace Migration page.
2. On the Workspace projects which need migration page, ensure that both JPATutorial and JPATutorialEAR projects are selected and click **Next**.
3. Click **Next** on the Migration Project Resources page.
4. On the Undefined Server Runtime Page, select the version of WebSphere Application Server that you want to define as the targeted runtime and then click **Next**.
5. Click **Finish** on the first Complete Migration Startup page.
6. When the migration completes, click **OK** in the Migration Validation window.

Set the target server

Setting the target server for the Web project enables you to test the resources that you will be creating in this tutorial. To set the target server:

1. In the Enterprise Explorer view of the Web perspective, right-click **JPATutorial** and click **Properties**.
2. In the properties list, click **Server**.
3. From the list of servers, select the version of WebSphere® Application Server that you selected as the targeted runtime when you migrated the imported resources. Click **Apply**.
4. In the Enterprise Explorer view of the Web perspective, right-click **JPATutorialEAR** and click **Properties**.
5. In the properties list, click **Server**.
6. From the list of servers, select the version of WebSphere Application Server that you selected as the targeted runtime when you migrated the imported resources. Click **Apply**.

Lesson checkpoint

You have now imported the JPATutorial web project and set the target server.

You are now ready to continue with the other lessons in this tutorial.

< Previous | Next >

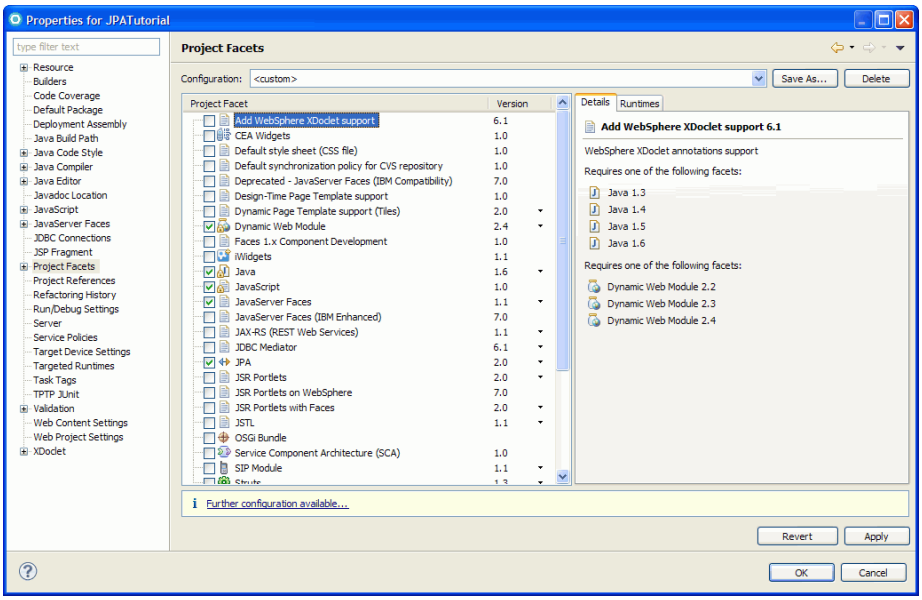
Lesson 1.2: Add JPA support to the project

In this lesson you will learn how to add JPA support to the web project that you imported in the previous lesson.

Before you begin, you must complete Lesson 1.1: Import the required resources.

Add the JPA facet to the project

1. Right-click the JPATutorial project and then click **Properties**. The Properties page opens.
2. Click **Project Facets**. The Project Facets property page opens.
3. Under Project Facets, select **JPA**, and then click **Finish**.



JPA 2.0 is now listed as a facet of the project.

4. Click **OK**. The Properties window closes.

Lesson checkpoint

You have completed Lesson 1.2. In this lesson, you learned how to add JPA support to a web project.

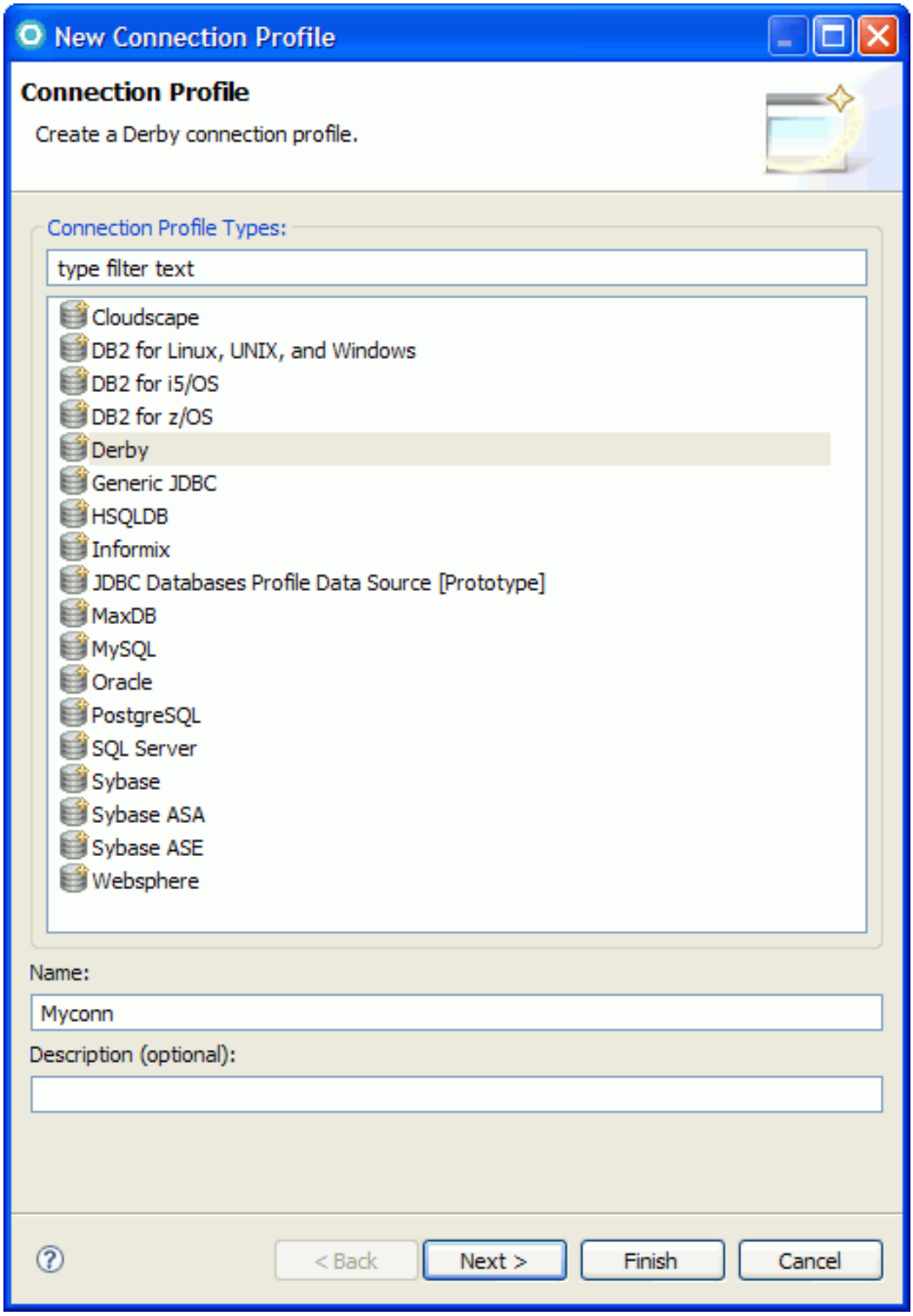
Lesson 1.3: Create a connection to a database

In this lesson you will learn how to add a connection to a sample Derby database.

Before you begin, you must complete Lesson 1.2: Add JPA support to the project.

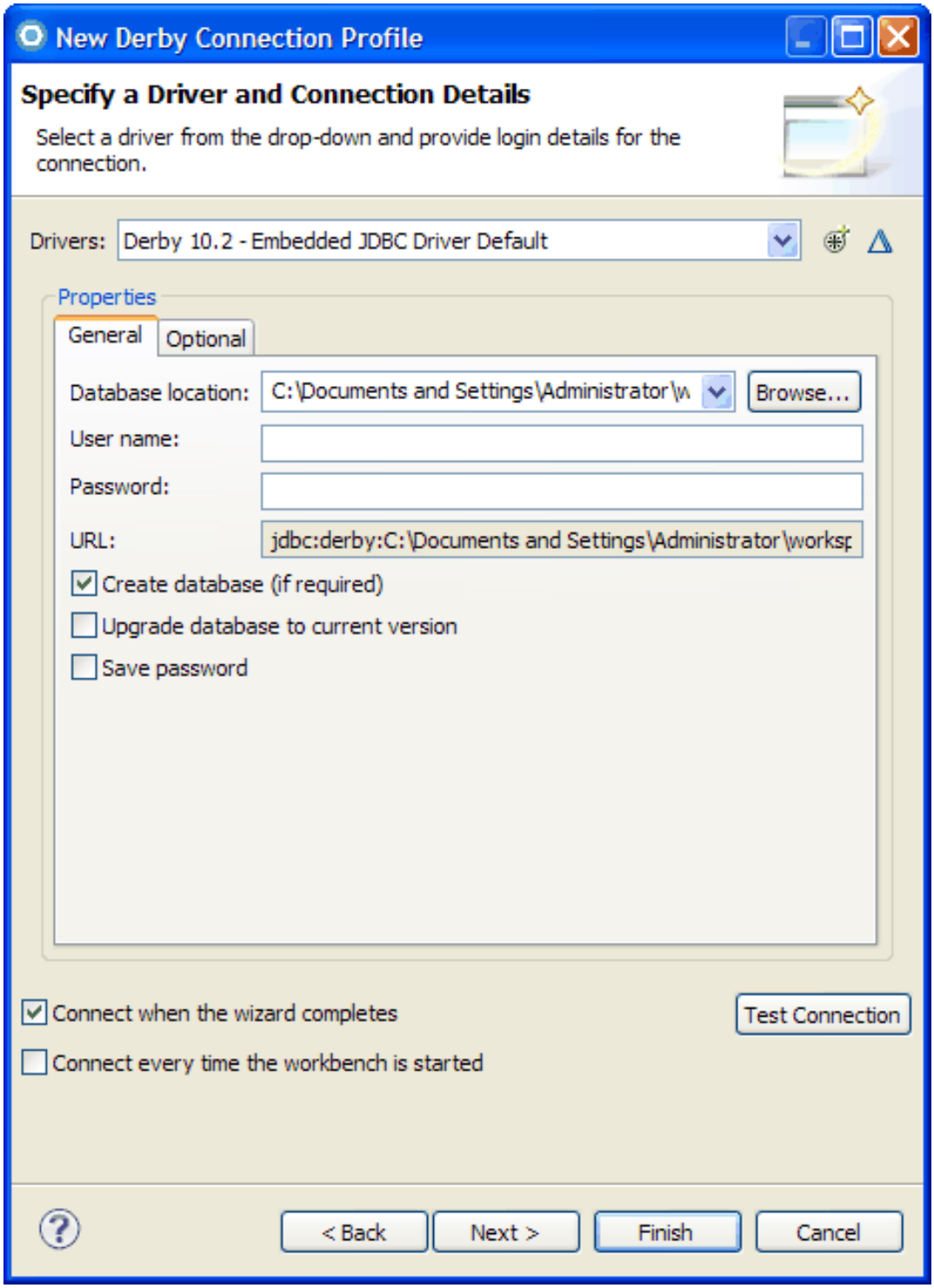
Add a connection to the Derby sample database

1. In Project Explorer, right-click **JPATutorial** and select **Properties**. The Properties for JPATutorial window opens.
2. Click **Java Persistence**.
3. Make sure that **RAD JPA 2.0 Platform** is selected in the Platform list.
4. Under **Connection**, click **Add connection**. The New Connection Profile wizard opens.
5. In the connection profile type list, click **Derby**. In the Name field, type a name for your connection (for example, name it Myconn) and then click **Next**.

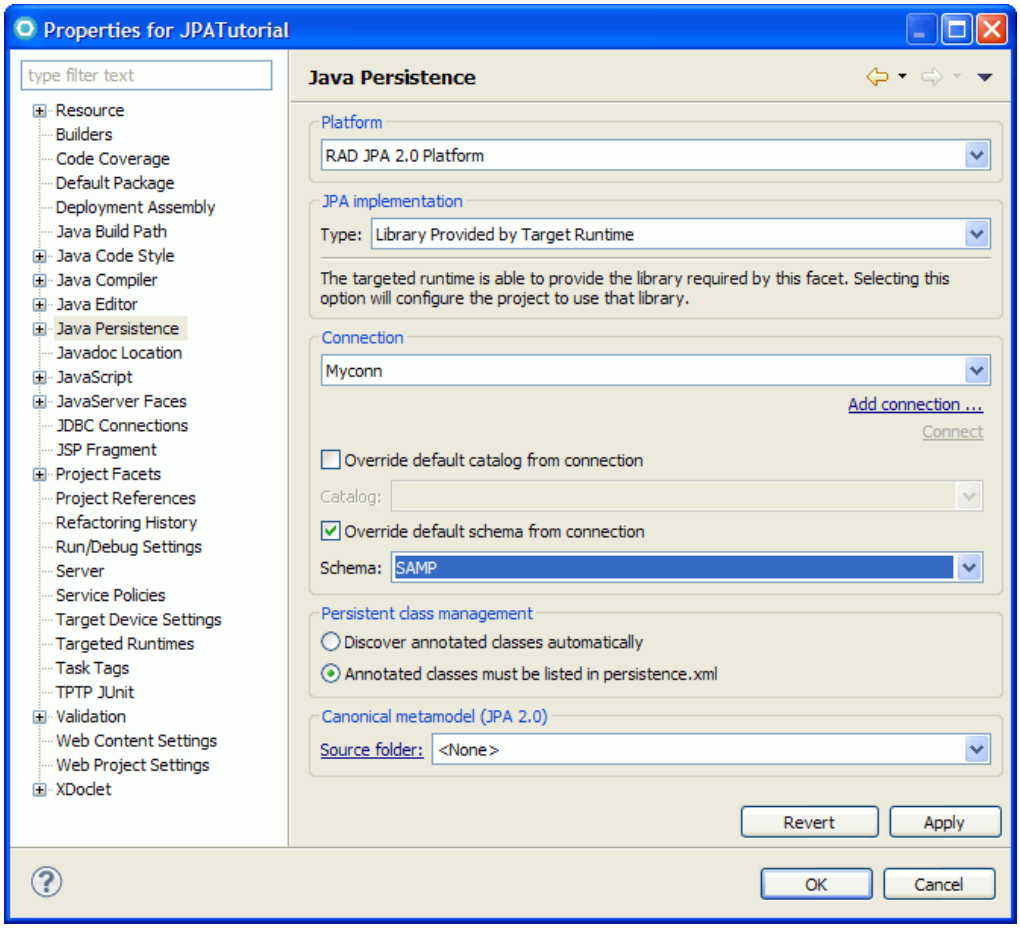


The New Derby Connection Profile wizard opens

6. From the Drivers list at the top of the wizard, select Derby 10.2 - Embedded JDBC Driver Default.
7. On the General tab of the Specify a Driver and Connection Details page of the Derby Embedded Database wizard, click **Browse** and select *<workspace_location>*`JPATutorial\WebContent\sample` and click **OK**.



8. Click **Test connection** to ensure that you can connect to the Derby database. On the Success dialog-box, click **OK**. Click **Next**.
9. Click **Finish** to complete the connection to the derby database.



10. On the JPA properties page, select **Override default schema from connection** and then select **SAMP** in the **Schema** list. Click **Apply** to apply the changes to your Web project, and then click **OK**.

Lesson checkpoint

You have completed Lesson 1.3. In this lesson, you learned how to add a connection to a Derby database.

< Previous | Next >

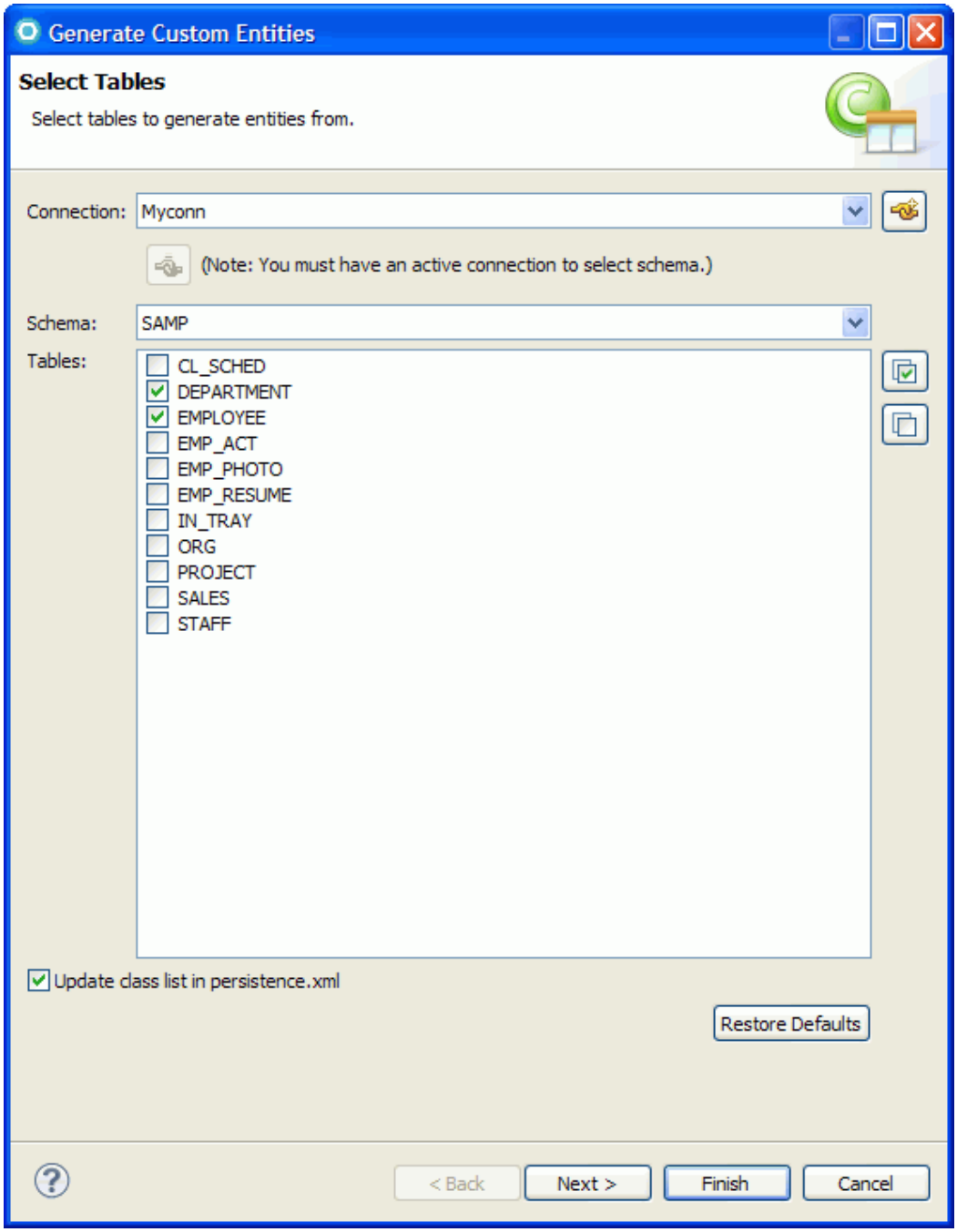
Lesson 1.4: Create JPA entities from the database tables

In this lesson you will learn how generate JPA entities from tables in the sample Derby database that you connected to in the previous lesson.

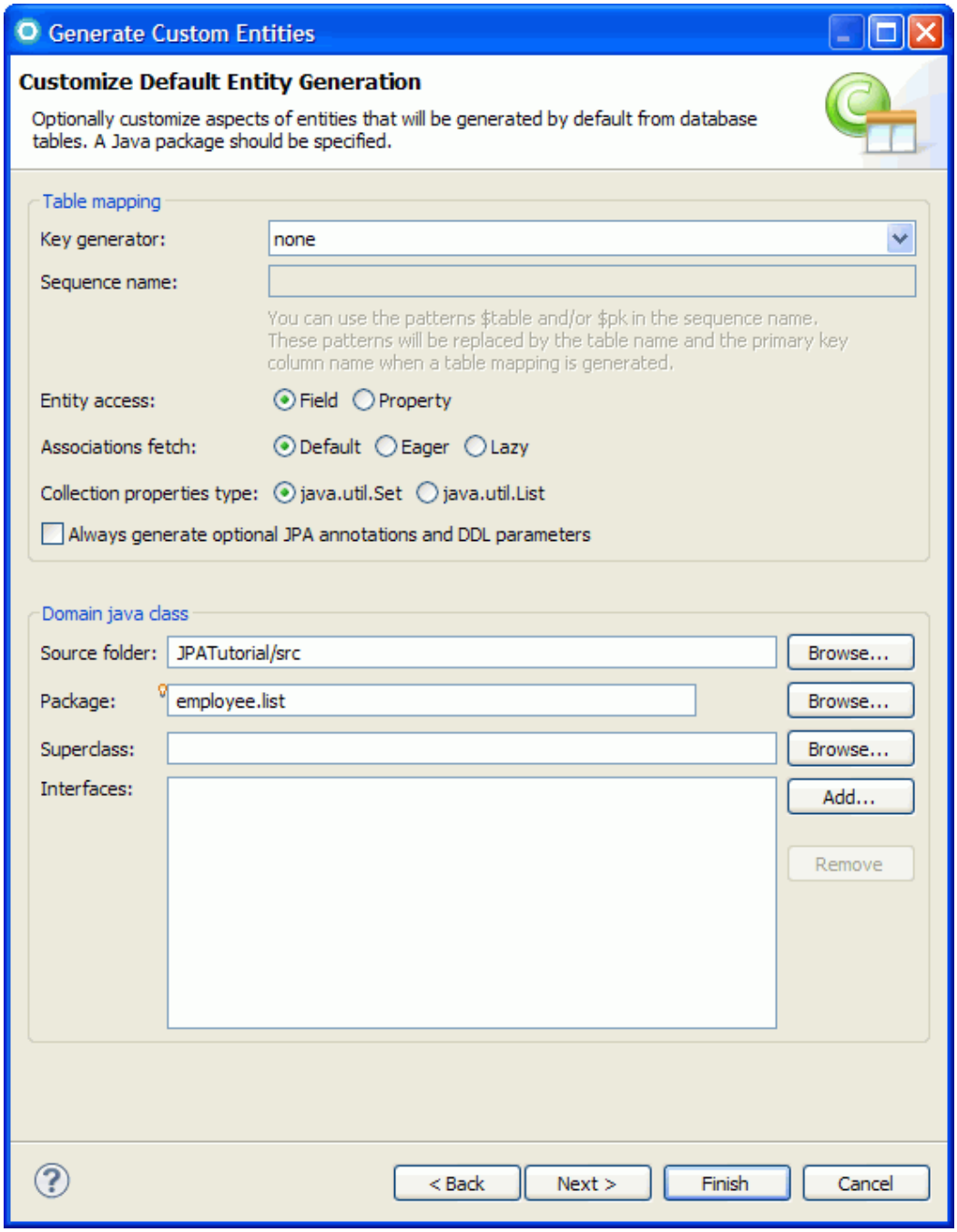
Before you begin, you must complete Lesson 1.3: Create a connection to a database.

Generate JPA entity beans

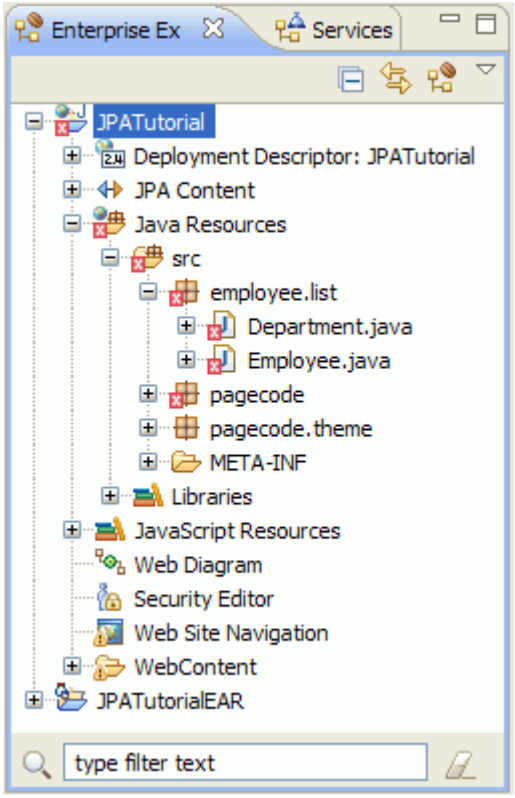
1. Right-click the project and then click **JPA Tools** > **Generate Entities from Tables**.
2. In the Database Connection window, ensure that the database connection that you created in the previous lesson is selected.
3. For Schema, select **SAMP** from the drop-down list. If the drop-down list does not work, click **Reconnect** and then try selecting the SAMP schema again.
4. In the list of tables, select **DEPARTMENT** and **EMPLOYEE**.



5. Ensure **Update class list in persistence.xml** is selected, and then click **Next**.
6. On the Table Associations page, click **Next**.
7. On the Customize Default Entity Generation page, type `employee.list` in the Package field and then click **Finish**.



The package employee.list is created and populated with two entities, Employee.java and Department.java.



Lesson checkpoint

You have completed Lesson 1.3. In this lesson, you learned how to generate JPA entities from tables in the sample Derby database.

< Previous | Next >

Lesson 1.5: Add primary keys and named queries to the entities and configure the runtime database connection

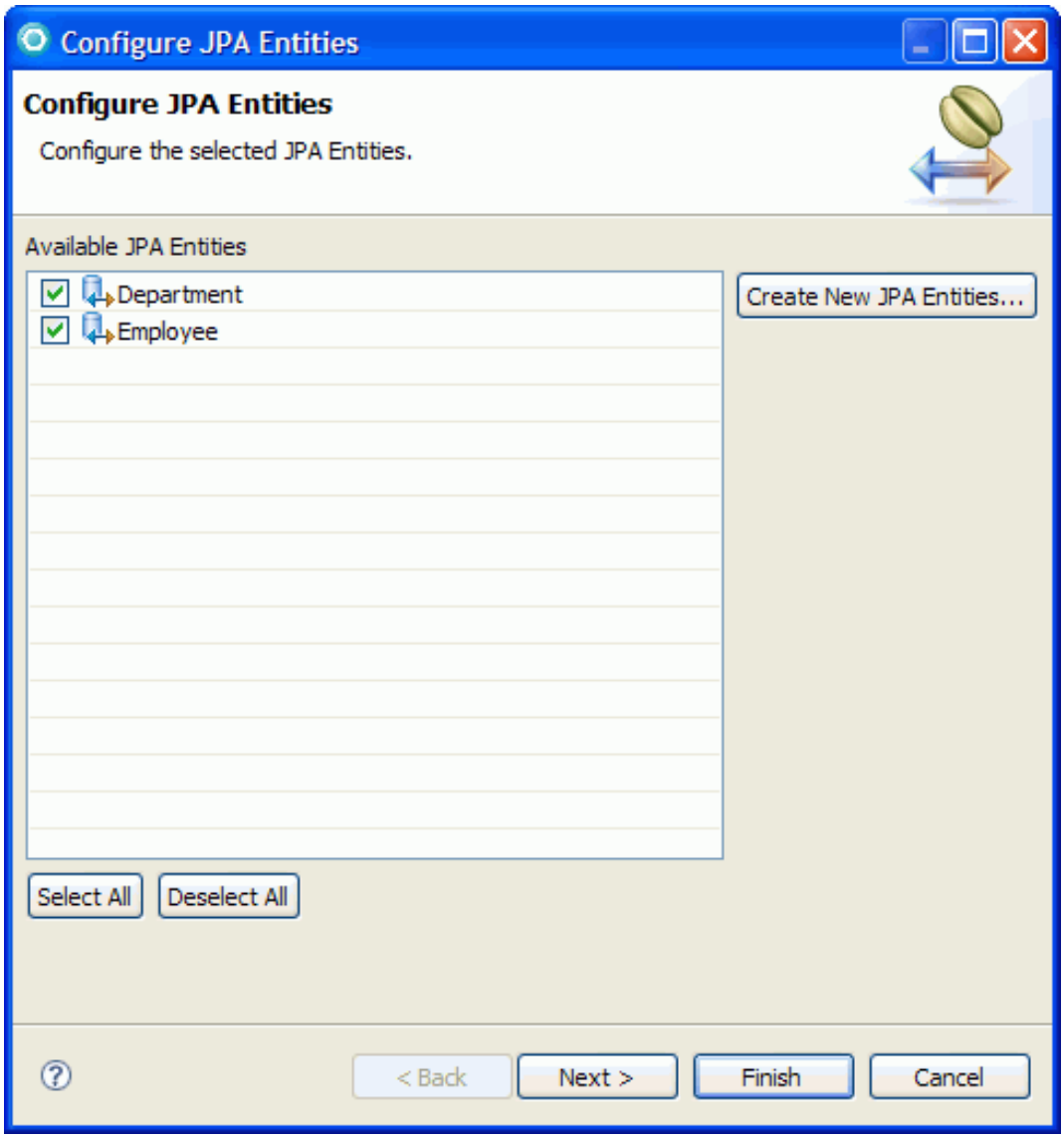
In this lesson you will learn how to add primary keys and named queries to the JPA entities that you generated in the previous lesson. You will also learn how to configure a runtime connection to the Derby database.

Before you begin, you must complete Lesson 1.4: Create JPA entities from the database tables.

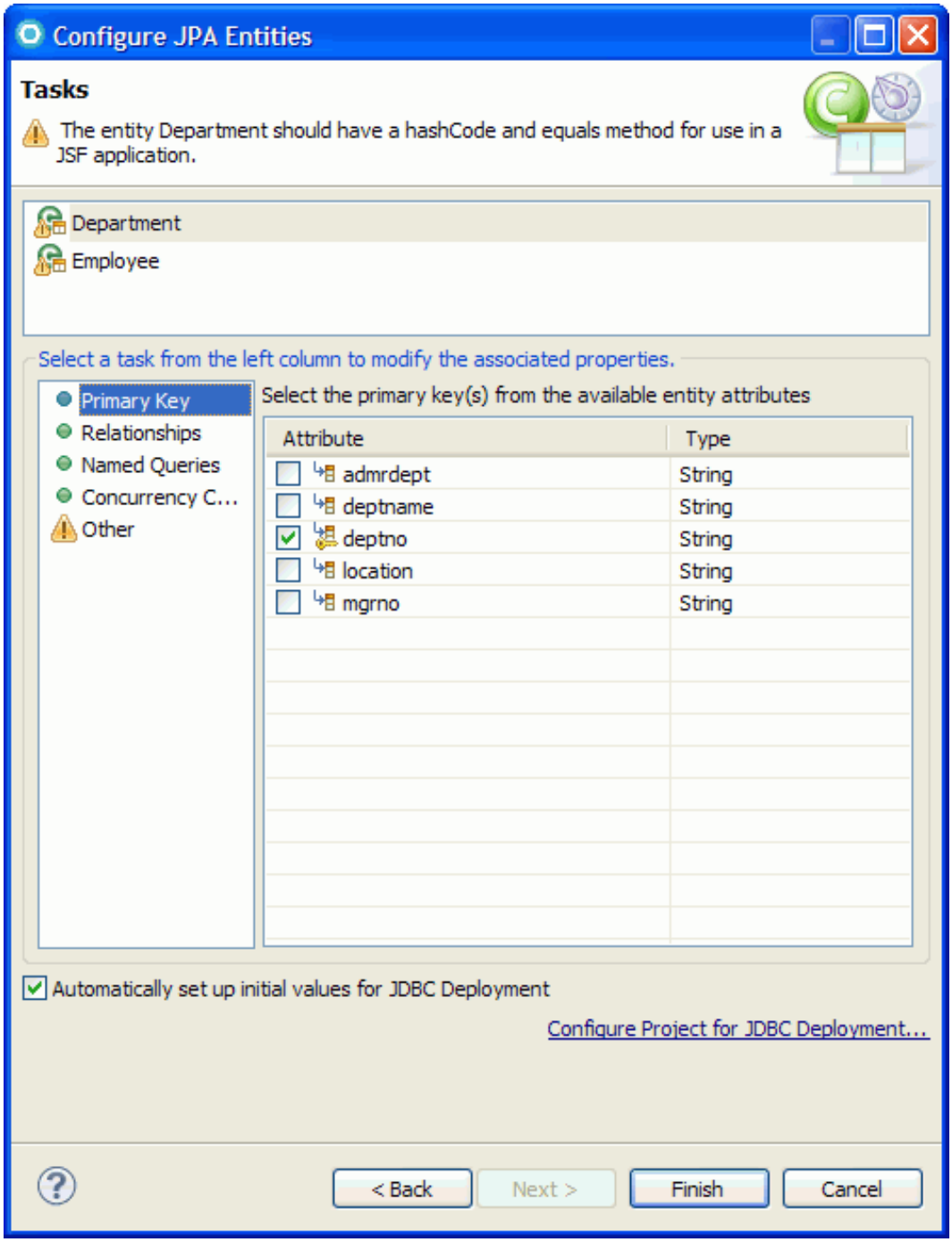
Configure the entity beans

JPA entities must include an @Id annotation to define the entity's primary key. The JPA Tools can add the required primary key annotation for you. Perform the following steps to configure the entity beans:

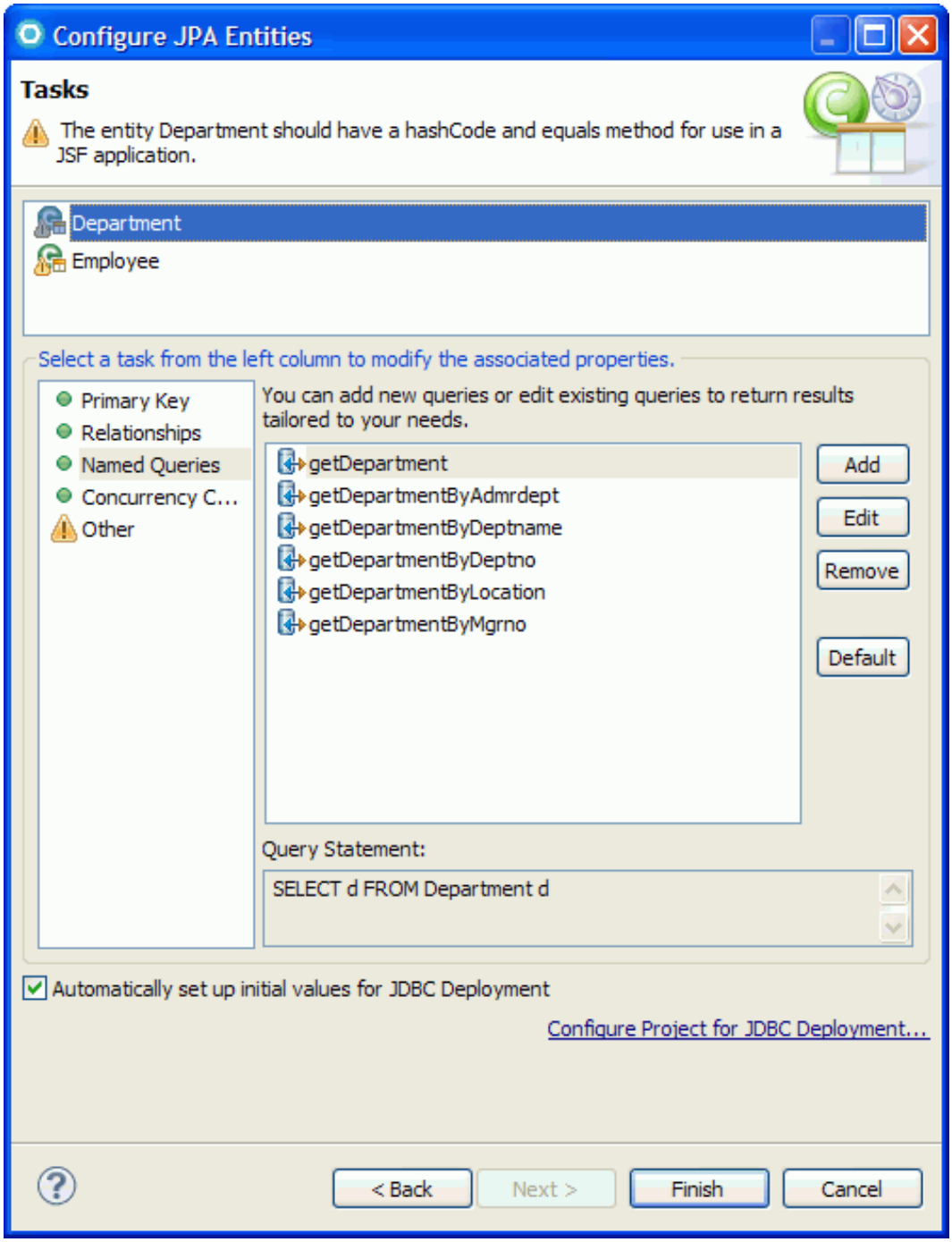
1. In the Enterprise Explorer, right-click on one of the entity beans - Employee.java or Department.java - and select **JPA Tools** > **Configure JPA Entities**.
2. On the Configure JPA Entities Page, ensure that both entities are selected. Click **Next**.



3. On the Tasks Page, click **Department**. Click **Primary Key**, and then select **deptno** as the primary key.

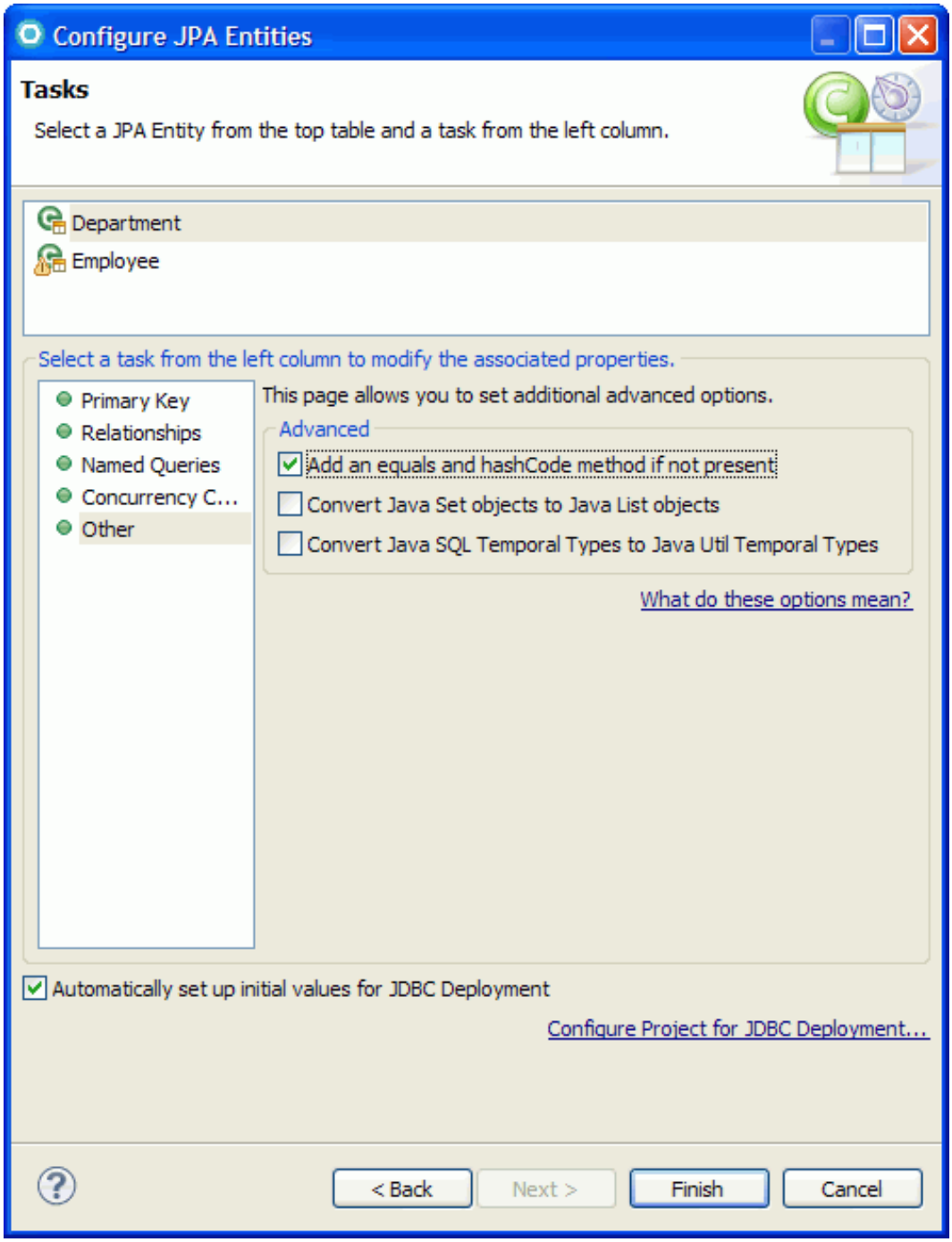


4. Click **Named Queries** and then click **Default**.



Note: If you do not see the "getDepartment" named query, click **Add** and then **OK**.

5. Click **Other** and then click **Add an equals and hashCode method if not present**.

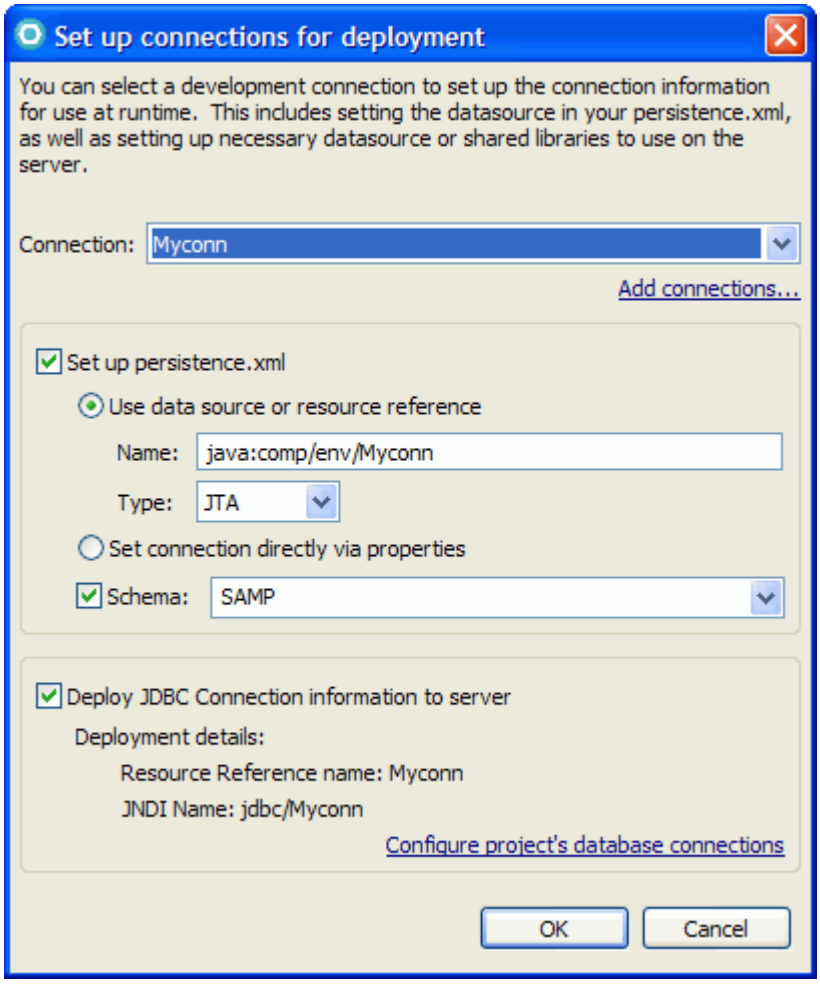


6. On the Tasks page, click **Employee**. Click **Primary Key**, and then select **empno** as the primary key.
7. Click **Named Queries** and then click **Default**.**Note:** If you do not see the "getEmployee" named query, click **Add** and then **OK**.
8. Click **Other** and then click **Add an equals and hashCode method if not present**.
9. Click **Finish**. The Configure JPA Entities wizard adds the primary keys, named queries and other methods to the entities.

Configure the runtime connection

Perform the following steps to configure the runtime connection to the Derby database:

1. Right-click the project in the Enterprise Explorer and click **JPA Tools** > **Configure Project for JDBC deployment**.
2. For Schema, select **SAMP**. Leave all other selections at their defaults and click **OK**.



Lesson checkpoint

You have completed Lesson 1.5. In this lesson, you learned how to configure your JPA entity beans. You also learned how to configure a runtime connection to a database.

< Previous | Next >

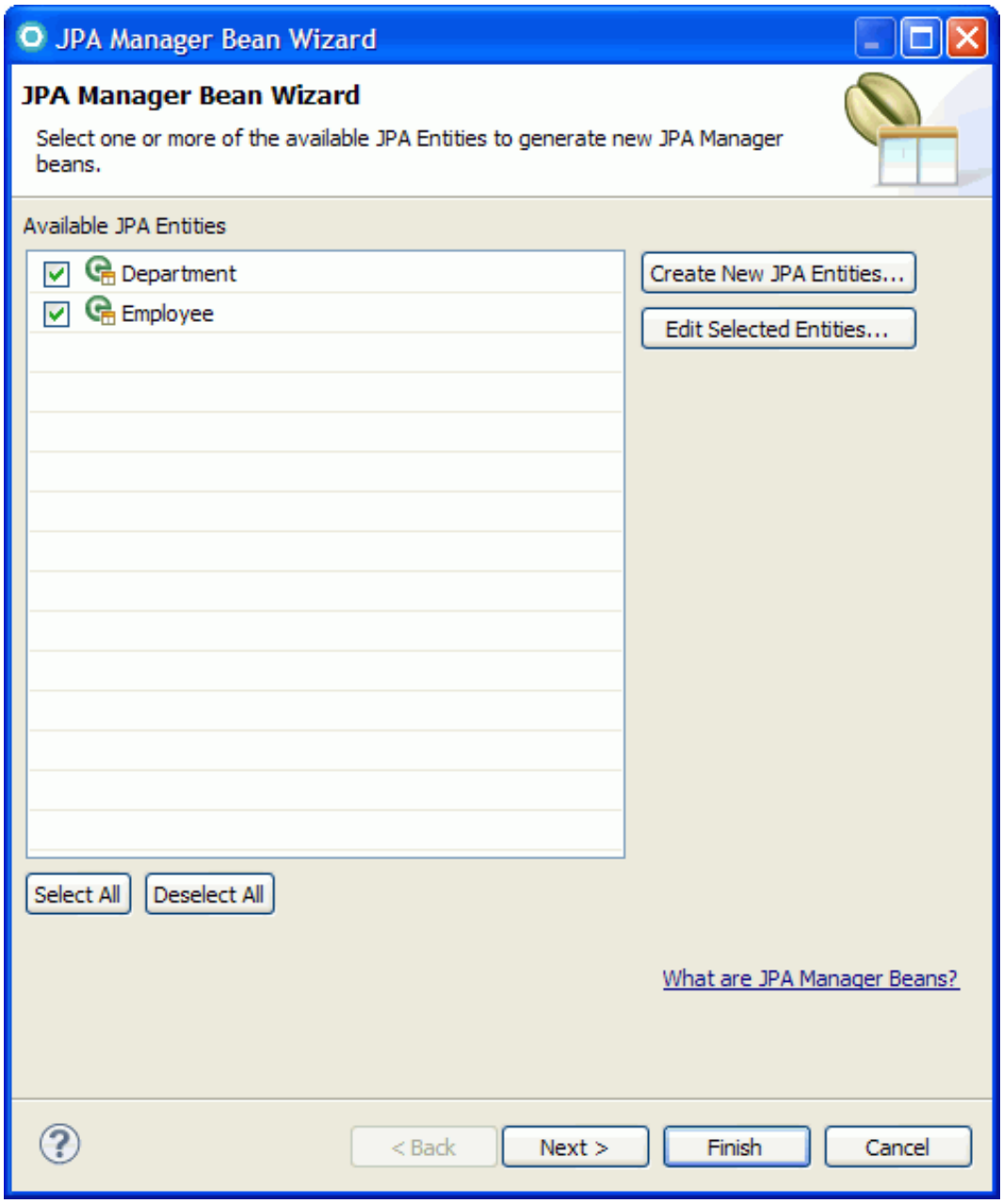
Lesson 1.6: Create JPA manager beans

In this lesson you will learn how to generate JPA manager beans for the JPA entities that you created in the previous lessons. JPA Manager beans are used to manage all of the data access related to your JPA entities, such as retrieving data through queries and updating your entities. They fill the role that would normally be filled by a session bean in an EJB environment. All of the business logic related to an entity is performed by the JPA Manager bean.

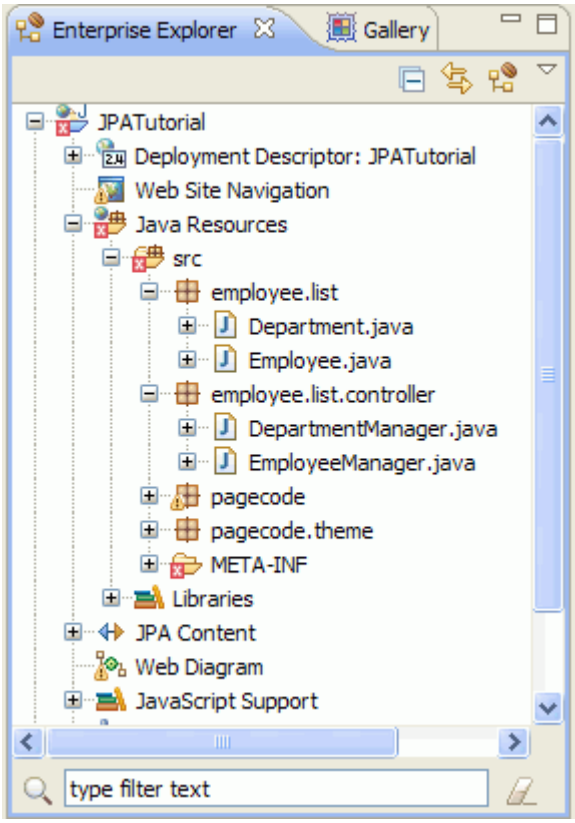
Before you begin, you must complete Lesson 1.4: Create JPA entities from the database tables.

Generate entity managers

1. In the Enterprise Explorer view, right-click on one of the entity beans (Department.java or Employee.java) and select **JPA Tools** > **Add JPA Manager Beans**.
2. In the JPA Manager Bean Wizard, select both **Department** and **Employee**.



3. Click **Finish**. This creates a package named employee.list.controller and JPA Manager beans for each entity.



4. Modify the method that is used to populate the list of departments:

   A. Open DepartmentManager.java.

   B. Locate the following method:`public List<SelectItem> getDepartmentSelectList() {`

```
List<Department> departmentList = getDepartment();
List<SelectItem> selectList = new ArrayList<SelectItem>();
MessageFormat mf = new MessageFormat("{0}");
for (Department department : departmentList) {
 selectList.add(new SelectItem(department, mf.format(
   new Object[] { department.getDeptno() },
   new StringBuffer(), null).toString()));
}
return selectList;
}
```

   C. Delete the method and replace it with the following method:`public List<SelectItem> getDepartmentSelectList() {`

```
List<Department> departmentList = getDepartment();
List<SelectItem> selectList = new ArrayList<SelectItem>();
MessageFormat mf = new MessageFormat("{0}");
for (Department department : departmentList) {
 selectList.add(new SelectItem(department.getDeptno(), mf.format(
   new Object[] { department.getDeptname() + " (" + department.getDeptno() + ")" }, new StringBuffer(),
   null).toString()));
}
return selectList;
}
```

Lesson checkpoint

You have completed Lesson 1.6. In this lesson, you learned how to create entity managers for your JPA entity beans.

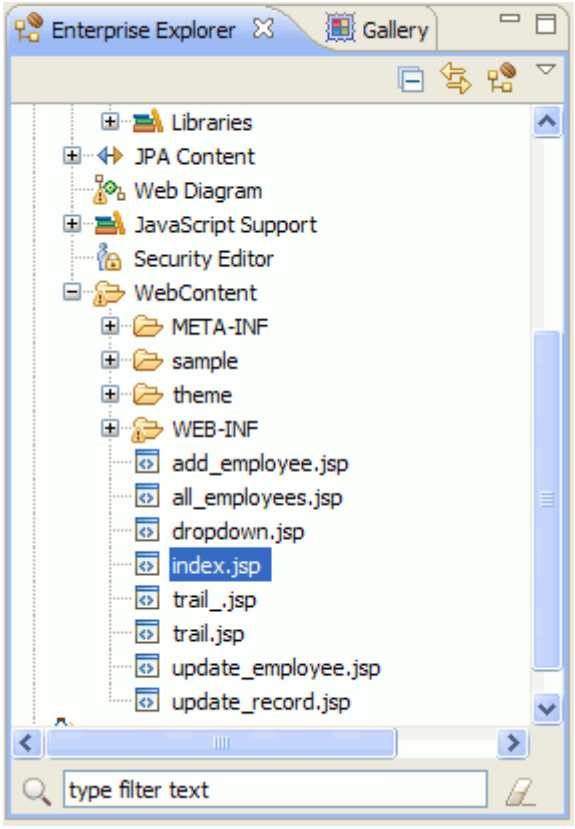
Lesson 1.7: Run the Web application to test the entities

In this lesson you will learn how to run the Web application on the server.

Before you begin, you must complete the lesson Lesson 1.6: Create JPA manager beans.

To run the web project on the server:

1. In the Enterprise Explorer view, expand **JPATutorial** > **Webcontent**.
2. Right-click the index.jsp file, and click **Run As** > **Run on Server**.



3. A web browser opens and displays the Employee List application.

The Employee List application demonstrates the use of Java™ Persistent API (JPA) to access employee data. The application maps persistent entities to columns in a database table and presents the information on a web site where users can create, read, update, and delete data.

Lesson checkpoint

Congratulations! You have completed this tutorial.

< Previous | Next >

< Previous

Summary

This tutorial guided you through the detailed steps to add JPA 1.0 support to a web project and to create and configure entity beans. You also created JPA manager beans and ran a Web application to test your work.

Lessons learned

If you completed all of the lessons, you should now be able to do the following tasks:

- Add JPA support to a web project
- Create a connection to a database
- Create JPA entities from database tables
- Add primary keys and query methods to entities
- Configure a runtime database connection
- Create JPA manager beans for entities

Additional resources

For more information about JPA, consult the online help (**Help** > **Help Contents**).

< Previous

Next >

Tutorial: Create a Hello World Java Application

This tutorial shows you how to create a simple Java™ application using the tools included in the Rational® Software Development Platform.

Learning objectives

In this tutorial you will learn how to:

- Create a Java project
- Create Java source files
- Run a Java program on a server
- Debug a Java program
- Use editor features to help you complete or fix Java code
- Rename source files without breaking dependencies
- Use retrieval aids to quickly find Java artifacts
- Use workbench views to improve your efficiency.

Time required

1 hour

Next >

Module 1: Developing a Java program

In this module, you will learn the basics of creating and running a Java™ program in the workbench.

Learning objectives

After finishing, you will know how to do the following tasks:

- Create a Java project
- Create Java source files
- Run a Java program on a server
- Debug a Java program.

Time required

25 minutes

Lessons in this module

- **Lesson 1.1: Create a Java project**

  This video shows you how to create a Java project.

- **Lesson 1.2: Run the Java program**

  This video shows you how to run your Java program on a server.

- **Lesson 1.3: Debug the Java program**

  This video shows you how to debug your Java program.

Lesson 1.1: Create a Java project

This video shows you how to create a Java™ project.

Lesson 1.2: Run the Java program

This video shows you how to run your Java™ program on a server.

Lesson 1.3: Debug the Java program

This video shows you how to debug your Java™ program.

Module 2: Using productivity features of the workbench

In this module, you will learn how to use workbench features to enhance your productivity.

Learning objectives

You will learn to use these features:

- Use editor features to help you complete or fix Java™ code
- Rename source files without breaking dependencies
- Use retrieval aids to quickly find Java artifacts
- Use workbench views to improve your efficiency.

Time required

60 minutes

Lessons in this module

- **Lesson 2.1: Code assist**

  This video shows you how to use code assist when you edit a Java file.

- **Lesson 2.2: Quick fix**

  This video shows you how to use quick fix to update Java code.

- **Lesson 2.3: Refactoring**

  This video shows you how to refactor Java code.

- **Lesson 2.4: Local history**

  This video shows you how to view changes made to your Java project files.

- **Lesson 2.5: Search**

  This video shows you how to search your Java project.

Lesson 2.1: Code assist

This video shows you how to use code assist when you edit a Java™ file.

Lesson 2.2: Quick fix

This video shows you how to use quick fix to update Java™ code.

Lesson 2.3: Refactoring

This video shows you how to refactor Java™ code.

Lesson 2.4: Local history

This video shows you how to view changes made to your Java™ project files.

< Previous

Lesson 2.5: Search

This video shows you how to search your Java™ project.

< Previous

Develop an application by using editable UML, topic, and browse diagrams

When you develop an application, you can use different types of diagrams to organize and capture new ideas, to document and communicate these ideas, or to understand an existing application.

Watch tutorial

Learning objectives

This tutorial shows you how to use different types of diagrams to design, document, and explore application elements such as Java™ code, Enterprise JavaBeans, and database objects. Specifically, this tutorial shows you how to do the following things:

- Use editable Unified Modeling Language (UML) diagrams to organize and capture ideas
- Use topic diagrams to automate documentation
- Use browse diagrams to discover and understand workspace artifacts

Time required

4 minutes

Lesson: Using editable UML, topic, and browse diagrams to develop an application

This tutorial introduces the editable Unified Modeling Language (UML), topic, and browse diagram types and demonstrates how to use them to design, document, and explore application elements in your development workspace.

Next >

Tutorial: Creating a JAX-RS web service

The following tutorial will walk you through creating a JAX-RS application. Any JAX-RS v1.0 implementation can be used, but for this tutorial we will be using the IBM implementation which is installed when you install a WebSphere Test Environment. For this tutorial you should have the WebSphere Application Server v7 or v8 test environment installed.

Learning objectives

In this tutorial, you will learn to:

- Integrate a JAX-RS implementation into the workbench, and target projects to use this implementation
- Create a bottom-up web service which uses JAX-RS, using quick fixes to speed up the process.

Time required

To complete the entire tutorial you will need 30 minutes.

Prerequisites

- For this tutorial you should have the WebSphere Application Server v7 or v8 test environment installed, with the Web 2.0 v1.0.1 Feature Pack.

If you do not want to complete the tutorial manually, you can import the application by clicking the following link. Note that this sample code is targeted to WebSphere Application Server v8 and cannot be deployed to WebSphere Application Server v7. You can then proceed to the test section of Lesson 2. Import the JAX-RS Address Book tutorial sample code

If you want to watch a JAX-RS web service being created, a demonstration is available at the following URL: Creating and testing JAX-RS (RESTful) web services

When you are ready, begin Creating a server and web project.

Next >

Lesson 1: Creating a server and web project

The web service needs to reside in a web project with the JAX-RS facet enabled.

Create a JAX-RS enabled server

Before creating the web service you need to have a server which has Java™ 5.0 or later JVM support defined and started. By default a server is created for you when you install WebSphere® Application Server. This server can be seen in the Servers view. However if you want to create a new WebSphere Application Server do the following:

1. From the **File** menu, select **New** > **Other** > **Server** > **Server** > **Next**.
2. Select **WebSphere Application Server v7.0** or **WebSphere Application Server v8.0** as the server type. Click **Next**.
3. If this runtime has not been created in your workspace, you will be prompted to select the installation directory for the server. Click **Next**.
4. Accept the default server port and name. For this tutorial the default server name used will be `server1`. Click **Finish**.
5. Wait for the server to start. once it has started the Console view will display `Server server1 open for e-business`. If the server does not start automatically select it in the Servers view and click the start icon:  .

Create a JAX-RS enabled web project

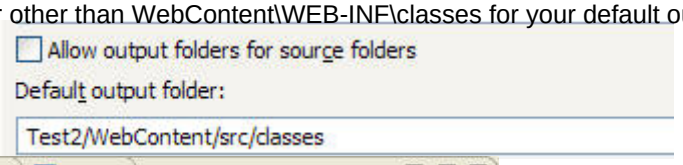
The JAX-RS web service needs to reside in a project with the JAX-RS facet enabled.

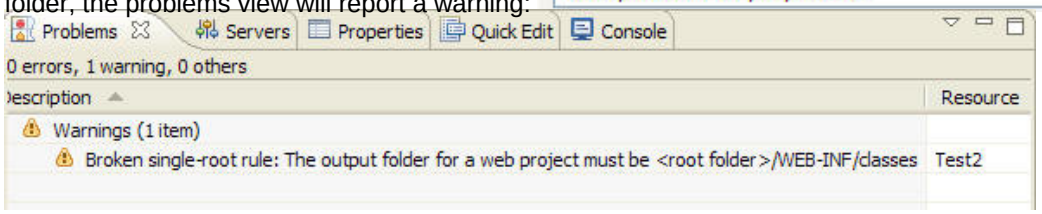
1. In the Java EE perspective, right-click your enterprise application project and select **New** > **Web Project** to open the web project wizard.
2. In the **Name** field, type a name for your new web project. For this tutorial, use `JAXRS`.
3. In the Project Templates section, select the type of web template you want to use: For this tutorial, select `Simple`.

Option	Description
Dojo Toolkit	Configures the project to have Dojo capabilities. The Dojo resources can be in the project itself, a separate project, or a remote location accessible via HTTP.
JavaServer Faces	Enables the project to be deployed with JSF capabilities. Configuration is provided for either JSP or Facelets.
REST Services	A project configured for REST Services based on JAX-RS
Simple	This creates a basic web project.

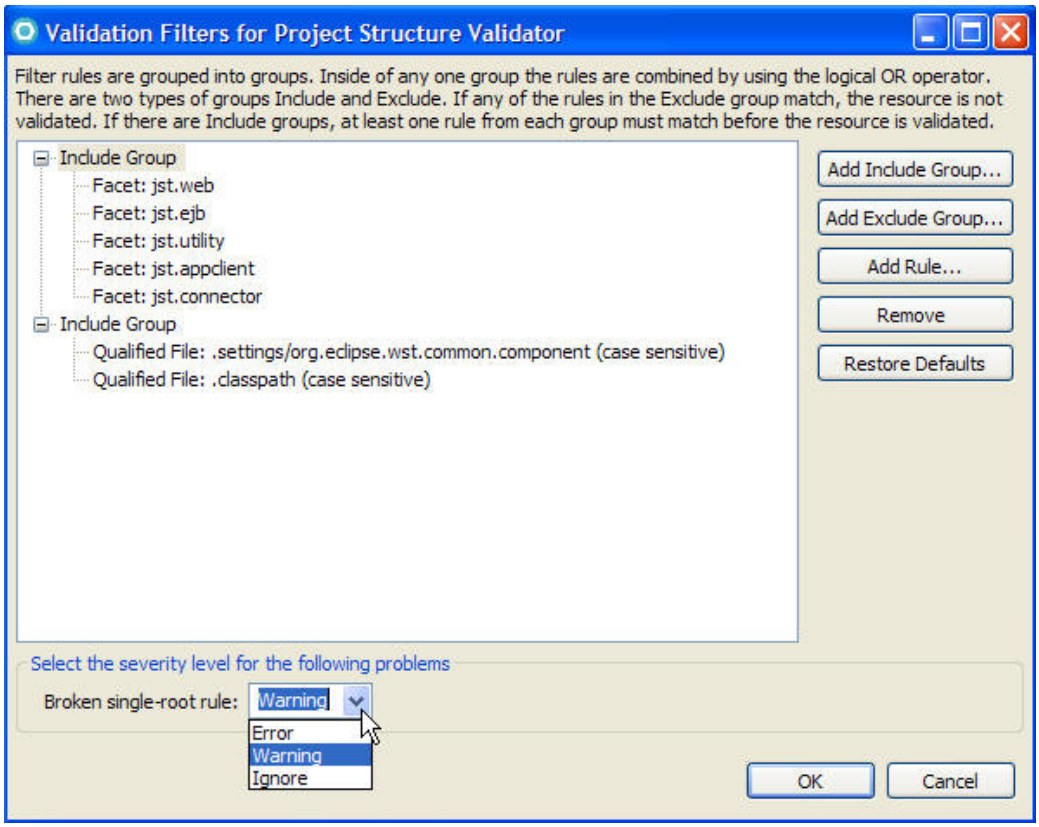
4. In the Programming Model section, select the programming model you want to use: For this tutorial, select `Java EE`.
   - Client-side only (HTML, JavaScript,...)
   - Java EE
   - OSGi
5. Click **Next** to configure your new web project.
6. On the deployment page, from the list of available configuration options, click **Deployment** to open the Deployment configuration page.
   - In the **Target runtime** field, select the v7 or v8 WebSphere Application Server that you installed earlier in the tutorial.
   - In the **Web module version** field, accept the default (which is automatically selected based on which WebSphere Application Server you selected).
   - In the **EAR Membership** field, select **Add project to an EAR**, and ensure that `JAXRSEAR` is the EAR project name.
   - Under the Deployment section, select **Change Features**. On the Project Facets page, select **JAX-RS (REST Web Services)**, version **1.1**, and click **OK**.

7. From the list of available configuration options, click **Java** to open the Java configuration page.
   - In the **Source folders on build path** field, accept the default **src** directory, or click **Add Folder**, **Edit...** or **Remove** to specify a folder for your source files.
   - In the **Default output folder:** field, specify a folder for your output files or accept the default value (WebContent\WEB-INF\classes). **Important:** If you choose a folder other than WebContent\WEB-INF\classes for your default output folder, the problems view will report a warning:





   The default for single rootedness problems is set to warning. To change this setting, select **Window** > **Preferences** > **Validation** > **Project Structure Validation**. Click the **...** settings field, and select
   A. **Error**
   B. **Warning**
   C. **Ignore**



8. From the list of available configuration options, click **REST Services** to open the REST Services configuration page. In the **JAX-RS Implementation Library** field, select **IBM WebSphere Application Server v<x> JAX-RS Library**. If you are using a version of WebSphere Application Server earlier than v8, check **Include library with this application** and select to include it as a **Shared Library**. Click **Udate Deployment Descriptor**. Ensure that the following values appear:
   - In the **JAX-RS servlet name:** field, ensure that **JAX-RS Servlet** appears.

- In the **JAX-RS servlet class name:** field, ensure that **com.ibm.websphere.jaxrs.server.IBMRestServlet** appears.
- In the **URL mapping patterns:** field, ensure that **/jaxrs/*** appears.

Learn more about libraries

9. From the list of available configuration options, click **Web Module**.. On the Web Module configuration page,
   - In the **Context root** field, type the name of your web project root, or accept the default (which is the name of your web project).
   - In the **Content directory** field, type the name of your content directory, or accept the default (WebContent).
   - Select **Generate web.xml deployment descriptor** if you want to create a deployment descriptor. You can also add a deployment descriptor to your web module later.You need to use a `web.xml` to configure security constraints and other behavior.
10. Click **Finish**.

The facet adds the library, servlet information, and support for JAX-RS annotations processing and JAX-RS quick-fixes.

Now you are ready to begin the next module: Creating and testing the web service.

< Previous | Next >

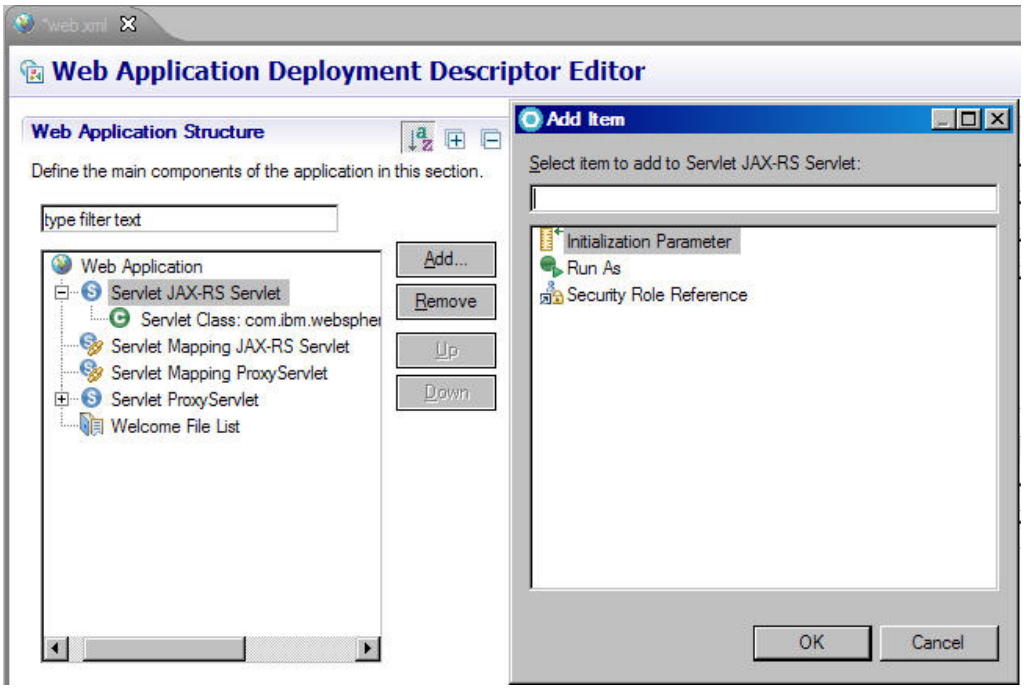
< Previous

Lesson 2: Creating and testing the web service

Once you have created a JAX-RS enabled project, you can import the Java files used by the application and test the web service.

Create a JAX-RS web service

1. Import the project which contains the Java classes needed for the application by clicking the following link:Import the Address Book tutorial resources
2. In your web project, create a package called `com.test` (right-click **Java Resources** > **src** and select **New** > **Package**). Import the following classes from the imported project into the package:
   - `AddressBook.java`
   - `AddressBookApplication.java`
3. Open `WebContent/WEB-INF/web.xml`. In the Design view, select the Servlet (JAX-RS Servlet) and click **Add** and add an Initialization parameter to the JAX-RS servlet, leaving the name and value fields empty. Save `web.xml` ignoring any errors that might be displayed.



4. In the Problems view, right-click the `param-name` warning and select Quick Fix. Select to browse for an existing sub-class, and select the AddressBookApplication.
5. Save `web.xml`.

Test the JAX-RS web service

1. In the Servers view, right-click your server and select **Add and Remove**, and add the JAX-RS EAR to the server. Restart the server.
2. To retrieve all addresses in the Address Book application, open a Web browser and enter the following URL: `http://localhost:<default_host_port>/<application_name>/jaxrs/addresses` For example, following the naming convention used in this tutorial and the default port, `http://localhost:9080/JAXRS/jaxrs/addresses`**Note:** You can determine the default host port name in the WebSphere Application Server Admin Console server configuration tab.

3. Enter the following URL: `http://localhost:<default_host_port>/<application_name>/jaxrs/addresses/<address index>` The `address index` is a number between 0 and 5 which represent the 6 addresses listed in `AddressBook.java`. The address assigned to that index value will display.

< Previous

Next >

Tutorial: Creating a secured JAX-WS web service from an WSDL file

This tutorial will walk you through the steps to create a JAX-WS web service and client, and to secure it using a policy set. It will generate code similar to that in the JAX-WS RSP address book sample.

Learning objectives

In this tutorial, you will learn to:

- Create a JAX-WS enabled server and Web project
- Create the address book web service and test its business logic using the Generic Service Client
- Create the address book web service client and test it using sample JSPs
- Secure the web service using the WebSphere Application Server

To complete this tutorial, you will need approximately 2 hours. If you decide to explore other facets of web services while working on the tutorial, it could take longer to finish.

Prerequisites

In order to complete this tutorial end to end, you should be familiar with:

- Web services concepts
- WebSphere Application Server concepts

When you are ready, begin Lesson 1: Creating a server and Web project.

Related information:

Sample: WebSphere JAX-WS address book RSP web service

Next >

Lesson 1: Creating a server and web project

In this lesson you will learn how to create a server and web project for use with web services.

Create a JAX-WS enabled server

Before creating the web service you need to ensure you have a WebSphere Application Server v8.0 server defined and started. By default a server is created for you when you install WebSphere Application Server. This server can be seen in the Servers view. However if you want to create a new server do the following:

1. From the **File** menu, select **New** > **Other** > **Server** > **Server** > **Next**.
2. Select **WebSphere Application Server v8.0** as the server type. Click **Next**.
3. If this runtime has not been created in your workspace, you will be prompted to select the installation directory for the server. Click **Next**.
4. Accept the default server port and name. For this tutorial the default server name used will be `server1`. Click **Finish**.
5. Wait for the server to start. once it has started the Console view will display `Server server1 open for e-business`. If the server does not start automatically select it in the Servers view and click the start icon:  .

Create a web project for the web service

The web services wizards can create a web project for the web service and enable the facets for you, however in this tutorial you will create the project manually.

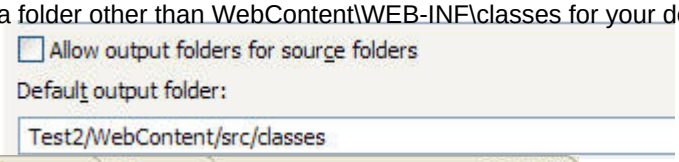
1. In the Java™ EE perspective, right-click your enterprise application project and select **New** > **Web Project** to open the web project wizard.
2. In the **Name** field, type a name for your new web project. For this tutorial, use `jwsAddressBook`.
3. In the Project Templates section, select the type of web template you want to use: For this tutorial, select `Simple`.

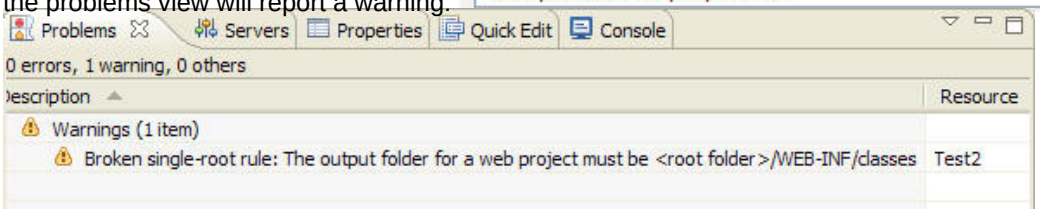
Option	Description
Dojo Toolkit	Configures the project to have Dojo capabilities. The Dojo resources can be in the project itself, a separate project, or a remote location accessible via HTTP.
JavaServer Faces	Enables the project to be deployed with JSF capabilities. Configuration is provided for either JSP or Facelets.
REST Services	A project configured for REST Services based on JAX-RS
Simple	This creates a basic web project.

4. In the Programming Model section, select the programming model you want to use: For this tutorial, select `Java EE`.
   - Client-side only (HTML, JavaScript,...)
   - Java EE
   - OSGi
5. Click **Next** to configure your new web project.
6. On the deployment page, from the list of available configuration options, click **Deployment** to open the Deployment configuration page.
   - In the **Target runtime** field, select the v7 or v8 WebSphere Application Server that you installed earlier in the tutorial.
   - In the **Web module version** field, accept the default (which is automatically selected based on which WebSphere Application Server you selected).
   - In the **EAR Membership** field, select **Add project to an EAR**, and ensure that `jwsAddressBookEAR` is the EAR project name.

- Under the Deployment section, select **Change Features**. On the Project Facets page, select **JAX-RS (REST Web Services)**, version **1.1**, and click **OK**.

7. From the list of available configuration options, click **Java** to open the Java configuration page.
   - In the **Source folders on build path** field, accept the default **src** directory, or click **Add Folder**, **Edit...** or **Remove** to specify a folder for your source files.
   - In the **Default output folder:** field, specify a folder for your output files or accept the default value (WebContent\WEB-INF\classes). **Important:** If you choose a folder other than WebContent\WEB-INF\classes for your default output folder, the problems view will report a warning:



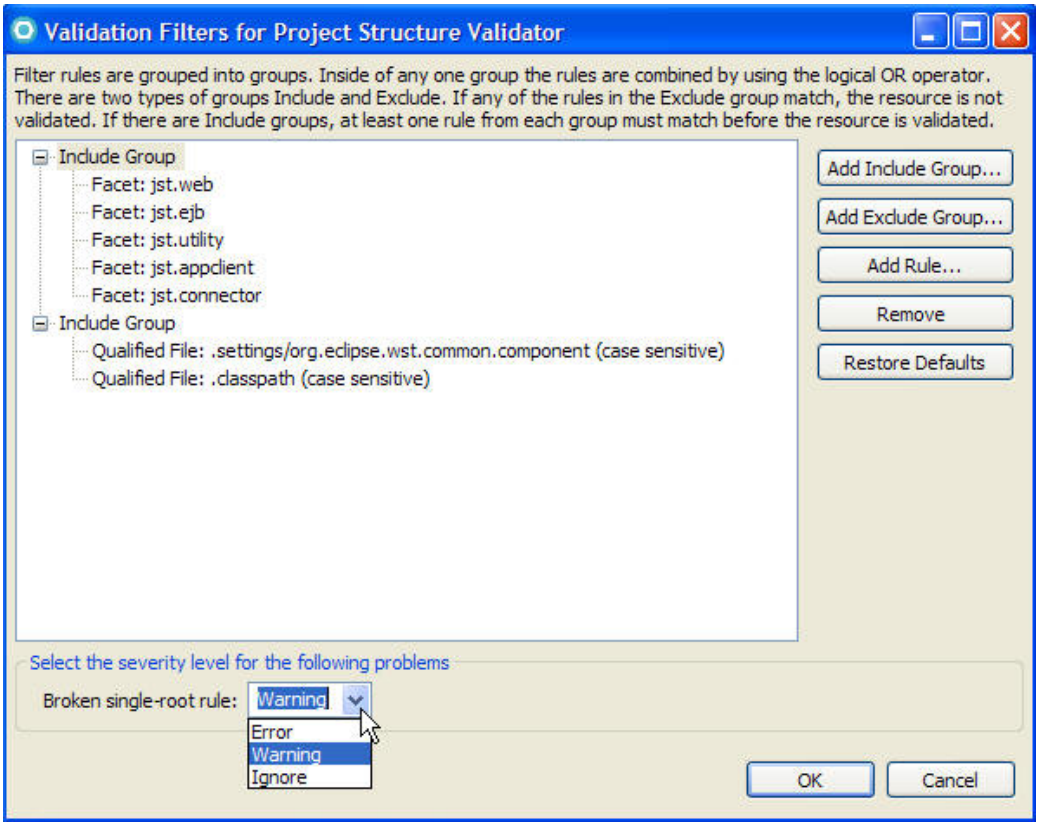


   The default for single rootedness problems is set to warning. To change this setting, select **Window** > **Preferences** > **Validation** > **Project Structure Validation**. Click the **...** settings field, and select

   A. **Error**

   B. **Warning**

   C. **Ignore**



8. From the list of available configuration options, click **Web Module**.. On the Web Module configuration page,
   - In the **Context root** field, type the name of your web project root, or accept the default (which is the name of your web project).
   - In the **Content directory** field, type the name of your content directory, or accept the default (WebContent).

- Select **Generate web.xml deployment descriptor** if you want to create a deployment descriptor. You can also add a deployment descriptor to your web module later.You need to use a `web.xml` to configure security constraints and other behavior.

9. Click **Finish**.

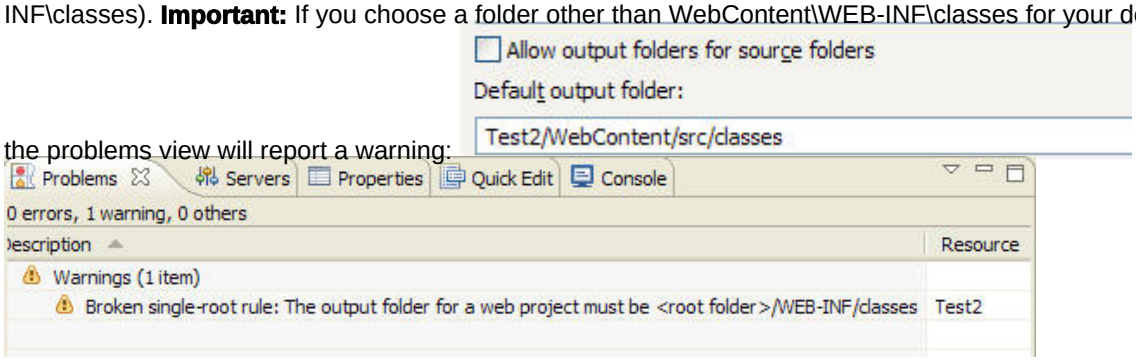
Create a web project for the web service client

The web services wizards can create a web project for the client and enable the facets for you, however in this tutorial you will create the project manually.

1. In the Java EE perspective, right-click your enterprise application project and select **New** > **Web Project** to open the web project wizard.
2. In the **Name** field, type a name for your new web project. For this tutorial, use `jwsAddressBookClient`.
3. In the Project Templates section, select the type of web template you want to use: For this tutorial, select `Simple`.

Option	Description
Dojo Toolkit	Configures the project to have Dojo capabilities. The Dojo resources can be in the project itself, a separate project, or a remote location accessible via HTTP.
JavaServer Faces	Enables the project to be deployed with JSF capabilities. Configuration is provided for either JSP or Facelets.
REST Services	A project configured for REST Services based on JAX-RS
Simple	This creates a basic web project.

4. In the Programming Model section, select the programming model you want to use: For this tutorial, select `Java EE`.
   - Client-side only (HTML, JavaScript,...)
   - Java EE
   - OSGi
5. Click **Next** to configure your new web project.
6. On the deployment page, from the list of available configuration options, click **Deployment** to open the Deployment configuration page.
   - In the **Target runtime** field, select the v7 or v8 WebSphere Application Server that you installed earlier in the tutorial.
   - In the **Web module version** field, accept the default (which is automatically selected based on which WebSphere Application Server you selected).
   - In the **EAR Membership** field, select **Add project to an EAR**, and ensure that `jwsAddressBookEAR` is the EAR project name.
   - Under the Deployment section, select **Change Features**. On the Project Facets page, select **JAX-RS (REST Web Services)**, version **1.1**, and click **OK**.
7. From the list of available configuration options, click **Java** to open the Java configuration page.
   - In the **Source folders on build path** field, accept the default **src** directory, or click **Add Folder**, **Edit...** or **Remove** to specify a folder for your source files.
   - In the **Default output folder:** field, specify a folder for your output files or accept the default value (WebContent\WEB-INF\classes). **Important:** If you choose a folder other than WebContent\WEB-INF\classes for your default output folder, the problems view will report a warning:

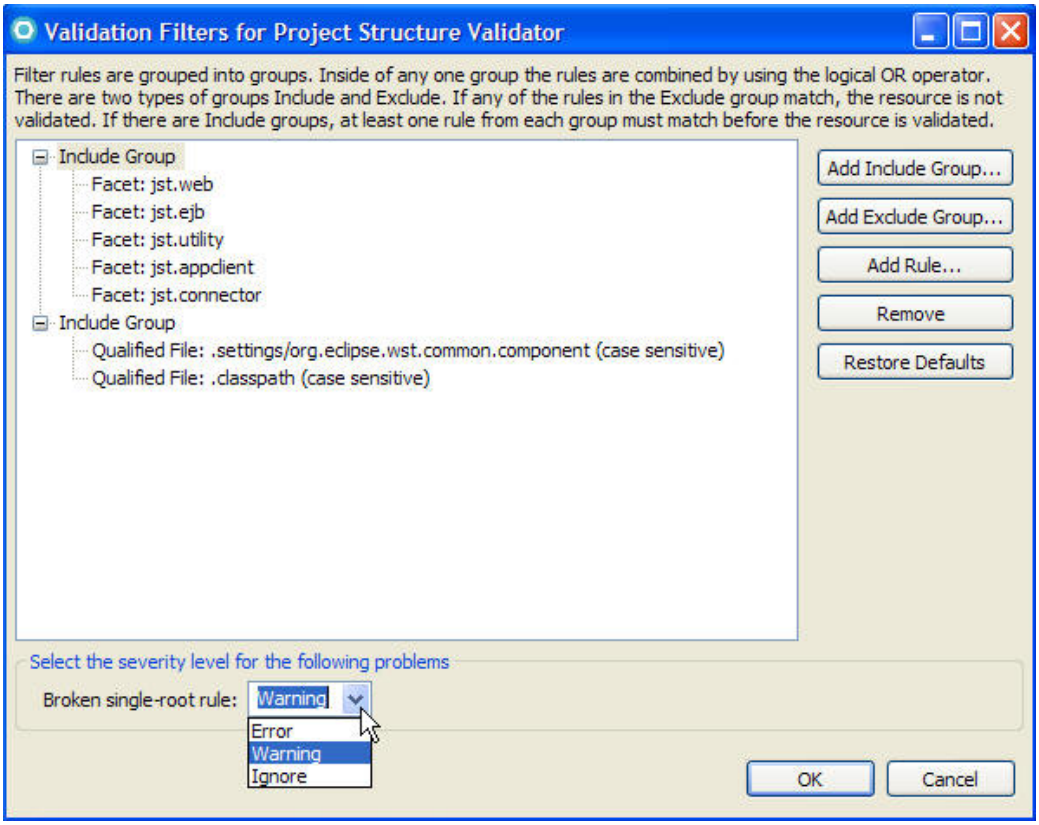


The default for single rootedness problems is set to warning. To change this setting, select **Window** > **Preferences** > **Validation** > **Project Structure Validation**. Click the **...** settings field, and select

A. **Error**

B. **Warning**

C. **Ignore**



8. From the list of available configuration options, click **Web Module**.. On the Web Module configuration page,
   - In the **Context root** field, type the name of your web project root, or accept the default (which is the name of your web project).
   - In the **Content directory** field, type the name of your content directory, or accept the default (WebContent).
   - Select **Generate web.xml deployment descriptor** if you want to create a deployment descriptor. You can also add a deployment descriptor to your web module later.You need to use a `web.xml` to configure security constraints and other behavior.
9. Click **Finish**.

Import the address book WSDL file

Import the required WSDL file. The import wizard will import a simple project containing the WSDL file.Import the Address Book tutorial resources

Lesson Checkpoint

Now you are ready to begin Lesson 2: Creating the web service.

Lesson 2: Creating the web service

In this lesson you will learn how to create the top-down Java address book web service from the WSDL file imported in the previous lesson.

Before you begin, you must complete Lesson 1: Creating a server and web project.

1. Switch to the Java EE perspective: **Window > Open Perspective >Other > Java EE**.
2. In the Enterprise Explorer view, select the WSDL file you imported in the previous lesson.
3. Right-click the WSDL and select **Web Services > Generate Java bean skeleton**.
4. Web Services page: select **Top down Java™ bean Web service** as your web service type and ensure the `AddressBook.wsdl` is set as the service definition. Choose the following options:
   A. Select the stages of web services development that you want to complete using the slider. The slider sets the defaults on the remaining wizard pages, but you can override the default settings on each page as you proceed. For this tutorial select **Start service**. This will create all the code required for the web service, deploy it to the server and attempt to start the server.
   B. Select your server: click the server link and ensure that the WebSphere v8 Server is selected.
   C. Select your runtime: ensure the IBM® JAX-WS runtime is selected.
   D. Select the service project: Select the `jwsAddressBook` project created in the previous lesson.
   E. Select the service EAR project: Select the `jwsAddressBookEAR` created in the previous lesson.
   F. The client will be created later, so ensure the client slider is set to **No client**.
   Click **Next**.
5. WebSphere® JAX-WS Top Down Web Service Configuration page:
   - Output folder: Accept the default location where the generated Java skeleton will be generated: `jwsAddressBook/src`.
   - Target package: Accept the default package name
   - Enable wrapper style: Enables wrapper style mapping from WSDL to Java. For WSDL documents that implement a document/literal wrapped pattern, a root element is declared in the XML schema and is used as an operation wrapper for a message flow. Separate wrapper element definitions exist for both the request and the response. Accept the default.
   - Enable MTOM support: If you select this check box the SOAP Message Transmission Optimization Mechanism will be enabled to optimize the transmission of binary content. Accept the default.
   - Version of JAX-WS code to be generated: Starting with WebSphere Application Server v8.0, you can generate JAX-WS 2.1 compliant code. Select 2.1 for this tutorial.
   - Copy WSDL to project: Select this to copy the WSDL file into the service project. Since you will create the client at a later time select this check box.
   - Generate serializable JAXB classes: In WebSphere Application Server v7.0 and v8.0 when you enable the Java 6 facet, you can choose to generate JAXB classes which implement java.io.Serializable. Classes that do not implement this interface will not have any of their state serialized or deserialized. Do not select this for the tutorial.
   - Specify JAX-WS or JAXB binding files: this allows you to use JAX-WS or JAXB custom binding files. Do not select this for the tutorial.
   - Customize service implementation class name: this allows you to change the default port name to service implementation class name mapping. Do not select this for the tutorial.
   - Generate schema library: Selecting this will run the JAX-WS Schema to Java compiler to generate a schema. Do not select this option for this tutorial.
   - Generate Web service deployment descriptor: For JAX-WS web services deployment information is generated dynamically by the runtime; static deployment descriptors are no longer required. Selecting this checkbox will generate them. This tutorial does not require deployment descriptors.

Click **Next**.

6. Web Service Publication page: Use this page to publish your web service to a UDDI registry using the Web Services Explorer. This is unnecessary for this tutorial. Click **Finish**.

All the required code for the web service is generated

Adding the business logic to the skeleton bean

The skeleton implementation bean generated by the web service wizard `AddressBookPortImpl.java` does not contain any business logic. It does contain the annotation `@javax.jws.WebService` which tells the runtime that it is a JAX-WS web service.

In order to make the address book web service function as expected, you need to add code to this bean. It should be opened in an editor automatically after generating the web service, but if not it can be found in: `jwsAddressBook/Java Resources/src/com.addressbook`.

1. Replace the current saveAddress method:

```
public boolean saveAddress(PersonType person) {
        return false;
    }
```

with the following static field and method:

```
private static Hashtable<String,AddressType> addresses = new Hashtable<String,AddressType>();
    public boolean saveAddress(PersonType person) {
     addresses.put(person.getName(),person.getAddress());
        return true;
    }
```

2. Replace the current findAddress method:

```
public AddressType findAddress(String name) throws FindAddressFault {
        return null;
    }
```

with the following:

```
public AddressType findAddress(String name) throws FindAddressFault {
        return addresses.get(name);
    }
```

3. Several error markers may be displayed. To correct these, organize your imports by clicking `Ctrl+Shift+o`. Select to import `java.util.Hashtable`. Once this is done the errors should be grayed out.
4. Save the updated implementation bean.

Test the web service using the Generic Service Client

The Generic Service Client allows you to test a web service without building a client. You can select the operation you want to test, enter the required information, and the result will display in the Status pane.

1. Select the generated WSDL file `jwsAddressBook/WebContent/WEB-INF/wsdl/AddressBook.wsdl`, right-click and select **Web Services** > **Test with Generic Service Client**. Alternatively you can select the service under the project's Services node or the JAX-WS web services node in the Services view, and launch the Generic Service Client from there.
2. Select the `SaveAddress` operation.
3. Enter values in each of the fields and click **Invoke**.
4. Select the `FindAddress` operation.
5. Enter the name you chose when invoking the saveAddress operation and click **Invoke**.
6. The information you saved to this name will display in the Status pane.

Lesson Checkpoint

Now you are ready to begin Lesson 3: Creating the web service client.

< Previous | Next >

Lesson 3: Creating the web service client

In this lesson you will learn how to create a client for the web service.

Before you begin, you must complete Lesson 2: Creating the web service.

Web service clients are created from a WSDL document which describes where the web service is deployed and what operations this service provides. You can use either the static WSDL file generated by the web service wizard, or when creating JAX-WS web services you can use the dynamic WSDL file generated by the runtime based on information it gathers from the annotations added to the Java classes. For this tutorial the dynamic WSDL file will be used.

View the dynamically generated WSDL

If your server is started you should be able to quickly test your annotated bean to ensure it is a web service and generate the dynamic WSDL file.

1. Determine the port your web service is using if you do not already know it. To do this:
   A. Launch the WebSphere Application Server Administrative Console by right-clicking your server in the Servers view and selecting **Administration** > **Run administrative console**.
   B. Expand **Servers** > **Server Types** in the left pane, and select **WebSphere application servers**.
   C. Select your server name from the list. By default this is **server1**.
   D. On the Configuration tab, search for the Communications heading and expand **Ports**.
   E. The port used is WC_defaulthost. This tutorial will use the example port number 9081. Replace this number with your own port in any following steps.
2. Launch the Web browser by clicking this icon: 
3. Enter the following URL in the browser: `http://localhost:`*`9081`*`/jwsAddressBook/AddressBookService` and hit return. A page displays stating that this is a web service.
4. Add `?wsdl` to the end of the above URL and hit return again: `http://localhost:`*`9081`*`/jwsAddressBook/AddressBookService?wsdl`. The dynamically generated WSDL file will be displayed. This file can be used the same way as you would use the static WSDL file that the web service wizard generated as long as the server is running.

Create the address book web service client and test JSP

1. Click **File > New > Other**. Select **Web Services** in order to display the various web service wizards. Select the **Web Service Client** wizard. Click **Next**.
2. Web Services page: Enter the URL of the dynamically generated WSDL file that you discovered in the previous section in the **Service definition** field: `http://localhost:`*`9081`*`/jwsAddressBook/AddressBookService?wsdl`.
   A. Select the stages of web service client development that you want to complete using the slider. For this tutorial select **Test client**. This will generate all the required client code and provide various testing options for the client.
   B. Server: Ensure the WebSphere v8.0 server is selected.
   C. Web service runtime: Ensure the JAX-WS runtime is selected. Although you could create a JAX-RPC client to access the address book web service because it does not use any JAX-WS specific options such as SOAP 1.2, for this tutorial we will be exploiting the asynchronous capabilities of JAX-WS so a JAX-WS client is necessary.
   D. Client project: Select the client project created in lesson 1: `jwsAddressBookClient`.
   E. Client EAR project: Select the `jwsAddressBookEAR`. For JAX-WS web services, the server and client projects can share the same EAR.
   F. Monitor the web service: This will send the web service traffic through the TCP/IP Monitor, which allows you to watch the SOAP traffic generated by the web service and to test this traffic for WS-I compliance. The TCP/IP Monitor is not

capable of monitoring web services secured with the RSP policy set, so do not select this option..

Click **Next**.

3. WebSphere® JAX-WS Web Service Client Configuration page:
   - Output folder: Accept the default folder where the client's Java™ classes will be generated.
   - Target package: The web services client wizard generates a number of Java files from the specified WSDL. By default it will create a package name based on the namespace specified in the WSDL file. To override this default behavior you can specify your own package name for the namespace in the WSDL file. Accept the defaults for this tutorial.
   - Generate portable client: Selecting this checkbox would allow you to move your web service client code from one machine to another or from one instance of WebSphere Application Server to another. It is not required by this tutorial.
   - Enable asynchronous invocation for generated client: If you select to enable an asynchronous client, for each method in the web service two additional methods will be created. These are polling and callback methods which allow the client to function asynchronously. Select this option.
   - Specify JAX-WS or JAXB binding files: If you have created JAX-WS or JAXB custom binding files, select this check box to use them to create this web service. Do not select this for the tutorial.
   - Customize client proxy class name: You can accept the default proxy name or enter your own. Do not select this for the tutorial.
   - Generate schema library: Selecting this will run the JAX-WS Schema to Java compiler to generate a schema. Do not select this option for this tutorial.
   - Generate web service deployment descriptor: For JAX-WS web services deployment information is generated dynamically by the runtime; static deployment descriptors are no longer required. Selecting this checkbox will generate them. This tutorial does not require deployment descriptors.
   - Version of JAX-WS code to be generated: Starting with WebSphere Application Server v8.0, you can generate JAX-WS 2.1 compliant code. Select 2.1 for this tutorial.

   Click **Next**.
4. Web Service Client Test page:
   - Test the generated proxy: Select this to test the functionality of the client.
   - Select your test facility. You can test the generated proxy in the Universal Test Client or the Web Service Explorer, or you can generate sample JAX-WS 2.0 JSPs. For this tutorial, select the JAX-WS JSPs.
   - Folder: You can select the folder where the JSP will be located. Accept the default.
   - Methods: Select the methods to expose. The asynchronous methods should be listed as well. Ensure all methods are selected.
   - Run test on server: this will start the server for you automatically.

   Click **Finish**. The sample JSP will open in a browser window.

Test the web service client using sample JSPs

The following steps will test the web service synchronously.

1. The TestClient.jsp should be launched automatically if you selected the options selected above. If it is not displayed, select `jwsAddressBookClient/WebContent/sampleAddressBookPortProxy/TestClient.jsp`, right-click and select **Run As** > **Run on Server**
2. To test the service synchronously, select the saveAddress method and enter information in the name field. All other fields are optional. Click **Invoke**.
3. Select the findAddress method, enter the name you used during the saveAddress method, and click **Invoke**. The information saved by the saveAddress method should display in the results pane.
4. To test the service asynchronously, in the Quality of Service pane select the **Enable asynchronous invocation** checkbox.
5. Select the findAddress method, enter the name you used during the saveAddress method, and click **Invoke**.

6. A new link will display indicating that the method is in progress. Click the link to display the method response in the Results pane.

Once you have tested the web service leave the JSP open so that you can test the service again once the web service and client have been secured.

Lesson Checkpoint

Now you are ready to begin Lesson 4: Secure the web service with the RSP policy set.

< Previous | Next >

Lesson 4: Attach the WS-I RSP policy set to the web service

In this lesson you will learn how to secure the web service with one of the default policy sets packaged with WebSphere Application Server.

Before you begin, you must have completed the steps in Lesson 3: Creating the web service client.

Alternately, you can import the WebSphere JAX-WS address book web service sample which has nearly identical content to that created in the preceding lessons.

Sample: WebSphere JAX-WS address book web service

You can use the policy sets that are included with this product to simplify configuring the qualities of service for your web services and clients. Several policy sets are included in the workbench. Alternately you can use the administrative console to create your own policy sets and import them.

In this tutorial you will attach the Reliable Secure Profile (RSP) default policy set to the web service and client. The WS-I RSP default policy set consists of instances of the WS-Security, WS-Addressing and WS-ReliableMessaging policy types. This policy set provides the following features:

- Reliable message delivery to the intended receiver by enabling WS-ReliableMessaging
- Message integrity by digital signature that includes signing the body, timestamp, WS-Addressing headers and WS-ReliableMessaging headers using the WS-SecureConversation and WS-Security specifications
- Confidentiality by encryption that includes encrypting the body, signature and signature confirmation elements, using the WS-SecureConversation and WS-Security specifications

1. In the Java EE perspective Services view expand the JAX-WS Web Services node. The address book Web service and client should be under their respective folders.
2. Select the address book service, right-click and select **Manage Policy Set Attachment**.
3. Select the jwsAddressBookEAR as the service EAR project and click **Add**.
4. You can apply a policy set at the service, port or operation level. Different policy sets may be applied to various endpoints and operations within a single web service. However the service and client must have the same policy set settings. For this tutorial you will apply the policy set to the entire service, so the Endpoint and Operation Name fields can be left blank.
5. From the Policy Set drop-down list, select **WS-I RSP**, and for the Binding ensure **Provider Sample** is selected. This is a provider-side general binding packaged with WebSphere Application Server. Click **OK**. The service should now be listed in the Application table and the WS-I RSP policy. Click **Finish**.

Once a policy set has been attached to a web service a policyAttachements.xml file is generated in the EAR META-INF folder. This file will be appended for each additional policy set setting added to any service within the EAR.

Lesson Checkpoint

Now you are ready to begin Lesson 5: Secure the web service client with the RSP policy set.

Lesson 5: Secure the web service client with the WS-I RSP policy set

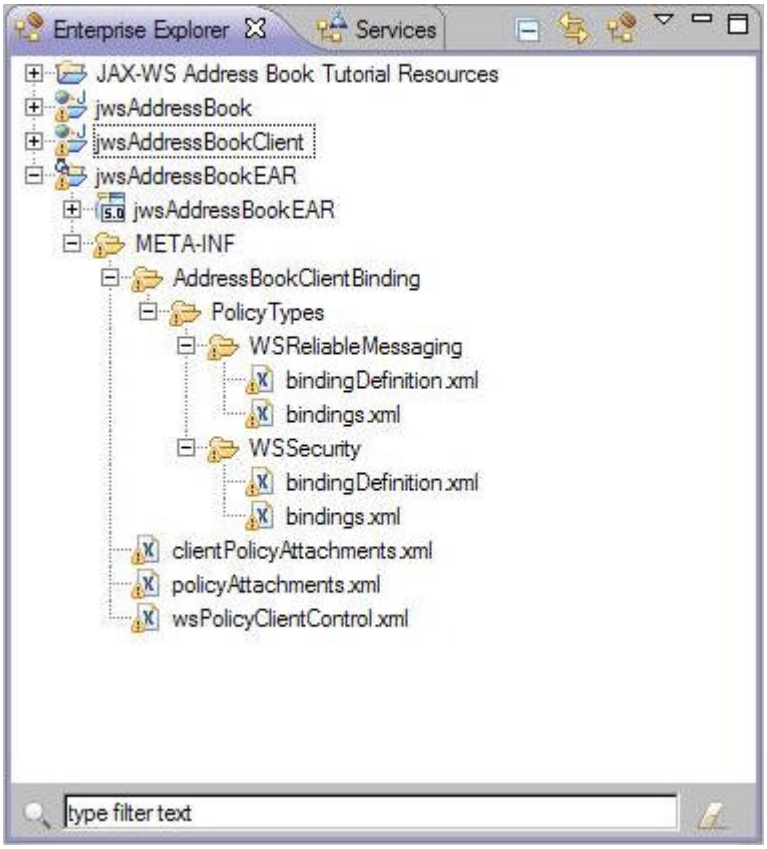
You can add security to a web service client by attaching policy sets to the client. Each attachment specifies an endpoint, a policy set, and a binding. Because each configuration is specific to an application and a user, you must configure a binding for some policy types.

Before you begin, you must have completed the steps in Lesson 4: Attach the RSP policy set to the web service.

For a given web service and a client of that service, the policy sets and bindings configuration must match for the service to function correctly.

1. In the Java EE perspective Services view expand the JAX-WS web services node. Select the address book client, right-click and select **Manage Policy Set Attachment**.
2. Ensure that the jwsAddressBookEAR is selected, and click **Next**.
3. In the Application section, click **Add** to attach a policy set to the endpoint and specify the bindings.
   A. Since the service is secured at the service level rather than the endpoint or operation level, the client will be secured at this level as well. Select the AddressBookService from the Service Name drop-down list and leave the Endpoint and Operation Name fields empty.
   B. In the Policy Set field, select WS-I RSP from the list.
   C. In the Binding field ensure **Client Sample** is selected. This is a client-side general binding packaged with WebSphere Application Server.
   D. Click **OK**.
4. The policy types contained by the policy set you selected are listed in the Bindings Configuration table. The configuration for these policy types are already complete.
5. Click **Finish** to complete the wizard.

`clientPolicyAttachments.xml` is created in the `META-INF` folder of the jwsAddressBookEAR, as well as the client side bindings.



Testing the secured web service using the TestClient.jsp

When you ran the TestClient.jsp earlier the web service was not secure. The SOAP traffic contained the information being sent to and from the web service in clear (unencrypted) text.

1. Select the SaveAddress method, enter information in each of the fields and click **Invoke**.
2. Select the Find Address method, enter the name used in the previous step and click **Invoke**. The web service should function in the exact way that it did before it was secured with the information entered in step 1 being displayed in the Results pane.

Lesson Checkpoint

Finish your tutorial by reviewing the materials in the Summary

< Previous | Next >

< Previous

Summary

You have just completed development, securing, and testing of a web service that uses JAX-WS as its runtime environment.

In this tutorial you:

- Created a server and web projects that supported the JAX-WS runtime.
- Used the web service wizard to generate a web service from an imported WSDL document.
- Used the WSDL file dynamically generated by the JAX-WS runtime to create a web service client.
- Tested the web service using the TestClient.jsp test facility using synchronous and asynchronous invocation.
- Attached the RAMP default policy set to the web service.
- Attached and configured the RAMP default policy set to the web service client.
- Tested the secured web service and examined the impact of the policy sets on its function.

When you have completed the tutorial and no longer require the Web projects for testing, remove the EAR from the server, and delete the projects from your workspace.

Additional resources

For more information about web services, please consult the online help (**Help** > **Help Contents**).

< Previous

Next >

Tutorial: Creating web services and an EJB skeleton from a WSDL file

Learning objectives

You will learn how to create a WS-I compliant JAX-WS web service from a WSDL file. The wizard generates an EJB skeleton that contains a set of methods corresponding to the operations described in the WSDL document. When the EJB is created, each method has a trivial implementation that can be easily replaced by editing the EJB.

Time required

To complete this tutorial, you will need approximately **1 hour and 30 minutes**. If you decide to explore other facets of web services or EJBs while working on the tutorial, it could take longer to finish.

Prerequisites

In order to complete this tutorial end to end, you should be familiar with:

- Basic web services concepts, such as SOAP and WSDL
- Basic XML
- EJB programming

When you are ready, begin Lesson 1.1: Set up the workspace and create the required projects

Next >

Lesson 1.1: Set up the workspace and create the required projects

Create a WebSphere Application Server v7.0 or v8.0 server

To create a WebSphere Application Server do the following:

1. From the **File** menu, select **New** > **Other** > **Server** > **Server** > **Next**.
2. Select **WebSphere Application Server v7.0** or **WebSphere Application Server v8.0** as the server type. Click **Next**.
3. If this runtime has not been created in your workspace, you will be prompted to select the installation directory for the server. Click **Next**.
4. Accept the default server port and name. For this tutorial the default server name used will be `server1`. Click **Finish**.
5. Wait for the server to start. once it has started the Console view will display `Server server1 open for e-business`. If the server does not start automatically select it in the Servers view and click the start icon:  .

Setting the WS-I compliance level

WS-I refers to web service interoperability; this includes interoperability across platforms, operating systems, and programming languages.

The WS-I organization sets out standards collected in documents called Profiles that define the requirements needed to make a web service interoperable. The Rational Developer products validate web services against the WS-I Simple SOAP Binding Profile 1.0 (WS-I SSBP) and the WS-I Attachments Profile 1.0 (WS-I AP). For more information on WS-I, refer to their Web site: http://www.ws-i.org/

By default, the WS-I SSBP compliance level is set to **Ignore**. With this setting, no warning will be given if non-compliant choices are made. This compliance level is used by the web service wizards and the WSDL validation tool. This sample generates a WS-I compliant web service, therefore you should set the WS-I compliance level to **Require**.

You can change the WS-I compliance level by following the proceeding steps:

1. On the main menu bar, click **Window** > **Preferences**. The Preferences dialog box opens.
2. Expand the **General** > **Service Policies** branch and expand **Profile Compliance** > **WS-I BP 1.1 + SSBP 1.0**, and select the **Require compliance** option from the drop-down list
3. Click **OK**.

Creating the web service EJB project

The remaining steps in this tutorial will be done in the Java EE perspective. If you are asked if you want to change to another perspective after performing a task, select **No**.

The EJB project will contain the business logic for the web service as well as the WSDL file.

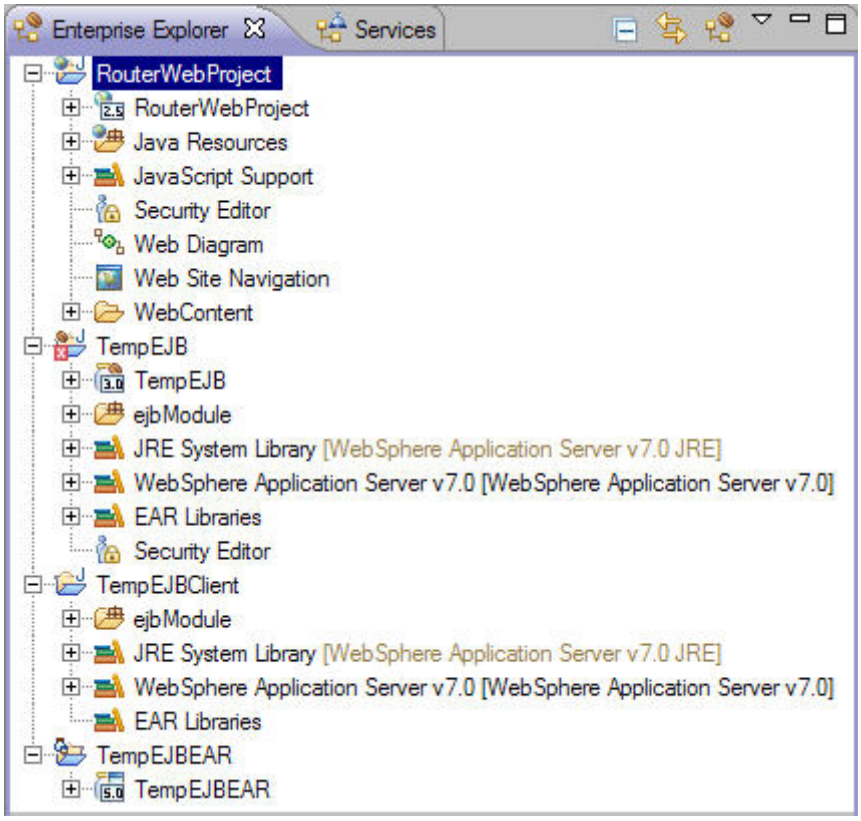
1. On the main menu bar, click **File** > **New** > **Project** > **EJB** > **EJB Project**. Click **Next**.
2. Type `TempEJB` in the Project name text field. Under Target Runtime ensure that the target server is WebSphere Application Server v7.0 or v8.0. In the **EAR Project Name** field, enter `TempEJBEAR` as the EAR name. Click **Next**.
3. Clear the checkbox for creating a client JAR module. The web services wizard will create this module for you. Click **Finish**.

The EJB project that will contain the web service logic and the associated EAR are created. The EJB project will have an error associated with it because it does not contain an enterprise bean. The bean will be generated by the web services wizard.

Creating the web service router project

EJB web services require a router project. This project contains the router servlet that acts as the endpoint for the service and will call out to the EJB. If you are using SOAP over JMS as your transport method the router project needs to be an EJB project. If you are using SOAP over HTTP as we are in this tutorial, the router project should be a Web project. The project you create must be added to the same EAR as the EJB project that will contain the enterprise bean. This project should not contain any of the business logic for your web service.
You can create a Web project by following these steps:

1. On the main menu bar, click **File** > **New** > **Project** > **Web** > **web Project**. Click **Next**.
2. Type `RouterWebProject` in the Name text field. Under Target Runtime ensure that the target server is WebSphere Application Server v7.0 or v8.0. In the **EAR Project Name** field, ensure `TempEJBEAR` is selected. This will ensure that the enterprise bean that you will create later and your router project are both referenced in the same EAR. Click **Finish**.
3. You have now created your router project and your workspace should look similar to the following:



Adding the projects to the server

You can associate the project with the server that your web service will run on by following these steps:

1. Right-click the server in the Servers view and select **Add and Remove**. If the Servers view is not open in your workspace, open it from the **Window** menu by selecting **Show View** > **Servers**.
2. In the window that opens, select TempEJBEAR which contains your router and EJB projects, and click **Add**.
3. Click **Finish**.

Lesson checkpoint

Now you are ready to begin Lesson 1.2: Import and validate the WSDL file.

< Previous | Next >

Lesson 1.2: Import and validate the WSDL file

The Temperature Conversion WSDL document has been provided for you. The WSDL file that you will use in this tutorial converts temperature from Fahrenheit to Celsius and Celsius to Fahrenheit.

Before you begin, you must complete Lesson 1.1: Set up the workspace and create the required projects.

You can create a new WSDL document or import an existing one. The Temperature Conversion WSDL document used in this tutorial has been provided for you in a simple project. Complete the following steps to import the Temperature Conversion WSDL document into the workbench:

1. In the Enterprise Explorer view, expand **TempEJB**, right-click and click **New** > **Folder** to create a folder called `wsdl`. Click **Finish**.
2. Import the simple project containing the WSDL file
3. Expand the imported simple project called TempConversionWSDL. The ConvertTemperature.wsdl file will be located here.
4. Right-click the ConvertTemperature.wsdl file and select **Move**.
5. Select the `wsdl` folder you created in the TempEJB project as your destination and click **OK**.

Validating the WSDL document

The WSDL Validator can validate WSDL semantics and WS-I compliance.

You can validate the Temperature Converter WSDL document by following the proceeding steps:

1. Select the ConvertTemperature.wsdl document in the Project Navigator.
2. Right-click, and click **Validate**. Any errors will be displayed in the Tasks view.

If no errors occur during validation, you can proceed to create the web service.

Lesson checkpoint

Now you are ready to begin Lesson 1.3: Create the web service.

Lesson 1.3: Create the web service

Before you begin, you must complete Lesson 1.2: Import and validate the WSDL file.
Before you attempt to create a web service it is strongly suggested that you start the WebSphere Application Server on which the web service will run. Although you can start the server in the web service wizards, since it may take several minutes to start depending on the speed of your machine, starting the server before you begin will both increase the speed with which you complete the wizard and reduce the chance that the wizard will generate an error because the server is taking too long to start.
To start the server, select the server in the Servers view and select **Start**: 

If the Servers view is not open in your workspace, open it from the **Window** menu by selecting **Show View** > **Servers**.

Create a web service from a WSDL file

The web service wizard assists you in creating a new web service, configuring it for deployment, and deploying the web service to a server. Once your web service is deployed, the wizard assists you in generating the client proxy and sample application to test the web service.
When you have completed testing, you can publish your web service to a UDDI Business Registry using the Export wizard.

1. In the Project Explorer, select the **ConvertTemperature.wsdl** document in your EJB project.
2. Click **File** > **New** > **Other**. Select **Web Services** in order to display the various web service wizards. Select the **Web Service** wizard. Click **Next**.
3. Select the following options on the first page of the wizard:
   - Web service type: Top down EJB Web service
   - Service definition: ensure the ConvertTemperature.wsdl file that you imported is selected.
   - Level of service generation slider: move the slider to Test service. The slider sets the defaults on the remaining wizard pages, but you can override the default settings on each page as you proceed.
   - Service configuration: ensure that WebSphere v7.0 or v8.0 Server and the IBM WebSphere JAX-WS runtime environment are selected. Click **Service project** and enter `TempEJB` as your service project name. TempEJBEAR should be selected as your service EAR project.
   - Level of client generation slider: move the slider to Test client.
   - Client configuration: ensure that WebSphere v7.0 or v8.0 Server and the IBM WebSphere JAX-WS runtime are selected. The wizard will create a client and client EAR project. You can accept the default names or enter a different name.
   - Monitor the web service.

   Click **Next.**
4. On the Web Service Configuration page, leave all the default options selected, and click **Next**.
5. On the Router project configuration page, select `RouterWebProject` as your http router project if it is not already selected, and click **Next**.
6. In the Web Service Test page, you can select a test facility to test your web service before a client or proxy is developed. Select Web Services Explorer as the test facility for your web service and click **Launch**. This step may take several seconds for the WebSphere Application server to start.
7. The Web Services Explorer is displayed in a Web browser. Select **fahrenheitToCelsius** or **celsiusToFahrenheit** from the operations list. Enter a number in the value field and click **Go**. A trivial implementation of each of these operations is provided, and a default value of -3 is returned. If both operations complete successfully, close the browser window and click **Next** in the web services wizard.

8. In the Web Service Client Configuration page, keep the default selections. Click **Next**.
9. In the Web Service Client Test page, ensure **Test the generated proxy** and **Run test on server** are both selected. In the Methods section. ensure that all methods are selected, or click **Select All** to select all methods. If you want to publish your web service to a UDDI Registry, click **Next** to configure the Web Service Publication options. However this step will not be covered in this tutorial. Otherwise, click **Finish**.
10. The sample application is launched in a Web browser. You can use this application to test the web service by selecting a method in the Methods frame, entering an input value in the Inputs frame, and clicking **Invoke** to view the result in the Result frame. Do not close the TestClient.jsp browser window yet - it will be used to test the web service traffic for WS-I compliance later in this tutorial.

Lesson checkpoint

Now you are ready to begin Lesson 1.4: Implement the temperature conversion methods.

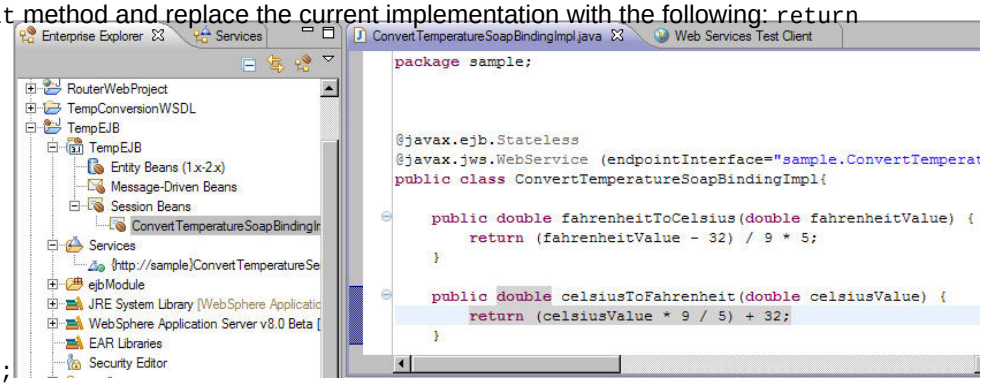
< Previous | Next >

Lesson 1.4: Implement the temperature conversion methods

Before you begin, you must complete Lesson 1.3: Create the web service.

Trivial implementations of the **fahrenheitToCelsius** and **celsiusToFahrenheit** methods were automatically generated when you created a web service from your WSDL document. In this section you will replace these trivial implementations with more meaningful code, and perform the necessary steps to test your new methods.

1. In the Enterprise Explorer view, select **ConvertTemperatureSoapBindingImpl.java** under **TempEJB** > **Session Beans** .
2. Locate the `fahrenheitToCelsius` method and replace the current implementation with the following: `return (fahrenheitValue - 32) / 9 * 5;`
3. Locate the `celsiusToFahrenheit` method and replace the current implementation with the following: `return (celsiusValue * 9 / 5) + 32;`



4. Save your updates by clicking **File** > **Save**.
5. Restart the EAR by expanding your server in the Servers view and right-clicking **TempEJBEAR** > **Restart TempEJBEAR**.
6. In the Web Services Test Client, test your **fahrenheitToCelsius** and **celsiusToFahrenheit** methods.

Lesson checkpoint

Now you are ready to begin Lesson 1.5: Validate the web service traffic WS-I compliance.

Lesson 1.5: Validate the web service traffic WS-I compliance

Before you begin, you must complete Lesson 1.4: Implement the temperature conversion methods.
To ensure that the SOAP envelope request and response pairs are WS-I compliant, you need to direct your web service traffic through the TCP/IP Monitor:
When creating a web service using the web service or web service client wizards, you can select to set up and run the TCP/IP Monitor automatically. Since you chose this option when creating the web service, the TCP/IP monitor view should be in your workspace. If it is not, you can open this view by selecting **Window** > **Show View** > **Other** > **Debug** > **TCP/IP Monitor**.
Alternately, you can set up the TCP/IP Monitor manually by completing the following steps:

1. In the sample application, invoke the getEndPoint method. Record this endpoint.
2. Create a server to act as the TCP/IP Monitor:
   A. From the **Window** menu, select **Preferences**.
   B. In the Preferences window, expand **Run/Debug** and then select **TCP/IP Monitor**.
   C. Select the **Show TCP/IP Monitor View when there is activity** check box.
   D. Under the TCP/IP Monitors lists, click **Add**. A New Monitor dialog box opens.
   E. Specify the following settings:

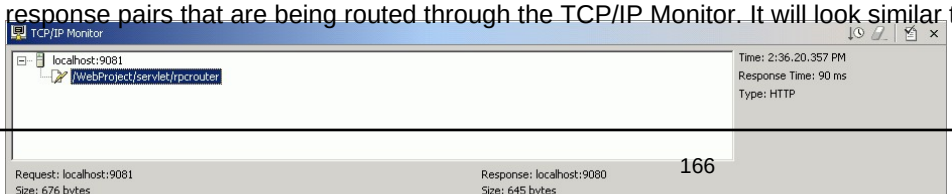
Option	Description
Local monitoring port	Specify a unique port number on your local machine.
Host name	Specify the host name or IP address of the machine where the server is running.
Port	Specify the port number of the remote server.
Type	Specify whether the request type from the Web browser are sent by HTTP or TCP/IP. If the HTTP option is selected the requests from the Web browser are modified so that the HTTP header points to the remote machine and separated if multiple HTTP requests are received in the same connection. If the TCP/IP option is selected, all the requests are sent byte for byte.

3. In order to route the web service through the monitor, the endpoint of the web service client needs to be changed. The TCP/IP Monitor listens on port 9081. In the Web browser window used in step 1, invoke the setEndPoint method, and change the endpoint so that it directs to port 9081. For example, the default would be: `http://localhost:9081/web_module_context_root/servlet/rpcrouter` Invoke the getEndPoint method again to ensure that your change has been implemented.

Routing traffic and verifying WS-I compliance

You can route traffic through the TCP/IP monitor and test the traffic for WS-I compliance by following the proceeding steps:

1. Select a web service method in the Methods pane. Invoke this method.
2. Change to the TCP/IP Monitor view by clicking the TCP/IP Monitor tab in the Servers view. This will display request and response pairs that are being routed through the TCP/IP Monitor. It will look similar to the following picture:



3. To ensure that your web service SOAP traffic is WS-I compliant, you can generate a log file by clicking the  icon. In the dialog box that opens, select a name for the log file and specify where you want it to be stored. This log file will be validated for WS-I compliance. You can open the log file in an XML editor to examine its contents.

Lesson checkpoint

Finish your tutorial by reviewing the materials in the Summary.

< Previous | Next >

< Previous

Summary

You have created a WS-I compliant Temperature Conversion Web service and an EJB skeleton from a WSDL file, and learned to implement some basic EJB methods.

Lessons learned

If you have completed all of the exercises, you should now be able to:

- Set the WS-I compliance level.
- Create a Web project called WebProject.
- Import the Temperature Conversion WSDL document.
- Validate the WSDL, and validate the WSDL file's WS-I compliance.
- Create the web service and skeleton EJB, and test the methods included in the web service using the Web Services Explorer.
- Implement the fahrenheitToCelsius and celsiusToFahrenheit methods (optional).
- Deploy the web service to the WebSphere Application Server.
- Test the web service traffic for WS-I compliance.

More information

For more information about web services, WSDL, SOAP, and the WebSphere runtime environments, consult the online help: Developing web services

For more in-depth technical articles on web services, consult DeveloperWorks

< Previous

Developing Java applications

The Java development tools (JDT) project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself. The Java visual editor is a tool which allows you to visually construct Graphical User Interfaces based on Swing, SWT or AWT. The Session Initiation Protocol (SIP) Tools Feature provides a development environment for the creation of new SIP-based services. The feature enhances the existing Java EE Platform, Enterprise Edition development perspective to allow for the creation of SIP and converged HTTP/SIP applications. The enhanced Java EE perspective also includes support for the Servlet ARchive (SAR) archive format and includes a wizard for editing SIP deployment descriptors. You can bundle SAR archive files within a Java EE application archive, just like other Java EE components.

Overview

You can read the following topics before creating a Java application, or before using the Java Visual Editor, or creating a SIP application. These topics provide planning and technology overview information that may be useful if you are new to Java applications or developing Java applications in this development environment.

- **About the visual editor for Java**
- **Java development overview**
- **SIP application overview**

Getting started

If you are already familiar with Java applications technology the following topics will help you set up your workspace for Java applications development, and guide you through the development process.

- **Preparing your workspace for Java development**
- **The visual editor Design and Source views**
- **Setting visual editor preferences**
- **Develop a SIP application**

Samples and tutorials

The following samples and tutorials are included with this product:

- **Sample: SWT Browser sample**
- **Sample: SWT Simple Text Editor**
- **Sample: Swing Slider Game**
- **Sample: Swing Simple Text Editor**
- **Tutorial: Basic Java Tutorial**
- **Tutorial: Project Configuration**

Resources for learning available on the Web

In addition to the information found in this infocenter, the following links provide additional learning material.

IBM® Redbooks®: Rational® Application Developer V7 Programming Guide

Rational Application Development certification prep, Part 2: Java development

Developing SIP and IP Multimedia Subsystem (IMS) Applications

Developing Java in the visual editor

This information describes how to use the visual editor for Java™ to develop and test visual Java classes and applications.

About this task

Choose one of the following topics for more information:

- About the visual editor for Java
- Designing Java classes with the visual editor
- Testing and debugging in the visual editor

About the visual editor for Java

The visual editor for Java™ is a code-centric Java editor that helps you design applications that have a graphical user interface (GUI). The visual editor is based on the JavaBeans component model and supports visual construction using the Standard Widget Toolkit (SWT), the Abstract Window Toolkit (AWT), or Swing.

The visual editor is designed to work with .java source files, letting you edit the source and work on the visual design simultaneously. The visual editor does not have its own perspective. If you use the visual editor in the Java perspective, the visual editor for Java uses any customization that you have made to the position of the Tasks view, the Console view, and the Outline view.

In order to use the visual editor for Java, ensure that you installed the appropriate optional components:

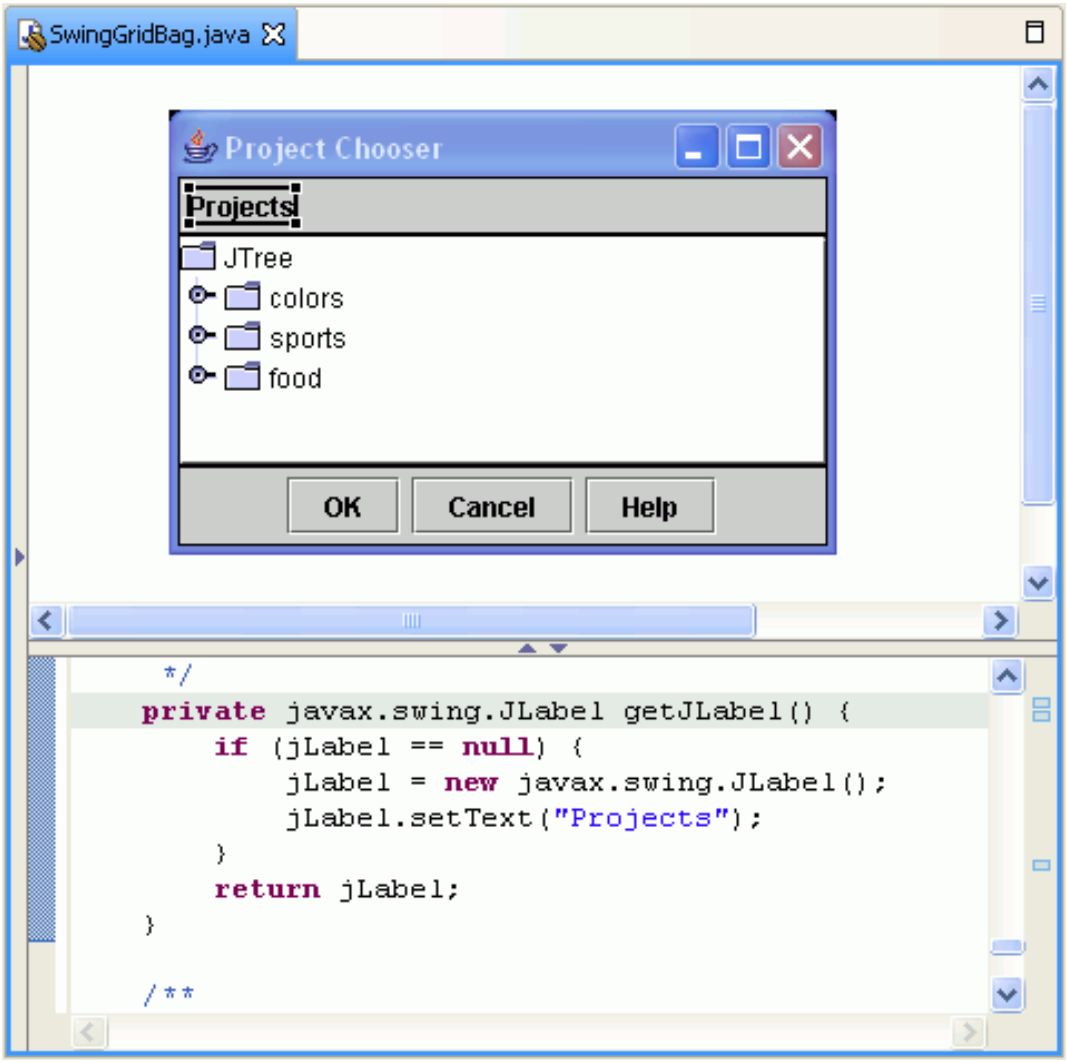
1. Open Installation Manager.
2. Click **Modify**, and click **Next**.
3. In the Modify Packages page, select your product, and click **Next**.
4. In the Features list, select **Java client application editor**, and click **Next**:

5. Click **Modify**.

You can use the visual editor for Java to create a visual class using a wizard, then design the class using visual components, or Java beans, from a design palette. This class can be an executable application (a class with a main method) or it can be a Java bean that you intend to include in another class. To be used as a Java bean by the visual editor for Java there is no interface that must be implemented and the only requirement is that the editor knows how to instantiate the class. For most purposes this means that the class should be public and have a public null constructor. The null constructor cannot be abstract, and it can be explicit or implicit, although there are a few exceptions that the visual editor for Java recognizes, for example java.awt.Dialog subclasses (which are instantiated using the java.awt.Frame argument constructor) or SWT controls. Certain other classes are recognized and can be dropped or sub-classed, such as Eclipse RCP views or editors. The visual editor adds an explicit constructor that calls the initialize method, which is used to set up the initial state of the Java bean.

The visual editor Design and Source views

The visual editor for Java™ is divided into two main windows: the Design view and the Source view. The Design view previews the Java beans in a graphical representation, while the Source view shows the associated Java code.
Visual beans that are SWT, Swing, or AWT components, or subclasses of these components, are shown in the Design view with their graphical representation. As you modify the Java beans in the Design view, the source is updated. Conversely, as you change the Java source, the Design view updates to reflect the changes you make. Also, if you make changes to your .java source file in another editor, your changes are reflected in the Design window of the visual editor.
The visual editor for Java, by default, shows the Design view above the Source view on a split pane. The Design view is a WYSIWYG surface, or canvas, where you can compose the GUI that you are building. The Source view shows the contents of the Java file. The collapsible palette is on the right side of the Design view by default. The palette includes selection tools and components that you can add to the design.



You can change the default so that the Design and Source views are notebook tabs, allowing you to click back and forth between them. To change the default, click **Window** > **Preferences** > **Java** > **Visual Editor**. Select the **Appearance** tab, then click **On Separate notebook tabs**. Close the visual editor for Java, then open it again to refresh your preference change.

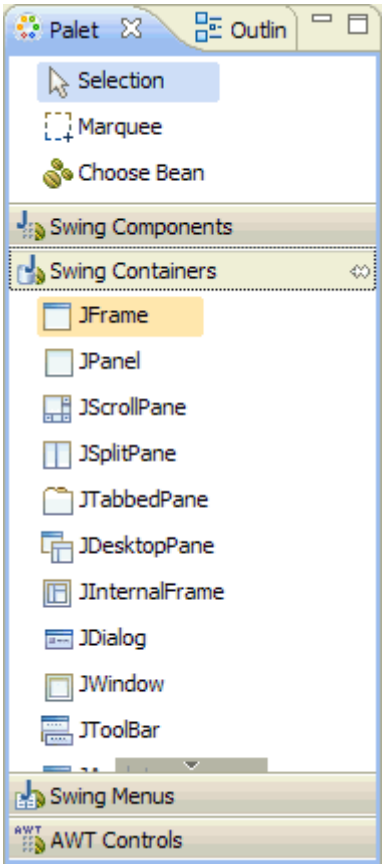
The visual editor palette

The visual editor for Java™ includes a palette of components and other selection tools for you to use in the Design view. By default, the design palette is collapsed on the right side of the Design view in the visual editor for Java. You can resize the palette, collapse it, and choose whether you want the palette collapsed by default. For more information, see Customizing the appearance of the visual editor for Java.

Tip: You can access the same Java components found on the palette by clicking the arrow next to the Choose bean toolbar button.

The design palette includes the following selection tools:

- **Selection** - The Selection tool is the default cursor tool used for selecting existing components in the Design view. You can select multiple components in both the Java beans view and the Design view by holding down the Ctrl key.
- **Marquee** - The Marquee tool is a selection tool that you use to drag a rectangular selection area. All components that lie completely within the selection rectangle are selected.
- **Choose bean** - The Choose bean tool opens the **Choose a bean** selection dialog where you can search for the component that you want to drop onto the Design view. The component that you select becomes the active element for the cursor.



The design palette also includes common components, containers, menus, and controls for SWT, Swing, and AWT. The components are organized by type using drawers that you can expand, collapse, and pin open.

When you select a tool or a component from the design palette, it becomes the active behavior for your cursor. For example, if you select an SWT Button on the palette and then move your cursor over the Design view, the visual editor gives visual cues about the availability and location of placement. When you click in a valid position, the Button is dropped onto the design. After you drop the component, your cursor reverts to the default selection cursor.

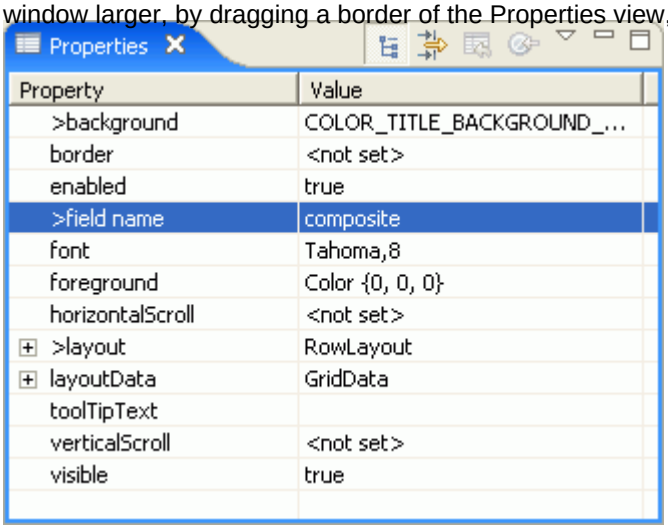
The SWT controls and containers are only included in the palette for projects that include the Standard Widget Toolkit (SWT) library on the Java build path. This library is automatically added to your project when you create a SWT visual

class or if you are using a plug-in project that has org.eclipse.ui as a required plug-in. You can also manually add the library on the Java Build Path page of the project properties. Remember that SWT controls and containers can only be used with SWT applications.

The visual editor Properties view

The Properties view shows the properties for the Java™ bean that is selected in either the Design view or the Java Beans view. You can use the Properties view to change the value of a given property.

Each property typically has a get and set method associated with it, and the value of each property is shown in the Properties view. To see the full list of properties, you might need to enlarge the property sheet by making the workbench window larger, by dragging a border of the Properties view, or double-clicking the Properties view tab.



The Properties view also includes several toolbar buttons that you can use to customize the view or work with values:

- **Show Categories** - The properties for beans provided with the visual editor for Java are not categorized.
- **Show Advanced Properties** - Click this button to display more advanced properties for the Java bean.
- **Restore Default Value** - Click this button to change the value of the selected property to the default value.
- **Set the Active Property being Edited to Null** - For properties that can be set to Null, click this button to set the value to Null.

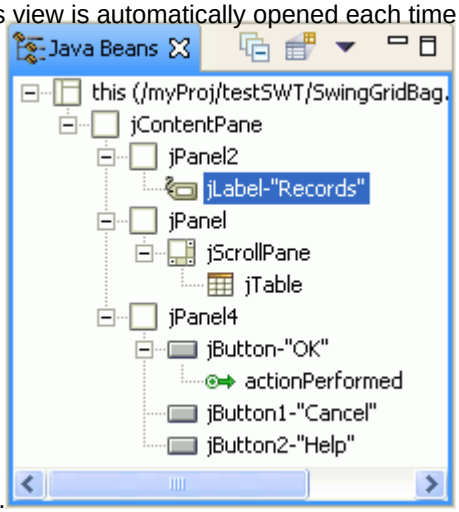
Related tasks:

Changing component properties in the Properties view

The Java Beans view

The Java™ Beans view of the visual editor for Java shows a navigation tree of the components, events, and listeners used by the Java class that you are composing. You can use this view to work with the components.

By default, the Java Beans view is automatically opened each time the visual editor for Java is launched, unless you have changed your preferences.



The following are key features of the Java Beans view:

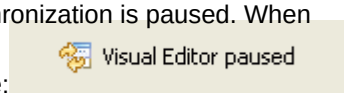
- You can use the Java Beans view to directly work with the beans. By right-clicking on a component in the Java Beans view, you can perform various actions such as cut, copy and paste, delete, and update text.
- You can drag components to reorder and nest them.
- The selection between the Java Beans view and the Design view is synchronized both ways. If you select items in the Design view, they are also selected in the Java Beans view.
- In the Java Beans view, the icon shown for the entry is the same icon used in the palette to represent the component type.
- The label for each component in the Java Beans view is the name of the instance variable that is used in the Java code for the component. For some types of components, the label also contains details from the instance itself, for example the text of a button or label.

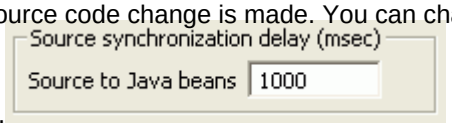
Note: A special part called `this` is shown on the Java Beans view to represent the instance of the class itself that is being composed. You cannot delete the `this` element.

Java code and visual design synchronization in the visual editor

By default, the visual editor for Java™ continuously synchronizes the source code and the Design view. This synchronization can be paused or reloaded manually at any time, and the delay before synchronization occurs can be set as a preference.

The following are key features of the synchronization process and how you can adjust it for performance:

- **Pause** - Click the pause button on the main visual editor toolbar to suspend all synchronization. While synchronization is paused, you can only make changes to the source. This removes the overhead of parsing source, but you cannot make any changes using the Java beans view, Properties view, or Design view while synchronization is paused. When synchronization is paused, the Design view is greyed out to indicate the paused state: 

- **Reload** - When synchronization is paused, the reload button is displayed in the main toolbar. Click the reload button to reparse the code, resynchronize, and revert to the default synchronization mode.
- In Sync - The status of the synchronization is displayed at the bottom of the workbench.
  - If the source code and the model of the Java beans are synchronized, the status displays **In Sync**.
  - If the synchronization is paused, the status displays **Sync Paused**.
  - If synchronization has not yet taken place for a change, the status displays **Out of Sync** until the next synchronization occurs.
  - If you do a reload and the visual editor is unable to parse the code, the status displays **Parse Error**.
- You can change the delay time before automatic synchronization occurs to adjust for performance. By default, synchronization with the Design view occurs one second after a source code change is made. You can change the delay on the Code Generation tab of the Visual Editor preferences page. 

- If the source code contains syntax errors, the toolbar button shows the synchronization error button. The visual editor will wait until the errors are fixed before trying to synchronize again, or you can click the button to attempt a reload.

Java code generation and parsing in the visual editor

The visual editor generates and parses Java™ code as you design your visual class. The visual editor uses a particular style for its generated Java code, and it has rules for parsing Java code.

The visual editor for Java generates valid Java code that can be properly compiled and run. In addition, when the visual editor parses existing Java code, it uses rules to determine which elements of the code to try to visualize in the Design view.

Rules for parsing code for display on the Design view

The visual editor for Java attempts to visualize code in the Design view if the code meets *any* of the following criteria:

- The component is a visual class. Visual classes are subclasses of `java.awt.Component` or `org.eclipse.swt.widgets.Widget` or they implement the interface `org.eclipse.ui.IWorkbenchPart` (such as Eclipse RCP views or editor parts). Most standard widgets and controls are already modeled by the visual editor.
- The declaration includes a `// @jve:decl-index` annotation. When you drop a non-visual component onto the Design view, the visual editor generates an annotation that says the code should be visualized on the Design view. An example of a full annotation looks like this:`// @jve:decl-index=0:visual-constraint="381,79"`
  If you type your own Java code for a non-visual component and you want it to be included on the Design view, be sure to include an annotation similar to this.
- The field name begins with the letters `ivj`. This rule exists for code that was generated in earlier versions of the visual editor for Java.
- The component is referenced by another element that meets any of the above criteria.

There are other requirements that the visual editor checks for before visualizing a component on the graphical Design view:

- Fields must be instantiated within a `get` method, or the bean must be initialized by an initialization method that is listed on the Pattern Styles tab of the visual editor Preferences page.
- There must not be a compilation error on the line.
- A class must be valid to load and instantiate.
- Most array initialization expressions cannot be evaluated.
- Complex arguments for a method can be understood as long as the entities making up the argument are modeled. For example, an expression involving string concatenation using the + operator is evaluated properly in most cases.

Most expressions are successfully parsed, but not all expressions can be correctly evaluated. In this case a warning sign is shown against the Java bean in the views, and the reason for the failure is shown in the status line when the bean is selected on the Design view or Java Beans view. A warning icon also displays on the canvas:

Code generated by the visual editor

- The visual editor generates default constructors that call the method `initialize()`, which sets the values of the properties for the class.
- For applets, the code to set the initial property values is called `init()`. This is not called by the constructor, as it is executed by the applet browser itself.
- **Optional:** You can specify that the visual editor generate `try{}catch()` blocks for components. The `try{}catch()` blocks continuously catch every exception thrown during initialization, and the risk of exceptions being suppressed could increase. Therefore, it is better to let the exception pass through instead. You can select this option on the Code Generation tab of the visual editor preferences (**Window** > **Preferences** > **Java** > **Visual Editor**). The following code shows a JPanel initialized with the `try{}catch()` block:`private JPanel getJPanel1() {`

```
  if (jPanel1 == null) {
   try {
    jPanel1 = new JPanel();
```

```
    }
    catch (java.lang.Throwable e) {
     // TODO: Something
    }
   }
   return jPanel1;
  }
```

The following code shows a JPanel without the `try{}catch()` block of code:

```
private JPanel getJPanel() {
 if (jPanel == null) {
  jPanel = new JPanel();
 }
 return jPanel;
}
```

- **Optional:** You can also specify that the visual editor add a comment marking each expression that it generates. This could be useful for distinguishing hand-written code from generated code. The following line of code is an example of what the comment looks like:`this.add(getJPanel(), null);  // Generated`

  To turn on this option, select the **Generate a comment for new expressions** check box on the on the Code Generation tab of the visual editor preferences.
- For Swing/AWT, although the visual editor generates methods such as `getPanel()` that instantiate and return a single Java bean, this is not a requirement. A method can instantiate more than one Java bean, and the return value of the method is not important for recognizing whether the field is a Java bean. Forthe fields `anOKButton` and `ivjTableModel` to be included as Java beans, they will need to be instantiated within a `get` method in the class.
- For SWT, the visual editor generates `private void createComposite()` methods for every class extending Composite, and any children beans are initialized within the same method.
- If the edited class extends a Java bean, the instance being edited is represented with a special Java bean called a 'this' part. The 'this' part cannot be deleted from the Design view or Java Beans view, and its properties are initialized in the `initialize()` method. A 'this' part is only shown in the Design view and Java Beans view if there are any properties that are available to set on the Properties view. The set methods for the properties are generated in the `initialize()` method, or if the class extends `java.awt.Applet` the `init()` method is used.
- If the edited class implements org.eclipse.ui.IWorkbenchPart, the generated code for the child controls is added to the createPartControl(Composite parent) method.

Setting visual editor preferences

You can set preferences that affect the appearance of the visual editor, the way the visual editor generates and parses Java™ code, and other preferences.

About this task

Choose one of the following topics for more information:

- Customizing the appearance of the visual editor for Java
- Specifying grid display preferences for containers
- Making the visual editor the default Java editor
- Specifying code generation preferences for the visual editor for Java

Customizing the appearance of the visual editor for Java

You can control how and whether views and palettes are displayed in the visual editor.

About this task

By default, the visual editor for Java™ shows the Design and Source views on a split pane, and the component palette is collapsed against the right side of the Design view. If you prefer to have more screen space for your work, you can stack the Design and Source views on separate tabs, rather than tiling them. You can also move the divider to show more or less of the Source view. In addition, you can resize the palette by dragging its border, or you can show or hide it by clicking the small arrow at the top of the palette.

Procedure

1. Specify general appearance preferences using the visual editor Preferences page (**Window** > **Preferences** to open the Preferences window, and go to the **Java** > **Visual Editor**):
   A. On the Appearance tab, specify whether you want the Design and Source views to be tiled (**Above each other with a split pane**) or stacked (**On separate notebook tabs**). This preference goes into effect the next time you open a Java file.
   B. Select **Open Properties view** to automatically display the Properties view when the editor opens.
   C. Select **Open Java Beans view** to automatically display the Java Beans view when the editor opens.
   D. Click **OK** to apply your preferences and close the Preferences window.
2. Open a Java file in the visual editor and customize the appearance and layout of the visual editor:
   A. Click and drag the divider between the Design and Source views, or use the arrows on the divider, to adjust the size of the Design and Source views. For example, if you are not looking at the code at all, you can collapse the divider by clicking the small down arrow.
   B. Drag the border of the palette to adjust its width, or use the small arrow at the top of the palette to collapse or open the palette. For example, if you have finished adding components to your Java class, you can close the palette to give you more design area to work with.

Specifying grid display preferences for containers

When working with Swing GridBagLayout, SWT GridLayout, or null layout, you can specify a preference to show or hide the grid feedback in the Design view.

About this task

By default, the grid feedback for these layout managers is displayed whenever the container or a child component is selected. However, you can choose to not display the grid feedback.

Note:

- When you are dropping or moving components within Swing GridBagLayout, the grid markers and the column and row numbers are always displayed.
- When you are dropping or moving components within SWT GridLayout, the grid markers are always displayed.

You can specify the grid display preferences on the Visual Editor preferences page:

Procedure

1. From the main menu, click **Window** > **Preferences** to open the Preferences window.
2. Expand **Java** and select **Visual Editor**.
3. Select **Show grid when container or its children are selected**. By default, this check box is selected. If you clear this check box, the grid feedback does not display, even when the container or a child component is selected.
4. Optional: For null layout, in the **X/Y grid spacing** field, enter a value in pixels to specify the distance between the grid dots. **Note:** You can override this universal preference in individual containers that use null layout by specifying custom X and Y distances using the Customize Layout  window.

Related tasks:

Using GridBagLayout (AWT)
Using GridLayout (SWT)
Using null layout

Making the visual editor the default Java editor

By default, Java™ classes are opened in a source code editor. You can make the visual editor the default Java editor for all .java files.

About this task

The workbench associates file types with certain editors so that the proper editor opens when you launch a file. For Java files with a .java file extension, the default editor is the basic Java source code editor. Unless you change the default editor in your workbench file association preferences, Java files do not automatically launch in the visual editor.

To make the visual editor the default application for .java files:

Procedure

1. Click **Window** > **Preferences** to open the Preferences window.
2. Expand the **Workbench** node and select **File Associations**.
3. In the **File types** list, select ***.java**.
4. In the **Associated editors** list, select **Visual Editor** and click **Default**.

Results

Note: Changing the default Java editor affects what happens when you open any .java file, including JSP files and servlets.

Specifying code generation preferences for the visual editor for Java

There are several code generation preferences that you can set for the visual editor for Java™.

About this task

To specify code generation preferences:

Procedure

1. From the main menu, click **Window** > **Preferences** > **Java** > **Visual Editor** to open the visual editor preferences.
2. On the Code Generation page, you can select the following preferences:
   - **Generate try{}catch() block**
   - **Generate a comment for new expressions**

   The following code shows a JPanel initialized with the `try{}catch()` block:
```
private JPanel getJPanel1() {
 if (jPanel1 == null) {
  try {
   jPanel1 = new JPanel();
  }
  catch (java.lang.Throwable e) {
   // TODO: Something
  }
 }
 return jPanel1;
}
```
   The following code shows a JPanel without the `try{}catch()` block of code:
```
private JPanel getJPanel() {
 if (jPanel == null) {
  jPanel = new JPanel();
 }
 return jPanel;
}
```
   The following line of code is an example of what the generated comment looks like:
```
this.add(getJPanel(), null);  // Generated
```
3. Click **OK** to save your changes and close the Preferences window.

Designing Java classes with the visual editor

You can use the visual editor to visually compose and preview Java™ classes using Standard Widget Toolkit (SWT), Swing, and AWT components.

Before you begin

About this task

The following Web sites provide more information about Java, the Eclipse Rich Client Platform, and Swing:

- Java™ Platform, Standard Edition 6 API Specification
- Standard Widget Toolkit (SWT)
- Eclipse Rich Client Platform
- Swing API document (package javax.swing)

Using the visual editor, you can add Java beans, or components, from different palettes, manipulate them in the Design view and Java Beans view, and edit the properties in the Properties view. The visual editor also includes a Source view where you can see and modify the generated Java code. You can change your code in either the Source view or in the Design view.

Opening a Java class in the visual editor

If you have not made the visual editor for Java™ the default editor for .java files, you can still open Java classes in the visual editor.

About this task

To open an existing Java class in the visual editor for Java:

Procedure

1. In the Package Explorer view of the Java perspective, select the .java source file for the class that you want to edit in the visual editor.
2. Do *one* of the following steps:
   - Right-click and select **Open With** > **Visual Editor** from the pop-up menu. The workbench remembers your editor choice for this file until you choose a different editor.
   - Double-click the file. If the visual editor is the default editor for .java files, or if you most recently opened the file with the Java visual editor, the file opens in the visual editor.

Creating a new Java visual class

You can use the New Java™ Visual Class wizard to quickly create a visual class that you can then design within the visual editor.

Before you begin

Before you can create a new Java visual class, you must create a Java project or plug-in project where you can place the visual class. A plug-in project that is enabled for Rich Client Platform (RCP) application development is required if you want to create a new RCP view or editor.

About this task

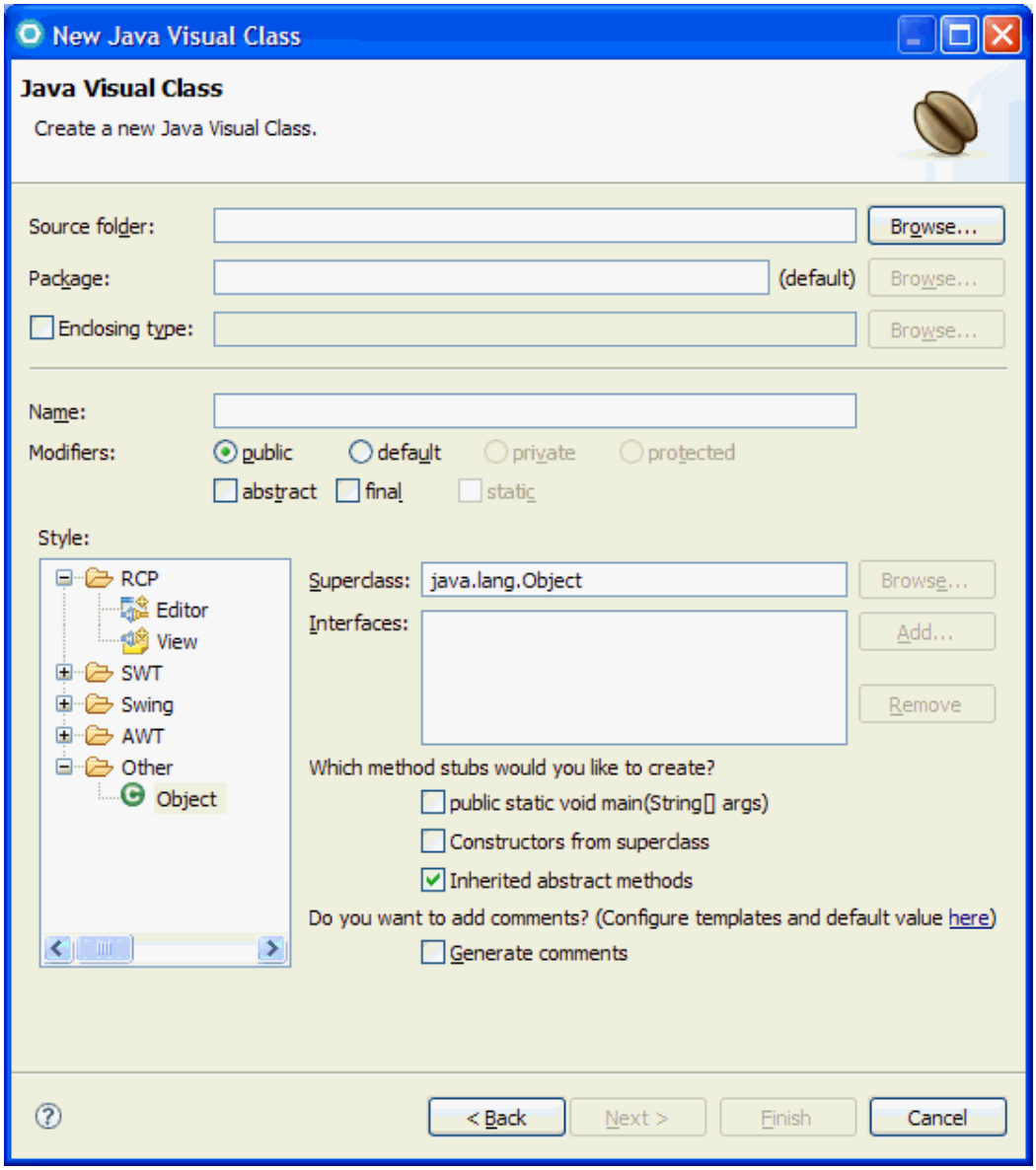
The wizard generates new visual classes based on a few quick selections. For example, you can specify the toolkit or style that you want to use (RCP, SWT, Swing, AWT, or Other), the initial container or composite that you want the visual class to instantiate (for example, an RCP view, an SWT composite, or a Swing JFrame), and whether you want the visual class to be an application that includes a `public static void main(String[] args)` method.

Tip: If you select the Swing Application style, the wizard generates a Swing application that includes commonly used menus and event handlers already built into it.

To create a new Java visual class for use in the visual editor for Java:

Procedure

1. In the Java perspective, click **File** > **New** > **Visual Class**. The New Java Visual Class wizard opens.



2. In the **Source Folder** field, enter the workspace folder where the class is to be saved. This defaults to your current project.
3. In the **Package** field, enter the name of the Java package where you want the visual Java class to be packaged.**Tip:** If you open the wizard from the pop-up menu of the package, this field defaults to that package name.
4. Ensure that the **Enclosing type** check box is cleared. Selecting this option creates an inner class in another class that you specify, so no new visual class is created.
5. In the **Name** field, enter the name for the new Java visual class.
6. Select *one* of the following modifiers to specify access control for the class:
   - **public** - makes the class available to any other class that wants to use it
   - **default** - sets no modifier, making it available to any other class in the same package
7. Optional: Select *one* of the following modifiers for the class:
   - **abstract** - indicates that the class serves in a superclass role
   - **final** - indicates that the class cannot be subclassed
8. In the **Style** list, select the toolkit and visual element that you want your new visual class to extend. For example, you can select RCP view, SWT composite, or Swing JPanel. **Note:** An RCP view or editor can only be created in a plug-in project that is enabled for Rich Client Platform (RCP) application development

   The **Superclass** field displays the appropriate class name. If you select **Other** as the style, you need to specify the superclass that you want to extend or accept the default `java.lang.Object`. The `java.lang.Object` superclass can be used, for example, for an SWT application that uses an SWT Shell.

9. Optional: To import and implement an additional interface in your new visual class, click **Add** and select the interface, then click **OK**. The interfaces to be implemented are listed in the **Interfaces** field. Repeat this step for each interface.
10. Optional: Select any of the following check boxes to generate additional method stubs in your new visual class:
    - **public static void main(String[] args)** - generates a stub main() method for running the class as a Java application. For SWT applications, the visual editor also generates the necessary display loop in the main method.
    - **Constructors from superclass** - generates constructor stubs to initialize methods inherited from the superclass.
    - **Inherited abstract methods** - generates stubs for additional abstract methods inherited from the superclass.
11. Select **Generate comments** if you want the wizard to add comments to the source code, as configured in the project properties.
12. Click **Finish**.

Results

The wizard generates a new .java file that includes the new visual class, and the class opens in the visual editor for Java

Related tasks:

Creating an RCP editor in the visual editor

Creating an RCP view in the visual editor

Working with Java components visually

With the visual editor for Java™, you can add, remove, resize, modify, and change the properties of components, including visual and non-visual elements.

Adding a component to a Java visual class

You can use the visual editor palette or the Choose a Bean dialog to add components to your visual Java™ class.

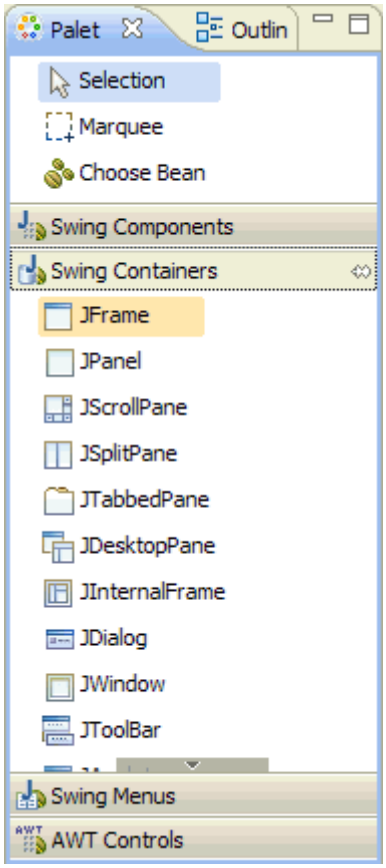
About this task

For example, for an SWT class, you can select the Button control (on the palette under the SWT Controls drawer) and drop it onto your application in the Design view.

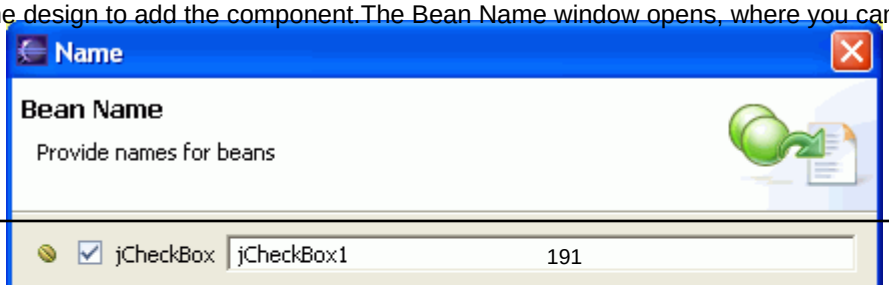
To add a component to your visual class in the Design view:

Procedure

1. Select the component that you want to add to your Java class by doing *one* of the following steps:
   - In the visual editor palette, select the component that you want to add to the design.



   - Click the down arrow next to the Choose Bean icon in the toolbar, and select a component from the list.
   - Click **Choose component** in the palette to open the Choose a Bean dialog. Enter the name of the component. As you type, the dialog displays available and valid classes in the Java build path for your current Java project. Valid classes have a null constructor rather than the static getInstance() pattern. Select a component and click **OK**.
2. Move your pointer over the Design view to the point where you want to add the component. The mouse pointer shows a plus sign when it is positioned over a valid drop location. Depending on the position where you are dropping the component and which layout managers are being used, if any, you might see other visual cues to help you see where the component is added.
3. Left-click the design to add the component. The Bean Name window opens, where you can specify the name for the



Select **Do not ask again** to not show this window each time you add a bean. The visual editor uses a default name. This choice can be changed in the visual editor preferences.

Results

The Design view shows the visual representation of the component, and the Source view shows the Java code that was generated when you added the component.

Cutting, copying, and pasting in the visual editor for Java

The visual editor supports cutting, copying, and pasting in the Design view and Java™ Beans view. You can copy single widgets or containers with their children.

About this task

When you are designing a visual class, often you want to quickly cut or copy an element then paste several instances of the same element in another spot in your application. When an item is copied, not the properties of the item are included, but typically the key visual properties are included in the paste. For example, when you copy and paste a button, the text for the button is included. When you cut or copy a Swing container or SWT composite, the child controls are copied too. The standard keyboard shortcuts are supported:

- Ctrl+X (cut)
- Ctrl+C (copy)
- Ctrl+V (paste)

To cut, copy, and paste components using the visual editor for Java:

Procedure

1. In the Design View or Java Beans view, right-click the element that you want to cut or copy, and select **Cut** or **Copy**.
2. Do *one* of the following to paste the item:
   - In the Design view, click the canvas to load your cursor with the copied component. Now you can drop the component in an appropriate location the same way you would drop an item from the palette. For example, you can click and drag to specify the initial size and bounds of the dropped component.
   - In the Java Beans view, right-click the container where you want to paste the item, and select **Paste**.

Using the keyboard to add components

You can use the keyboard to drop a component from the palette onto the Design view of the visual editor for Java™.

About this task

To drop a component from the palette using the keyboard only, use one of the following options:

Procedure

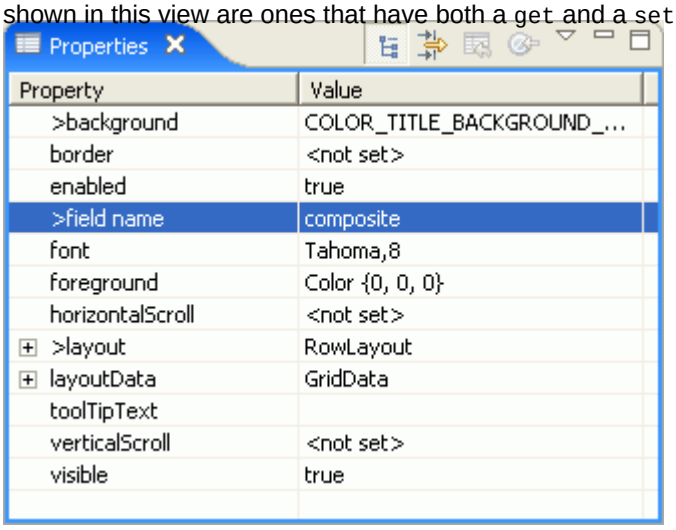
- Use the Source view to type or enter the Java code.
- Use the MS accessibility tool to map the mouse to the keyboard and use as follows:
  - Use the workbench keyboard shortcut to get to the toolbar (F10, and tabs), then select the ChooseBean Action with the arrow key down. From the menu items, select the component you want to drop using the enter key.
  - Use the numeric keypad to move your loaded cursor to the area in the canvas you want to drop (you can change the location later from the source code).
  - Use the + numeric key to drop the bean. You can use the Property view or source view to change the properties of the component.

Changing component properties in the Properties view

You can change the properties of a component by using the Properties view.

About this task

The entries in the Properties view show the properties available on the selected component. By default, the properties shown in this view are ones that have both a `get` and a `set` method, meaning that they can be read and written.



To view and change the properties for a component:

Procedure

1. In the Design view or Java™ Beans view, select the component whose properties you want to view or change.**Tip:** Here are a few tips for working with the Properties view:
   - A plus sign (+) next to a property name indicates that property itself has other properties. Click the plus sign to expand the view and see all the bean properties. A keyboard shortcut for clicking the plus sign is the right arrow key.
   - A right angle bracket (>) next to a property name indicates that the property is explicitly set in code, not returned from the live bean. If you select the property that is flagged with the > symbol, the line of Java code that sets the value is highlighted in the Source view. To toggle the > symbol on or off, select **Show set values** from the Properties view menu.
   - To distinguish null values from empty values, select **Show null values** from the Properties view menu. The characters `<null>` then displays in the Value column when the result of calling the property's get method on the Java bean returns null.
   - To show more advanced properties, click the Properties menu button and select **Show Advanced Properties** from the Properties view menu.
   - To show properties that are read only (only have a `get` method), select **Show read only properties** from the Properties view menu.

2. In the Properties view, select the property that you want to change, then update the **Value** column with the new property value.
   - To set a value to be explicitly null, click the **Set the Active Property Being Edited to Null** button on the Properties view toolbar. The `set` method is called with a null argument.
   - To restore a property to its default value, click the **Restore Default Value** button on the Properties view toolbar. When a value is restored to its default value the statement that sends the `set` method is deleted from the source, and the `set` method is explicitly called with the original default value, to restore the component to its default state. The original value is also sent to the component if you delete the `set` method in the Source view.

- Some properties, such as color, have their own custom property editors that open when you click the ellipse button in the value column. If the property editor does not open in front of the main workbench window, select the property editor on the task bar to bring it to the front.

Related concepts:

The visual editor Properties view

Resizing visual components in the visual editor for Java™

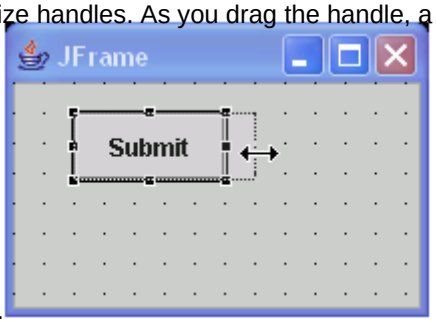
In most cases, if the container of a component does not control the size of the component, you can use the selection handles to resize visual components, or you can change the size in the Properties view.

About this task

For example, in a Border layout, the components typically fill the entire space that is created by the layout manager, and a selected component does not show any resize handles. With layout set to null, a selected component shows resize handles that you can use to resize freely. If no resize handles are available, you can try using the appropriate size property in the Properties view.To resize a component using the resize handles:

Procedure

1. In the Design view, select the component that you want to resize.
2. Click and drag one of the resize handles. As you drag the handle, a dotted line shows where the component is sized to when you release the mouse.



3. Release the mouse button.

Directly editing components in the Design view

Some text properties can be directly edited on the Design view of the visual editor.

About this task

When you work in the visual editor for Java™, you can change the properties of many components just by manipulating them in the Design view. For example, you can edit the text strings for most components that include labels. The new values are reflected in the Source view and in the Properties view. Adding text strings can also affect visual layout, which is reflected in the Design view. For example, a long text string for a button can affect the alignment and size of the button.

Tip: You can also directly edit items by selecting **Set [property]** from the pop-up menu, in both the Java Beans view and the Design view. For example, you can right-click a JLabel and select **Set Text**.

To directly edit the text for a component in the Design view:

Procedure

1. Select the component in the Design view.
2. Click the component again. A direct edit text field displays in the upper left corner of the component.
3. Type the text that you want to use for the component, and press **Enter**. 

Results

The text that you typed is reflected in the Design view, in the source (the `set` method), and in the Properties view for that particular property.

Reordering components in a Java visual class

The Java™ Beans view displays components in the order in which they are added to their parent. You can use the Java Beans view to change this order.

About this task

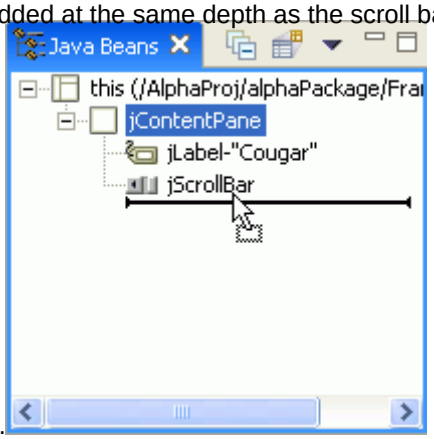
When the source is parsed, the order that displays in the Java Beans view is determined by the order of the methods that create the relationship, such as `add(Component,Object)`.

Tip: Depending on the layout manager being used, you can reorder most components on the Design view by dragging and dropping them. Your changes are reflected in the Java Beans view.

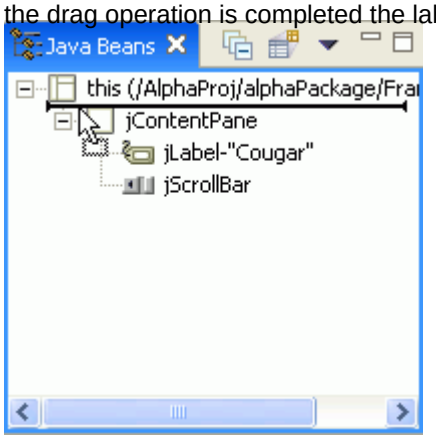
To use the Java Beans view to reorder children components:

Procedure

1. In the Java Beans view, click and drag the component that you want to reorder.
2. Release the mouse in the new position where you want the component to be placed: Use the feedback lines to determine proper placement:
   - As you drag the component, feedback is drawn as a line that shows the position of where the bean are to be moved. The following figure shows a label that has been selected and dragged below the scroll bar. The left edge of the feedback shows that the label is added at the same depth as the scroll bar and the label. So, it remains a child component of the navigation pane.



   - As the mouse is moved around, the left edge of the feedback line is always drawn at the depth level in the tree where the item is inserted. The following figure shows that the left edge of the feedback line is level with the frame itself, and if the drag operation is completed the label becomes a top-level Java bean like the frame itself.



Deleting components from a Java visual class

You can use the visual editor to quickly delete a component from a Java™ class.

About this task

Procedure

1. In the Java Beans view or Design view of the editor, select the component or components that you want to delete.
2. Do *one* of the following steps:
   - Right-click and select **Delete** from the pop-up menu.
   - Press the Delete key on the keyboard.
   - Click the Delete  button on the main toolbar.

Results

The components that you selected are deleted and removed from both the Source and the Design views of the visual editor. To undo your change, you can click **Edit** > **Undo**, or the Undo  button on the main toolbar.

Laying out UI components using the visual editor

You can use the visual editor for Java™ to visually arrange and lay out user interface (UI) components in your Java application. The editor supports various SWT and Swing layout managers.

Before you begin

Before you can lay out UI components, your Java class must contain a container, such as JFrame, JPanel, Composite, or Shell, that allows for a layout manager to be set.

About this task

When you design an application, you should make the user interface easy to use, intuitive, well-organized, and visually appealing. You can customize the layout of your application to improve its usability and make it fit your needs.
When you add containers in a Java visual class, the editor uses default settings for the layout and alignment of components within those containers. For example, the default layout manager in the visual editor for JPanel is FlowLayout, while other containers have layout set to null.
To specify a layout manager for a container, set the layout property:

Procedure

1. In the Design view or Java Beans view, select the container that you want to specify a layout manager for.
2. In the Properties view, locate the layout property, and click its value column.
3. Select the layout manager that you want to use for the container.

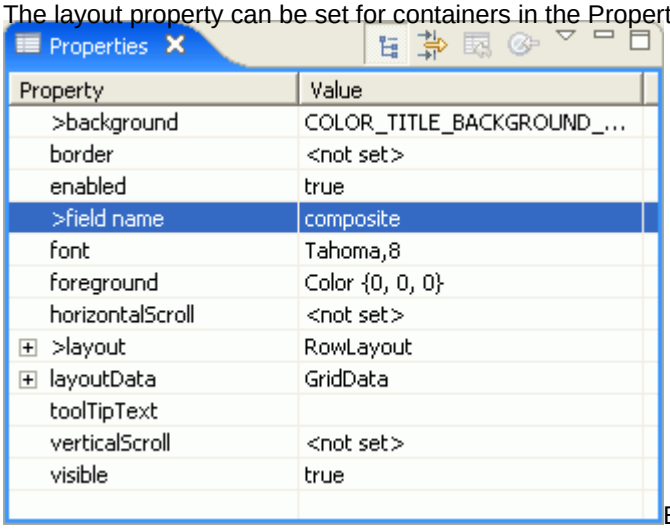
Layout managers and containers

Layout managers are objects that control the size and position of components within a container.

Some Java™ beans allow relationships between themselves. For example, the class java.awt.Container allows instances of java.awt.Component to be added to it as part of its components relationship. An example of java.awt.Container is a panel or a frame. An example of java.awt.Component is a button or a text box. The SWT class org.eclipse.swt.widgets.Composite allows instances of org.eclipse.swt.widgets.Control to be added as part of its controls relationship. In the visual editor for Java, containment relationships are shown in the Java Beans view and in the Design view. In other words, containers can contain components or other containers.

A layout manager is an object that controls the size and position of components within a container. You can set the layout manager on a container to null and allow each component to size and position itself. However, in a null layout the position of the GUI components will not react to a window being resized. With a layout manager, rather than having a fixed size and position for each component, the container's layout manager delegates the sizing and positioning of its components.

The layout property can be set for containers in the Properties view:



Each layout manager has its own rules for distributing the components, but most layout managers place each component so that it is at least as big as its preferred size. The preferred size of components with user visible strings, such as button or label, is calculated dynamically. So, as the string length changes due to different locales or fonts, the layout manager will reposition each component. As the window is resized by a user at run time, the layout manager also repositions each component to optimize the new overall size. This makes layout managers invaluable for building user interfaces that will be deployed in an environment where label strings, fonts, or window size may vary (due to globalization, for example).

Swing and Abstract Window Toolkit (AWT) containers use the same set of layout managers, which are mostly AWT objects. Standard Widget Toolkit (SWT) composites use their own SWT layout managers.

Note: If you use a layout manager not supplied in the visual editor, the Design view will still construct the container correctly and render the components. However, you will not be able to update constraints using the Properties view or Design view.

Customizing UI layout in the visual editor

In the visual editor for Java™, you can use the Customize Layout window to modify layout settings for the currently selected container and UI component.

Before you begin

Your Java class must include a container, such as JFrame, JPanel, or Shell, that allows for a layout manager to be set.

About this task

Some layout effects can be achieved by visually dropping, dragging, and resizing UI components in the Design view. The visual editor includes the Customize Layout window to help you further refine and work with your layout settings. The Customize Layout window is a non-modal dialog box that you can open and leave open, like a toolbox, while you work in the Design view. The Customize Layout window then displays relevant layout properties and settings, depending on the layout manager for the selected container or the layout manager of the selected container of the component.

The Customize Layout window includes two tabbed pages:

- **Layout** - The Layout page includes options that affect the layout in general. For example, for layouts that use grids, the Layout page might have grid spacing and margin settings, or it might allow you to specify the number of columns or rows in the grid. Again, these settings vary based on the layout manager. Not all layout managers include customizable settings on the Layout page.
- **Component** - The Component page includes layout options for the selected component. For example, certain layout managers use grids where you can specify the alignment of a component within the grid, how many cells a component spans, or whether the component should grab excess space within its cell. Not all layout managers include settings on the Component page.

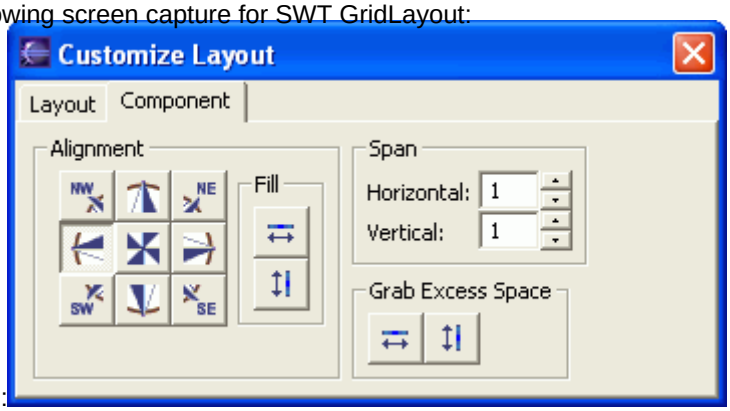
To open the Customize Layout window, do *one* of the following:

Procedure

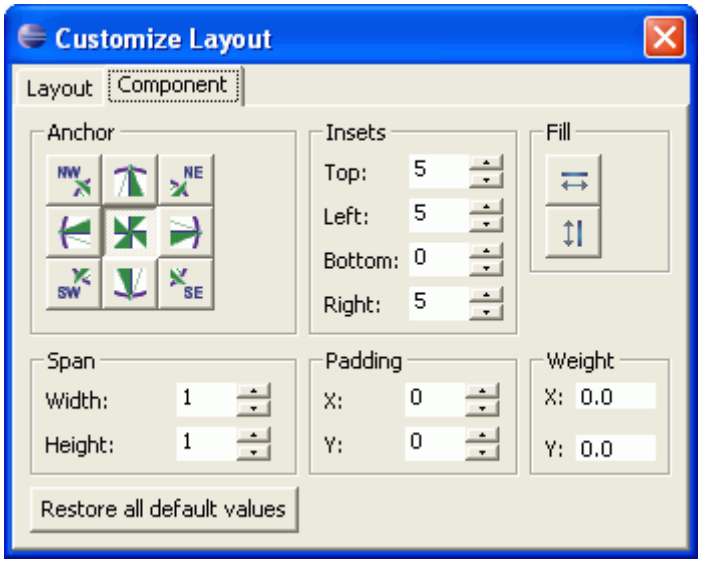
- Click the Customize Layout  toolbar button.
- In the Design view, right-click a container or component, and select Customize Layout from the pop-up menu.

Example

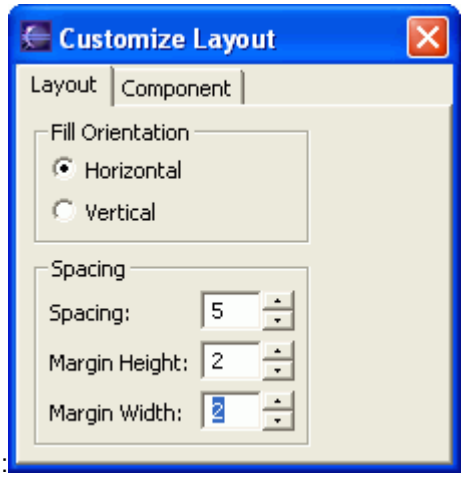
The Customize Layout window includes different options for different layout managers. For example, the Customize Layout manager looks like the following screen capture for SWT GridLayout:



- SWT GridLayout example:
- Swing GridBagLayout example:



- SWT FillLayout example:



Using SWT layout managers

You can work with SWT layout managers with the visual editor for Java™.

About this task

Using RowLayout (SWT)

The SWT RowLayout layout manager lays out its components in rows, but is more robust than FillLayout.

About this task

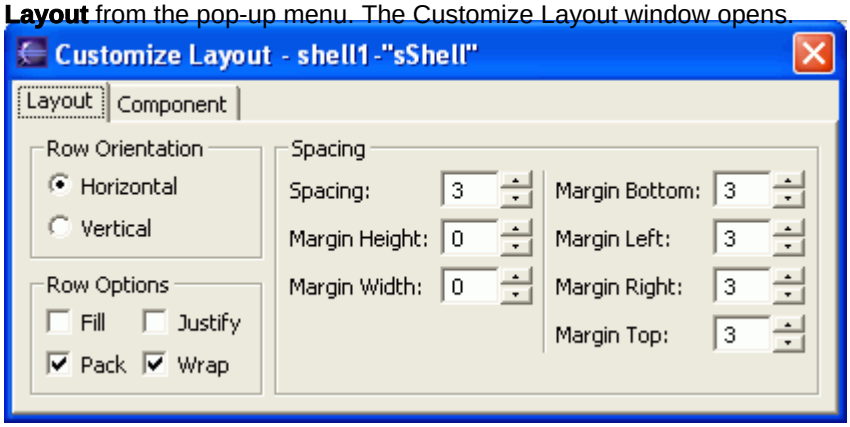
With RowLayout, you can set the rows to be in vertical or horizontal style. For the horizontal style, widgets are placed to the left and right of each other. In vertical style, widgets are placed above and below each other.

Dropping or moving widgets within a RowLayout works similarly to other layout managers in the visual editor. As you move your cursor on the Design view, a black bar indicates where the widget is dropped or moved. If the RowLayout is set to the vertical style, a horizontal black bar isabove or below any existing widgets. If the RowLayout is set to the horizontal style, a vertical black bar shows placement to the left or right of existing widgets.

To set the RowLayout options for a container using RowLayout layout manager:

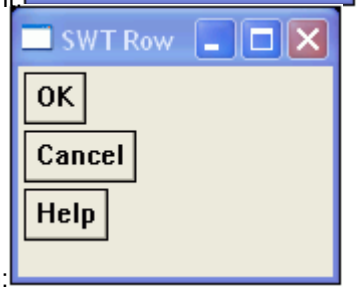
Procedure

1. In the Design view or Java™ Beans view, right-click the container that is using RowLayout and select **Customize Layout** from the pop-up menu. The Customize Layout window opens.



2. On the Layout page, specify an orientation for the rows:
   - **Horizontal** - standard row orientation, left to right:

   - **Vertical** - a top-to-bottom orientation, or column:



3. Specify sizes in pixels for the margins and the space between the widgets:
   - **Spacing** - sets the spacing, or padding, between widgets
   - **Margin Height** - sets the height in pixels of the top and bottom margins
   - **Margin Width** - sets the width in pixels of the left and right margins
   - **Margin Bottom** - sets the size of the bottom margin
   - **Margin Left** - sets the size of the left margin

- **Margin Right** - sets the size of the right margin
- **Margin Top** - sets the size of the top margin

Note: If you set margin height or margin width, then also specify the bottom, top, left, or right margins, the values cumulate. For example, if you set a margin height of 5 and a margin top of 5, the margin at the top of the row layout is 10.

4. Specify other RowLayout options:
   - **Fill** - specifies that the controls in a row should be all the same height for horizontal layouts, or the same width for vertical layouts
   - **Justify** - specifies that extra space remaining in the composite is allocated as margins between the widgets
   - **Pack** - specifies that all controls in the layout take their preferred size
   - **Wrap** - specifies that a control is wrapped to the next row or column if there is insufficient space in the current row or column

Related concepts:

Layout managers and containers

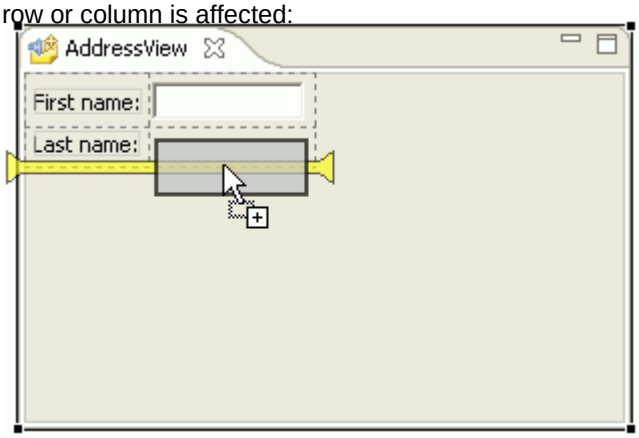
Using GridLayout (SWT)

You can use the GridLayout with SWT containers to arrange widgets in a grid arrangement.

About this task

The SWT GridLayout layout manager arranges widgets in rows and columns. The visual editor provides feedback as you add widgets to the grid. When you add a widget, you can add it to an empty grid cell, or you can force the grid to add a new row or column and place the widget in a new cell. When you add new rows or columns, "empty" cells are created as needed. Because GridLayout requires each cell to contain a widget, a filler label with no text is added the empty cells actually include a filler label with no text.

The visual editor displays a grid border and placement indicators to help you determine where widgets are located within the grid and where a widget is placed in relation to other widgets. A yellow bar that spans the entire width or height of the grid indicates that an entire row or column is added. A yellow bar that spans a single row or column indicates that only that row or column is affected:



For more help, read the following sections of this topic:

- Adding or moving components in GridLayout
- Setting the alignment of a component within its cell in the grid
- Spanning a widget across grid cells
- Specifying the number of columns and rows in the grid

Related concepts:

Layout managers and containers

Related tasks:

Specifying grid display preferences for containers

Adding or moving components in GridLayout

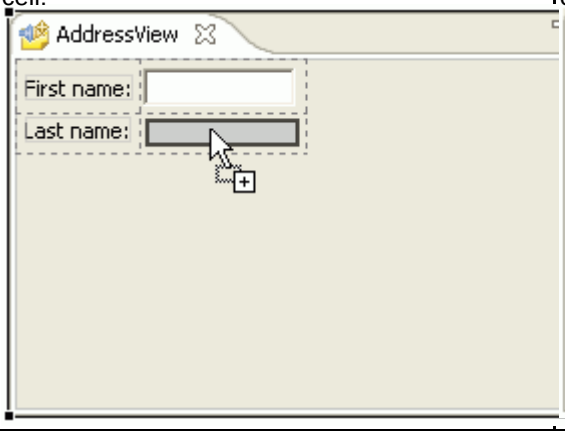
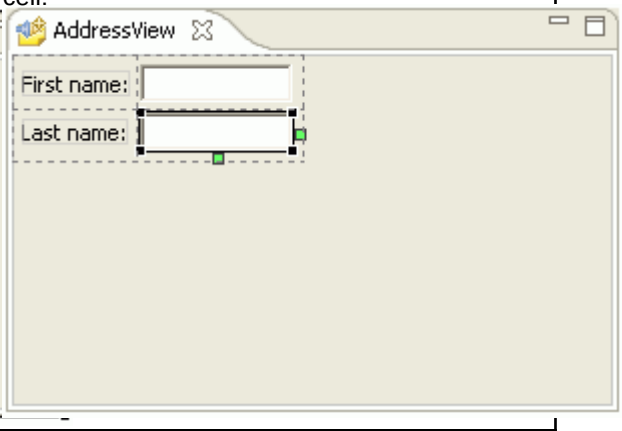
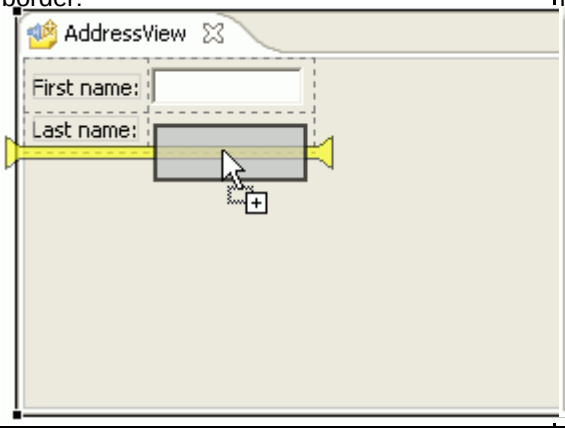
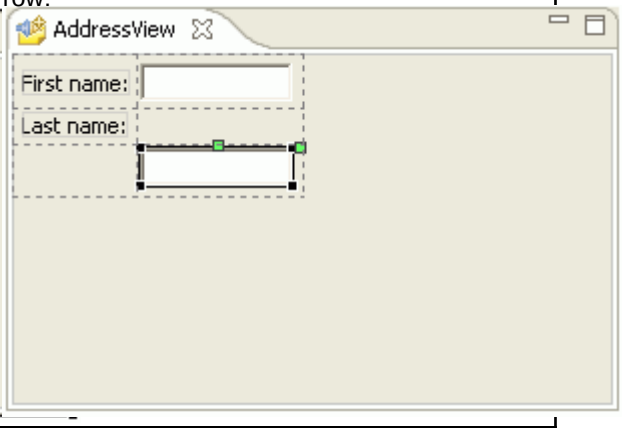
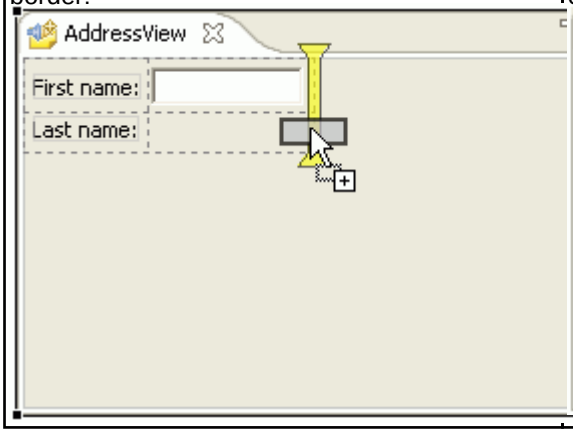
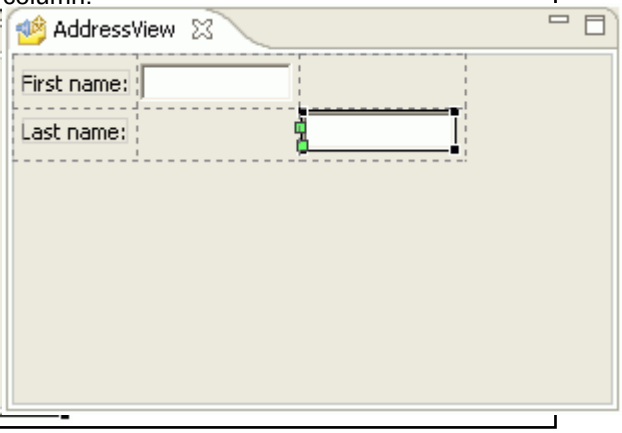
Procedure

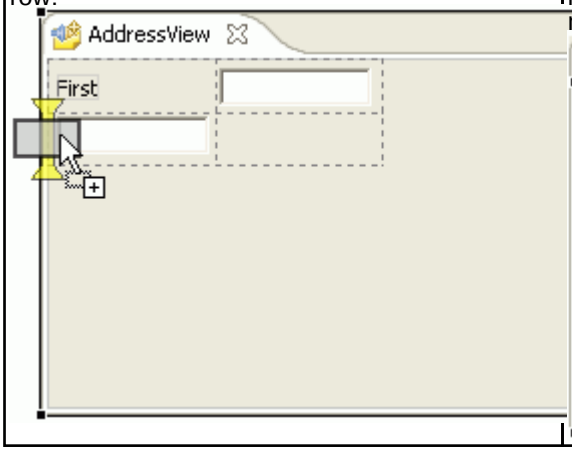
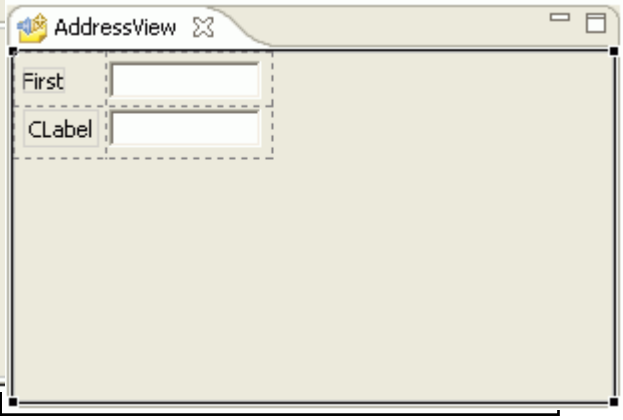
1. Select the component from the palette that you want to add to your Grid layout, or click and drag the existing component that you want to move within your Grid layout.
2. Move your mouse pointer over the position where you want to drop or move the component. As you move the cursor, visual feedback indicates the placement of the widget if you release your mouse button. A yellow bar indicates that a new row or column is created

3. Click or release your mouse to drop the component in the position that you selected. Depending on the placement, the visual editor moves widgets and create rows or columns as necessary.

Example

Table 1. Example drop placements and results

Cursor position and feedback	Result
Cursor placed in an empty cell: 	Widget is added to cell: 
Cursor placed on horizontal grid border: 	Widget is added to cell in new row: 
Cursor placed on vertical grid border: 	Widget is added to cell in new column: 

Cursor placed on vertical grid border, on single row:	Widget (CLabel) is added to cell, other widgets in row are moved:
	

Setting the alignment of a component within its cell in the grid

Procedure

1. Right-click the component and select **Customize Layout**. The Customize Layout dialog opens.
2. In the Alignment section of the Component page, select one of the compass buttons to align the component. For example, if you click the **NW** (North-West) button, the component is aligned in the upper left corner of the grid cell.
3. Click or release your mouse to drop the component in the position that you selected. In Grid layout, you can only place an item to the left or right of another item. Depending on the number of columns and whether other components span columns, the visual editor moves widgets and creates rows as necessary.
4. Optional: To force the row or column that holds the component to grab extra space in the container, click the **Fill horizontal** or **Fill vertical** button. If multiple rows or columns are grabbing extra space, they divide up the excess space equally.

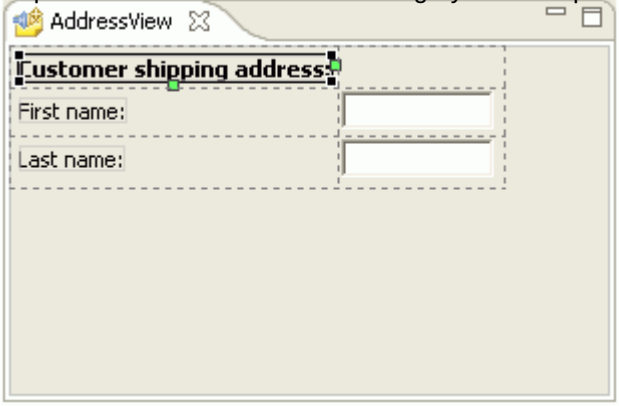
Spanning a widget across grid cells

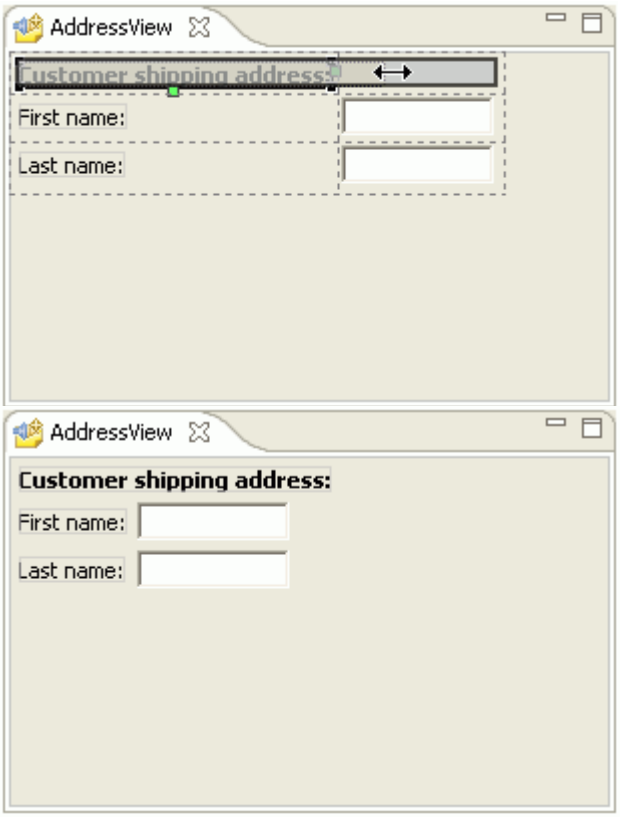
About this task

To span a widget across multiple grid cells, do *one* of the following steps:

Procedure

- Click a resize handle on the widget, and drag the component across cells until the dark grey box occupies the cells that you want to span, then release your mouse button:





- On the Customize Layout window, enter the exact number of cells that you want the widget to span. If you increase the **Vertical** span, the component spans to the cell below its current position. If you increase the **Horizontal** span, the component spans to the cell to the right of its current position.

Specifying the number of columns and rows in the grid

About this task

When you add or move widgets, the number of columns and rows can change dynamically. You can also manually specify the number of columns:

Procedure

1. In the Design view, select the container that uses Fill layout.
2. Click the **Customize Layout** Toolbar button. The Customize Layout dialog opens.
3. On the Layout notebook tab, enter a new value in the **Number of columns** field.
4. Optional: You can force all the Grid columns to be the same width by selecting the **Make columns equal width** check box.

Using FillLayout (SWT)

The visual editor makes it easy to work with the SWT FillLayout manager.

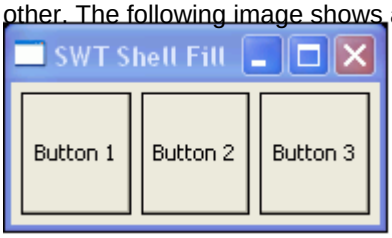
Before you begin

To work with Fill layout, you need to create an SWT visual class with a Shell or a Composite or another container.

About this task

The SWT Fill layout manager simply lays out components in order, filling the entire space of the container. The components do not wrap, but are resized to fit the space, unless the verticalScroll or horizontalScroll properties are set. In addition, the Fill layout has margin height, margin width, and spacing properties that affect the spacing of components within the container.

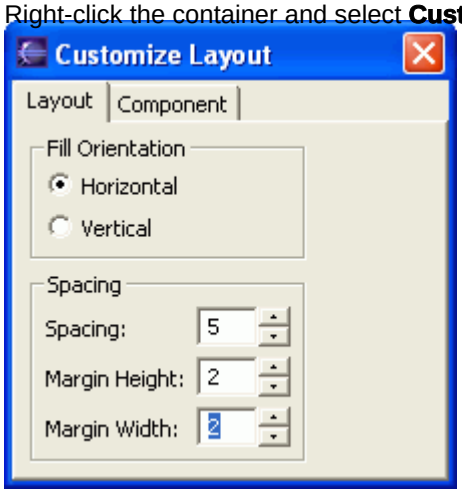
Also, you can set the type to either vertical or horizontal to dictate how the components are laid out in relation to each other. The following image shows a horizontal FillLayout with margin height, margin width, and spacing each set to 5:



To work with FillLayout:

Procedure

1. Select your SWT container, such as Shell or Composite.
2. In the Properties view, change the layout property to FillLayout.
3. Right-click the container and select **Customize Layout** on the pop-up menu to open the Customize Layout window:



4. Set the following properties as required:
   - **Spacing** - sets the distance between components.
   - **Margin height** - sets the distance between the components and the top and bottom inner edges of the container.
   - **Margin width** - sets the distance between the components and the left and right inner edges of the container.
5. Choose one of the following options for Fill Orientation:
   - **Horizontal** - sets a left to right orientation
   - **Vertical** - sets a top to bottom orientation
6. Optional: To add a scroll bar to the container, set the value for either of the following properties in the Properties view:
   - horizontalScroll - adds a scroll bar at the bottom of the container for scrolling horizontally
   - verticalScroll - adds a scroll bar at the right side of the container for scrolling vertically

The scroll bar does not function in the Design view. To see how the scroll bar functions, run your visual class.

Related concepts:

Layout managers and containers

Using FormLayout (SWT)

You can set an SWT container to use the SWT FormLayout layout manager, but the visual editor does not provide additional visual support.

About this task

You can set layoutData object properties for FormLayout in the Properties view, but the Design view does not provide advanced visual cues for placement or layout of components within FormLayout.
It is recommended that you update the source code for changes related to FormLayout.

Related concepts:

Layout managers and containers

Using Swing and AWT layout managers

You can work with Swing and AWT layout managers with the visual editor for Java™.

Using BorderLayout (Swing)

The visual editor provides visual cues for helping you work with the Swing BorderLayout.

About this task

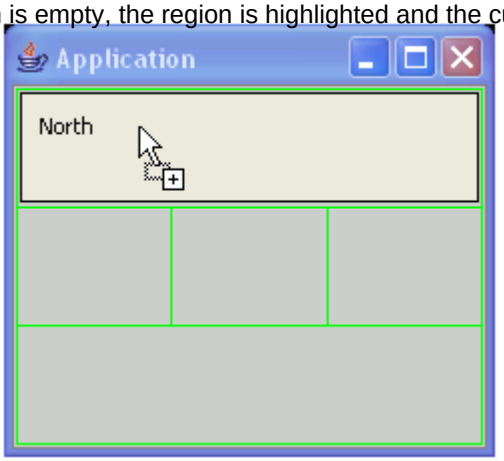
The BorderLayout manager lays out components into regions defined by compass directions. The class java.awt.BorderLayout implements LayoutManager2, and its constraint is a string that can be of the value "North", "South", "Center", "East", or "West".

Note: If you switch a container to BorderLayout and it has more than five components, only the first five components are added with constraints to the BorderLayout. The remaining components are moved from the container to the free form area of the Design view.

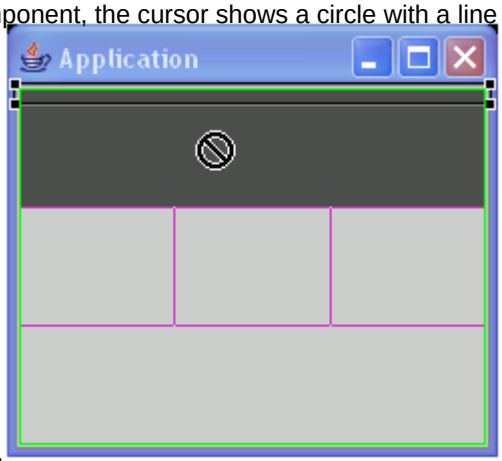
BorderLayout positions each component at one of the compass constraints along the edge with its preferred width or height, and the "Center" component occupies all the remaining space.

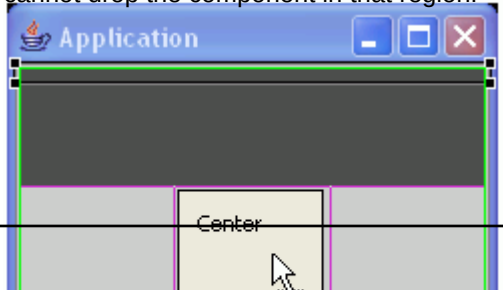
Procedure

- To add a component on a container that is using BorderLayout:
  1. Select the Swing component from the palette that you want to add to the BorderLayout.
  2. Move your mouse cursor over the Design view.
     - The BorderLayout shows five rectangular areas representing the five constraints of "North", "South", "Center", "East", or "West".
     - If the region is empty, the region is highlighted and the cursor shows a plus sign, indicating that you can drop the component.



     - If the region is already occupied by a component, the cursor shows a circle with a line through it, meaning that you cannot drop the component in that region.





3. Click the Design view in a valid region to add the component to the BorderLayout.

- To move components within a BorderLayout:
  1. In the Design view, select the component in the BorderLayout that you want to move to another region within the BorderLayout.
  2. Drag the component to another region and release your mouse button. The component is moved to the new region. If the region was already occupied by another component, the components switch places.
- To adjust the spacing for components within a BorderLayout, you can set the following properties on the BorderLayout:
  - **Horizontal gap** - sets the distance in pixels between the center component and the east and west components.
  - **Vertical gap** - sets the distance in pixels between the center component and the north and south components.

Results

The constraints value is used as the second argument to the method `add(Component,Object)` that adds the components to their parent container. For example, the code to initialize a JPanel might look like this:

```
private void initialize() {

      this.setLayout(new java.awt.BorderLayout());
      this.add(getJLabel(), java.awt.BorderLayout.NORTH);
      this.add(getJScrollBar(), java.awt.BorderLayout.WEST);
      this.add(getJButton(), java.awt.BorderLayout.EAST);
      this.setSize(193, 124);
}
```

Note: Two additional constants are used to support relative positioning based on the container's ComponentOrientation: "before line begins" and "after line ends". For example, in a container where ComponenetOrientation is `ComponentOrientation.LEFT_TO_RIGHT`, "Before line begins" maps to "West", and "After line ends" maps to "East". Mixing the two types of constants is not recommended. Unusual results may show in the graph viewer and at run time, since the relative constants take precedence.

Related concepts:

Layout managers and containers

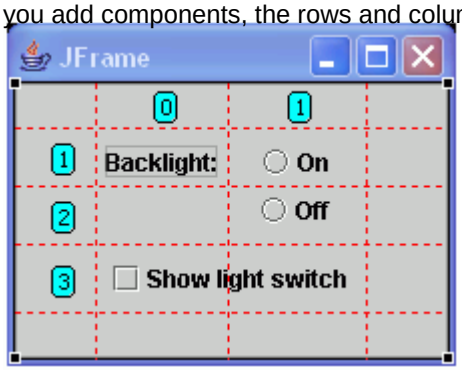
Using GridBagLayout (AWT)

You can use the AWT GridBagLayout with Swing and AWT containers to arrange components using a powerful and flexible grid arrangement.

About this task

The GridBagLayout layout manager arranges its components in rows and columns. The GridBagConstraints object contains information about the row and column where a component is placed, the number of cells the component spans, and how the component is sized and positioned within the cell.

The visual editor uses red, dotted lines to indicate the current borders of the grid. The column and row numbers are also labeled. The rows and columns begin their numbering with 0. So, the upper left corner cell is labeled originally as 0,0. As you add components, the rows and columns are numbered sequentially based on the previously used number.



Note: If a grid number is skipped, it indicates that there are no components that are using that specific grid x or grid y setting. This can occur if you add and remove lots of components or move components around. The GridBagLayout allows x and y values to be skipped, and simply lays out the x,y values in order, relative to other x,y values set for other components, even if values are skipped.

To specify how the grid displays, see Specifying grid display preferences for containers.

Related concepts:

Layout managers and containers

Related tasks:

Specifying grid display preferences for containers

Adding or moving components within GridBagLayout

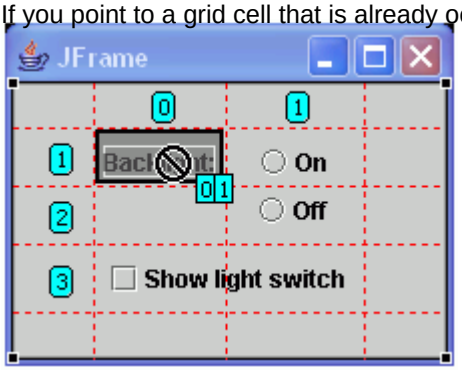
When you build your application visually using GridBagLayout, the visual grid markers help you to move and drop components in the required position.

About this task

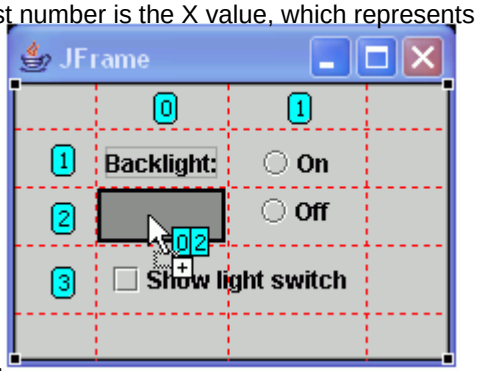
To add or move components within GridBagLayout:

Procedure

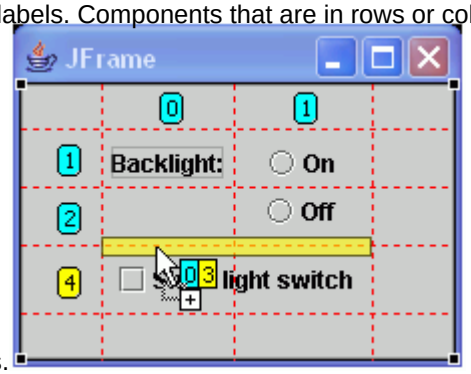
1. Select the Swing component from the palette that you want to add to your GridBagLayout, or click and drag the existing component that you want to move within your GridBagLayout.
2. Move your mouse pointer over the position where you want to drop or move the component:
   - If you point to a grid cell that is already occupied by another component, you cannot move or drop in that position.

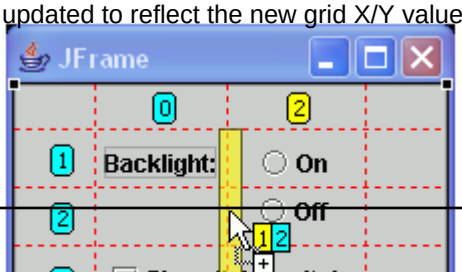


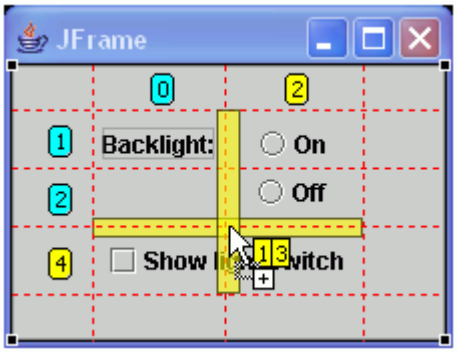
   - If you point to a grid cell that is empty, the cell turns dark gray to indicate the valid drop position. The cursor shows the X and Y grid values for the drop position. The first number is the X value, which represents the column, and the second number is the Y value, which represents the row.



   - If you position your cursor on a grid boundary, a yellow bar indicates that a new column or row number are added.
   - If any new column or row number is being added by your placement, the background color yellow shows on the cursor X/Y indicator and on the column and row labels. Components that are in rows or columns with yellow circles are updated to reflect the new grid X/Y values.







3. Click or release your mouse to drop the component in the position that you selected.

Customizing the layout of components in GridBagLayout

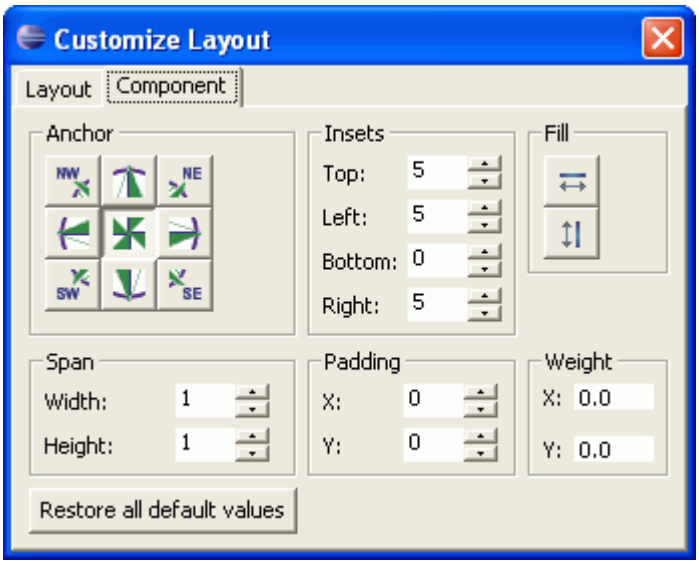
You can use the Customize Layout window to specify anchor position, insets, fill, span, padding, and weight for components within GridBagLayout.

About this task

In GridBagLayout, the visual editor uses default values for constraints, such as fill, anchor, weight x, and weight y, in order to display the component better. For example, when you drop a javax.swing.JTextField, the fill is set to HORIZONTAL (default value for JTextField is NONE), and the weight X value is set to 1 (default for JTextField value is 0).
When switching to a GridBagLayout from another layout manager, the visual editor for Java™ generates constraints for each component based on their existing bounds. An advantage of using GridBagConstraints instead of a null layout is that each component is sized according to its preferred size. So, if the size of the parent window changes or if the preferred sizes change at run time due to different language strings, the layout manager resizea and repositions all the components and make the best use of the available space.
To customize the layout of components within GridBagLayout:

Procedure

1. Select the component that you want to customize within the GridBagLayout, and click the **Customize Layout** toolbar button**Tip:** Select multiple components to set identical anchor, fill, and inset constraints on multiple components at the same time.
   The Customize Layout window shows the anchor position, insets, fill, span, padding, and weight values of the selected components on the GridBagLayout. If multiple components are selected, the anchor value and fill values only show a pressed button if all the components have the same value. The inset values shown are those set for the last component selected.



2. To adjust the anchor position of the component, click one of the compass buttons:
   - **Anchor northwest** - Positions the component in the upper-left corner of the grid cell.
   - **Anchor north** - Positions the component in the top center of the grid cell.
   - **Anchor northeast** - Positions the component in the upper-right corner of the grid cell.
   - **Anchor west** - Positions the component in the left middle of the grid cell.
   - **Anchor center** - Positions the component in the center of the grid cell.

- **Anchor east** - Positions the component in the right middle of the grid cell.
- **Anchor southwest** - Positions the component in the lower-left corner of the grid cell.
- **Anchor south** - Positions the component in the bottom center of the grid cell.
- **Anchor southeast** - Positions the component in the lower-right corner of the grid cell.

3. To adjust the padding between the grid cell border and the component, enter a value (in pixels) for any of the following fields:
   - **Top** - Sets the inset, or padding, above the component.
   - **Left** - Sets the inset, or padding, to the left of the component.
   - **Bottom** - Sets the inset, or padding, below the component.
   - **Right** - Sets the inset, or padding, to the right of the component.
4. To adjust how much the component fills the grid cell, click either or both of the fill buttons:
   - **Fill horizontal** - Specifies that the component occupies the full width of the grid cell.
   - **Fill vertical** - Specifies that the component occupies the full height of the grid cell.
5. To specify how many cells a component spans, enter values for **Width** and **Height**:
   - **Width** - Specifies the number of columns the component occupies (the X axis).
   - **Height** - Specifies the number of rows the component occupies (the Y axis).
6. To specify internal padding for a component, enter values (in pixels) for the following fields:
   - **X** -
   - **Y** -
7. To specify how to distribute extra space across rows or columns, enter values (in relative numeric values, in relation to the weights specified for other components) for the **X** and **Y** fields.The GridBagLayout manager calculates the weight of columns and rows to be the maximum weightx and weighty of all the components in a row and column, respectively. If the resulting layout is smaller than the area it needs to fill, the extra space in the layout is distributed to the columns and rows in proportion to the weights specified. A cell that has a weight of zero receives no extra space.
   For example, if a component in column 1, row 1 has a weight X of 3, and a component in column 2, row 1 has a weight X of 1, the total of the weight X values in row 1 add up to 4, and column 1 receives 75% of the extra space when space is distributed.
   - **X** - Specifies the weight for distribution of extra space between columns.
   - **Y** - Specifies the weight for distribution of extra space between rows.

Spanning components across GridBagLayout cells

You can drag a component in a GridBagLayout to make it span multiple cells. You can also specify the width and height of a component using Properties view or the Customize Layout window.

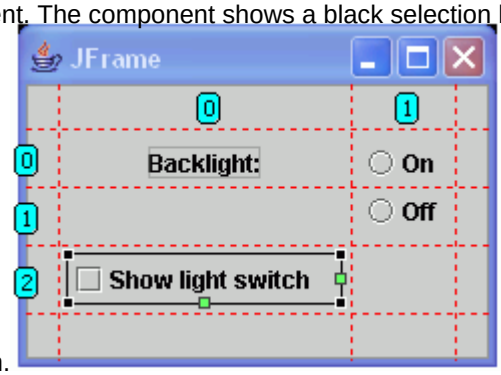
About this task

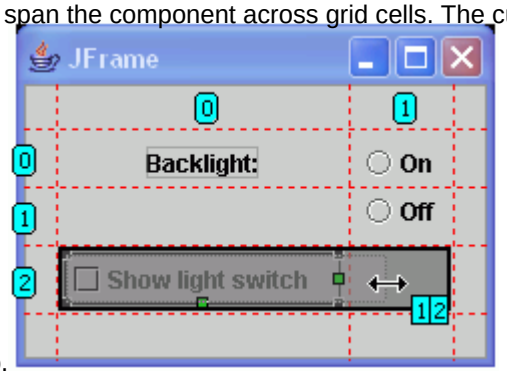
By default, when you drop a component into a GridBagLayout, the visual editor sets the x and y values to assign it to a cell. However, two additional constraints available for a component are gridheight and gridwidth, which are properties on the GridBagConstraints object that is instantiated for each component.

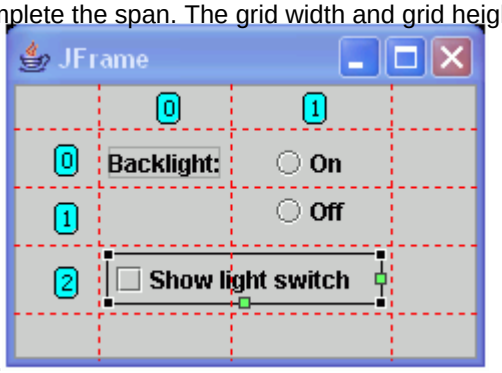
- The gridheight constraint is an integer that indicates the number of rows that the component spans.
- The gridwidth constraint is an integer that indicates the number of columns that the component spans.

To span a component across multiple cells in a GridBagLayout:

Procedure

1. Select the component. The component shows a black selection border around it with two green boxes, or handles, on the right and bottom. 

2. Click and drag a handle to span the component across grid cells. The cursor shows the X/Y coordinates for the grid cell where you are spanning to. 

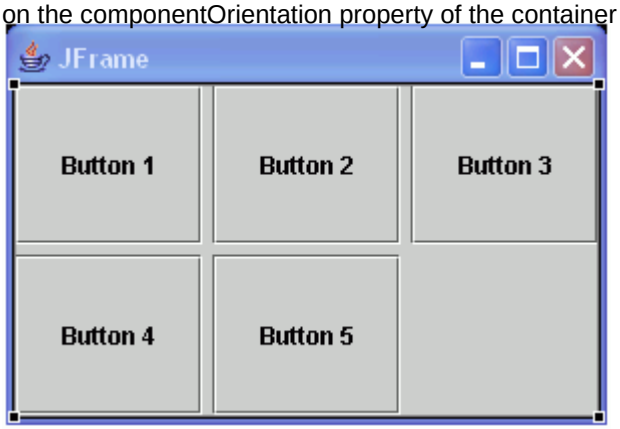
3. Release the mouse button to complete the span. The grid width and grid height constraints are set based on the number of rows or columns you spanned. 

Using GridLayout (AWT)

You can use the visual editor for Java™ to lay out Swing components using GridLayout.

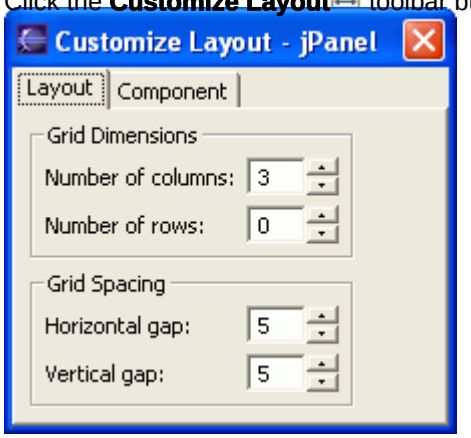
About this task

The layout manager java.awt.GridLayout lays out components in a grid of equal-sized rectangles in rows and columns. You can set the number of rows or columns that a GridLayout contains. As you add components, they are added in order based on the componentOrientation property of the container that is using GridLayout (RIGHT_TO_LEFT or LEFT_TO_RIGHT).



Procedure

- To set the grid dimensions for a GridLayout:
  1. In the Design view or Java Beans view, select the component that is using GridLayout.
  2. Click the **Customize Layout** toolbar button. The Customize Layout dialog opens.



  3. On the Layout notebook tab, enter a value for *either* **Number of columns** or **Number of rows**.**Important:** Specifying the number of columns affects the layout only when the number of rows is set to zero. If you set both the number of rows and the number of columns to non-zero values, the number of columns specified is ignored. In this case, the number of columns is determined from the specified number of rows and the total number of components in the layout. For example, if you specify three rows and two columns, and you add nine components to the layout, they are displayed as three rows of three columns.

- To set spacing for components with a GridLayout:
  1. Open the Customize Layout dialog for the component that is using GridLayout.
  2. On the Layout notebook tab, enter values for **Horizontal gap** and **Vertical gap**. Horizontal gap sets the distance in pixels between columns, while vertical gap sets the spacing between rows.

- To add a component to a GridLayout:
  1. Select a Swing component on the palette.
  2. Move your cursor over the Design view. A black bar provides feedback to indicate the position where the component are added.

3. Click the GridLayout container. The component is added to the grid of components.

- To move a component within a GridLayout:

1. On the Design view, click and drag the component that you are moving. As you move your cursor, a black bar provides feedback to indicate the position where the component are moved.
2. Release your mouse to move the component.

Related concepts:

Layout managers and containers

Using BoxLayout (Swing)

You can use the visual editor to layout Swing components using BoxLayout on the X or Y axis.

About this task

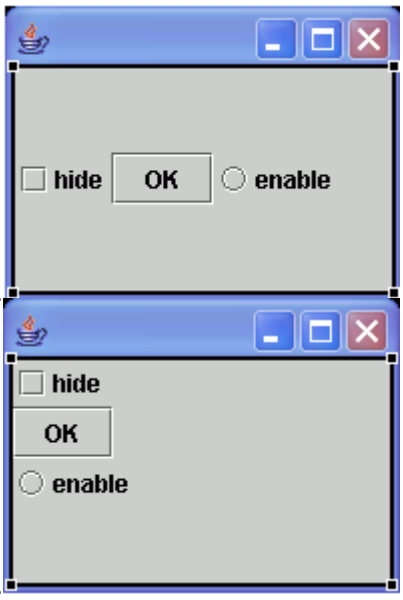
BoxLayout is a Swing layout manager that allows multiple components to be laid out either vertically or horizontally. The components donot wrap, so, for example, a vertical arrangement of components stay vertically arranged when the frame is resized.

For all directions, components are arranged in the same order as they were added to the container. BoxLayout attempts to arrange components at their preferred widths (for horizontal layout) or heights (for vertical layout). For a horizontal layout, if not all the components are the same height, BoxLayout attempts to make all the components as high as the highest component. If that is not possible for a particular component, then BoxLayout aligns that component vertically, according to the component's Y alignment. By default, a component has a Y alignment of 0.5, which means that the vertical center of the component should have the same Y coordinate as the vertical centers of other components with 0.5 Y alignment. Similarly, for a vertical layout, BoxLayout attempts to make all components in the column as wide as the widest component. If that fails, it aligns them horizontally according to their X alignments.

To work with BoxLayout:

Procedure

1. Select a container in the Design view or Java™ Beans view.
2. In the Properties view, set the **layout** property to one of the following options:



   - **BoxLayout(X_AXIS)** - Components are laid out horizontally from left to right.
   - **BoxLayout(Y_AXIS)** - Components are laid out vertically from top to bottom.
3. Drop components from the palette onto the BoxLayout. A black bar next to the cursor indicates the placement of a component that is about to be added to the BoxLayout.

Related concepts:

Layout managers and containers

Using CardLayout (AWT)

The AWT CardLayout layout manager arranges components in a stack, like a stack of cards.

About this task

CardLayout treats each component in the container as a card, with only one card visible at a time. The ordering of cards is determined by the container's own internal ordering of its component objects, and the first component added to a CardLayout object is the visible component when the container is first displayed.
To work with CardLayout layout manager:

Procedure

1. In the Java™ Beans view or Design view, select a Swing container.
2. In the Properties view, set the **layout** property to CardLayout.
3. Use the palette to drop components onto the CardLayout. The components are stacked liked cards, with only the most recently added or currently selected component visible.
4. To affect the spacing of the components in the CardLayout, you can set the following properties in the Properties view:
   - **horizontal gap** - specifies the size in pixels of the left and right margins between the components and the border of the container.
   - **vertical gap** - specifies the size in pixels of the top and bottom margins between the components and the border of the container.

Related concepts:

Layout managers and containers

Using FlowLayout

The FlowLayout layout manager arranges components in a left-to-right flow, with wrapping lines. The advantages of the flow layout manager include its ease of use, and the guarantee that each component can be seen.

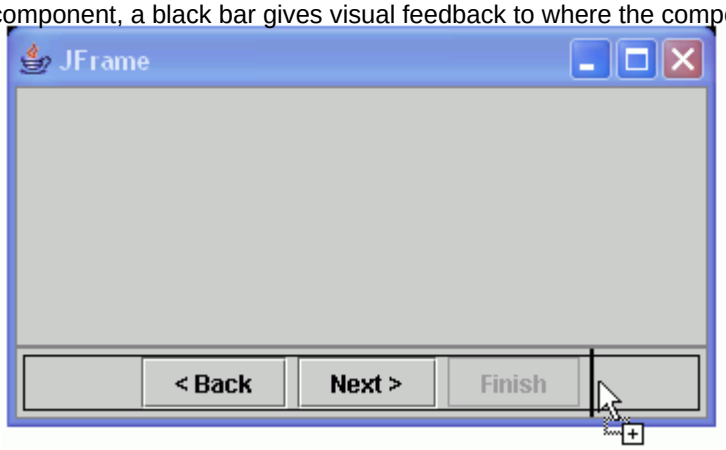
About this task

The FlowLayout layout manager (java.awt.FlowLayout) dynamically sizes each component according to its preferred size. It positions components so that they are evenly spaced. Flow layouts are typically used to arrange buttons in a panel. It arranges buttons left to right until no more buttons fit on the same line. Each line is centered by default, unless alignment is set to left or right.

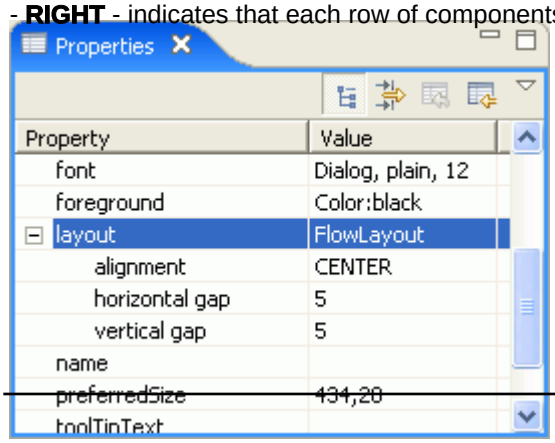
FlowLayout is the default layout manager for a JPanel.

Procedure

- To add a component to a FlowLayout:
  1. Select a Swing component from the palette.
  2. Point your cursor over the container that is using FlowLayout.
     - A black rectangle is drawn inside the boundaries of the container to provide feedback for where you are about to drop the component.
     - If the container already includes a component, a black bar gives visual feedback to where the component is dropped relative to the existing components:



- To reorder components in FlowLayout, do one of the following steps:
  - In the Design view, click and drag a component to a new position in the FlowLayout. The black bar gives visual feedback about the new position for the component.
  - In the Java™ Beans view, click and drag a component and move it above or below another component. Because FlowLayout determines behavior by the order of the `add()` methods, the Design view reflectsthe reordering.
  - You can explicitly set the position of a component using an index as an argument to the `add()` method.
- To specify the alignment of the flow layout, set the **alignment** property for the FlowLayout to one of the following options:
  - **LEFT** - indicates that each row of components is left-aligned
  - **CENTER** - indicates that each row of components is centered
  - **RIGHT** - indicates that each row of components is right-aligned



- To set the spacing between components in the FlowLayout, enter values for the following FlowLayout properties:
  - **horizontal gap** - specifies the distance in pixels between components in the same row
  - **vertical gap** - specifies the distance in pixels between rows of components

Related concepts:

Layout managers and containers

Using null layout

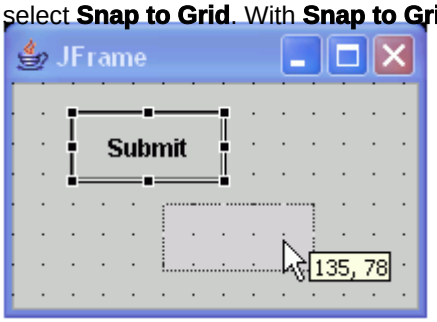
Layout managers help control the size and position of all components in a container. In containers using null layout, each component defines its own bounds to determine size and location.

About this task

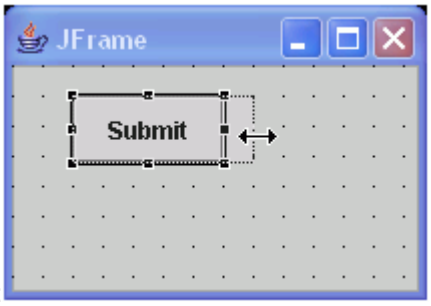
The visual editor provides grid markers that help you visually align components, and you can snap components to the grid. When you set to null the layout for an existing container that was previously using a layout manager, the Design view does not change in appearance. The visual editor calculates and sets the bounds for each component using the setBounds() method.

Procedure

- To specify universal settings for how the null grid markers display, see Specifying grid display preferences for containers. To customize the spacing of the grid markers for the selected container, open the **Customize Layout** window and specify the width, height, and margin sizes for the grid.
- To force items to snap to the grid markers as you drop them or move them on the null layout, right-click the container and select **Snap to Grid**. With **Snap to Grid** selected, you can easily align components visually.

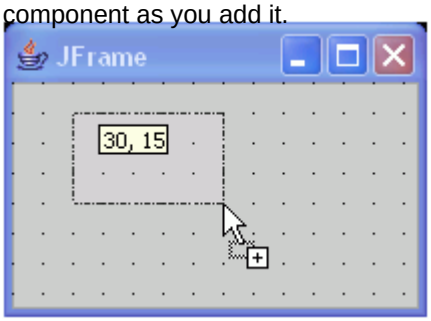


To override the snapping to grid for a component, hold down the Alt key while you drop or move the component. This allows you to place a component between grid markers.



- To resize a component in null layout, click and drag the handles of the component:

You can also set the size of a component as you add it from the palette. Simply hold down the left mouse button when you drop the component in the Design view, and drag the mouse pointer. In any other layout manager, you cannot size a component as you add it.



Example

Tip: You can start with the layout manager set to null and then change to a layout (like the GridBagLayout) to get the constraint settings calculated by the visual editor for Java™. The advantage of having each component control its bounds

is that you can be sure that a component is always the same size and in the same position at run time. The disadvantage of having fixed size and height is that the components does not adjust their bounds if the application is resized by the user. In addition, labels on components such as buttons or check boxes need to change size based on font or locale.

Related tasks:

Specifying grid display preferences for containers

Aligning components using X/Y alignment a

When you set the layout property for a component to null, you can use the Customize Layout window to align components based on their boundaries.

About this task

With layout set to null, each component is independently placed and sized on the user interface based on its bounds. To avoid the tedious and error-prone task of individually updating each component's bounds, you can use the visual editor to align and adjust the sizes and placement of components based on relationships.

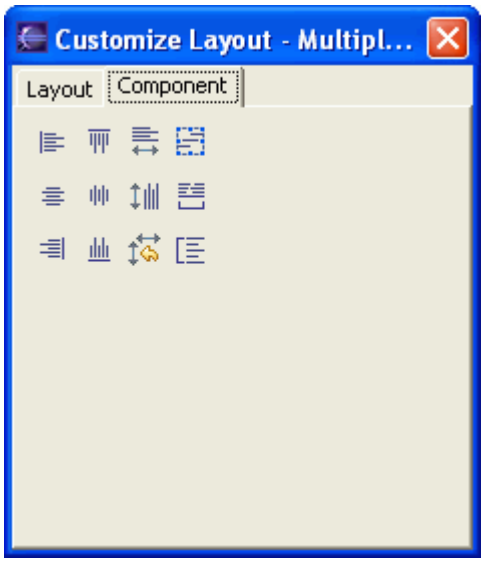
When layout is set to null, the Customize Layout window has a number of actions that work on a group of components. You can:

- Align components based on edges
- Align components based on center points
- Make components the same width and height
- Distribute components evenly, both vertically and horizontally, across a given space

To align, resize, and distribute components in null layout:

Procedure

1. In the Design view or Java™ Beans view, select two or more components that you want to align or adjust in relation to each other. You can use the marquee selection tool or hold the CTRL key to select multiple components. The last component that you select is the control component, or anchor. The anchor component is indicated by black resize handles, while the other selected components have white handles.
2. Do one of the following steps to open the Customize Layout window:
   - Right-click and select **Customize Layout** from the pop-up menu.
   - Click **Customize Layout** in the toolbar.



3. To align the components with each other, use the following buttons:
   - **Align left** - aligns the components with the left edge of the anchor component.
   - **Align center** - aligns the components with the center of the anchor component along a vertical plane.
   - **Align right** - aligns the components with the right edge of the anchor component.
   - **Align top** - aligns the components with the top edge of the anchor component.

- **Align middle** - aligns the components with the middle of the anchor component along a horizontal plane.
- **Align bottom** - aligns the components with the bottom edge of the anchor component.

4. To adjust the size of the selected components, use the following buttons:
   - **Match width** - makes the components the same width as the anchor component.
   - **Match height** - makes the components the same height as the anchor component.
5. To distribute the spacing of the components, first do one of the following to specify the bounds of the distribution area:
   - Click nothing to use the container as the distribution area (which is the default behavior), or click **Hide distribute box** if the box is already showing.
   - Click **Show distribute box** and drag the handles of the box to create the required size for your distribution.

   Then, use one of the following buttons to distribute the components:
   - **Distribute horizontally** - makes the components the same width as the anchor component.
   - **Distribute vertically** - makes the components the same height as the anchor component.

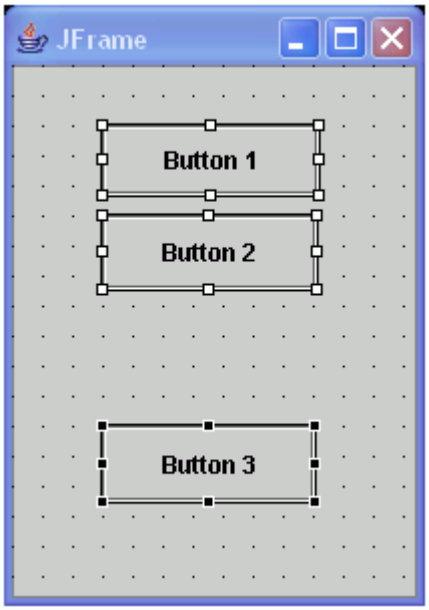
Example

For example, you can select three JButton components and specify that they are the same width and aligned with each other on the left bound. The last selected JButton component is used as the anchor, or control, for the other two buttons. The **Align left** action aligns the JButton components with the left edge of the last selected JButton component, and the **Match width** action makes the selected JButton components the same width as the control JButton component.

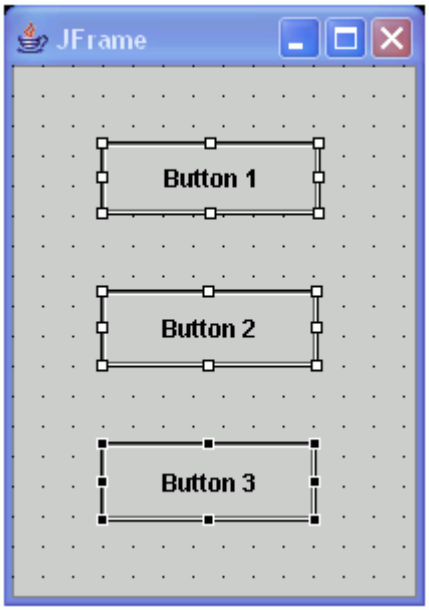
The alignment buttons on the Customize Layout window are only enabled if the following conditions are true:

- Two or more components are selected, and the parent container is not using a layout manager (layout is set to null).
- The components have no parent and have been placed on the Design view directly.

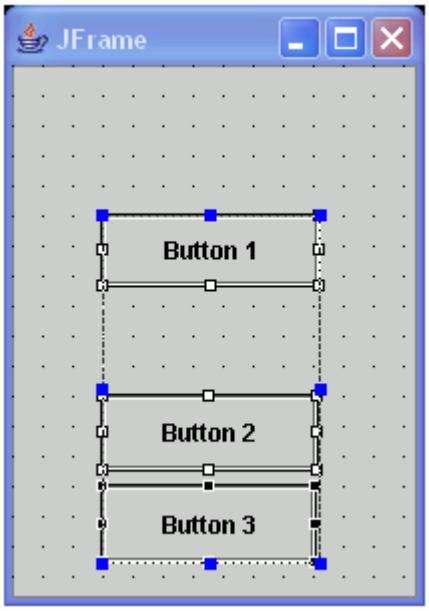
In addition to aligning components and matching width and height, you can distribute components. When components are distributed, their positions are changed so that they are evenly spaced within a bounding box defined by their parent container. For example, before clicking the **Distribute vertical** button , three buttons have uneven spacing, as shown in the following image:



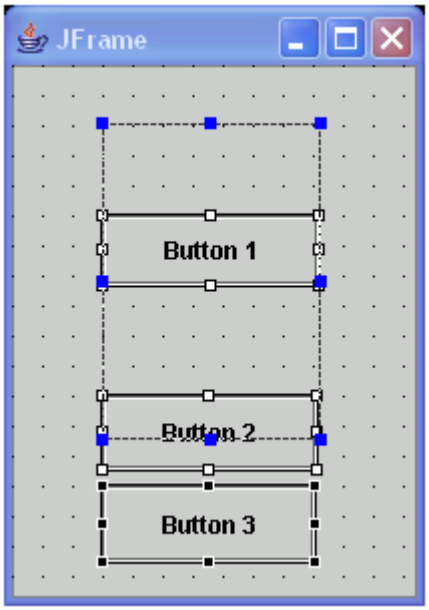
After clicking **Distribute vertical**, the components are spaced evenly, as shown in the following image:



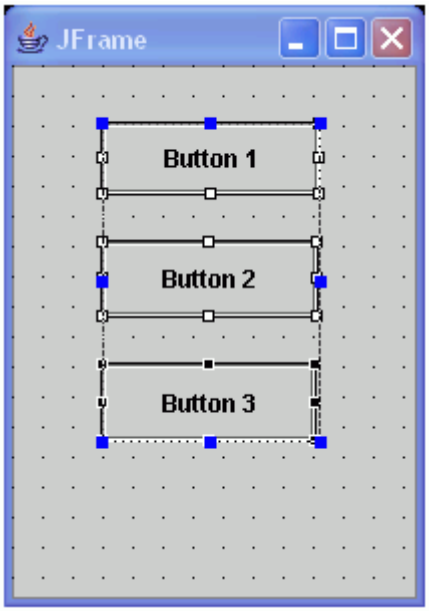
Notice that the three buttons are distributed within the height of the parent (the top and bottom edge of the frame).
For more control over the area used for the distribution, you can enable the distribution box by clicking the **Show distribute box** button. When you click this button, a box is drawn around the area of all the selected components. You can move and resize this box using its handles. When you use a distribution box and the box is active, the distribution box defines the area that is used to reposition the controls. Without a distribution box, the parent container defines the area used for distribution. The following image shows a distribution box:



You can move and resize the distribution box, as the following image shows:



If you do a vertical distribution while using the distribution box, the distribution takes place within the boundaries of the box, rather than the parent container, as is shown in the following image:



Working with SWT in the visual editor

The visual editor helps you develop applications using Standard Widgets Toolkit (SWT).

About this task

When you use the visual editor to create a new visual class based on an SWT composite or shell, the SWT library is added to the Java™ build path of the project, and the visual editor palette includes the supported SWT controls and containers. You can also manually add the library on the Java Build Path page of the project properties (right-click the project and select **Properties**). Remember that SWT controls and containers can only be used with SWT applications.

The topics in the SWT Programmer Reference section are taken from the Platform Plug-In Developer Guide and discuss SWT API reference in more detail.

Creating an RCP view in the visual editor

The visual editor for Java™ supports the creation and development of Rich Client Platform (RCP) views.

Before you begin

Before you can create a Java visual class, you must create a Java project or plug-in project where you can place the visual class. A plug-in project that is enabled for Rich Client Platform (RCP) application development is required if you want to create a RCP view or editor.

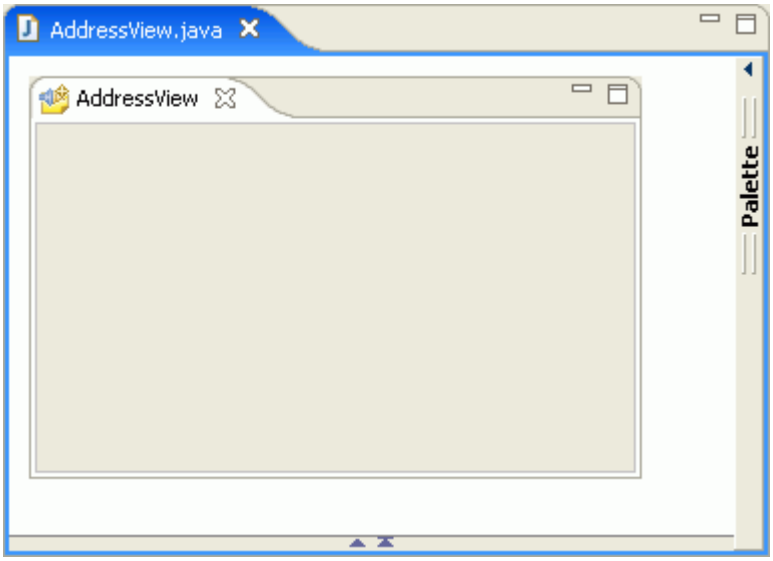
About this task

An RCP view is a class that extends org.eclipse.ui.part.ViewPart. You can use the New Java Visual Class wizard to quickly create a new visual class that extends ViewPart. The wizard creates the ViewPart and adds an SWT composite as the default container. You can then use the SWT controls and containers to design the contents of the view. **Tip:** The visual editor uses your workbench preferences for View tab position (top or bottom) and tab style (traditional or non-traditional) during design and when you run the view as a Java bean.

To create an RCP view:

Procedure

1. Complete the steps for Creating a new Java visual class, specifying RCP View in the **Style** list. The new Java class opens in the visual editor showing the ViewPart with a composite.



2. Use the SWT controls and containers to design the contents of the view.

What to do next

You can test the view independently by running it as a Java bean (**Run** > **Run As** > **Java Bean**).To incorporate the view into your plug-in, you need to register the view as an extension point in your project and include it in the proper perspective to display correctly as part of the RCP application.

Related tasks:

Creating a new Java visual class

Creating an RCP editor in the visual editor

Creating an RCP editor in the visual editor

The visual editor for Java™ supports the creation and development of Rich Client Platform (RCP) editors.

Before you begin

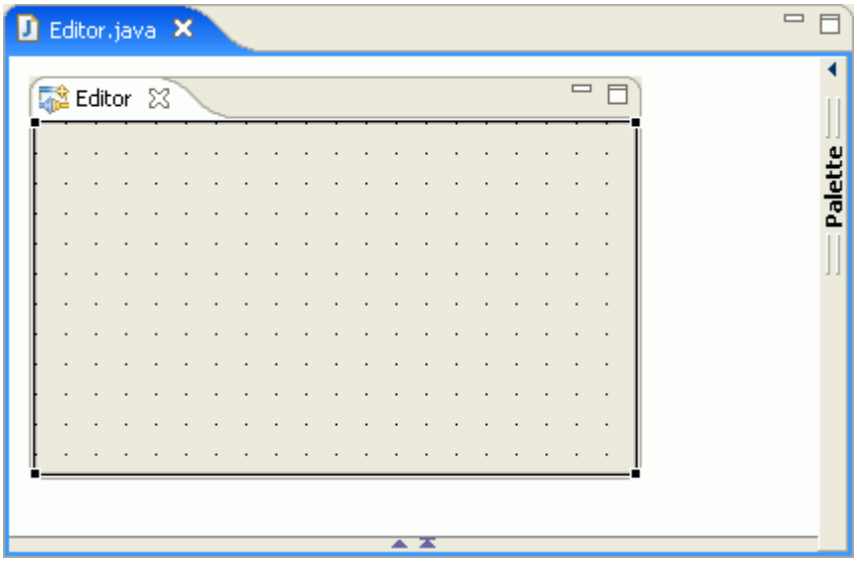
Before you can create a Java visual class, you must create a Java project or plug-in project where you can place the visual class. A Plug-in project that is enabled for Rich Client Platform (RCP) application development is required if you want to create a RCP view or editor.

About this task

An RCP editor is a class that extends org.eclipse.ui.part.EditorPart. You can use the New Java Visual Class wizard to quickly create a new visual class that extends EditorPart. The wizard creates the EditorPart and adds an SWT composite as the default container. You can then use the SWT controls and containers to design the contents of the editor.To create an RCP editor:

Procedure

1. Complete the steps for Creating a new Java visual class, specifying RCP Editor in the **Style** list. The new Java class opens in the visual editor showing the EditorPart with a composite.



2. Use the SWT controls and containers to design the contents of the editor.

What to do next

You can test the editor independently by running it as a Java bean (**Run** > **Run As** > **Java Bean**). To incorporate the editor into your plug-in, you need to register the editor as an extension in your project and programmatically display it as part of your application. For more information about adding an editor extension point to your Rich Client Platform application or rich client plug-in, see org.eclipse.ui.editors

Related tasks:

Creating a new Java visual class

Creating an RCP view in the visual editor

Changing the parent of an SWT Shell

You can use the visual editor to change the parent of an SWT Shell, to quickly create dialogs and other shells from an SWT application.

Before you begin

In order to change the parent of an SWT Shell, your visual class must include another Shell.

About this task

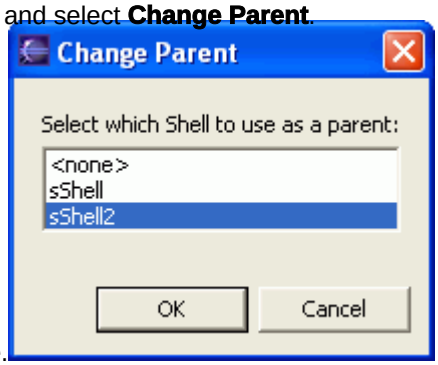
When you drop a new Shell on an SWT visual class, by default the Shell is created without a parent:`sShell1 = new Shell();`

If a Shell specified a parent, it would look like this:`sShell1 = new Shell(sShell);`

The visual editor provides a dialog to help you specify or change the parent of an SWT Shell.

Procedure

1. In the Design view, right-click the Shell whose parent you want to change, and select **Change Parent**.



2. In the Change Parent window, select the parent Shell that you want to use.
3. Click **OK**.

Results

The Java™ code is updated to reflect the new parent. The following code snippet shows an example:`sShell1 = new Shell(sShell2);`

Working with Swing in the visual editor

The visual editor for Java™ gives you visual control in working with Swing components. You can also customize the Swing look and feel.

About this task

For more information about Swing and the Java platform in general, see the java.sun.com Web site for articles and API documents, including the following:

- Java™ Platform, Standard Edition 6 API Specification
- Swing API document (package javax.swing)

Several Swing components have special behavior in the visual editor that is worth noting.

Changing the Swing look and feel

Swing provides the capability of changing the look and feel for your components. You can change the default Swing look and feel in the visual editor preferences.

About this task

Procedure

1. Open the Preferences window by clicking **Window** > **Preferences**, and select **Java** > **Visual Editor**.
2. On the **Appearance** tab, select the look and feel that you want to use.

Related tasks:

Adding a new Swing look and feel

Adding a new Swing look and feel

With the visual editor for Java™, you can use a look and feel other than those provided by Swing.

About this task

If you develop your own look and feel, or acquire a commercial or free look and feel, you can add the look and feel to the visual editor.

To add a new Swing look and feel:

Procedure

1. Add the JAR file with your look and feel to the Java Build Path.
   A. Download or create a new look and feel and save to a local directory.
   B. Extract to the Project directory where you want to include the new Look and Feel.
   C. In the Enterprise Explorer, right-click the project name.
   D. Click **Properties** > **Java Build Path** > **Libraries**.
   E. Click **Add External Jars** and browse to the JAR file that contains the look and feel, then click **OK**. Now, you are ready to apply the new Look and Feel to your application.
2. Add the look and feel to the visual editor preferences:
   A. Open the Preferences window by clicking **Window** > **Preferences**, and select **Java** > **Visual Editor**.
   B. Next to the **Swing Look and Feel** table on the Appearance tab, click **New**.
   C. Provide a **Name** and the **Class** for the new look and feel. The LookAndFeel class name is in the documentation for the look and feel. If you cannot locate the look and feel class name, you can find the class name by right-clicking the JAR file that you added to your project, and pressing **F4** to open its hierarchy.
   D. Click **OK** to close the Look and Feel dialog.
3. Select the check box next to the new look and feel.
4. Click **OK** to save your preferences.
5. Close your application, then open it again to see the new Look and Feel.

Related tasks:

Changing the Swing look and feel

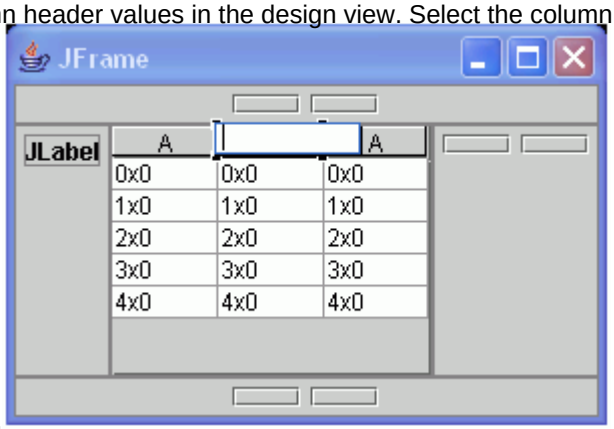
Swing JTable and TableColumn

You can use the visual editor to arrange data in rows and columns using Swing JTable and TableColumn.

When you drop a Swing JTable onto your design from the palette, the visual editor uses the table default table model (javax.swing.table.DefaultTableModel) to visualize the table. If you write your own Java™ code to assign a table model, the visual editor parses the source and attempt to instantiate and apply the table model to the instance of the JTable in the Design view.

A	B	C	D	E
0x0	0x1	0x2	0x3	0x4
1x0	1x1	1x2	1x3	1x4
2x0	2x1	2x2	2x3	2x4
3x0	3x1	3x2	3x3	3x4
4x0	4x1	4x2	4x3	4x4

- If you want to manually add and define the columns, you need to set the autoCreateColumnsFromModel property to false. Then you can drop TableColumn components from the palette onto the JTable. If the autoCreateColumnsFromModel is set to true, the JTable creates as many columns as the table model returns from the `getColumnCount()` method.
- If you add your own columns, you can directly edit the column header values in the design view. Select the column header and double click, then type the column header value.

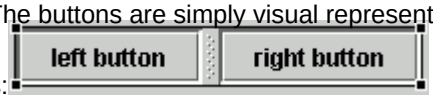


- You can also reorder the columns by selecting a column in the design view then dragging and dropping it in the new position.
- The JTable must be in a JScrollPane for the table headers to be displayed. The visual editor provides a JTable on JScrollPane component that you can drop onto your design.

JSplitPane and its components

The visual editor has special behavior to make it easy to work with a Swing JSplitPane container.

You can use the JSplitPane container to create two components with a split bar between them. When the application is running, a user can drag the split bar to adjust the space on the screen.

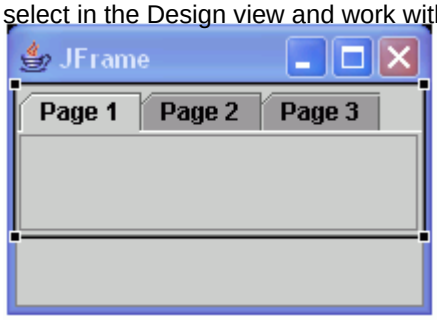
- When you first drop a JSplitPane container onto your design, the Design view shows a representation of two components separated by a split bar. The buttons are simply visual representations for the two sides, where you can drop your own components or containers: 

- The first component that is dropped onto JSplitPane occupies the left pane. The second component that is dropped occupies the right pane. If both panes in the JSplitPane are already occupied, the visual editor does not allow you to drop the component.
- The JSplitPane includes an orientation property. If you set the orientation to VERTICAL_SPLIT, the split bar runs horizontally, making top and bottom components. The default is HORIZONTAL_SPLIT.
- The visual editor generates code that uses a `set` method:`ivjJSplitPane.setTopComponent(getIvjJButton());`
  However, the visual editor also recognizes the following code method:
  `ivjJSplitPane.addComponent(getIvjJButton(),"top");`
- The visual editor renders JLayeredPane indexes in a single layer. In the case of multilayer and indexes, the visual editor might not show the components in the correct order.

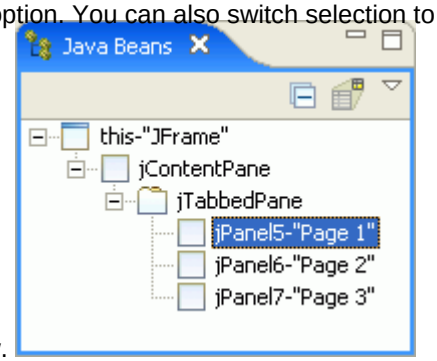
JTabbedPane and its pages

To create a notebook of tabbed pages, you can use the Swing JTabbedPane (`javax.swing.JTabbedPane`).

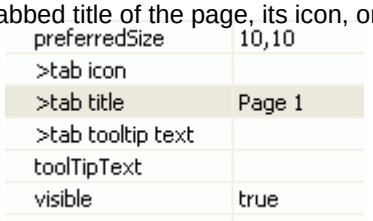
When you drop a component onto a JTabbedPane, the component becomes a new tabbed page. The component is also shown in the Java™ Beans view as a child of the JTabbedPane. Only one of the pages is the active page that you can select in the Design view and work with.



- To change the active page in the Design view, you can right-click the JTabbedPane and select a page using the **Switch To** option. You can also switch selection to a different page by selecting the component of the page in the Java Beans view.



- To change the tabbed title of the page, its icon, or its hover help, select the component and change the values in the Properties view.



- To reorder the tabbed pages, in the Java Beans view drag the component to the new position in the JTabbedPane.

Handling events with the visual editor

The Java™ Beans view of the visual editor for Java helps you work with events for your visual application.

Events, listeners, and adapter classes

Java™ beans events are signaled when an activity occurs, such as a button being pressed or a window being closed. The visual editor for Java shows events in the Java Beans view, and you can use this view to add and remove events.

The list of defined events for a Java bean is described in its BeanInfo class, which also controls the commonly used or preferred events.

You can add an event to a Java bean if you want something to happen (such as a database update when a button is pushed) when the Java bean is generated. The Java Bean raising the event is the source, and the object that gets called back when the event is raised is known as the *listener*. Each Java bean has an interface that allows it to notify the listeners for each event, as well as methods to add and remove listeners.

Typically, if the source Java bean has a method XXX, there is a listener interface, XXXListener, and two methods. It is important that XXXListener extends java.util.EventListener. If it does not, and unless a specific BeanInfo is supplied, the event cannot be discovered.

- addXXXListener(XXXListener aListener)
- removeXXXListener(XXXListener aListener)

The methods on the XXXListener interface itself depend on the semantics of the event, but the convention is that their signature is `void <eventOccurenceMethodName>(<EventStateObjectType> evt);`. It is important that XXXListener extends java.util.EventObject. If it does not, and unless a specific BeanInfo is supplied, the event cannot be discovered.

An example of an event is the Java bean java.awt.Component, which raises events when the mouse moves over it. The listener interface, java.awt.event.MouseMotionListener, implements the following two methods:

- void mouseDragged(MouseEvent evt);
- void mouseMoved(MouseEvent evt);

To add a mouse listener, the java.awt.Component has the following two methods:

- public void addMouseListener(MouseListener evt);
- public void removeMouseListener(MouseListener listener);

The second style of event is generated by a Java bean when a property value changes. An example of this is the 'enabled' property on javax.swing.JButton. A property that fires an event when its value is changed is known as a bound property. Instead of having a separate listener interface for each bound property, there is a generic listener interface `java.beans.PropertyChangeListener` which has a single callback method `void propertyCanged(PropertyChangeEvent evt);` The argument PropertyChangeEvent has three methods that can be queried by the receiver of the method:

String getPropertyName()	The name of the property that was changed on the Java bean that caused the event to fire
Object getNewValue()	The new value of the property
Object getOldValue()	The value of the property before it was changed

To register interest in the property changes of the Java bean, there are two methods: `void addPropertyChangeListener(PropertyChangeListener listener);void addPropertyChangeListener(String propertyName, PropertyChangeListener listener);`

The first of these methods is always present on a Java bean that has bound properties. However, the second is optional and depends on the style of event registration used by author of the Java bean. For example, AWT components use the first style of property change registration, while Swing components use both styles.

To use an event there are three objects:

1. The Java bean that raises the event (the source)
2. The class that receives notification from the source (the listener)
3. The class that implements the logic that occurs when the listener is called back.

Typically the last two are combined, so that the class that executes the logic either implements the listener interface directly or uses an inner class. The styles of code that the visual editor for Java recognizes and generates are covered in the section on Event Code Generation.

Adapter classes

Many listener interfaces have more than one callback method. An example is java.awt.FocusListener that has two methods; focusGained(java.awt.FocusEvent event) and focusLost(java.awt.FocusEvent event). When creating a listener class that implements the interface the Java compiler insists that all the interface methods are implemented, which often results in many empty methods being created to satisfy its requirements when only one or some of its methods actually contain code. The following statement shows a FocusListener being used to perform some logic when a Java bean gains focus. However, an empty focusLost method must be provided.

```
Java bean.addFocusListener(new java.awt.event.FocusListener() {
   public void focusGained(java.awt.event.FocusEvent e) {
       doFocusGainedCode();
    }
       public void focusLost(java.awt.event.FocusEvent e) {
    }
});
```

To avoid having many empty listener methods for many listeners, Adapter classes are provided. These implement the listener interface, and provide empty (no operation) implementation of its methods. The advantage is that the listener can extend these, and only specialize methods of choice without having to provide default implementations for the rest (these are inherited from the Adapter).

```
Java bean.addFocusListener(new java.awt.event.FocusAdapter() {
    public void focusGained(java.awt.event.FocusEvent e) {
        doFocusGainedCode();
        }
  });
```

Viewing events for a component

The Java™ Beans view shows all the events set on the components in your visual class.

About this task

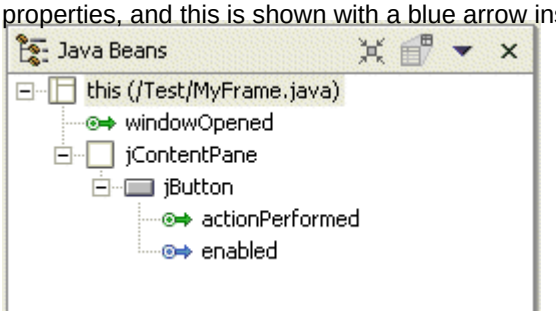
The Java Beans view has three modes for showing events:

- **No Events**
- **Show Events**
- **Expert Events**

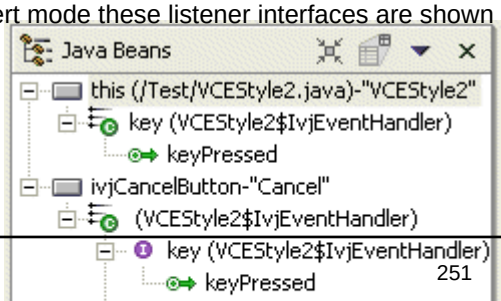
To view the events on a component:

Procedure

1. On the Java Beans view toolbar, click the menu (arrow) button.
2. Select **Show Events** or **Expert Events**.
   - **Show Events**When **Show Events** is selected, the events used by each component are shown as children in the tree. Events are marked with green arrows, property change events are marked with blue arrows.
     For an event to be used by a component, there must be a registered listener, and the callback method must have some code within it. The list of recognized source code patterns used by the visual editor are described in the source code patterns for events section. In the following image, the Java Beans view shows a JFrame with a windowOpened event, and it shows a button with an actionPerformed event. The button also has a PropertyChangeListener for its enabled properties, and this is shown with a blue arrow instead of green.



     Each event is made up of the source component (the JFrame or JButton in the example being used), a class that implements the listener interface that is added to the source using addXXXListener(XXXListener), and some code within the body of the callback method.
   - **Expert Events**In the **Expert Events** mode, each listener for the component is shown as a child tree node, and the events are shown beneath each listener. This increases the number of items in the tree, but it shows more detail about how the events are attached to the components. The option of the two modes lets you decide which level of detail you want to work with.
     In expert mode the icon used for the listener shows the type of listener class. If the listener is an anonymous inner class that implements the listener interface, the icon is used, and if the listener is an anonymous inner class that extends a listener adapter class, the icon is used.
     In addition to anonymous inner classes being used for listeners, named classes and shared listeners are also parsed and recognized by the visual editor. If the listener is not anonymous in expert mode, then the icon is . If the listener is shared by more than one component, the is used. If the listener class is used by a single event, then these are listed as children of the listener. However, if the listener class is used by more than one event listener interface for the component, in Expert mode these listener interfaces are shown as separate children of the listener class, as shown in



The source for this is shown in the following code statement. The inner listener class IvjEventHandler is used once by the first button (this) for a keyPressedEvent, and twice by the "Cancel" button, once for keyPressed (that is part of the key event) and another time for actionPerformed (that is part of the action event).

```
class IvjEventHandler implements java.awt.event.ActionListener, java.awt.event.KeyListener {    public void
actionPerformed(java.awt.event.ActionEvent e) {
  if (e.getSource() == VCEStyle2.this.getCancelButton())
  connEtoC3(e);
};
public void keyPressed(java.awt.event.KeyEvent e) {
   if (e.getSource() == VCEStyle2.this.getCancelButton())
   connEtoC2(e);
if (e.getSource() == VCEStyle2.this)
   connEtoC1(e);
};
public void keyReleased(java.awt.event.KeyEvent e) {};
public void keyTyped(java.awt.event.KeyEvent e) {};
};
```

Related concepts:

The Java Beans view

Adding events to a component

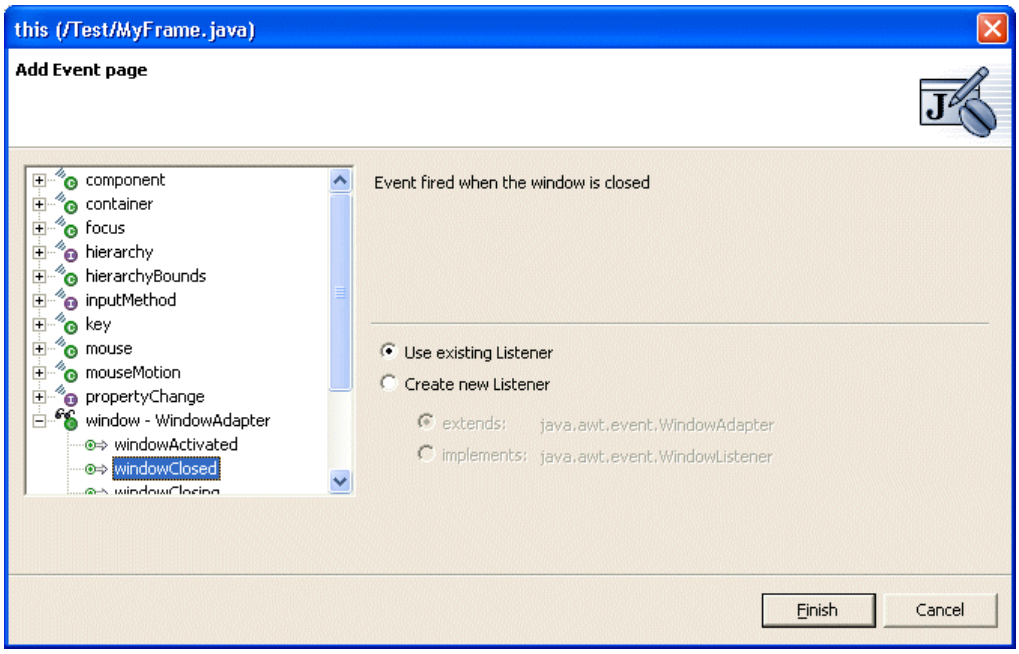
You can add an event to a component in the Design view or Java™ Beans view of the visual editor for Java.

About this task

To add an event to a component:

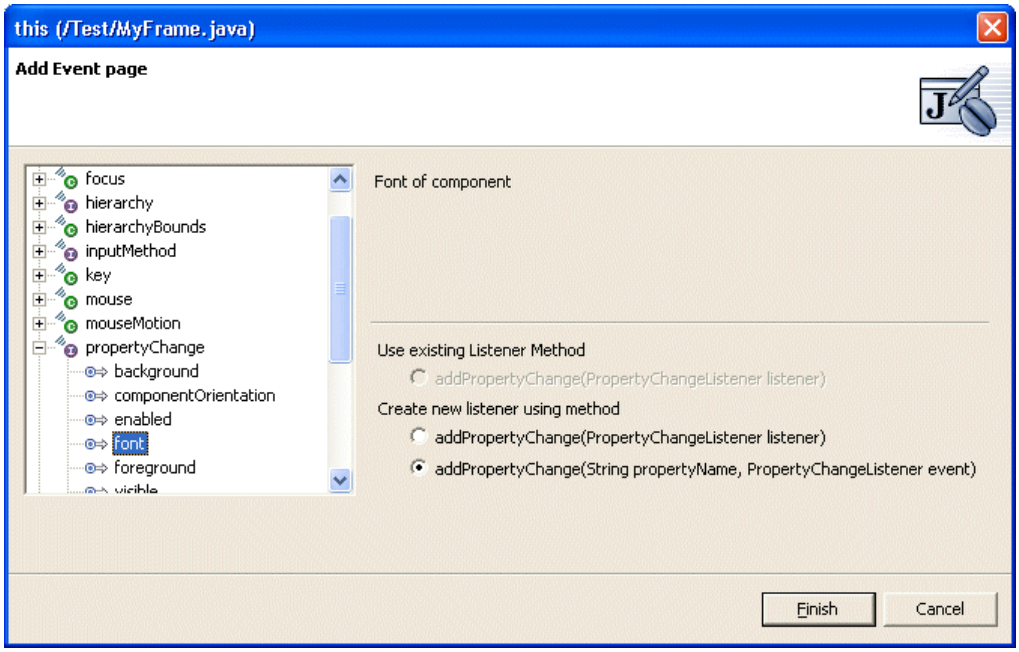
Procedure

1. Right-click the component in the Java Beans view or the Design view.
2. From the pop-up menu, click **Events**. The pop-up menu shows the preferred events for the component. Do one of the following steps:
   - Click one of the preferred events for the component. The event is added to the component. For example, for a JFrame the preferred events are windowClosed and windowOpened.
   - Click **Add Events** to open the Add Events window.
3. If you opened the Add Events window, you are presented with a list of categorized events to choose from. Select the event that you want to add, then click **Finish**.
   - The Add Event dialog shows all the available events for the component in a tree. The tree shows the event listeners as the first level of entries, and beneath each listener entry are the possible callbacks the event can raise.



   - You can use the dialog to specify whether you want to create a listener or use an existing one. When the dialog opens, it analyzes the existing listeners, and if it finds one that it believes can be reused, it marks the listener with a different icon. For existing listeners that extend an adapter class, the icon is used, otherwise, the is used. Within the listener itself, existing events are shown with a green arrow , and available events are shown with an incomplete arrow
   - You can specify whether you want to use the existing listener or create a new one. When a new one is created, you can choose whether you want it to extend the adapter class or implement the listener interface. If an option is not applicable, the radio buttons are disabled. For example, if there is no existing listener, or an existing listener is selected but the selected event is already used, the **Use existing Listener** is disabled. If there is no available adapter class for a new listener then the **extends: ADAPTER_CLASS_NAME** radio button is disabled.
   - If you select propertyChange in the Add Event dialog, you have additional options for adding the callback. A new listener can be created that can be added to the Java bBean using the method a

`addPropertyChange(PropertyChangeListener listener)`. If this occurs then the listener's generic callback method `propertyChange(PropertyChangeEvent event)` is used for all property events. On some JavaBeans (such as Swing components) the two argument method `addPropertyChange(String propertyName, PropertyChangeListener listener)` is available and selected by default to use for the new listener.



A listener that is added using the two argument method is specific to a particular property, so it cannot be reused for another property. A listener added with a single argument method has an `if` statement that checks the name of the property before processing the logic for each property callback as shown in the following code:

```
javaBean.addPropertyChangeListener(new java.beans.PropertyChangeListener() {
    public void propertyChange(java.beans.PropertyChangeEvent e) {
        if ((e.getPropertyName().equals("font"))) {
            System.out.println("propertyChange(font)");
       }
    }
});
```

This allows a PropertyChangeListener added with a single argument method to be used for more than one property (by having multiple if{} blocks used).

4.

Results

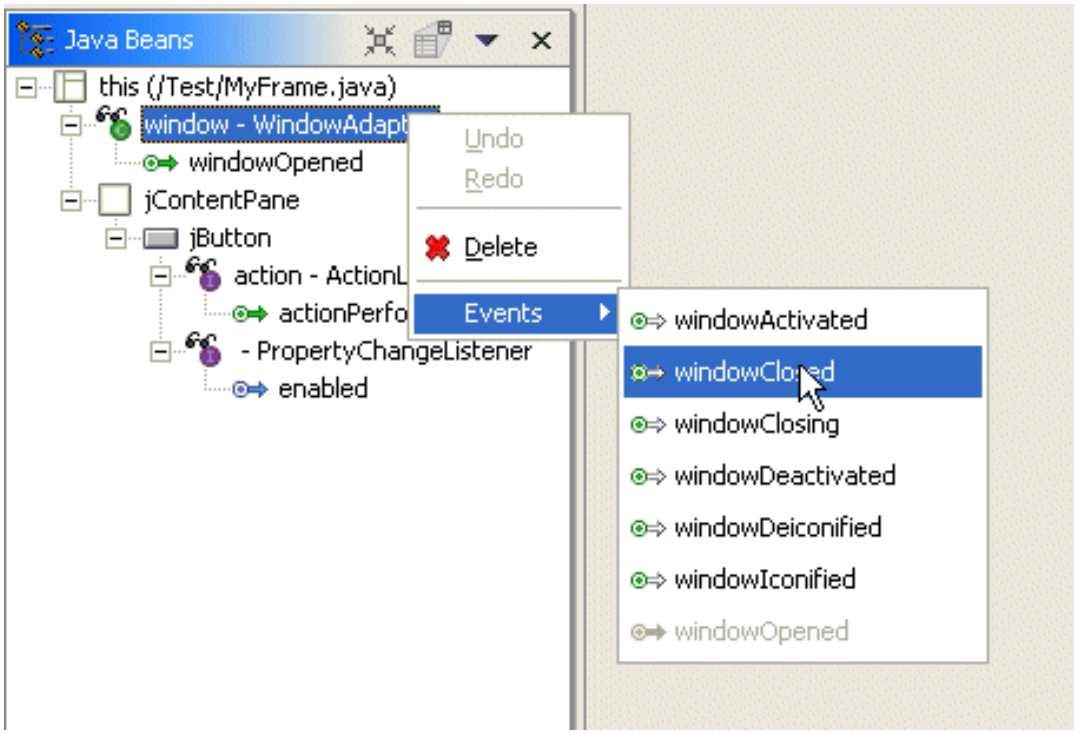
When you add an event, if there is an existing listener that can be used, then the callback method is added to it. Otherwise, a new listener is created. For an existing listener to be used, it must be an anonymous inner class on the Java bean that implements the listener interface. It must have an empty method body for the callback method, or extend the adapter class. and have no existing method for the event callback being added. For a property, an existing PropertyChangeListener is reused if it is added to the Java bean with the single argument method `addPropertyChange(PropertyChangeListener listener)`. If it does not already have code, processing the property is added.

If there is no candidate existing listener onto which the callback method can be added, then a new listener is created. This will be an anonymous inner class, and if an adapter class has been defined for the event then the listener extends this. Otherwise, it implements the listener interface. After the event is added a stub method is created with a //TODO comment. The stub method is an indicator of the source code that is executed when the event occurs, and you can then change this

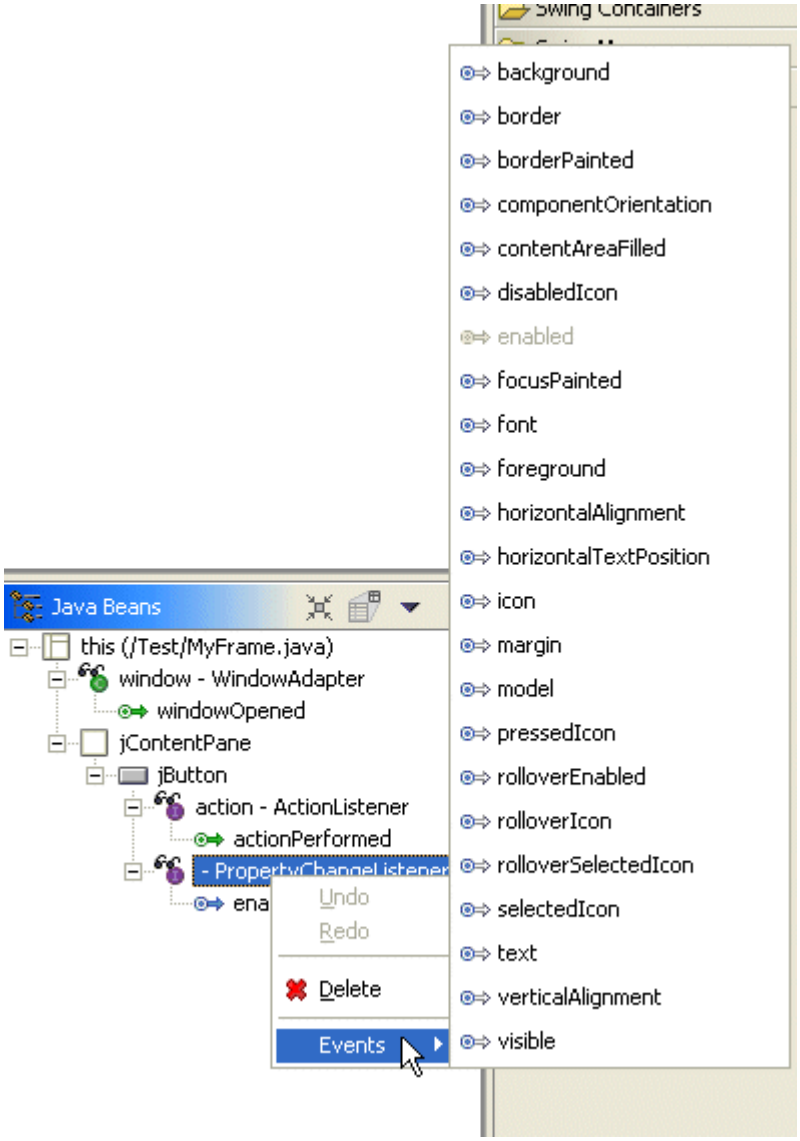
to perform your required behavior. The //TODO comment is displayed in the Tasks window, flagging incomplete methods. This is so that you can locate them later and remove the //TODO comment once the callback logic has been written.
In the previous example the windowOpened method exists. If this is a listener that extends the adapter, the same listener is reused because it does not already have a windowClosed method. The method `windowClosed(WindowEvent e)` is added, and the method stub and //TODO comment added as shown here:

```
this.addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosed(java.awt.event.WindowEvent e) {        System.out.println("windowClosed()"); // TODO Auto-generated
stub windowClosed()                    }
        public void windowOpened(java.awt.event.WindowEvent e) {
             callExistingWindowOpenedLogic();
            }
});
```

In expert mode, events can still be added to the Java bean as shown previously, but they can also be added to a listener in the Java Beans tree. The Events menu shows all the event callback methods on the listener, and any that are already used are disabled.



For a PropertyChangeListener, the pop-up menu shows all the bound properties on the Java bean. If any are already used by the PropertyChangeListener then they are disabled.



A listener added with a single argument method has an if statement that checks the name of the property before processing the logic for each property callback as shown in the following code:

```
javaBean.addPropertyChangeListener(new java.beans.PropertyChangeListener() {
    public void propertyChange(java.beans.PropertyChangeEvent e) {
            if ((e.getPropertyName().equals("font"))) {
                  System.out.println("propertyChange(font)");
            }
        }
   });
```

This allows a PropertyChangeListener added with a single argument method to be used for more than one property (by having multiple if{} blocks used), and when the second and subsequent property callback is added a new if{} blocks are added.

If the propertyChangeListener is added to the Java bean using the two argument method `addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` then it is specific to a particular property so cannot be reused for another property. In this case all the Events cascade menu children are disabled.

Related concepts:

The Java Beans view

Deleting events from a component

You can use the Java™ Beans view to delete an event that you added to a component in the visual editor for Java.

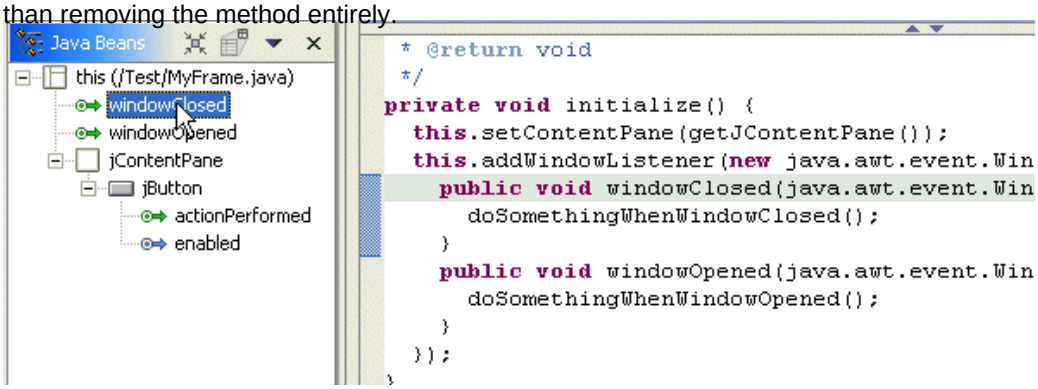
About this task

Tip: If you do not see any events in the Java Beans view, make sure that you click the Java Beans view menu arrow and select **Show Events** to show events, or select **Expert Events** to show events, adapters, and listeners.

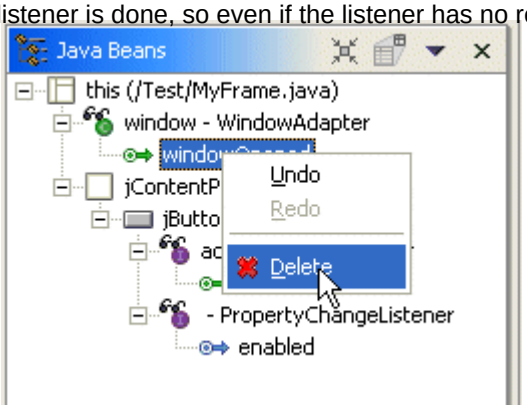
To delete an event from a component, do one of the following:

Procedure

- In the Java Beans view, select the event and press the Delete key.
- In the Java Beans view, right-click the event and select **Delete** from the context pop-up menu.
  - In **Show Events** mode, this removes from the Java source the callback method for the event. If the listener class implements an interface that requires a method body for compilation, then the method contents are cleared out, rather than removing the method entirely.

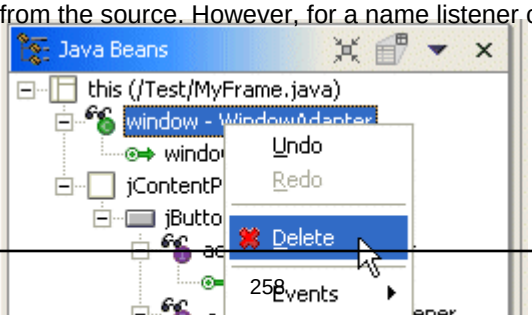


    If after you delete the callback method there are no remaining method body implementations on the listener and if the listener is an anonymous inner class, then the listener itself is removed with the method that registers it with the source. In the previous example, the Window listener only has a windowOpened callback, so when windowOpened is deleted the listener and the `addWindowListener(...)` statement that adds it to the source is removed.
  - In **Expert Events** mode, when a callback method is deleted it is removed from the listener, or replaced with an empty method body if the listener implements the interface and must have a method to successfully compile. Unlike in **Show Events** mode, no cascaded delete of the listener is done, so even if the listener has no remaining callback method bodies left is it not automatically removed.



    Delete in **Expert Events** mode can be thought of as deleting the selected tree item from its parent.

    The listener itself can also be selected and deleted. This removes the listener from the Java bean. If the listener is an anonymous inner class it is removed from the source. However, for a name listener class it remains after it has been



Related concepts:

The Java Beans view

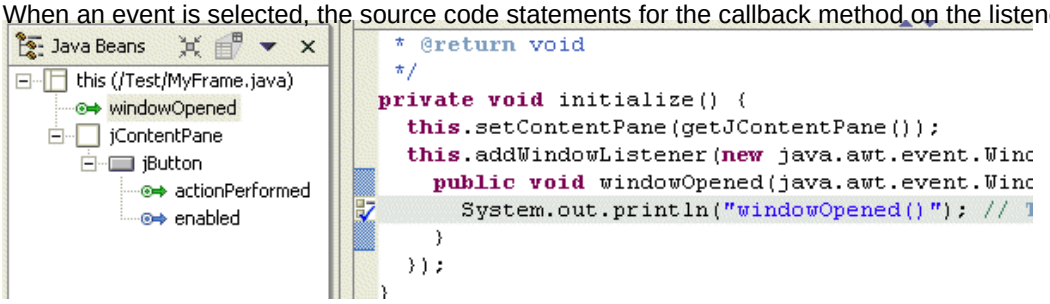
Viewing the source for an event

When you select an event in the Java™ Beans view, the Source view shows the relevant lines in the Java code.

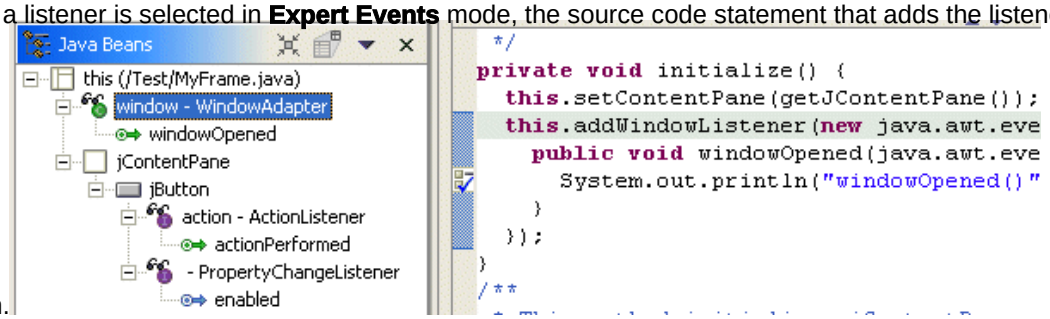
Procedure

To show the source for an event, select the event in the Java Beans view.

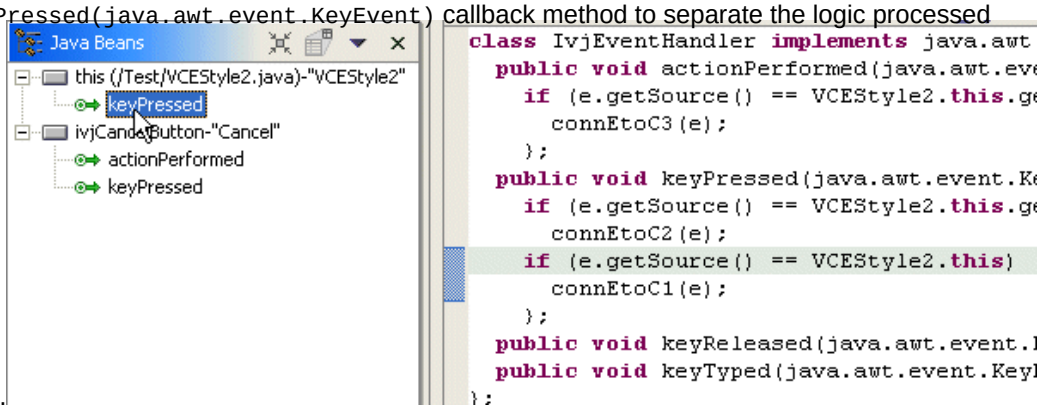
- When an event is selected, the source code statements for the callback method on the listener are shown.



- When a listener is selected in **Expert Events** mode, the source code statement that adds the listener to the source is shown.



- If the listener is shared (so the callback method can be used by more than one component), selection points to the relevant block of code responsible for processing the parent component of the event. The following image shows that the single inner class IvjEventHandler is used by both the class itself (this) and by a Cancel button. The keyPressed method has `if` blocks within the `keyPressed(java.awt.event.KeyEvent)` callback method to separate the logic processed when each button is pressed.



Related concepts:

The Java Beans view

Externalizing text strings with the visual editor

If you are designing a Java™ application to be deployed for use in different locales, any text that is visible to the user needs to be translated.

About this task

Rather than translating the string in the Java source file and having to create a new executable for each language, Java lets you use resource bundles to redirect the string through a properties file. For more information about externalizing strings, see Externalizing strings.

The visual editor for Java supports displaying the externalized strings. They show up normally with the actual values stored in the resource file.

However, if you change a text property of a component in the visual editor after you already externalized the string, the text property will overwrite the externalization, not change the resource file. To reflect changes to the resource file, the visual editor must be closed and reopened. Be careful when you choose strings to externalize. By default, all strings are externalized, even strings which are not displayed on the GUI, for example, font names.

Before externalizing the string, the string is presented in the source as a literal. For example:

```
ivjJFrame.setTitle("Hello World");
```

After using the Externalize Strings wizard, the string is retrieved from a static lookup in a file that retrieves the value from a resource bundle.

```
ivjJFrame.setTitle(Messages.getString("Hello_World_1"));
//$NON-NLS-1$
```

The string that is the argument to the setTitle method call is externalized, but the string that represents the bundle key is not because it is not a user visible string. To indicate that this string is not a user visible string so the Externalize Strings wizard does not attempt to retrieve this from a bundle, the comment `//$NON-NLS-1$` is appended to the statement with the `1`, indicating that it is referring to the first string occurrence on the line.

Testing and debugging in the visual editor

After you write a Java™ class using the visual editor for Java, you can run it to test and debug the runtime behavior.

About this task

If the class has a `public static void main(String[])` method, you can run or debug it as a Java application. However, when you write a Java bean using the visual editor for Java, typically it does not have a main method. Rather than adding a main method just to test the class, you can run or debug the class as a Java bean.

To debug a Java bean while you are testing it, you can switch to the Debug perspective, where the Java processes can be seen and where you can use breakpoints and step through the code.

Running your visual class as a Java bean or application

While you are developing your visual class, you can run it to test its appearance and behavior.

About this task

When you run a Java™ bean or application using the visual editor, a virtual machine is created that uses the class path specified in the Java Build Path of the project. The Java bean is then instantiated using its null constructor.

- If the Java bean is a visual class (is a subclass of `java.awt.Component`), an appropriate window is created to host the visual Java bean.
- If the bean is part of or inherits from an AWT Java bean, the window is a `java.awt.Dialog`.
- If the bean is a Swing class or inherits from a Swing class, the window is a `javax.swing.JDialog`.
- If the bean is an SWT class, the window is an SWT Shell. If the visual Java bean does not require a window because it itself is one, the Java bean will be made visible and given a default size after it has been instantiated.

When you run a class as a Java bean or application, a launch configuration is automatically created. The launch configuration is used to start a virtual machine that instantiates the class and allows it to be tested. If a launch configuration exists for the class you are running, that launch configuration is used.

To run a visual class as a Java bean or application:

Procedure

1. Open your visual Java class in the visual editor.
2. From the main menu, select one of the following options:
   - **Run** > **Run As** > **Java Application** if your class has a `public static void main(String[])` method.
   - **Run** > **Run As** > **Java bean** if your class has no main method.

Results

A virtual machine that instantiates the class is started, and the Java bean or application runs on top of the workbench. You can then test the behavior and performance of your Java class.

Configuring options for running a Java bean or application

You can specify different configurations for each Java™ bean or Java application that you run.

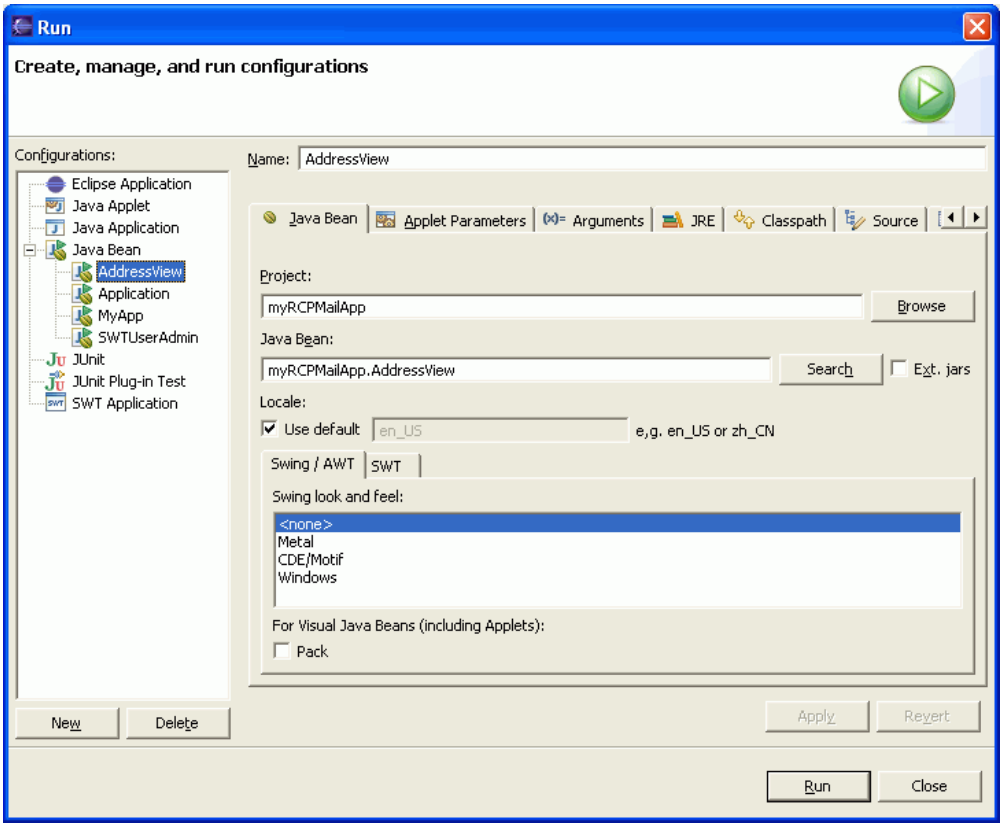
About this task

When you use the visual editor to run a Java bean or Java application, a new launch configuration is automatically created. The configuration lists the project, the Java Bean or main class, and other advanced options that dictate how the virtual machine appears and behaves when you run that Java bean or main class. You can modify the configuration to better suit your needs or to test different runtime environments.

To set up and configure the options for running a Java bean or application:

Procedure

1. From the main menu, click **Run** > **Run**. The Run window opens and lists the existing configurations.



2. In the **Configurations** list, do one of the following steps:
   - Select a configuration. Default configurations are created automatically when you run a Java bean or application.
   - Click **New** to add a new configuration.

   The options for the new or selected configuration are shown in the tabbed notebook pages to the right.
3. Specify the project and the Java bean or main class that this configuration is used for.
4. Specify other run options as required:
   - For a Java bean configuration:
     - You can specify a locale for the virtual machine to use.
     - For Swing or AWT, you can specify the Swing look and feel to use.
     - To cause the Java bean to be sized to fit the preferred size and layouts of its subcomponents, select the **Pack** check box.

- For a Java application configuration:
  - The tabbed pages for Arguments, JRE, Class path, Source, and Common include more advanced options. These advanced options let you control precisely how the virtual machine for the launcher is started, the defaults of which are set from the properties of the Java project.

Debugging a visual Java bean or application

If you want to see the Java™ processes and step through the code for your Java visual class while you are running it, you can choose the debug options.

About this task

Like the run commands, the debug commands start a virtual machine and create and use launch configurations. However, by selecting the debug menu options when running, you gain the added benefits of the Debug perspective.

Procedure

1. Open your visual Java class in the visual editor.
2. Switch to the Debug perspective by clicking**Window** > **Open Perspective** > **Debug** from the main menu.
3. From the main menu, select one of the following commands:
   - **Run** > **Debug As** > **Java Application** if your class has a `public static void main(String[])` method.
   - **Run** > **Debug As** > **Java bean** if your class has no main method.

   If you did not switch to the Debug perspective, the launch configuration automatically opens the Debug perspective.

Results

The Debug perspective displays the Java threads and processes that you can work with. You can use the standard Debug perspective options to step through your code using breakpoints.

Testing and deploying applets

You can test an applet using launch configuration options before deploying it to open in a browser.

About this task

An applet is a bean that is designed to be run inside a Web browser and inherits from `java.applet.Applet` or `javax.swing.JApplet`.

The Eclipse-based launcher has its own applet stub that is used to run the applet. This stub automatically starts the applet after it has been instantiated by calling its `start()` method. Unlike the JRE applet launcher, which the Java™ bean launcher does not use, the stub does not check authority profiles to see whether the applet can perform tasks such as input/output or opening sockets. This means you can quickly test the runtime appearance and behavior of your applet without worrying about security certificates.

To test an applet using the Java bean launcher:

Procedure

1. Create a launch configuration for your applet class. See Configuring options for running a Java bean or application for more information.
2. On the Applet Parameters page of the launch configuration, specify any parameters that you want to pass to the applet. These parameters are made available to the applet under test through a call to the `getParameter(String)` method.
   A. Click the **New** button to add a new parameter.
   B. To edit the values for the Name and Value of the parameter, select the text in the table and type new values.
3. Click **Apply** to save the launch configuration.
4. Click **Run** to run the applet using the parameters that you entered.

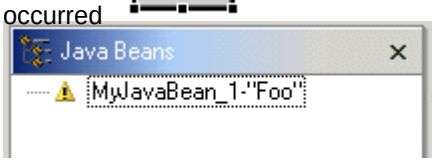
Results

When you have finished testing your applet and are ready to deploy it in a browser, you can use the simple HTML `<applet>` tag, as long as the class does not require any JDK 1.2 or higher features, such as any of the Swing classes. If your applet uses any JDK 1.2 or higher features, the HTML syntax to embed it into a page is more complex in order to deal with differences between browsers and to ensure that the applet uses the correct Java plug-in. For more information about applets and Java plug-ins, refer to java.sun.com.

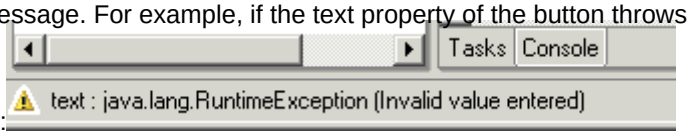
Java bean exceptions

During normal development, exceptions are sometimes thrown by the Java™ beans. The visual editor displays icons and visual markers to indicate exceptions.

For example, when a visual class is instantiated or when property values are applied, a visual class might throw an exception. If an exception is thrown when a property value is applied, the Java bean is recreated and all the other valid properties are applied. A warning sign is shown in the Design view and the Java beans view to indicate that an exception occurred 



When you select a component that has a warning, the status line shows you the property that threw the exception as well as the exception message. For example, if the text property of the button throws a run-time exception, the status bar shows the following status:



When the exception is thrown during the application of a property value, the Java bean is recreated and the property ignored. The warning sign indicates that the representation of the live Java bean, as shown in the Design view or the Properties view, is incomplete, as the errant property has not been applied.

In addition to exceptions thrown when property values are applied, there might be an exception thrown during instantiation of the Java bean. In this case it is not possible to partially create the Java bean, as was done for bad property values, so no live Java bean is present. To indicate this, for errors thrown during instantiation of the Java bean, a red x is shown in the Design view and Java Beans view. You can select the Java bean to see the exception message in the status line.

A `Too complicated` error, represented by a blue circle with an exclamation point, occurs when the initialization string for a property is too complicated for the visual editor to understand.

Advanced options for debugging Java beans

During normal development, you do not need to be concerned with the Virtual Machine (VM) that is used by each visual editor instance. Advanced users, however, can configure the workbench so that any messages sent to `System.out` or `System.err` of the VM are written to the `.log` file in the `.metadata` directory in your workspace.

The virtual machine (VM) that is used to execute the Java™ beans is not the same VM that the workbench is running within. A separate VM is created for each instance of the visual editor. The class path of this VM is set to be the entries in the Java build path for the project. If while the visual editor is open the Java build path is changed any open editors must be closed and re-opened to reflect the updated build path.

When the workbench is started it can be given a -DEBUG option that points to a text file:

```
-DEBUG file:/c:/temp/options.txt
```

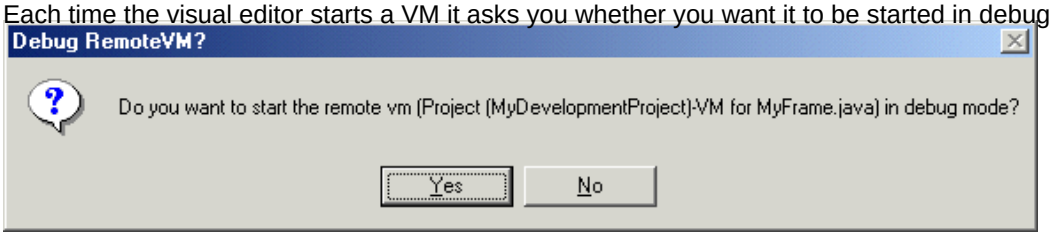
The debug file contains entries that are read by the visual editor. If the file contains the following line then any messages sent to `System.out` or `System.err` are shown in the console:

```
org.eclipse.jem.proxy/remote/debug/vmtraceout=true
```

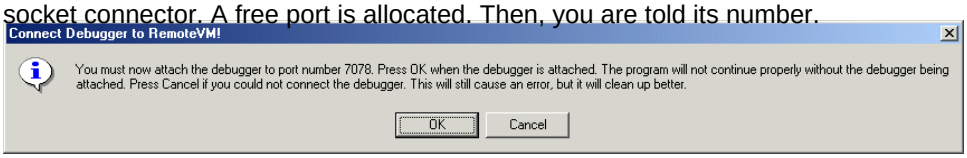
In addition to having console output shown from the VM used by the visual editor, you can start it in debug mode and perform remote debugging. This scenario is not optimized and might change in future versions of the visual editor. However, it is documented in case you need to perform debugging of your Java beans at design time. To perform remote debugging you need two workbenches running: one with the Java beans that you want to debug, and one that can perform socket-based remote debugging. The second workbench could be running on the same machine or a different machine. To enable debugging of the VM of the visual editor., the -DEBUG file must contain the line:

```
org.eclipse.jem.proxy/remote/debug/debugvm=true
```

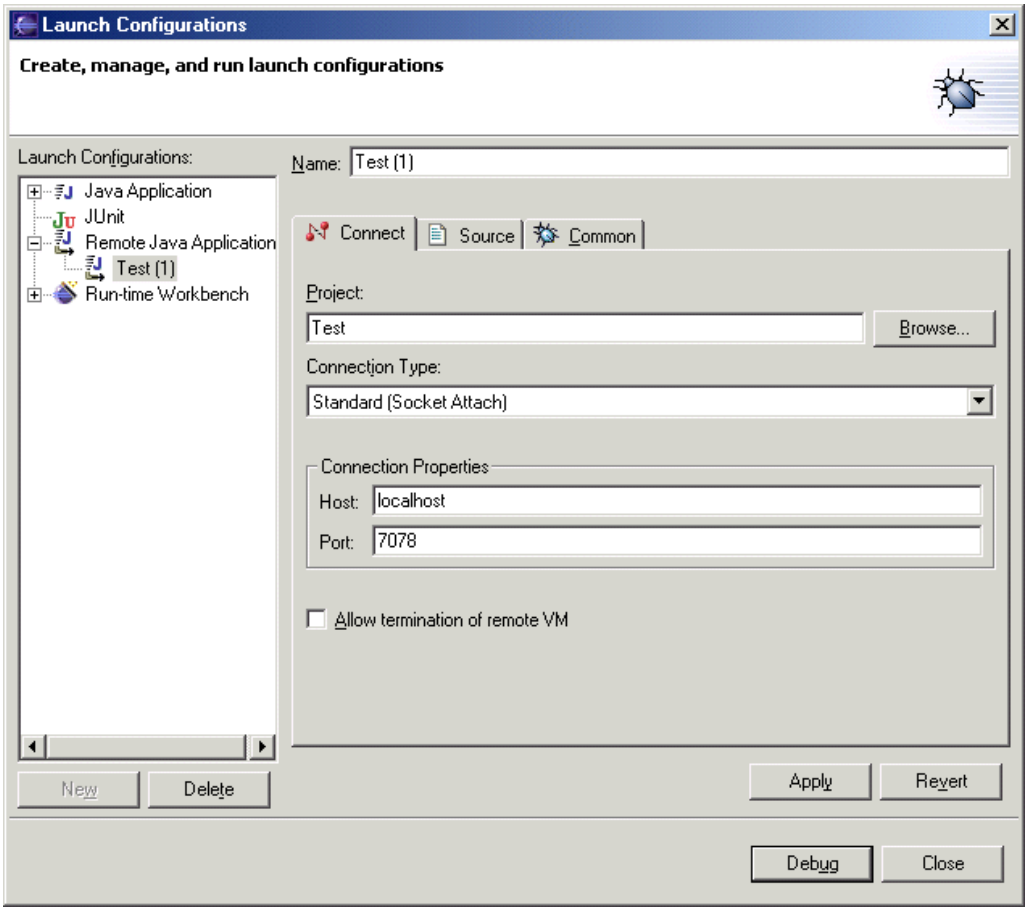
Each time the visual editor starts a VM it asks you whether you want it to be started in debug mode or not.



The previous dialog is for when the class `HelloFrame.java` in the project Test is opened with the visual editor. If you select Yes, or enter Y and press Enter in the console, then the VM is started in suspend mode with the `-Xdebug` using a socket connector. A free port is allocated. Then, you are told its number.



The next step is to attach a remote debugger to port number 7078. The instance of the workbench that is running the visual editor is suspended while the remote debugger attaches, so you must use a debugger from another application. If the other application is another instance of the workbench, you can begin remote debugging by using the menu option **Debug > Debug** and creating a new Remote Java Application configuration.



Enter the port number that was previously written to stdout, in this case 7078. If the debugger is on a different machine, in the **Host** field you must enter the IP address of the machine running the workbench that is being debugged. To begin the remote debugger, click **Debug** and then enter `y` in the stdin console of the workbench being debugged.

In addition to being prompted to debug the VM used by each instance of the visual editor, you see messages that prompt you to debug the project itself:

```
Do you want to start the remote vm (Project (Test)-Beaninfo) in debug mode? (Enter Y or N):
```

This is for the VM that is used for introspection and the `java.beans.Introspector` is run in, and can be distinguished from the VM for the visual editor because it does not specify a particular Java source file name. This VM is started the first time any bean info is required for a Java bean within a project, and remains active as long as the project remains open. By debugging this project you can step through and analyze any code in your BeanInfo classes as your Java beans are being introspected. Introspection occurs once for each Java bean class for each project it is used in. The result is held in a cache to help performance. If at any time you need to clear the cache of BeanInfo information held for a project you can **close** and **open** the project from its pop-up menu.

Package jve.generated

Interface Summary	
IActionBinder	IActionBinder includes methods for enabling and disabling the state of a bound visual element.
IActionBinder.ActionBinderListener	Interface used to register for action triggering notification.
IActionBinder1	IActionBinder1 includes flags and methods that can be used by a widget that is bound to an action to enable or disable itself.
IBoundObject	This interface represents a reference to an Object.
IDataObject	This interface extends the object reference capability of IBoundObject to allow the dynamic getting and setting of property values on the referenced object.
IDataSource	A data source is a factory for a facade of some type of data service.
IDataSourceService	A Data Source Service provided the interfaces necessary to perform a service call on a Data Source.
IFieldBinder	A FieldBinder contains a String text representation of a bound property of an object pointed to by a DataObject.
IFilterBinder	A filter binder provides a simple filter mechanism to filter objects.
IFilterBinder.FilterChangeListener	Interface used to register for filter change notification.
IRowsDataObject	This interface represents an array (rows) of objects.
IRowsDataObject.RowChangeListener	Interface used to register for row change notification.
ITableBinder	A Table binder represets an array of objects (rows) that are the basis for a table model.
ITableBinder.SelectionChangedEvent	Notice of a change of selection in the ITableBinder's visual table.
ITableBinder.SelectionChangedListener	Listen for changes to the ITableBinder's selected object.

Class Summary	
BasicDataObject	Default implementation of a data object.
DataSourceDataObject	Default implementation of a data object that is retrieved by a service call on a Data Source.
DataSourceDataRows	Default implementation of a row data object that is retrieved by a service call on a Data Source.
DataSourceService	Default implementation of IDataSourceService.
EJBDataSource	This data source provides access to an EJB service via a session bean.
IActionBinder.ActionBinderEvent	A action binder event is fired before and after the ActionBinder's action is performed.
IFilterBinder.FilterChangeEvent	A filter change event is fired when a user needs to reaccept.
IRowsDataObject.RowChangeEvent	A RowChangeEvent is fired when the contents of a row changes in the array maintained by the RowDataObject.
JavaBeanDataObject	This DataObject will provide an instance of the given Java Bean as a data object.
JavaBeanDataSource	This DataSource uses a null constructor to instantiate a Java bean.
JObjectTableBinder	This is a Swing Table binder whose row data is a property of another DataObject.
JRowTableBinder	This table binder get its source from a `IRowsDataObject`.
ObjectReference	This class is a wrapper that references an Object: boundObject.
PropertyHelper	This helper provide the details of resolving an Object/Property string into the final object.
SwingDataServiceAction	This action will execute a data source service with a single argument.
SwingPropertyFilter	This filter acts as a Document for a JTextComponent.
SwingTableBinder	Swing implementation of the ITableBinder interface.

SwingTextComponentBinder	Swing implementation of the IFieldBinder interface.
WebServiceDataSource	Provide access to a Web Service proxy as a data source.

Code migration from VisualAge for Java to the visual editor

This topic provides information about migrating Java™ code from VisualAge® for Java.

When you change a Java component using the visual editor for Java, the source code is updated to reflect the changes. The source code changes are reflected in `set` methods that change property values. However, some information used by the visual editor for Java is not stored in properties because it is only required at design time. This information includes the position of a Java bean on the free form surface.

To store this information so that the visual editor for Java can be reopened with the Java bean at the same position, the information is placed in a comment on the line that declares the Java bean. The following statement shows a JFrame component that is positioned at 16, 17:

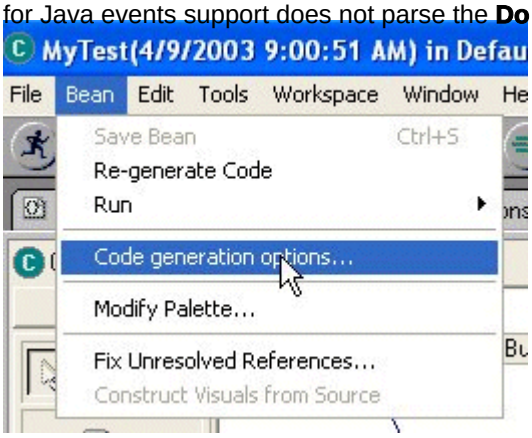
```
private javax.swing.JFrame ivjJFrame = null; // @jve:visual-info
decl-index=0 visual-constraint="16,17"
```

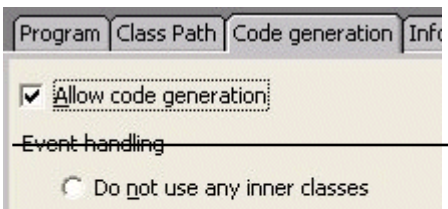
The comment representing the position of a component is not required, and if no comment is present then a default position is allocated when the visual editor for Java is opened. This default placement only applies to top level Java beans that are not contained within another and does not affect the placement of components within a container. The location of components within a container is determined by the layout manager of the container and the bounds or constraints of the component.

In VisualAge for Java, the position of the top-level Java beans (also referred to as free form parts) is not present in the source code. If you migrate a file that was written using VisualAge for Java's Visual Composition Editor (VCE), default positions are used. If you want to keep the positional information, then you can obtain a migration utility that is loaded into VisualAge for Java. The migration utility regenerates your classes with the position stored in a comment format. To get the utility, download the latest **Conversion tool for VisualAge for Java Visual Composition Editor applications** from www.ibm.com/support/us/

This migration utility is available as a tempfix that can be installed using VisualAge for Java's FixManager (from **Workspace > Tools > FixManager**). The utility migrates and exports classes that have been developed using VisualAge for Java's VCE to a format suitable for the visual editor. After installing this patch, you can select **VCE Code Generation/Export...** from the pop-up menu for projects, packages, or classes. Selecting this item launches a wizard that can regenerate the code for classes that have been previously saved with the VCE. The free form positions are saved in the comment format used by the visual editor.

If you have connections, you can first regenerate this code by selecting the **Use an inner class for each event** VCE code generation option before running this utility. However, some classes cannot be converted to this style due to a bug in VisualAge for Java. In this case, you can use the **Use one inner class for all events** VCE code generation option. The wizard also gives you the option to export the classes to a directory after the code generation is complete. The visual editor for Java events support does not parse the **Do not use any inner classes** VCE code generation style.





Since the VCE maintained its own model of the Java beans and their property values and relationships, it always regenerated the source in a top-down fashion from this model. Any modifications made by a user to the source were limited to pre-defined user code points in the source delimited by comments `//user code begin {1}` and `//user code end.` Also, to indicate that the methods for the Java beans were regenerated each time code generation was performed, the line `/* WARNING: THIS METHOD WILL BE REGENERATED. */` was added to the method comment. The migration utility has an option that removes these VCE-generated comments from the exported code (not the source code in VisualAge for Java), as they are no longer applicable outside the VCE. However, once the comments for the user code points have been removed from the source, the user code cannot be used within VisualAge for Java. The reason is that the presence of these comments is what protects the user code from being overwritten.

The visual editor for Java does not use a persistent object model for its Java beans and their property values and relationships, but rather parses the source each time. For this reason, the comments for user code points and for specifying method regeneration no longer apply, and modifications can be made freely to the source code. If the modifications alter the source code structure so that the visual editor for Java can no longer recognize the structure of the Java beans, you might not see them in the Design view or the Java Beans view. However, the source is altered to suit the editor's style, and your changes are preserved.

Using the Java integrated development environment

Read these topics to get started with the Java development environment.

Java development overview

The JDT project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself.

Try the Basic tutorial to get a first impression. Discover more in the Tips and Tricks section and learn about the latest features in What's new.

Basic tutorial

This tutorial provides a step by step walk-through of the Java development tools.

Next Section: Preparing the workbench

Preparing Eclipse

In this section, you will verify that Eclipse is properly set up for Java development.

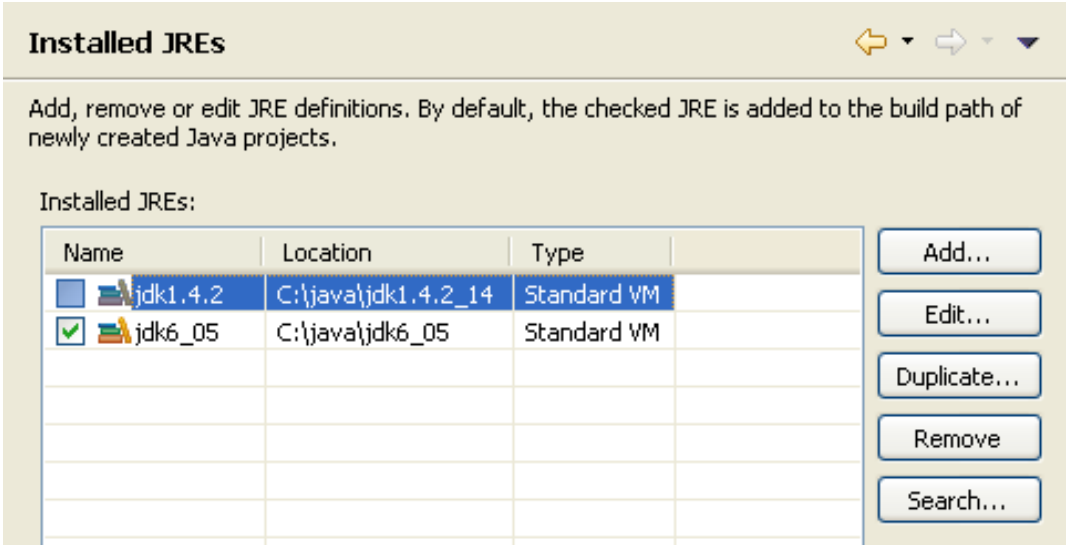
The following is assumed:

- You are starting with a new Eclipse installation with default settings.
- You are familiar with the basic Eclipse workbench mechanisms, such as views and perspectives.

If you're not familiar with the basic workbench mechanisms, please see the Getting Started chapter of the Workbench User Guide.

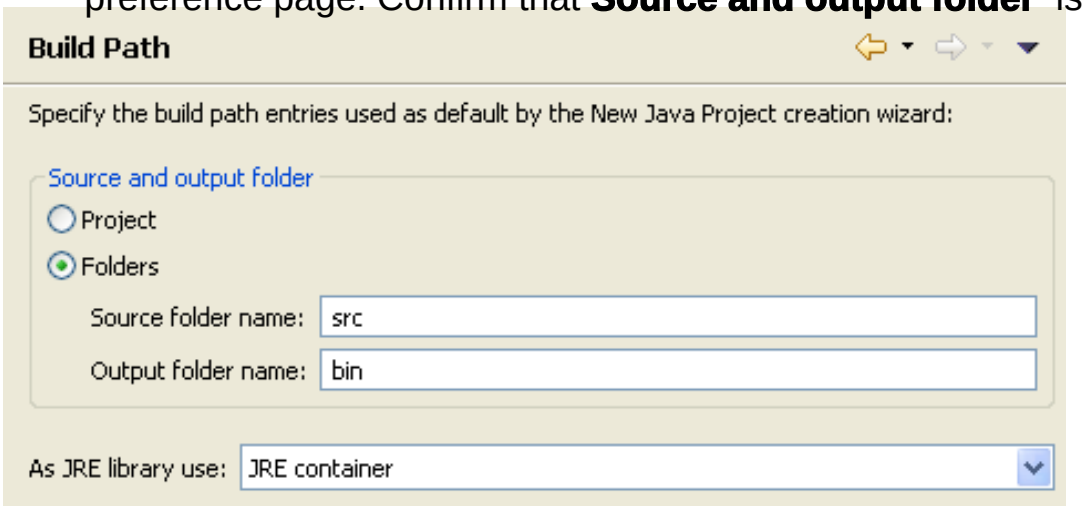
Verifying JRE installation and classpath variables

1. If you still see the Eclipse Welcome page, click the arrow icon to begin using Eclipse.
2. Select the menu item [Image: /topic/org.eclipse.help/command_link.png]**Window > Preferences...** to open the workbench preferences.
3. Select the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs** preference page to display the installed Java Runtime Environments. Confirm that a JRE has been detected. By default, the JRE used to run the workbench will be used to build and run Java programs. It should appear with a checkmark in the list of installed JREs. We recommend that you use a Java SDK instead of a JRE. An SDK is designed for development and contains the source code for the Java library, easing debugging. Additional SDKs can be added by searching the hard drive for installed SDKs. To do so, simply click the **Search...** button and specify a root folder for the search.
   If you work with code that does not yet use generics (as we do in this tutorial), we recommend that you install a Java SDK 1.4 as well, but leave the most recent version checked as default.

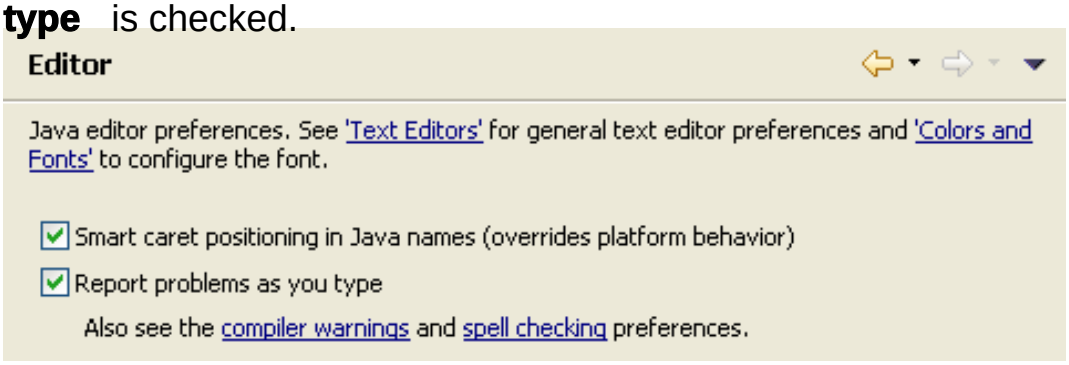


4. Select the [Image: /topic/org.eclipse.help/command_link.png]**General > Workspace** preference page. Confirm that the **Build automatically** option

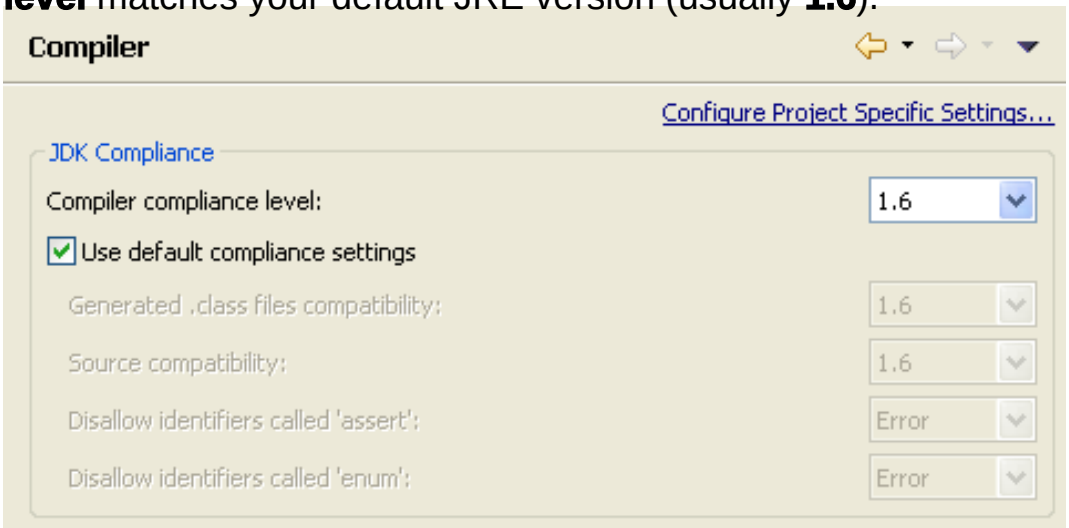
5. Select the [Image: /topic/org.eclipse.help/command_link.png]**Java > Build Path** preference page. Confirm that **Source and output folder** is set to **Folders**.



6. Select the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor** preference page. Confirm that option **Report problems as you type** is checked.



7. Select the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler** preference page. Confirm that option **Compiler compliance level** matches your default JRE version (usually **1.6**).



8. Click on **OK** to save the preferences.

Next Section: Creating your first Java project

Related concepts

Java projects
Classpath variables
Build classpath

Related tasks

Working with JREs

- Related reference

Installed JREs Preferences
Java Editor Preferences

Creating your first Java project

In this section, you will create a new Java project. You will be using JUnit as your example project. JUnit is an open source unit testing framework for Java.

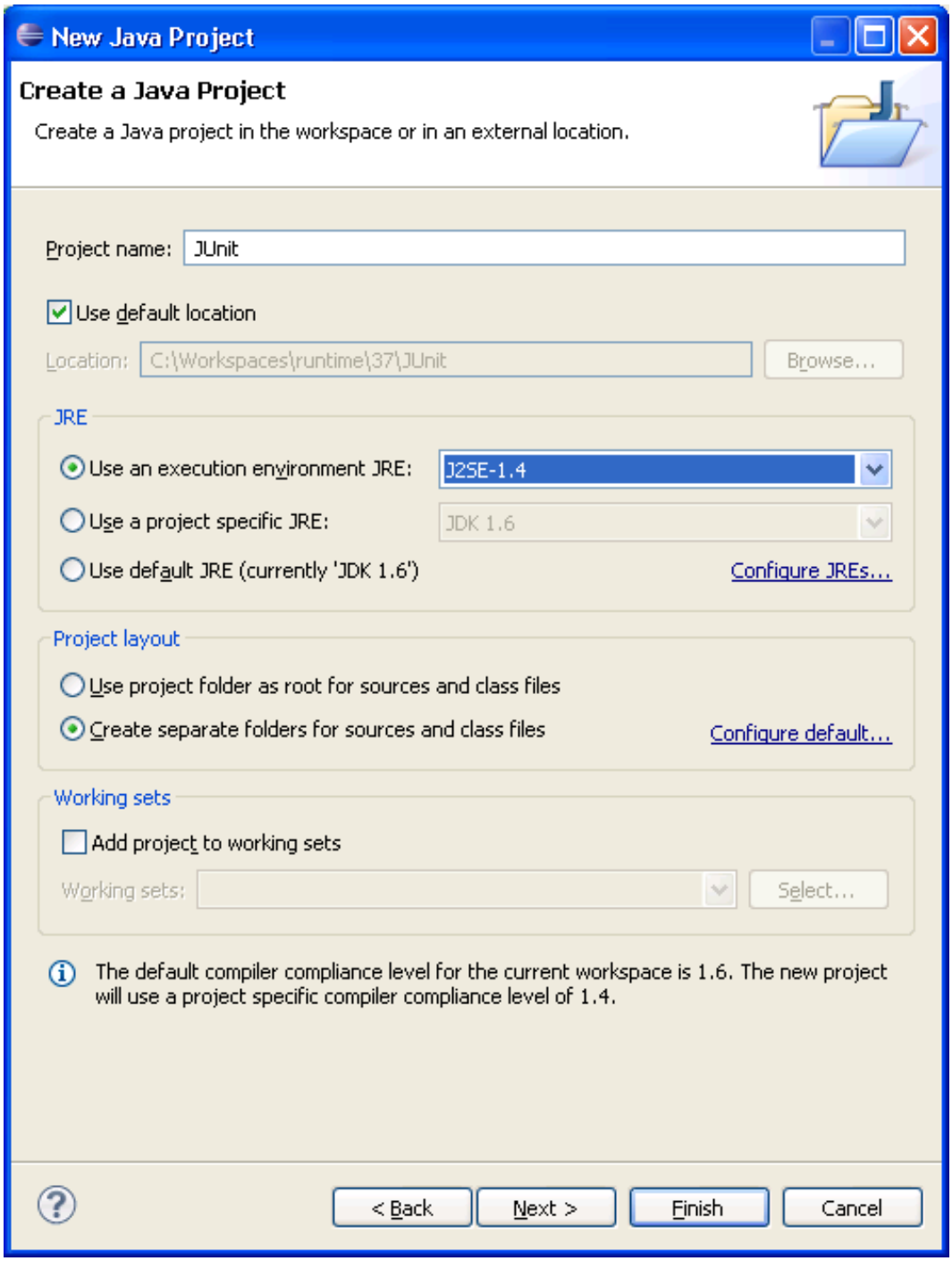
Getting the Sample Code (JUnit)

First you need to download the JUnit source code.

1. Click here to download the JUnit source code.
2. Save the archive (do not extract) to a directory from now on referenced as *<Downloads>*.

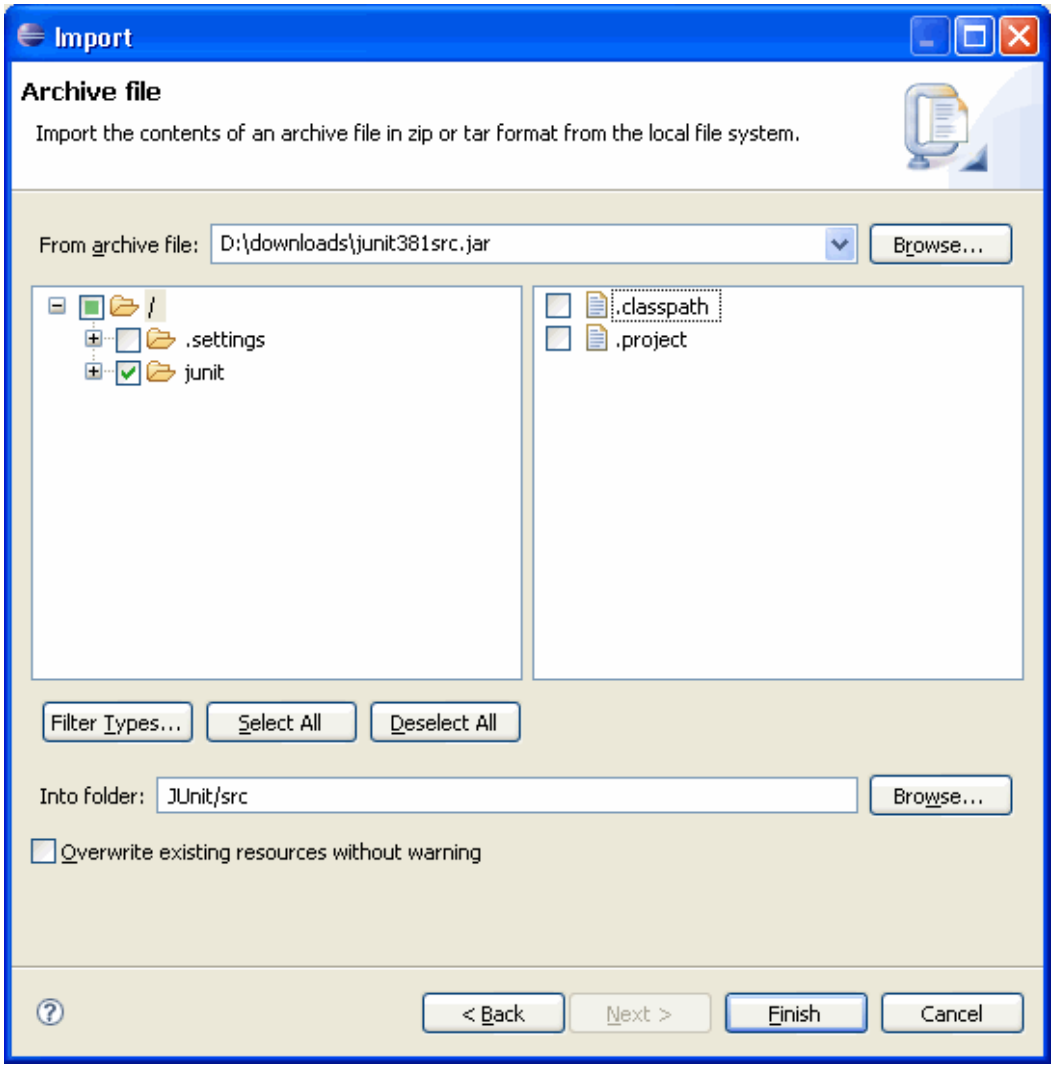
Creating the project

1. Inside Eclipse select the menu item **File > New > Project...**. to open the **New Project** wizard
2. Select **Java Project** then click **Next** to start the [Image: /topic/org.eclipse.help/command_link.png]**New Java Project** wizard:



On this page,
- type "JUnit" in the **Project name** field, and
- select "J2SE-1.4" in the **Use an execution environment JRE** field.
Then click **Finish**.

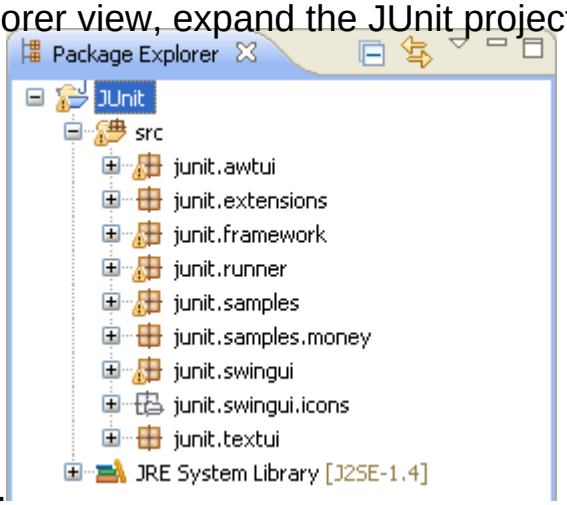
3. In the Package Explorer, expand the *JUnit* project and select the source folder *src*.
4. Select the menu item **File > Import...**.
5. Expand **General**, select **Archive file**, and click **Next**.
6. Click the **Browse** button next to the **Archive file** field and browse to select *<Downloads>/junit381src.jar*
   Note: This step assumes that you followed steps 1 and 2 in the **Getting the Sample Code** section above.
7. In the Import wizard, expand the root node, press **Deselect All** and select only the **junit** node. You can expand and select elements within the *junit* directory on the left pane to view the individual resources that you are importing on the right pane. *Note: Do not deselect any of the resources in the junit directory at this time. You will need all of these resources in the tutorial.*



8. Make sure that the JUnit project's source folder appears in the destination **Into folder** field. Then click **Finish**. In the import progress indicator, notice that the imported resources are compiled as they are imported into the workbench. This is because the **Build automatically** option is checked on the Workbench

preferences page.

9. In the Package Explorer view, expand the JUnit project and the src folder to view the JUnit packages.



Next Section: **Browsing Java elements using the package explorer**

Related concepts

Java projects
Java views

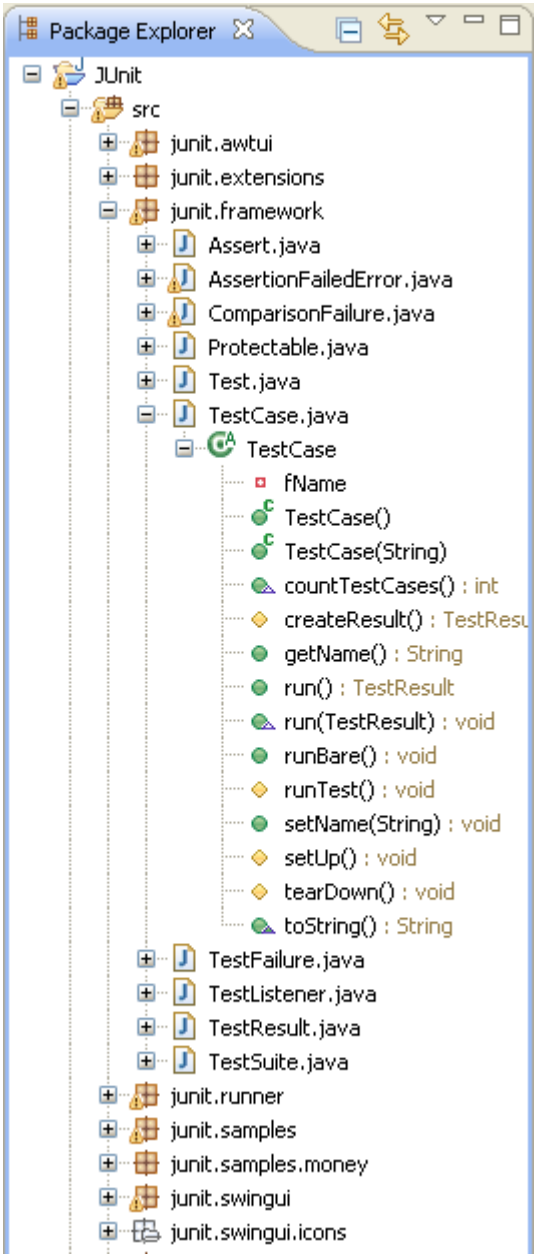
Related reference

New Java Project Wizard
Package Explorer View

Browsing Java elements using the package explorer

In this section, you will browse Java elements within the JUnit project.

1. In the Package Explorer view, make sure the JUnit project and the source folder are expanded so you can see the packages.
2. Expand the package `junit.framework` to see the Java files contained in the package.
3. Expand the Java file `TestCase.java`. Note that the Package Explorer shows Java-specific sub-elements of the source code file. The public type and its members (fields and methods) appear in the tree.



Next Section: Opening a Java editor

- Related concepts

Java views

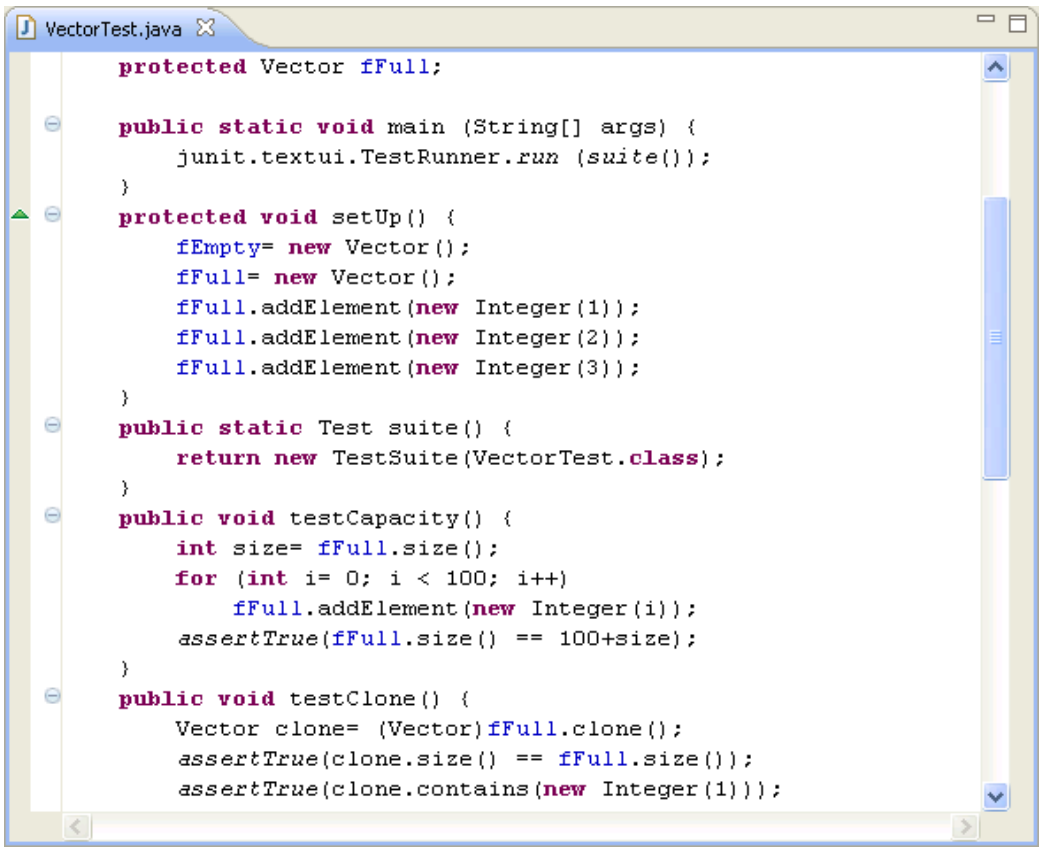
- Related reference

Package Explorer View

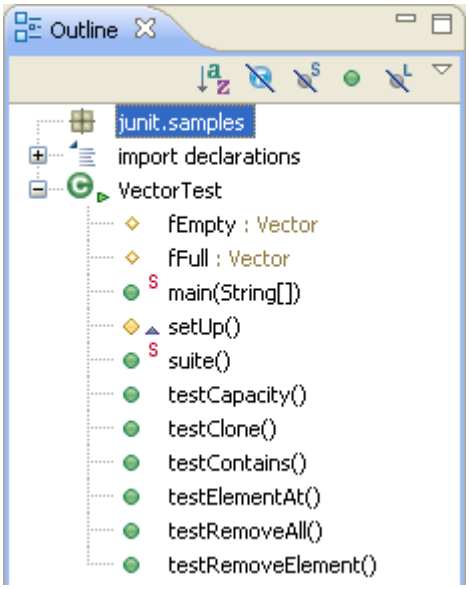
Opening a Java editor

In this section, you will learn how to open an editor for Java files. You will also learn about some of the basic Java editor features.

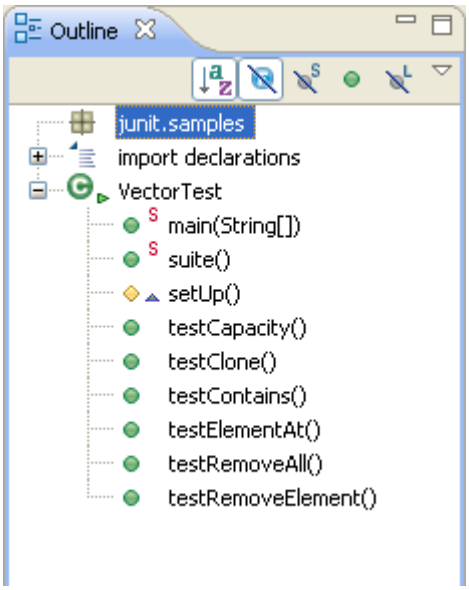
1. Expand the package *junit.samples* and select the file *VectorTest.java*. You can open *VectorTest.java* in the Java editor by double clicking on it. In general you can open a Java editor for Java files, types, methods and fields by simply double clicking on them. For example, to open the editor directly on the method *testClone* defined in *VectorTest.java*, expand the file in the Package Explorer and double click on the method.
2. Notice the syntax highlighting. Different kinds of elements in the Java source are rendered in unique colors. Examples of Java source elements that are rendered differently are:
   - Regular comments
   - Javadoc comments
   - Keywords
   - Strings



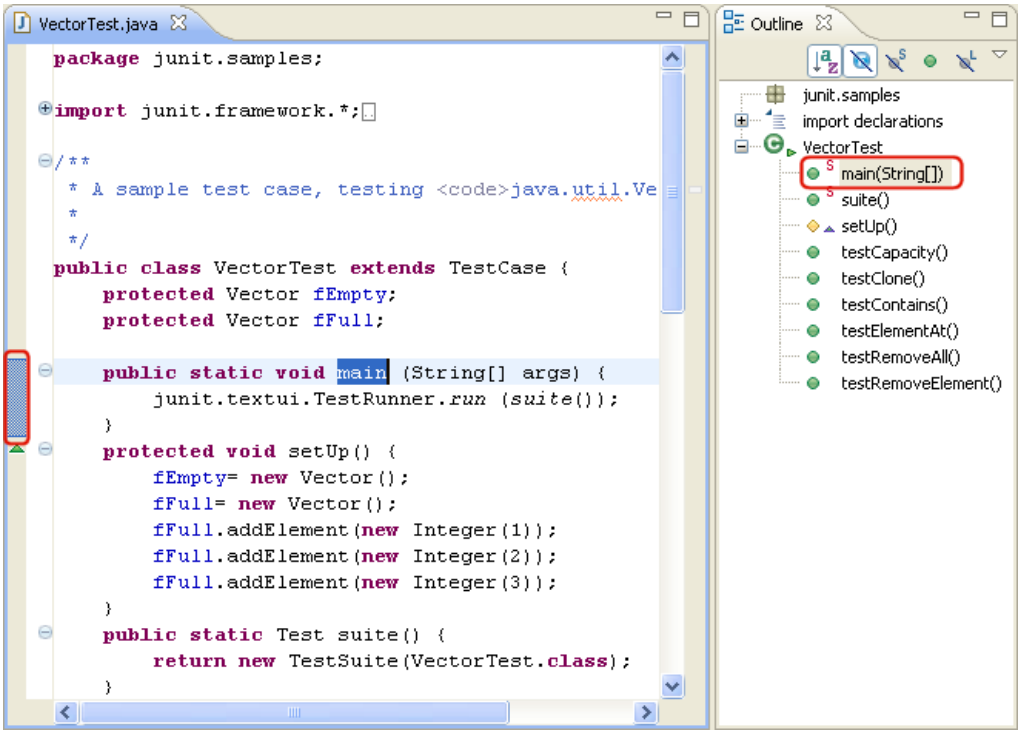
3. Look at the Outline view. It displays an outline of the Java file including the package declaration, import declarations, fields, types and methods. The Outline view uses icons to annotate Java elements. For example, icons indicate whether a Java element is static, abstract, or final. Different icons show you whether a method overrides a method from a base class or when it implements a method from an interface.



4. Toggle the **Hide Fields**, **Hide Static Members**, and **Hide Non-Public Members** buttons in the Outline view toolbar to filter the view's display. Before going to the next step make sure that the **Hide Non-Public Members** and **Hide Static Fields and Methods** buttons are not pressed.



5. In the Outline view, select different elements and note that they are again displayed in a whole file view in the editor. The Outline view selection now contains a range indicator on the vertical ruler on the left border of the Java editor that indicates the range of the selected element.



Next Section: Using quick views

Related concepts

Java views
Java editor
Sorting elements in Java views

Related reference

Java Outline View
Java Editor Preferences

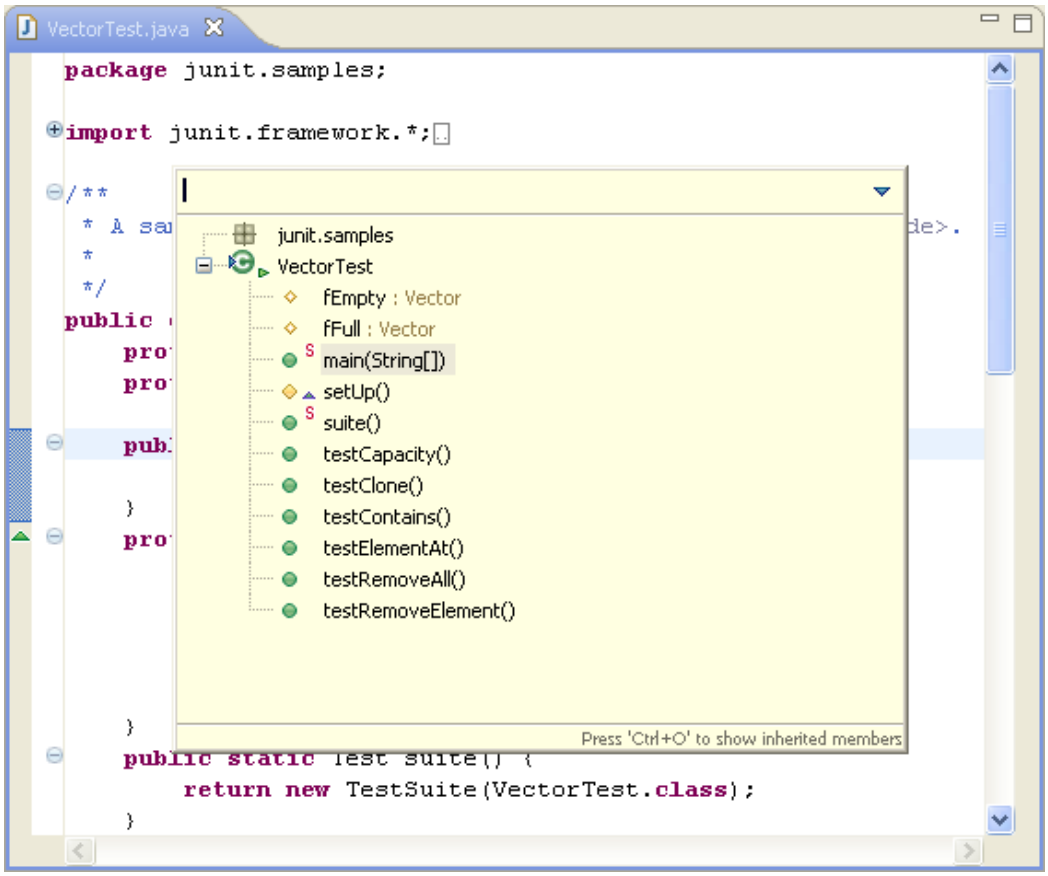
Using quick views

In this section, you will be introduced to the quick outline view. Quick views are in-place views which are shown on top of the editor area and can easily be controlled using the keyboard. A second quick view will be introduced in the Type Hierarchy section.

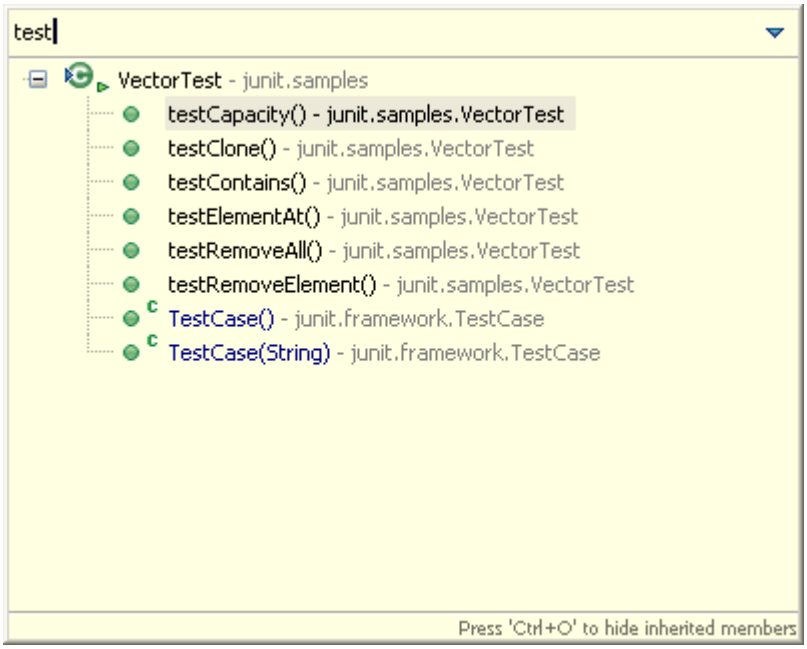
Quick Outline

To use the quick outline view in the Java editor:

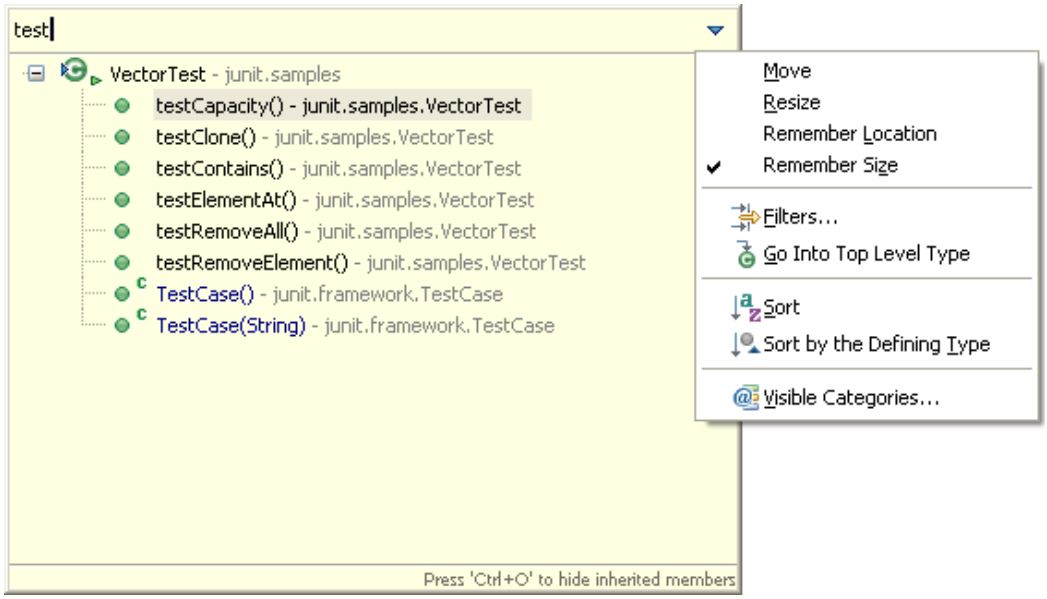
1. Open *junit.samples.VectorTest.java* file in the Java editor if you do not already have it open.
2. Press `Ctrl+O` or select **Navigate > Quick Outline** and you will see an in-place outline of the current source file.



3. Press `Ctrl+O` a second time and all inherited fields, types and methods are shown as well (for the types marked with ▸ on the left). Inherited members are shown in blue.
4. Start typing while the quick outline view is shown to filter the list of displayed elements. Further, use the arrow keys to navigate in the outline view and press `Enter` to reveal the selected element in the Java editor.



5. Click the triangle in the upper right corner to see the quick view menu:



The menu items can be divided into 3 categories:

- *Position* - Allows you to resize and move the quick view and to remember these settings
- *Filter* - Define filters so that not all members are shown in the quick outline.
- *Sort* - Sort the members by their defining type or alphabetically.

Note: `Ctrl+O` always opens the outline for the current Java editor. Press `Ctrl+F3` to open the quick outline for the currently selected type.

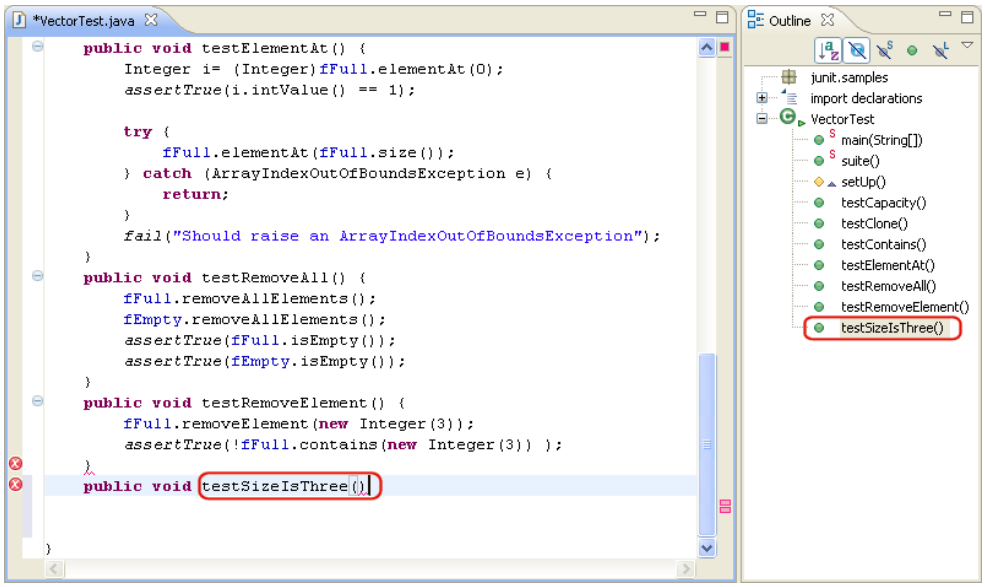
Next Section: Adding new methods

Adding new methods

1. Start adding a method by typing the following at the end of the *VectorTest.java* file (but before the closing brace of the type) in the Java editor:

```
public void testSizeIsThree()
```

As soon as you type the method name in the editor area, the new method appears at the bottom of the Outline view.

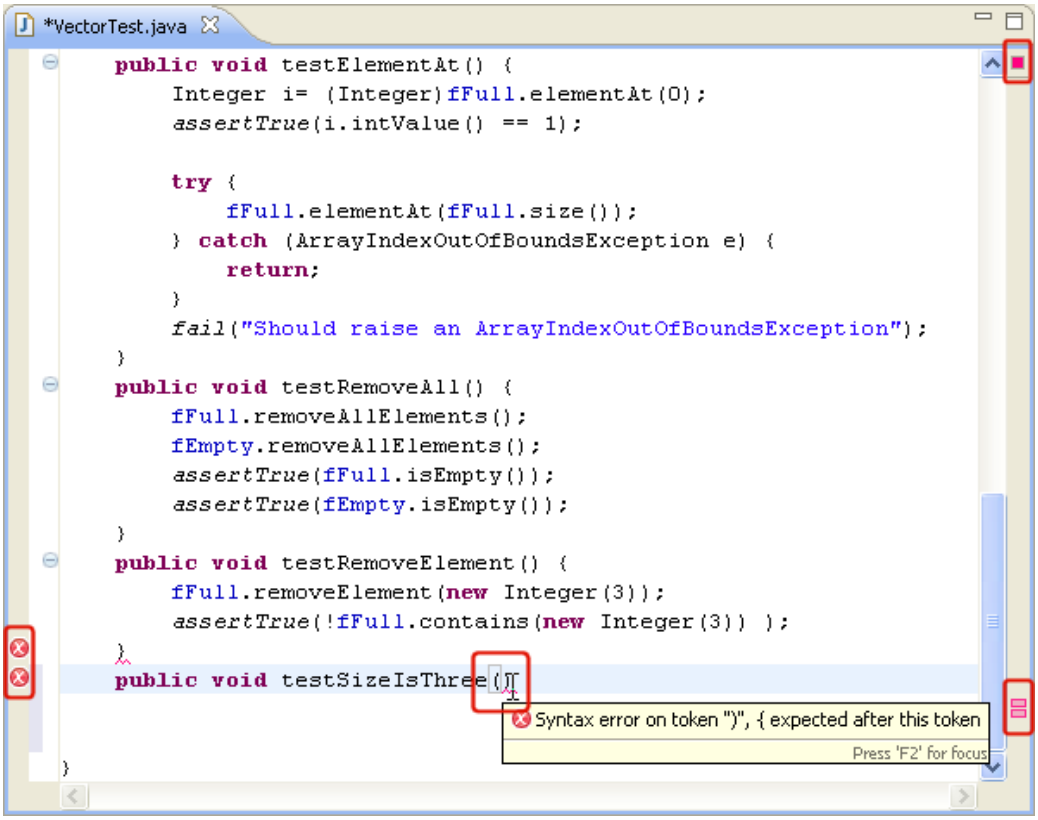


In addition,

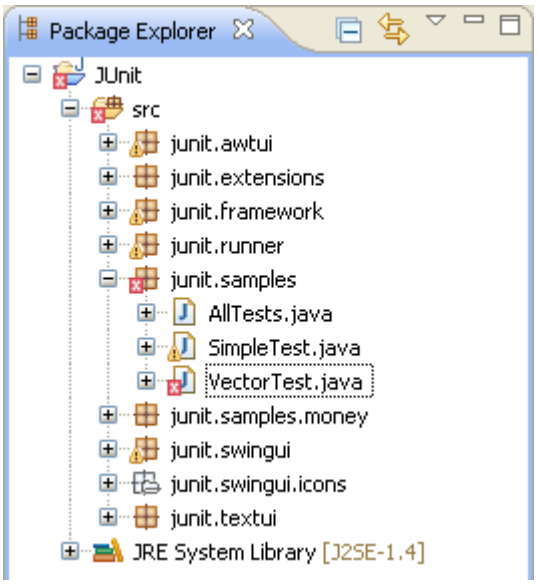
- error annotations (red boxes) appear in the overview ruler positioned on the right hand side of the editor,
- error icons appear in the vertical ruler positioned on the left of the editor,

- an error indicator appears in the top right corner of the editor,
- errors are marked in the text.

These error annotations indicate that the compilation unit is currently not correct. If you hover over the error in the text, a tool tip appears: *Syntax error on token ")", { expected after this token*. This is correct since the method doesn't have a body yet. Note that error annotations in the editor are updated as you type. This behavior can be controlled via the **Report problems as you type** option located on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor** preference page.



2. Click the **Save** button. The compilation unit is compiled automatically and errors appear in the Package Explorer view, in the Problems view and on the vertical ruler (left hand side of the editor). In the Package Explorer view, the errors are propagated up to the project of the compilation unit containing the error.



3. Complete the new method by typing the following:
   {

```
        //TODO: Check size
```

   Note that the closing curly brace has been inserted automatically.

4. Save the file. Notice that the error indicators disappear since the missing brace has been added.

Next Section: Using content assist

- Related concepts

Java editor

- Related reference

Java Editor Preferences

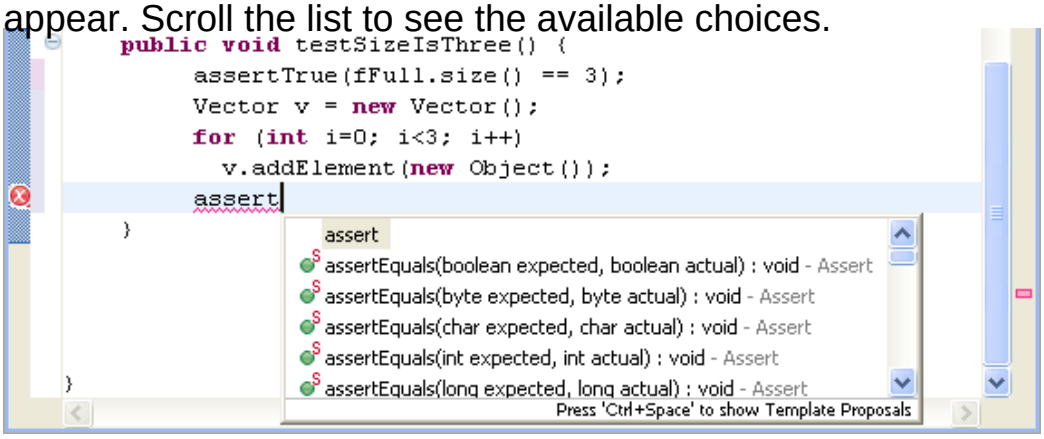
Using content assist

In this section you will use *content assist* to finish writing a new method. Open the file *junit.samples/VectorTest.java* in the Java editor if you do not already have it open and select the `testSizeIsThree()` method in the Outline view. If the file doesn't contain such a method see Adding new methods for instructions on how to add this method.

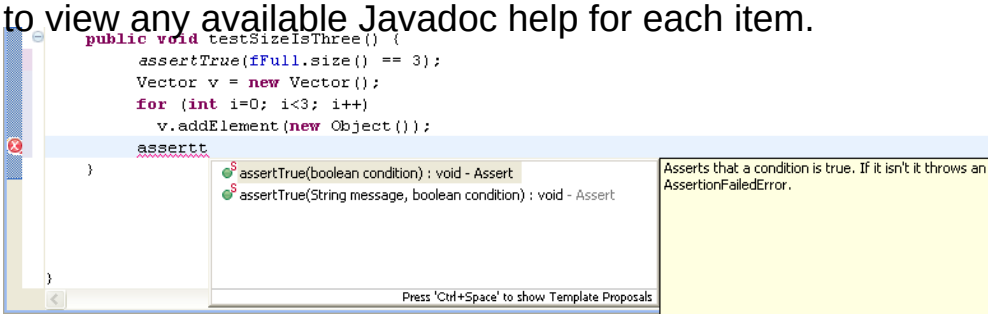
1. Replace the TODO comment with the following lines.

```
assertTrue(fFull.size() == 3);
  Vector v = new Vector();
  for (int i=0; i<3; i++)
    v.addElement(new Object());
  assert
```

2. With your cursor at the end of the word `assert`, press `Ctrl+Space` to activate content assist. The content assist window with a list of proposals will appear. Scroll the list to see the available choices.



3. With the content assist window still active, type the letter 't' in the source code after `assert` (with no space in between). The list is narrowed and only shows entries starting with 'assert'. Single-click various items in the list to view any available Javadoc help for each item.



4. Select `assertTrue(boolean)` from the list and press `Enter`. The code for the `assertTrue(boolean)` method is inserted.
5. Complete the line so that it reads as follows:

```
assertTrue(v.size() == fFull.size());
```

6. Save the file.

Next Section: Identifying problems in your code

Related concepts

Java editor

Related reference

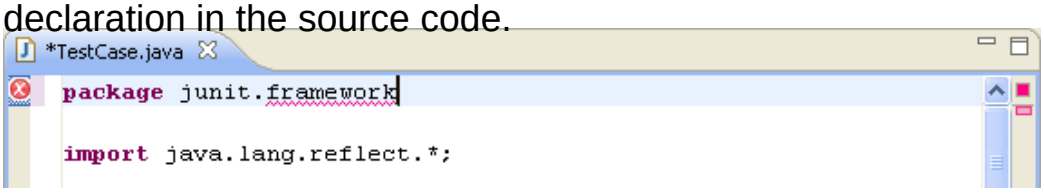
Java Content Assist
Java Editor Preferences

Identifying problems in your code

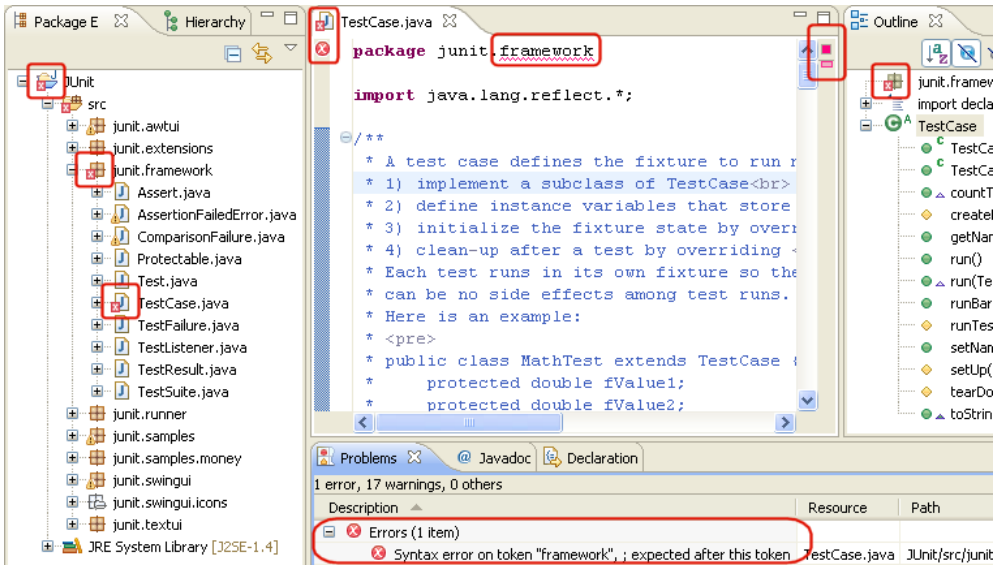
In this section, you will review the different indicators for identifying problems in your code.

Build problems are displayed in the Problems view and annotated in the vertical ruler of your source code.

1. Open *junit.framework/TestCase.java* in the editor from the Package Explorer view.
2. Add a syntax error by deleting the semicolon at the end of the package declaration in the source code.



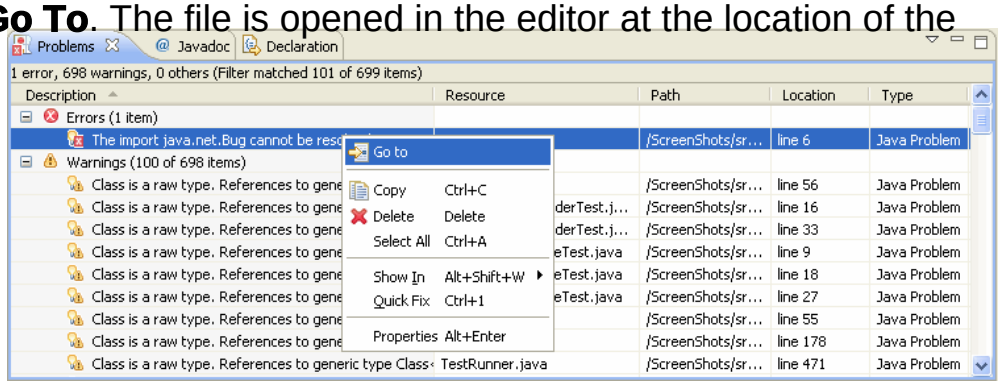
3. Click the **Save** button. The project is rebuilt and the problem is indicated in several ways:
   - In the Problems view, the problems are listed,
   - In the Package Explorer view, the Type Hierarchy or the Outline view, problem ticks appear on the affected Java elements and their parent elements,
   - In the editor's vertical ruler, a problem marker is displayed near the affected line,
   - Squiggly lines appear under the word which might have caused the error, and
   - The editor tab is annotated with a problem marker.



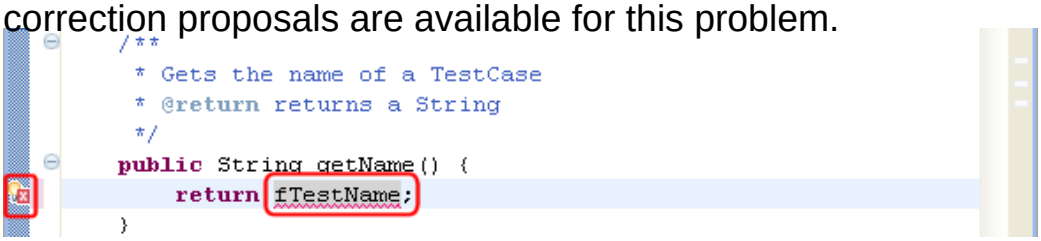
4. You can hover over the marked word in the editor to view a description of the problem. You can also hover over the problem markers in the vertical or overview rulers.

5. Click the **Close** ("X") button on the editor's tab to close the editor.

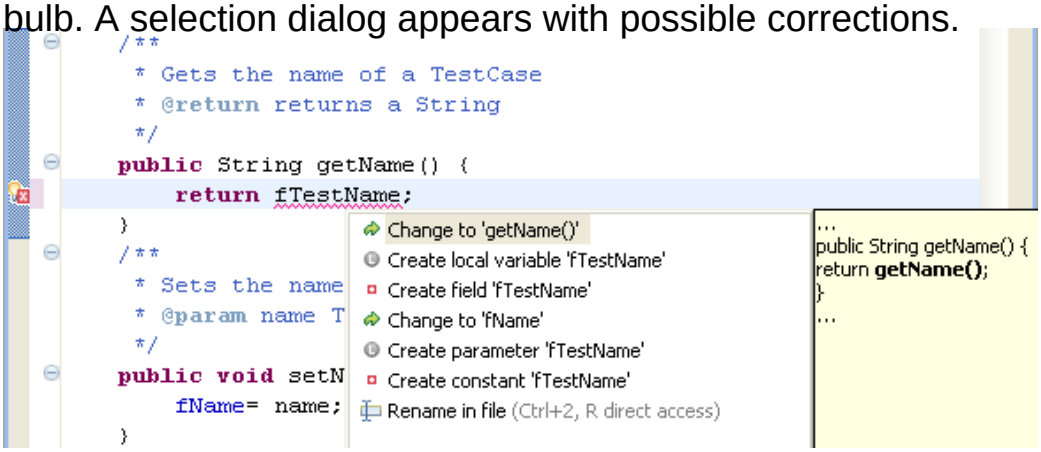
6. In the Problems view, select a problem in the list. Open its context menu and select **Go To**. The file is opened in the editor at the location of the problem.



7. Correct the problem in the editor by adding the semicolon. Click the **Save** button. The project is rebuilt and the problem indicators disappear.
8. In the Outline view, select the method *getName()*. The editor will scroll to this method.
9. On the first line of the method change the returned variable `fName` to `fTestName`. While you type, a problem highlight underline appears on fTestName, to indicate a problem. Hovering over the highlighted problem will display a description of the problem and applicable quick fixes.
10. On the marker bar a light bulb marker appears. The light bulb signals that correction proposals are available for this problem.



11. Click to place the cursor onto the highlighted error, and choose **Quick Fix** from the Edit menu bar. You can also press `Ctrl+1` or left click the light bulb. A selection dialog appears with possible corrections.



12. Select 'Change to fName' to fix the problem. The problem highlight line will disappear as the correction is applied.
13. Close the file without saving.
14. You can configure how problems are indicated on the [Image: /topic/org.eclipse.help/command_link.png]**General > Editors > Text Editors > Annotations** preference page.

Next Section: Using code templates

Related concepts

Java editor

Java views

Java builder

Related reference

Editor preference page
Quick fix

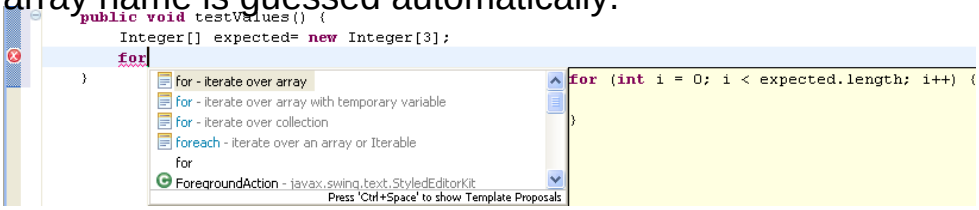
Using code templates

In this section you will use content assist to fill in a template for a common loop structure. Open *junit.samples/VectorTest.java* file in the Java editor if you do not already have it open.

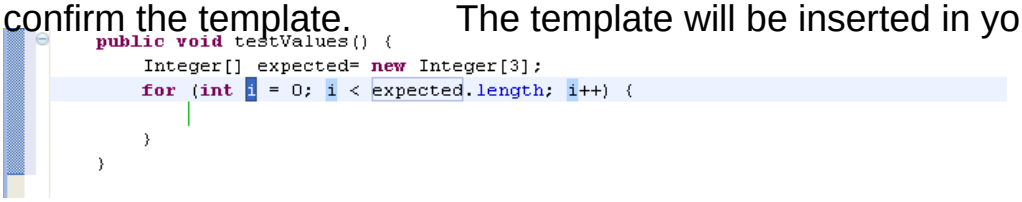
1. Start adding a new method by typing the following:

```
public void testValues() {
      Integer[] expected= new Integer[3];
      for
```

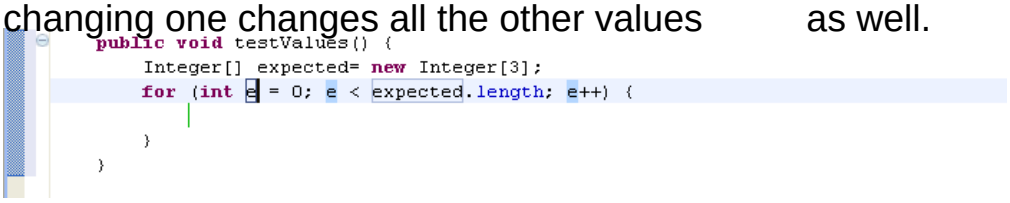
2. With the cursor at the end of `for`, press `Ctrl+Space` to enable content assist. You will see a list of common templates for "for" loops. When you single-click a template, or select it with the `Up` or `Down` arrow keys, you'll see the code for the template in its help message. Note that the local array name is guessed automatically.



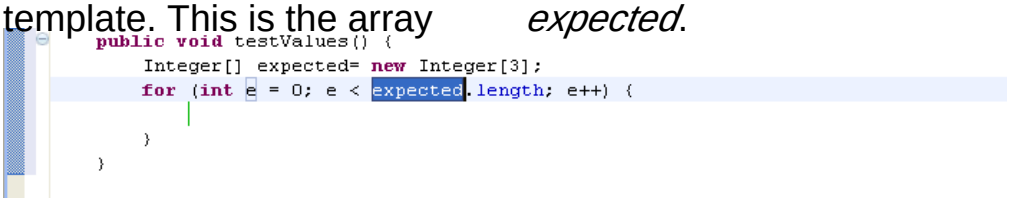
3. Choose the `for - iterate over array` entry and press `Enter` to confirm the template. The template will be inserted in your source code.



4. Next we change the name of the index variable from *i* to *e*. To do so simply press `e`, as the index variable is automatically selected. Observe that the name of the index variable changes at all places. When inserting a template all references to the same variable are connected to each other. So changing one changes all the other values as well.



5. Pressing the `tab` key moves the cursor to the next variable of the code template. This is the array *expected*.



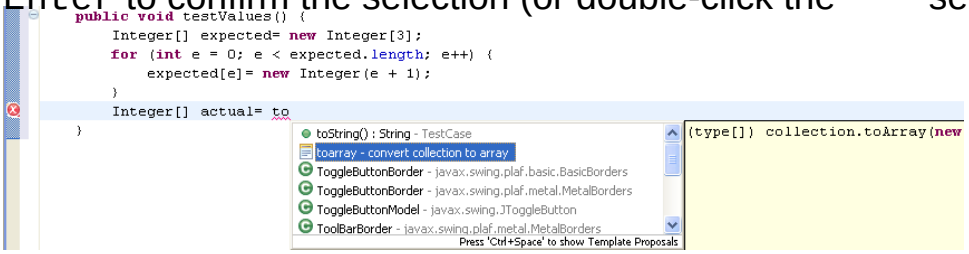
Since we don't want to change the name (it was guessed right by the template) we press tab again, which leaves the template since there aren't any variables left to edit.

6. Complete the `for` loop as follows:

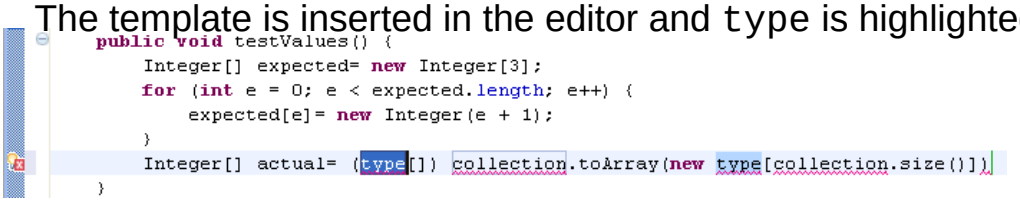
```
for (int e= 0; e < expected.length; e++) {
```

```
            expected[e]= new Integer(e + 1);
        }
        Integer[] actual= to
```

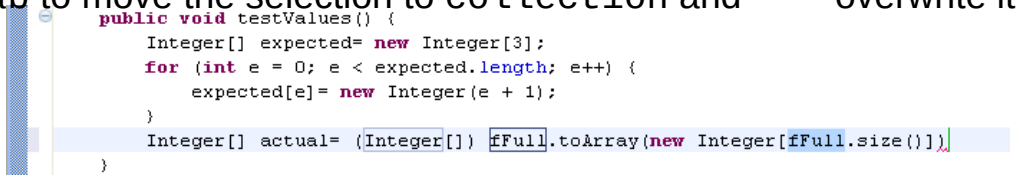
7. With the cursor at the end of `to`, press `Ctrl+Space` to enable content assist. Pick `toarray - convert collection to array` and press `Enter` to confirm the selection (or double-click the selection).



The template is inserted in the editor and `type` is highlighted and selected.



8. Overwrite the selection by typing `Integer`. The type of array constructor changes when you change the selection.
9. Press `Tab` to move the selection to `collection` and overwrite it by typing `fFull`.



10. Add a semicolon and the following lines of code to complete the method:

```
assertEquals(expected.length, actual.length);
  for (int i= 0; i < actual.length; i++)
      assertEquals(expected[i], actual[i]);
```

11. Save the file.

Next Section: Organizing import statements

Related concepts

Java editor
Templates

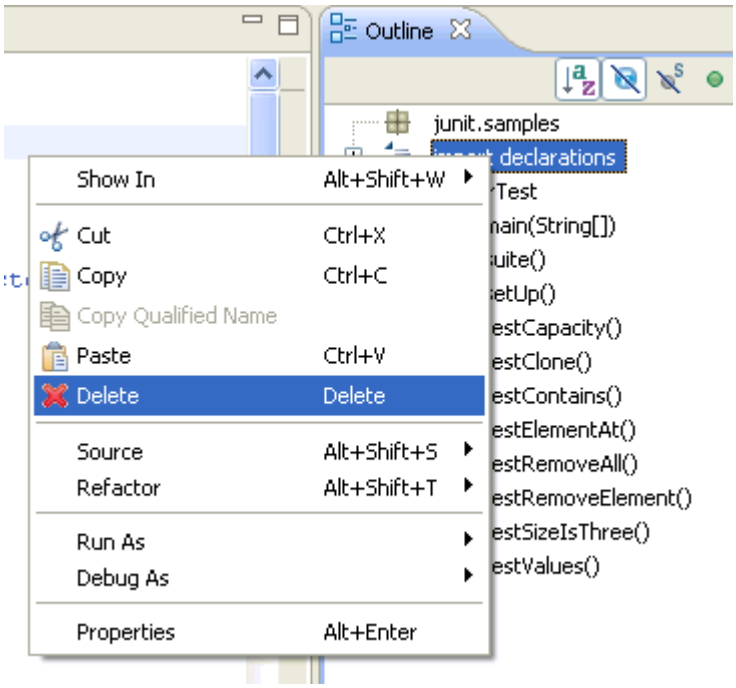
Related reference

Templates Preferences
Java Editor Preferences

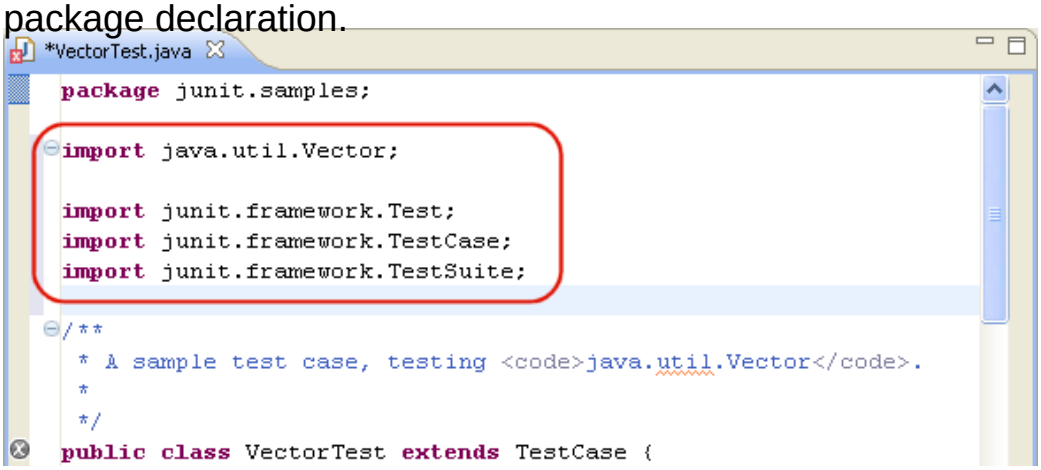
Organizing import statements

In this section you will organize the import declarations in your source code. Open *junit.samples/VectorTest.java* file in the Java editor if you do not already have it open.

1. Delete the import declarations by selecting them in the Outline view and selecting **Delete** from the context menu. Confirm the resulting dialog with **Yes**. You will see numerous compiler warnings in the vertical ruler since the types used in the method are no longer imported.



2. From the context menu in the editor, select **Source > Organize Imports**. The required import statements are added to the beginning of your code below the package declaration.



You can also choose **Organize Imports** from the context menu of the import declarations in the Outline view.

Note: You can specify the order of the import declarations using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Organize Imports** preference page.

3. Save the file.

Next Section: Using the local history

- Related concepts

Java editor

- Related reference

Organize Imports Preferences

Using the local history

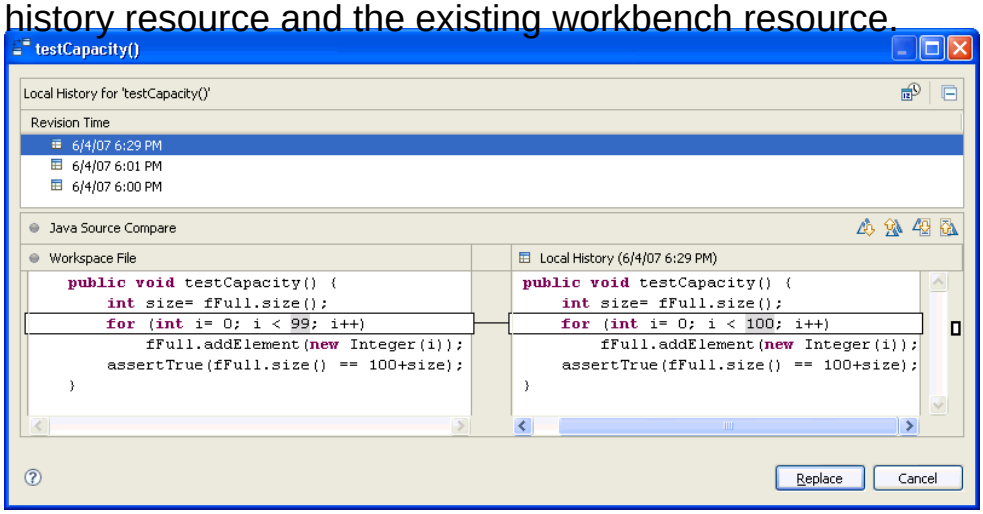
In this section, you will use the local history feature to switch to a previously saved version of an individual Java element.

1. Open *junit.samples/VectorTest.java* file in the Java editor and select the method *testCapacity()* in the Outline view.
2. Change the content of the method so that the 'for' statements reads as:

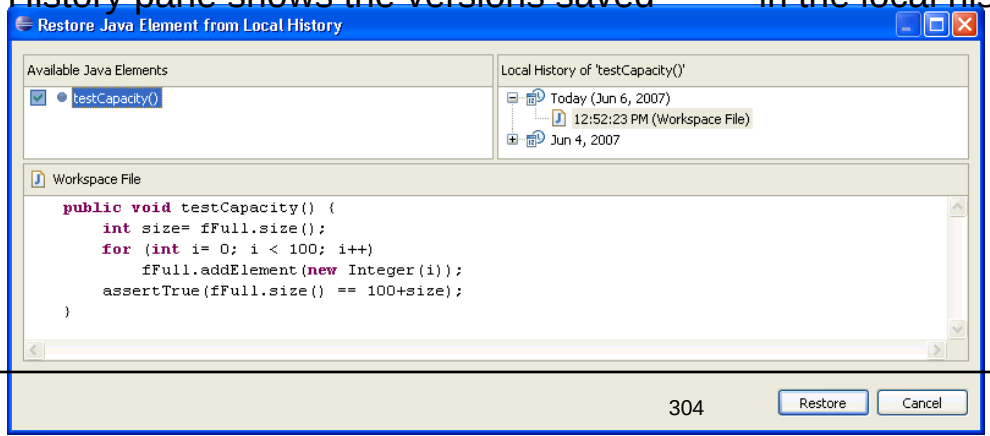
```
for (int i= 0; i < 99; i++)
```

Save the file by pressing `Ctrl+S`.

3. In the Outline view, select the method `testCapacity()`, and from its context menu, select **Replace With > Element from Local History**.
4. In the **Replace Java Element from Local History** dialog, the Local History list shows the various saved states of the method. The Java Source Compare pane shows details of the differences between the selected history resource and the existing workbench resource.



5. In the top pane, select the previous version, and click the **Replace button**. In the Java editor, the method is replaced with the selected history version.
6. Save the file.
7. Beside replacing a method's version with a previous one you can also restore Java elements that were deleted. Again, select the method *testCapacity()* in the Outline view, and from its context menu, select **Delete**. Confirm the resulting dialog with **Yes** and save the file.
8. Now select the type *VectorTest* in the Outline view, and from its context menu, select **Restore from Local History...**. Select and check the method *testCapacity()* in the Available Java Elements pane. As before, the Local History pane shows the versions saved in the local history.



9. In the Local History pane, select the earlier working version and then click **Restore**.

Next Section: Extracting a new method

- Related concepts

Java editor

- Related tasks

Using the local history

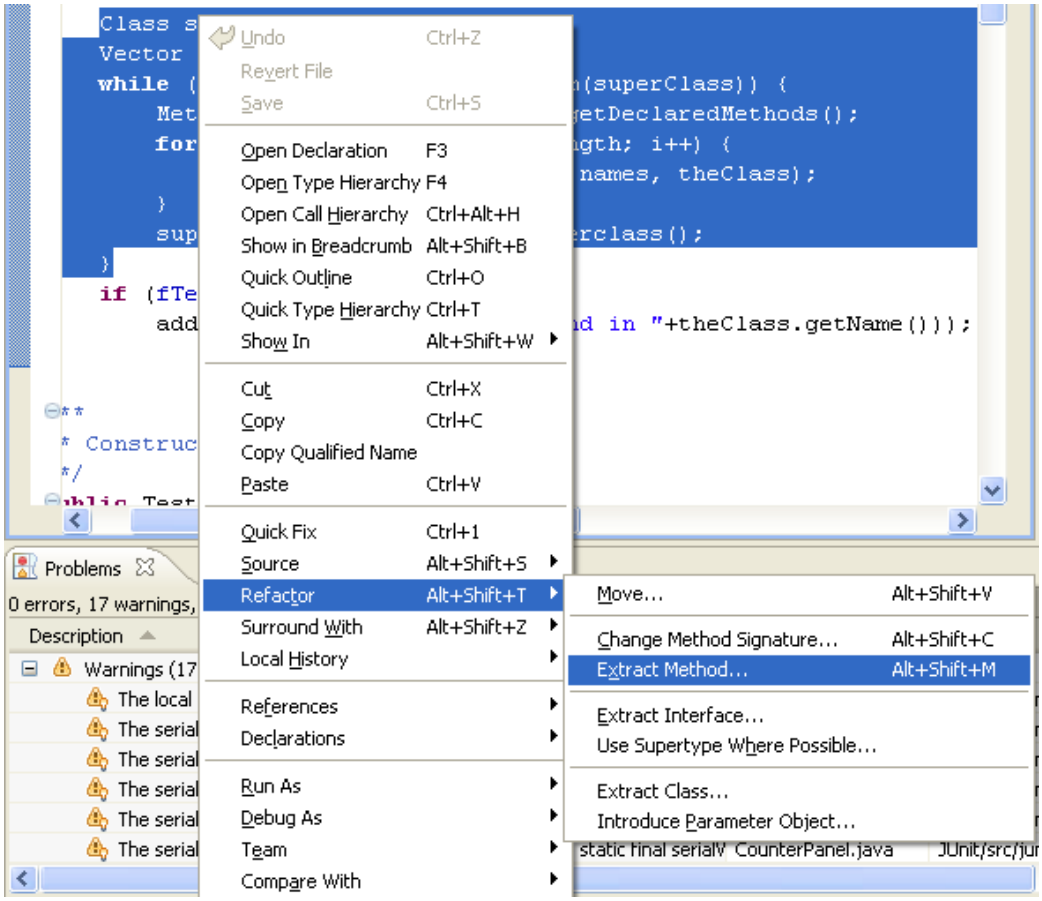
Extracting a new method

In this section, you will improve the code of the constructor of *junit.framework.TestSuite*. To make the intent of the code clearer, you will extract the code that collects test cases from base classes into a new method called *collectInheritedTests*.

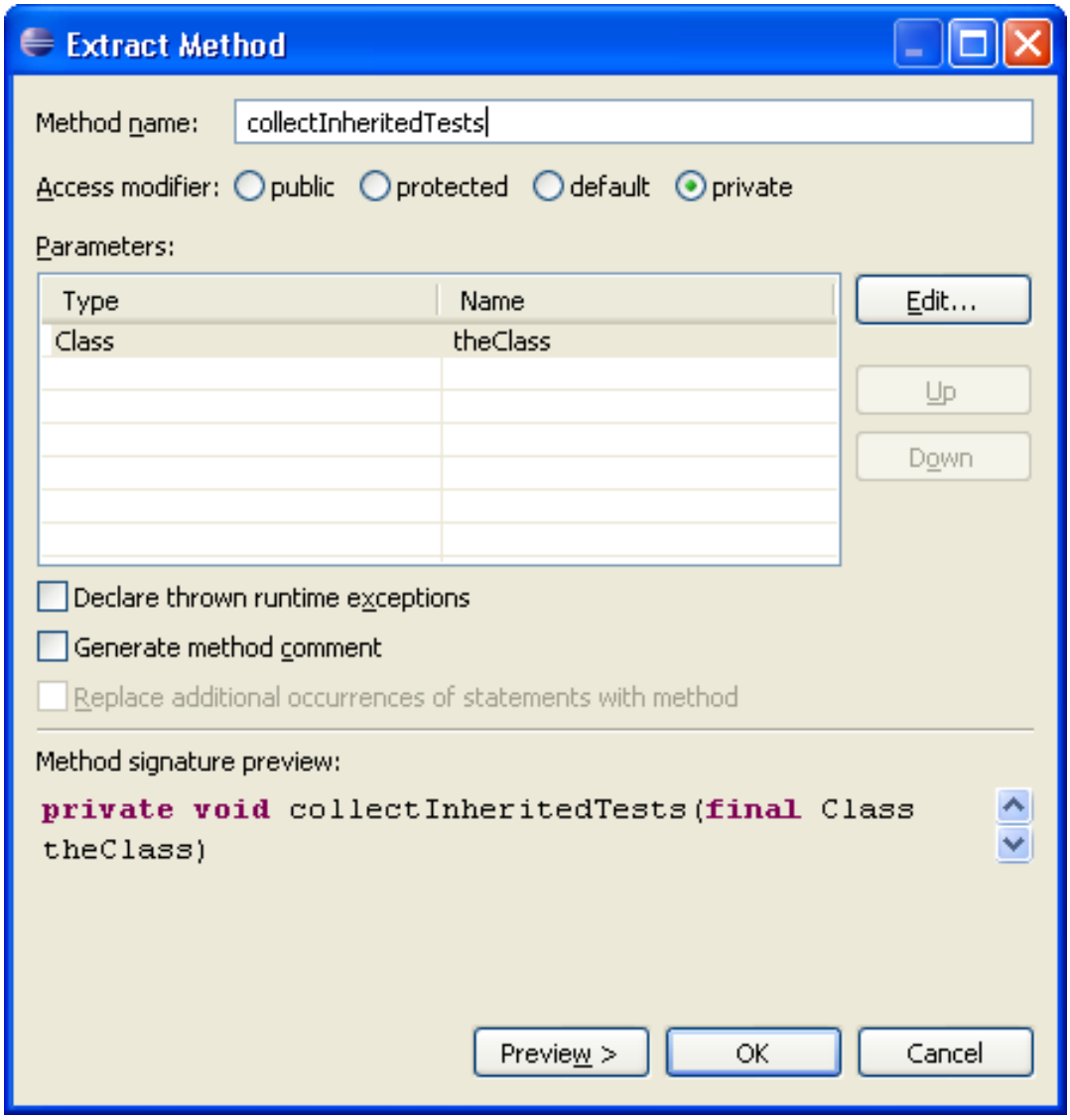
1. In the *junit.framework/TestSuite.java* file, select the following range of code inside the *TestSuite(Class)* constructor:

```
        Class superClass= theClass;
          Vector names= new Vector();
          while(Test.class.isAssignableFrom(superClass)) {
              Method[] methods=
superClass.getDeclaredMethods();
              for (int i= 0; i < methods.length; i++) {
                  addTestMethod(methods[i],names,
constructor);
              }
              superClass= superClass.getSuperclass();
          }
```

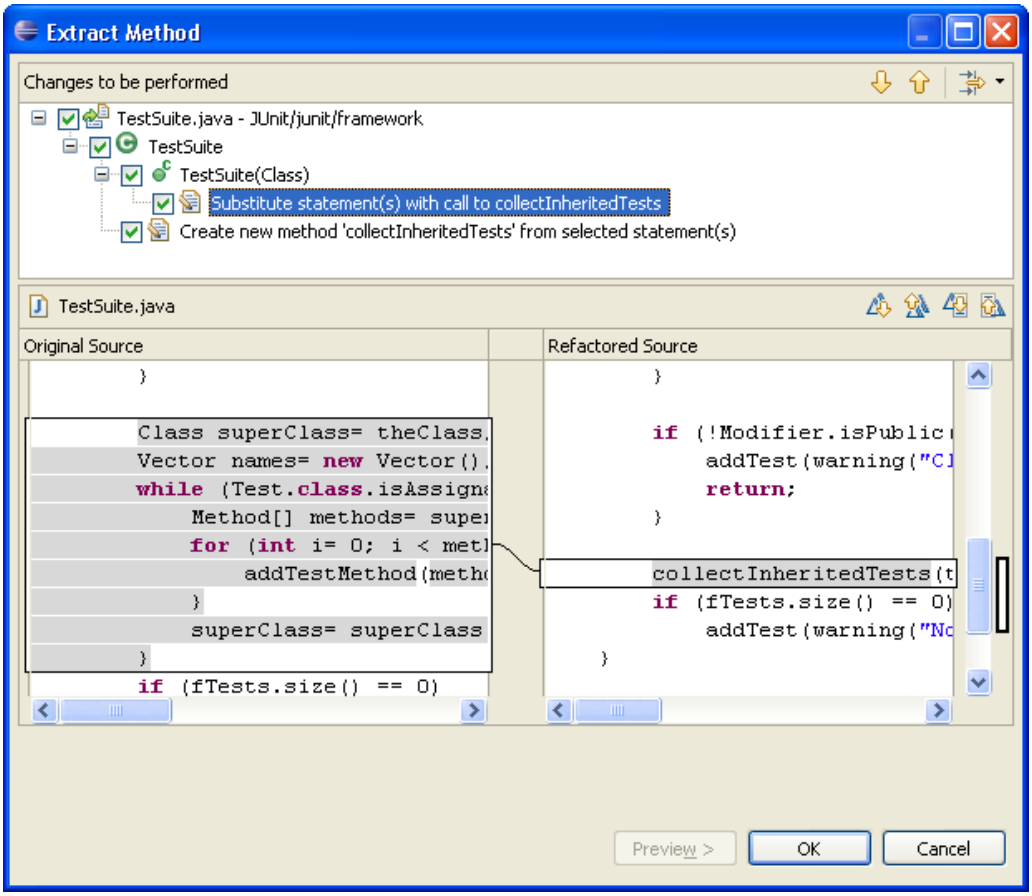
2. From the selection's context menu in the editor, select **Refactor > Extract Method...**.



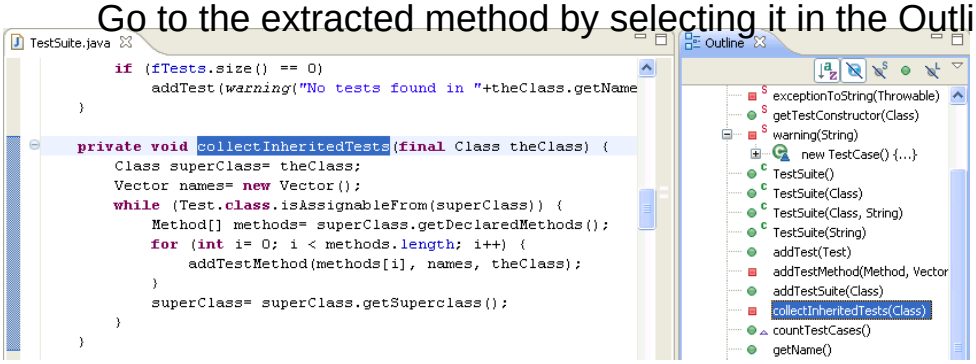
3. In the **Method Name** field, type *collectInheritedTests*.



4. To preview the changes, press **Preview>**.The preview page displays the changes that will be made. Press **OK** to extract the method.



5. Go to the extracted method by selecting it in the Outline view.



Next Section: Creating a Java class

Related concepts

Java editor
Refactoring support

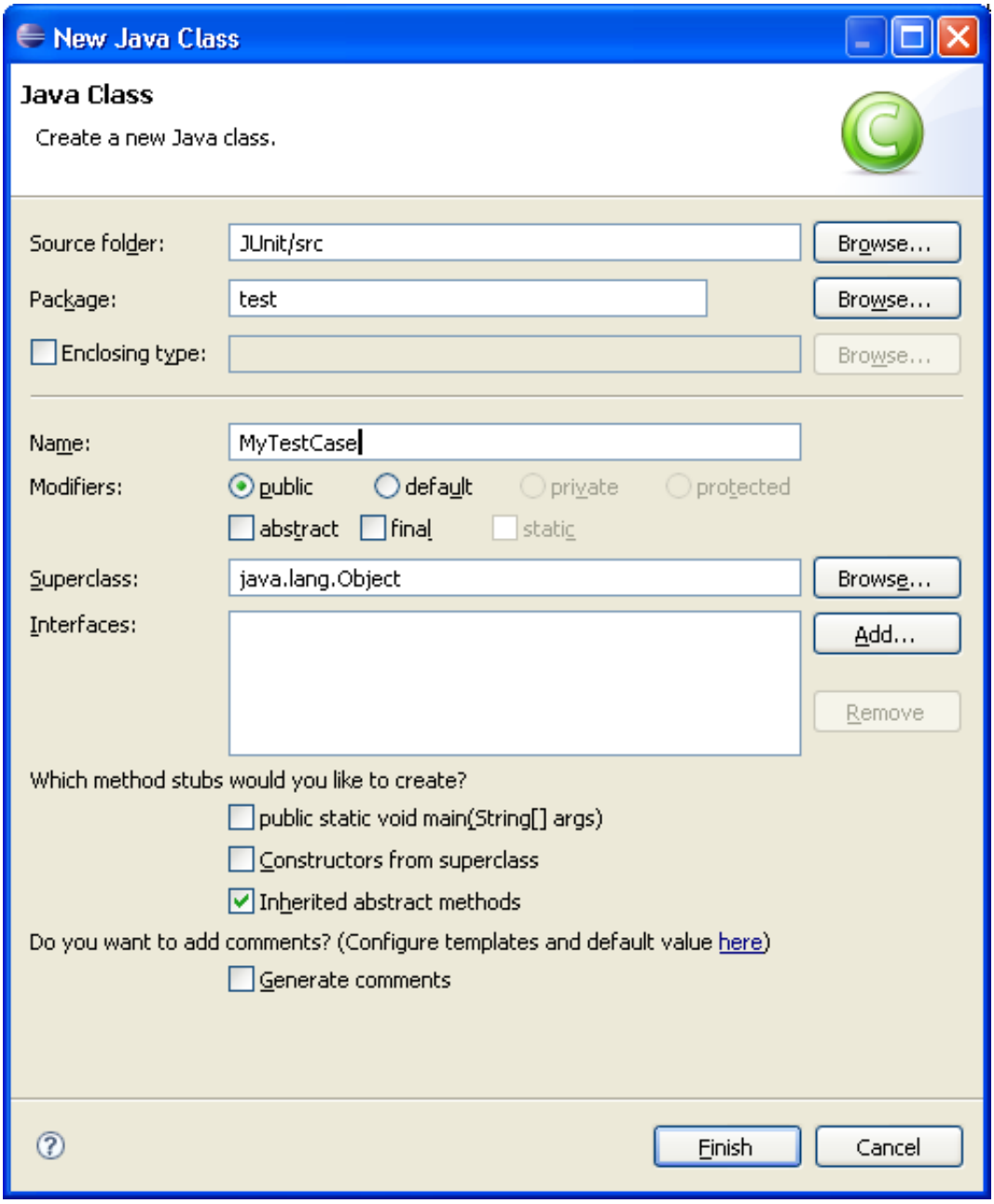
Related reference

Java Preferences

Creating a Java class

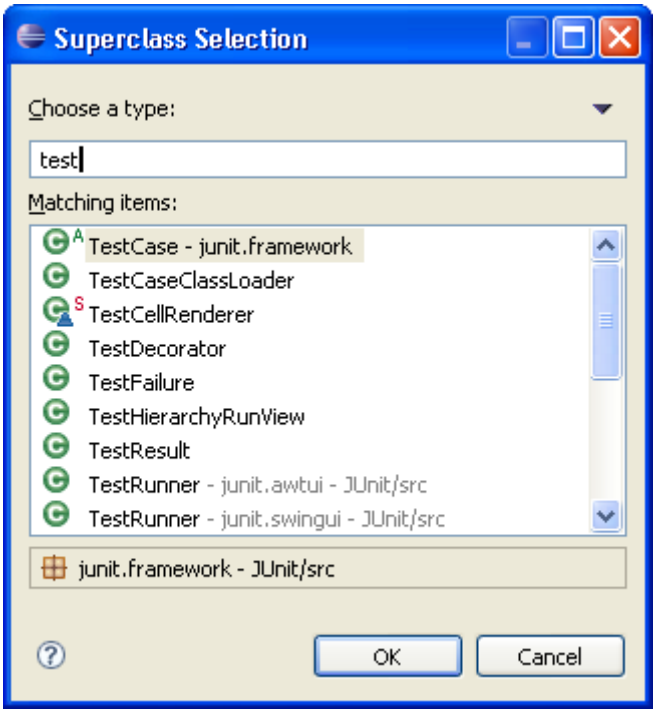
In this section, you will create a new Java class and add methods using code generation actions.

1. In the Package Explorer view, select the JUnit project. Click the **New Java Package** button in the toolbar, or select **New > Package** from the project's context menu .
2. In the **Name** field, type *test* as the name for the new package. Then click **Finish**.
3. In the Package Explorer view, select the new *test* package and click the **New Java Class** button in the toolbar.
4. Make sure that *JUnit* appears in the **Source Folder** field and that *test* appears in the **Package** field. In the **Name** field, type *MyTestCase*.

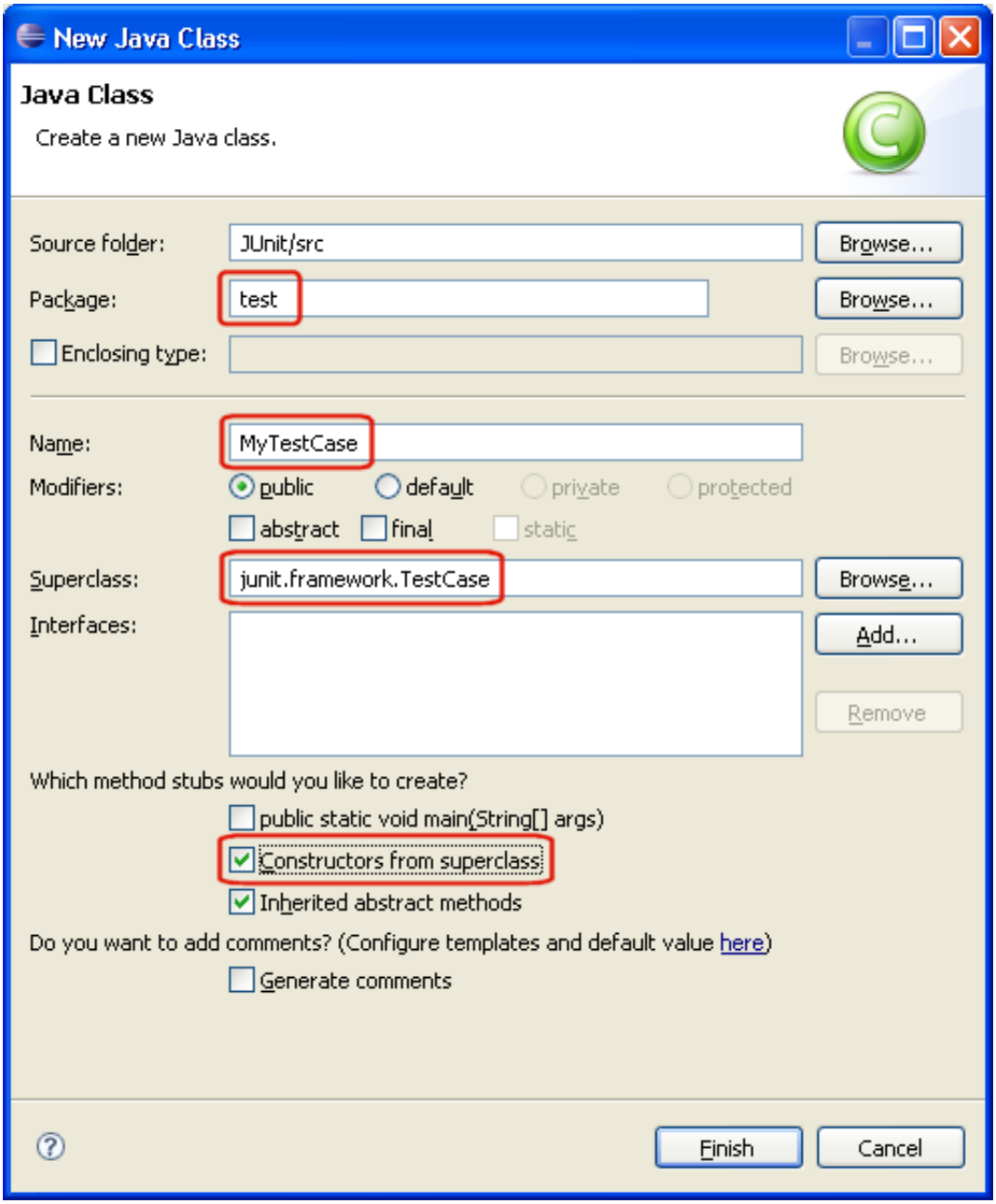


5. Click the **Browse** button next to the **Superclass** field.

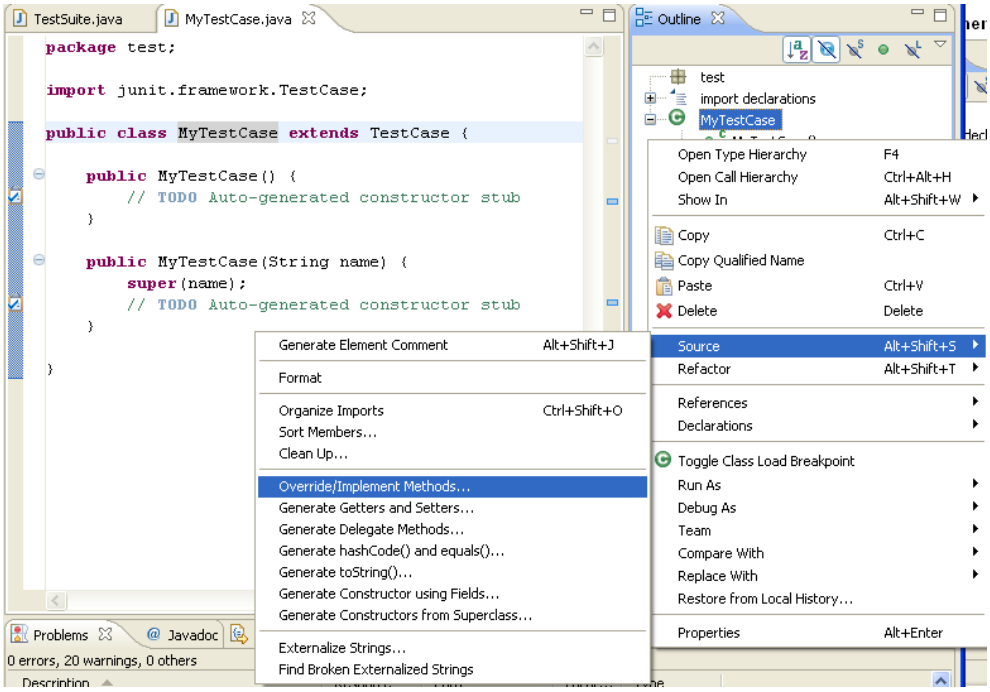
6. In the **Choose a type** field in the Superclass Selection dialog, type *Test* to narrow the list of available superclasses.



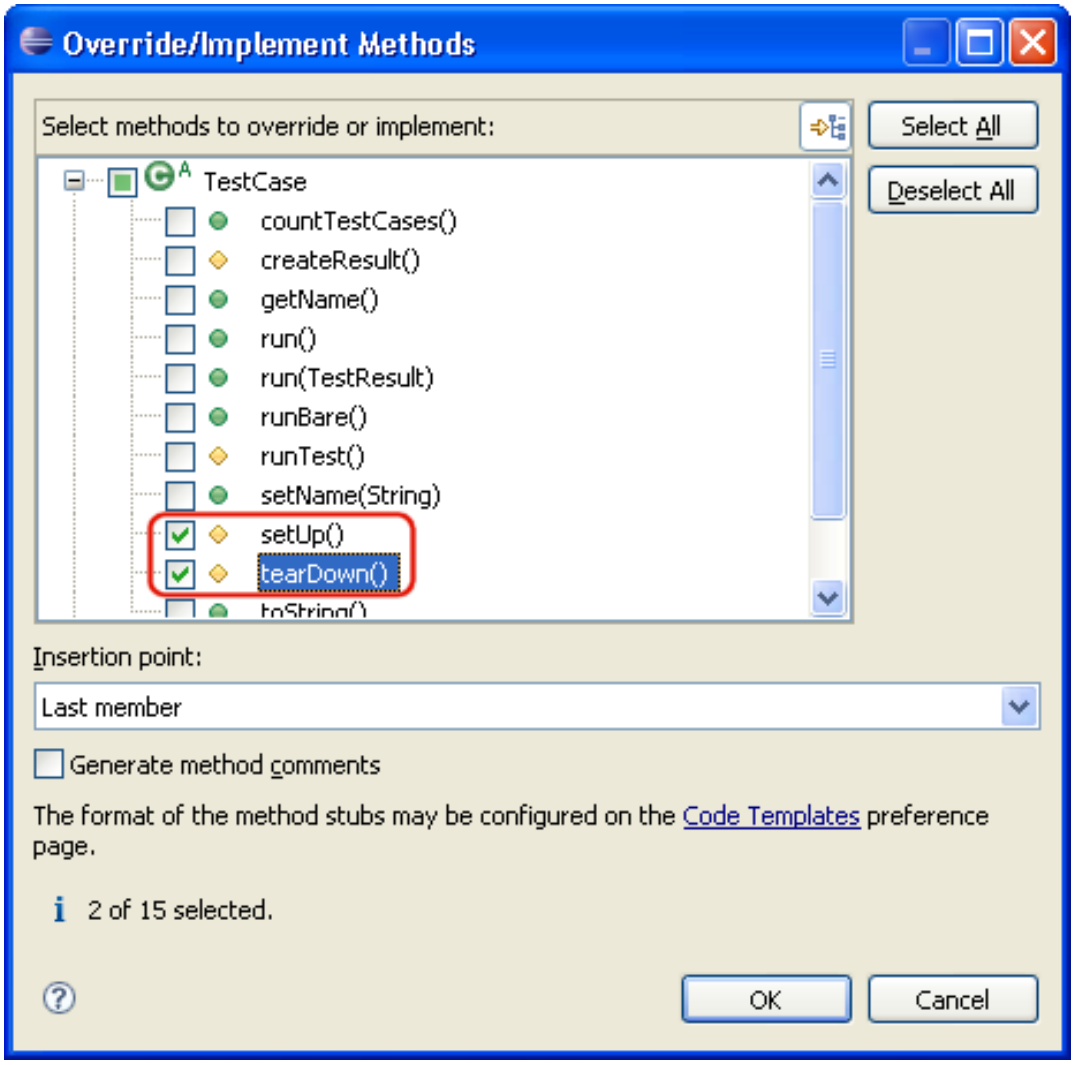
7. Select the *TestCase* class and click **OK**.
8. Select the checkbox for **Constructors from superclass**.
9. Click **Finish** to create the new class.



10. The new file is opened in the editor. It contains the new class, the constructor and comments. You can select options for the creation and configuration of generated comments in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Code Templates** preference page.
11. In the Outline view select the new class *MyTestCase*. Open the context menu and select **Source > Override/Implement Methods...**.

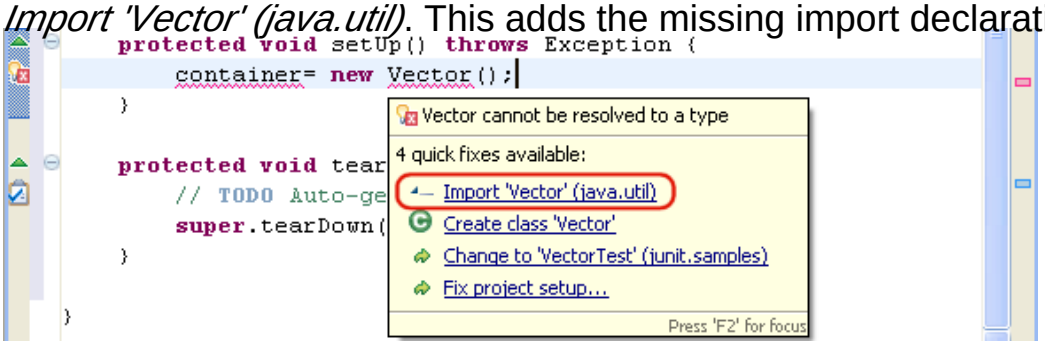


12. In the Override Methods dialog, check *setUp()* and *tearDown()* and click **OK**. Two methods are added to the class.

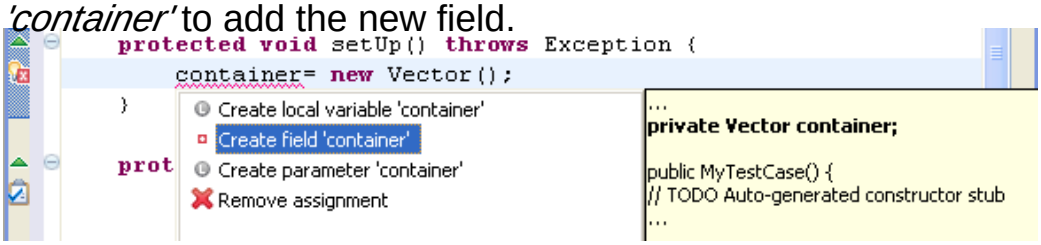


13. Change the body of *setUp()* to `container= new Vector();`

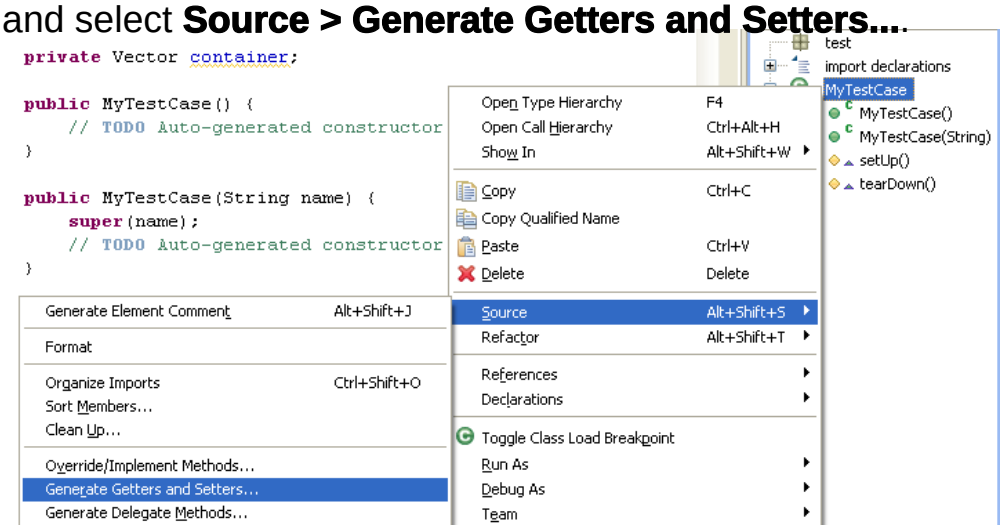
14. *container* and *Vector* are underlined with a problem highlight line as they cannot be resolved. A light bulb appears on the marker bar. Move the mouse over the underlined word *Vector*. A hover appears that shows the error message and contains quick fixes. Move the mouse into the hover and click *Import 'Vector' (java.util)*. This adds the missing import declaration.



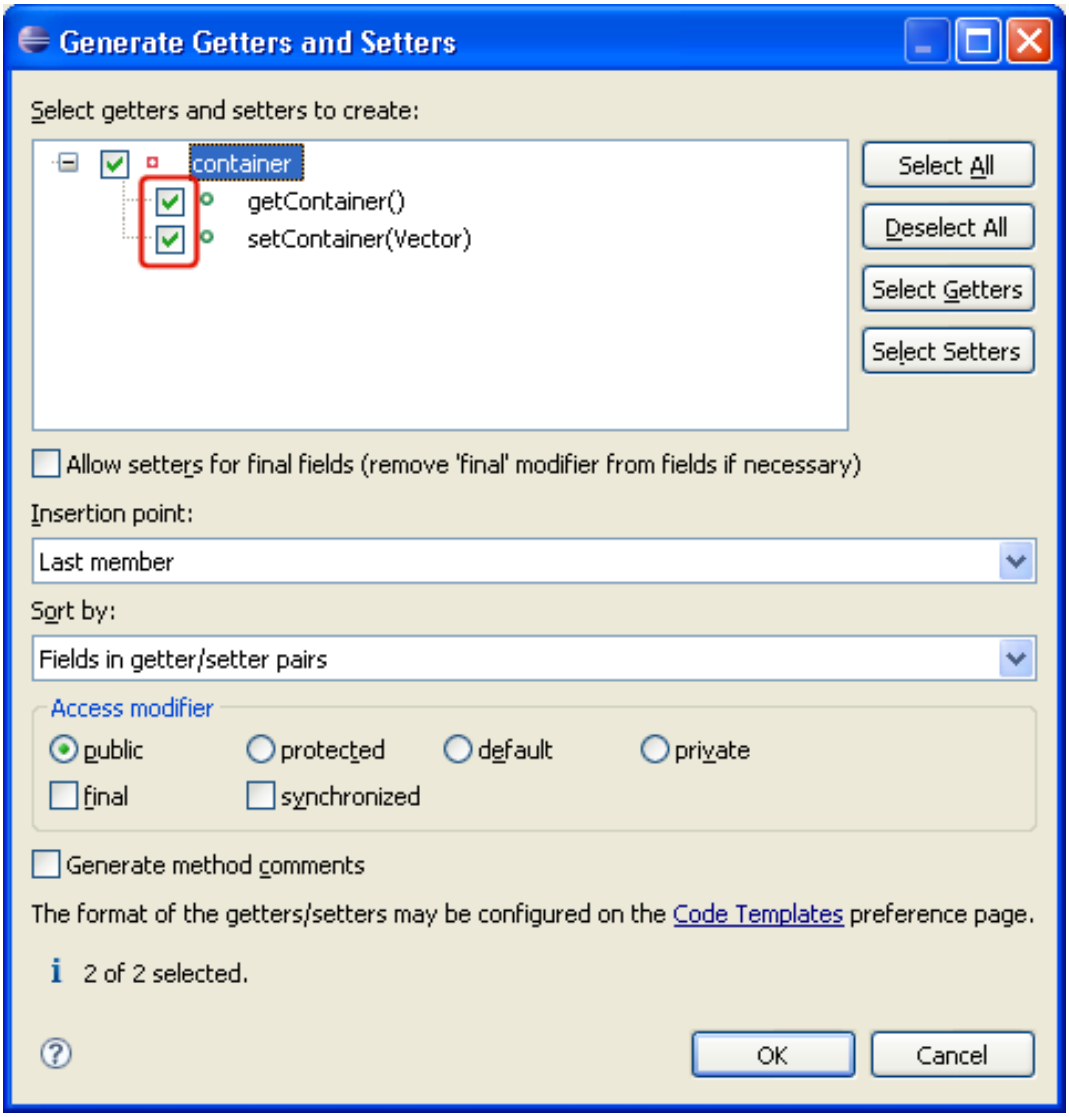
The blinking cursor should still be on the line that contains the error (if not, set it to the end of the line). Press `Ctrl+1`. This lets the cursor jump to the nearest error location and opens the quick fix proposals. Choose *Create field 'container'* to add the new field.



15. In the Outline view, select the class *MyTestCase*. Open the context menu and select **Source > Generate Getters and Setters...**.



16. The Generate Getter and Setter dialog suggests that you create the methods getContainer and setContainer. Select both and click **OK**. A getter and setter method for the field container are added.



17. Save the file.
18. The formatting of generated code can be configured in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Formatter** preference page. If you use a prefix or suffix for field names (e.g. fContainer), you can specify this in the [Image: /topic/org.eclipse.help/command_link.png] **Java > Code Style** preference page so that generated getters and setters will suggest method names without the prefix or suffix.

Next Section: Renaming Java elements

Related concepts

Java views

Java editor

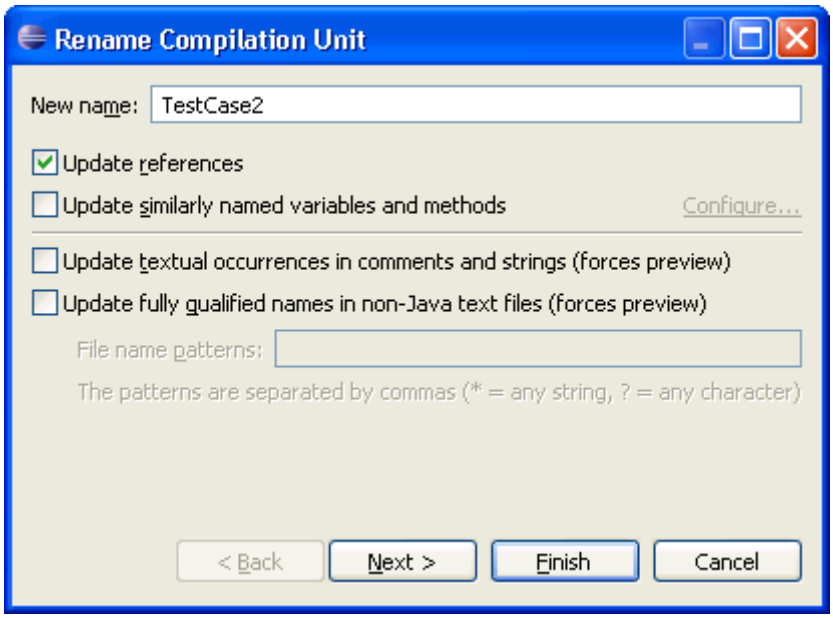
Related reference

New Java Class wizard
Source actions
Quick fix
Override methods dialog
Generate Getter and Setter dialog
Code formatter preference page
Code style preference page
Code templates preference page

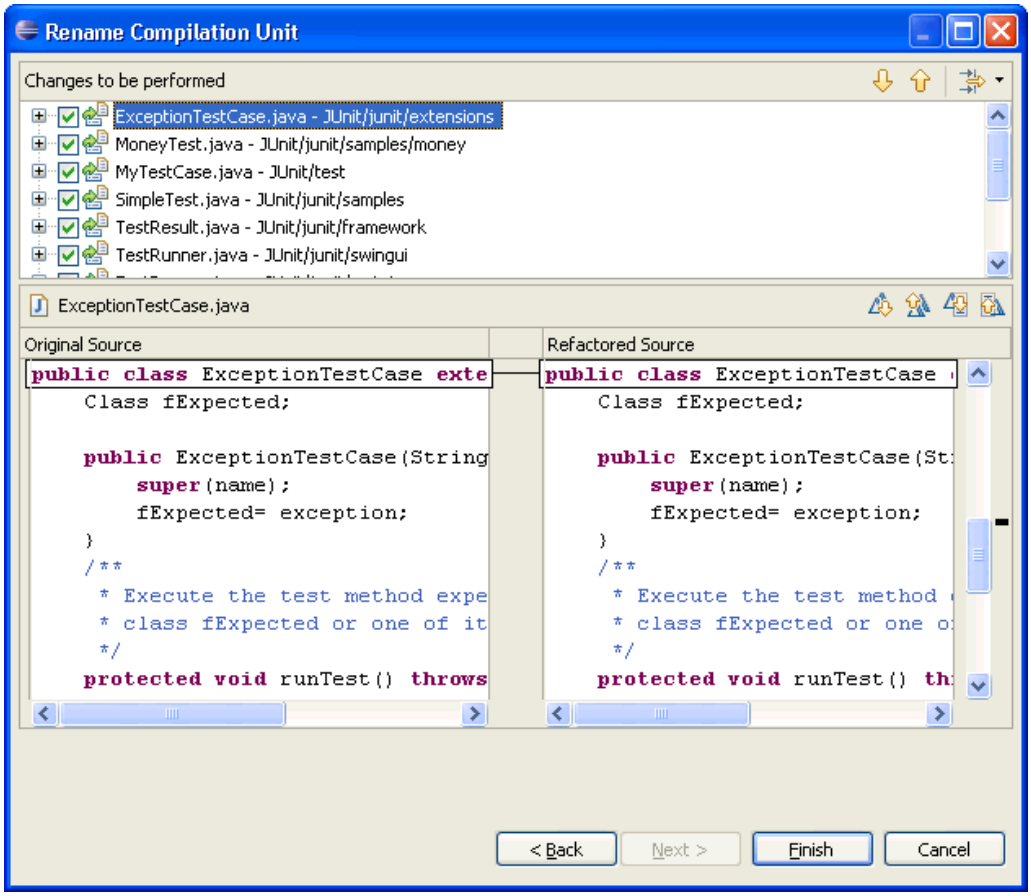
Renaming Java elements

In this section, you will rename a Java element using refactoring. Refactoring actions change the structure of your code without changing its semantics (behavior).

1. In the Package Explorer view, select *junit.framework/TestCase.java*.
2. From its context menu, select **Refactor > Rename**.
3. In the **New Name** field on the Rename Compilation Unit page, type "*TestCase2*".



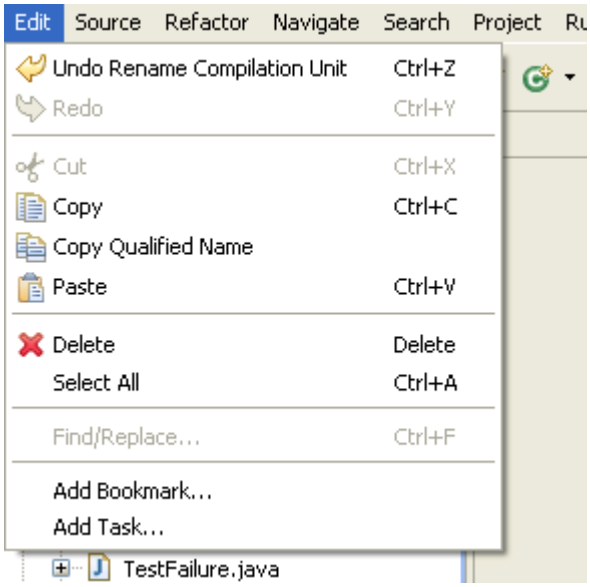
4. To preview the changes that will be made as a result of renaming the class, press **Next**.
5. The workbench analyzes the proposed change and presents you with a preview of the changes that would take place if you rename this resource. Since renaming a compilation unit will affect the import statements in other compilation units, there are other compilation units affected by the change. These are shown in a list of changes in the preview pane.



6. On the Refactoring preview page, you can scroll through the proposed changes and select or deselect changes, if necessary. You will typically accept all of the proposed changes.
7. Click **Finish** to accept all proposed changes.

You have seen that a refactoring action can cause many changes in different compilation units. These changes can be undone as a group.

1. In the menu bar, select **Edit > Undo Rename Compilation Unit**.



2. The refactoring changes are undone, and the workbench returns to its previous state. You can undo refactoring actions right up until you change and save a compilation unit, at which time the refactoring undo buffer is cleared.

Next Section: Moving and copying Java elements

- Related concepts

  Refactoring support

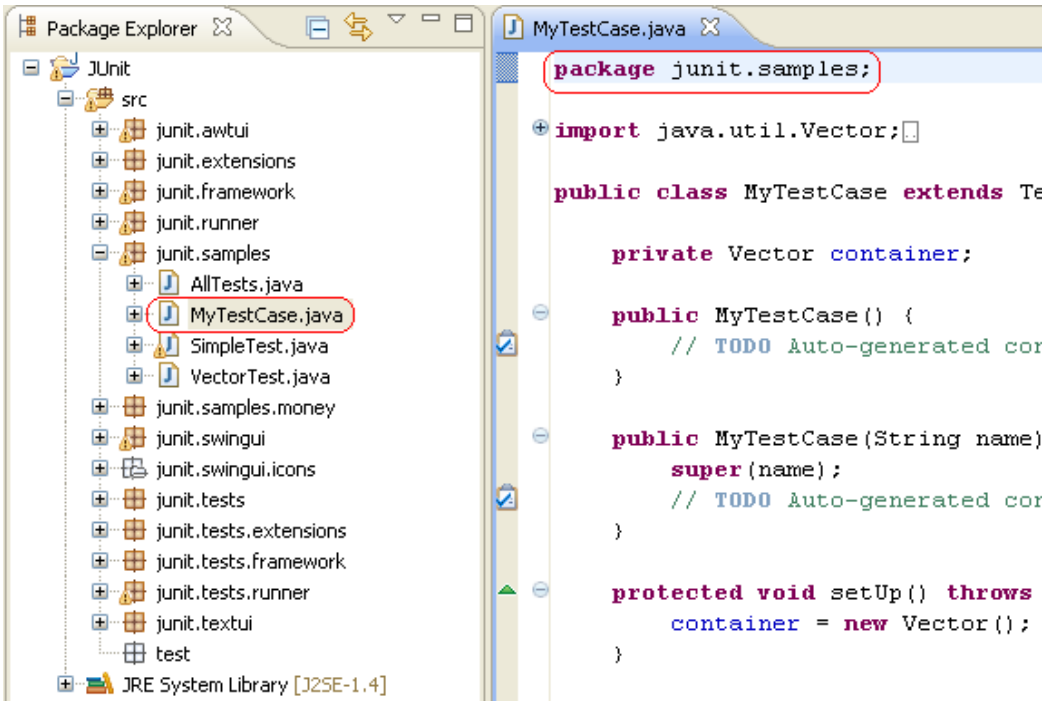
- Related reference

Refactoring actions
Refactoring wizard
Java preferences

Moving and copying Java elements

In this section, you will use refactoring to move a resource between Java packages. Refactoring actions change the structure of your code without changing its semantic behavior.

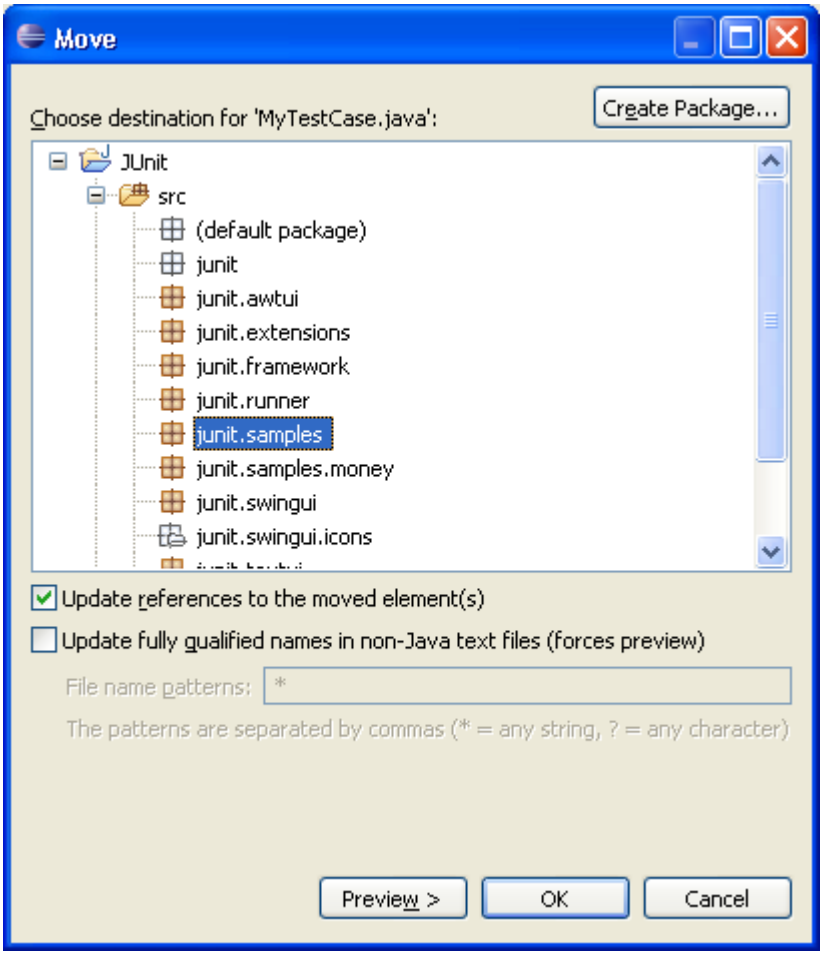
1. In the Package Explorer view, select the *MyTestCase.java* file from the *test* package and drag it into the *junit.samples* package. Dragging and dropping the file is similar to selecting the file and choosing **Refactor > Move** from the context menu.
2. You will be prompted to select whether or not to update references to the file you are moving. Typically, you will want to do this to avoid compile errors. You can press the **Preview** button to see the list of changes that will be made as a result of the move.
3. Press **OK**. The file is moved, and its package declaration changes to reflect the new location.



4. Use **Edit > Undo Move** to undo the move.

The context menu is an alternative to using drag and drop. When using the menu, you must specify a target package in the Move dialog, in addition to selecting the update references options you've already seen.

1. Select the *MyTestCase.java* file and from its context menu, select **Refactor > Move**.
2. In the Move dialog, expand the hierarchy to browse the possible new locations for the resource. Select the *junit.samples* package, then click **OK**. The class is moved, and its package declaration is updated to the new location.



Next Section: Navigate to a Java element's declaration

Related concepts

Java views

Refactoring support

Related reference

Refactoring actions

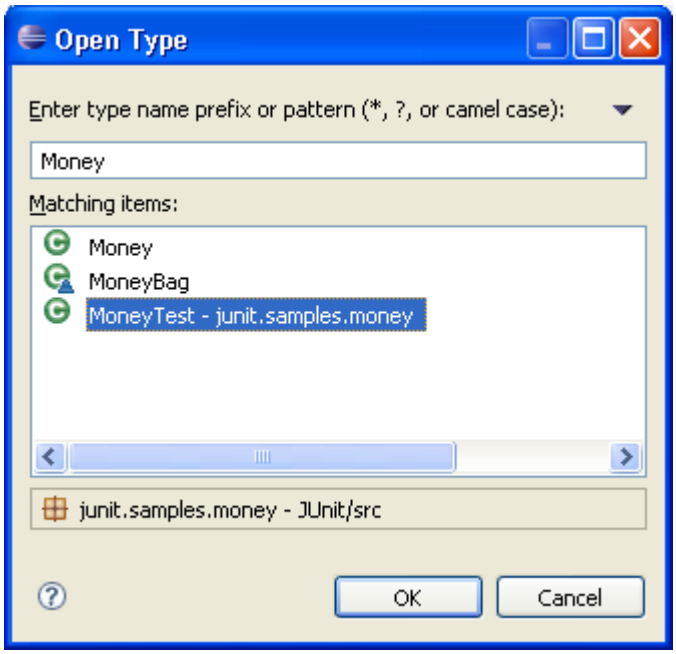
Refactoring wizard

Java preferences

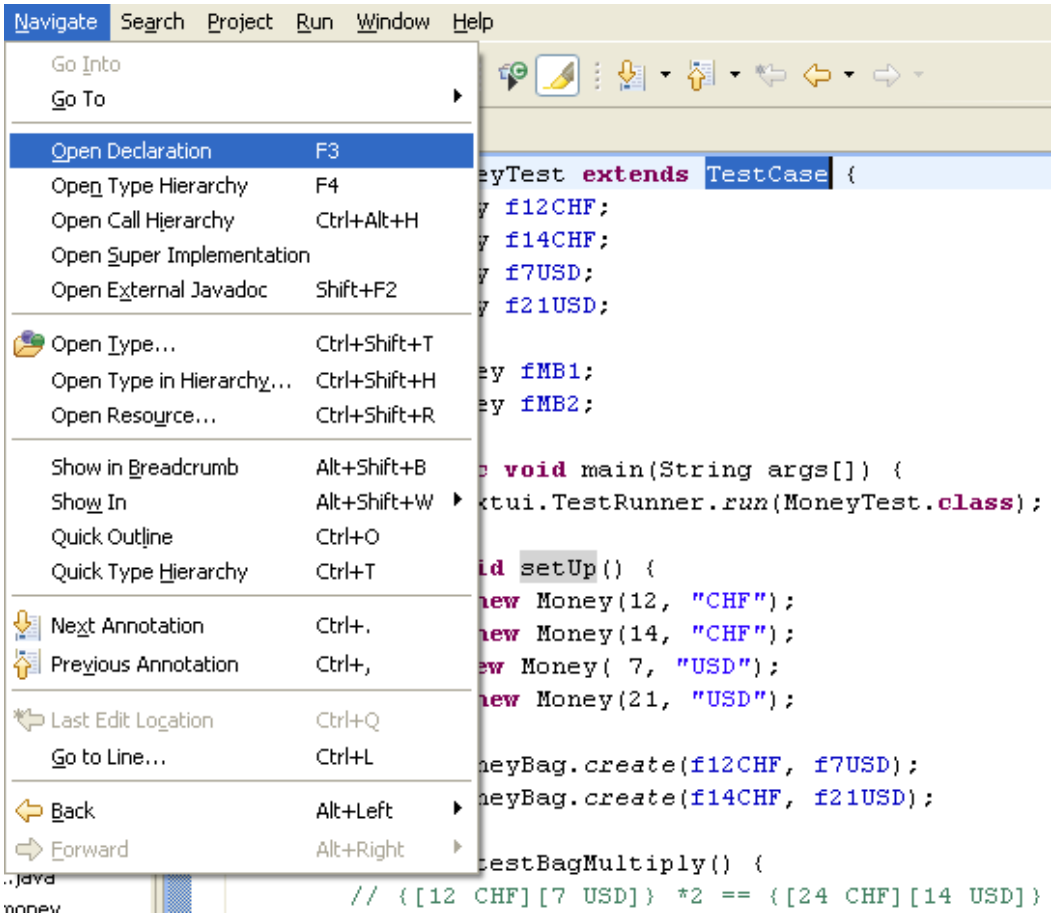
Navigate to a Java element's declaration

In this section, you will learn how to open a type in the Java Editor and how to navigate to related types and members.

1. Open the **Open Type** dialog by pressing `Ctrl+Shift+T`, choosing **Navigate > Open Type...**, or clicking the toolbar icon (). Type `Money`, press the `Arrow Down` key a few times to select *MoneyTest*, and then press `Enter` to open the type in the Java editor.

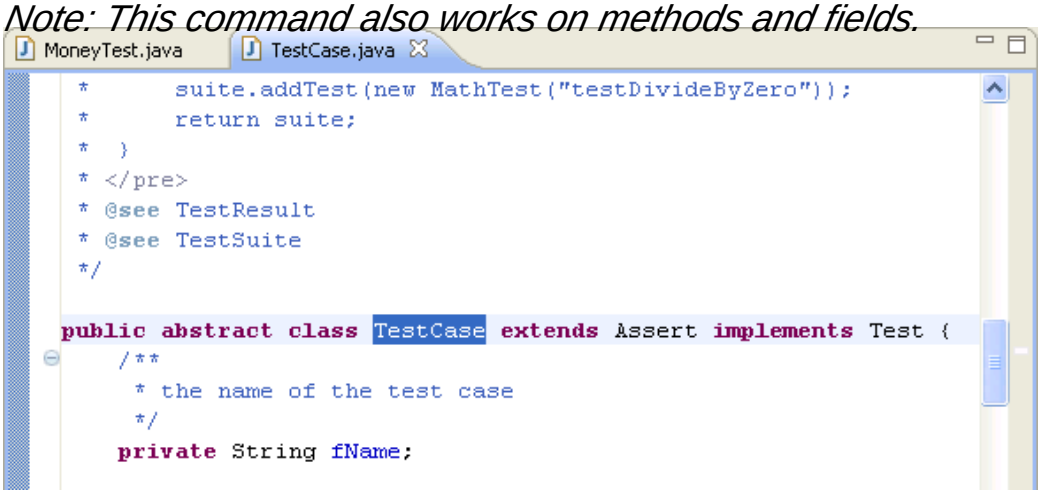


2. On the first line of the *MoneyTest* class declaration, select the superclass *TestCase* and either
   - from the menu bar select **Navigate > Open Declaration** or
   - press F3.

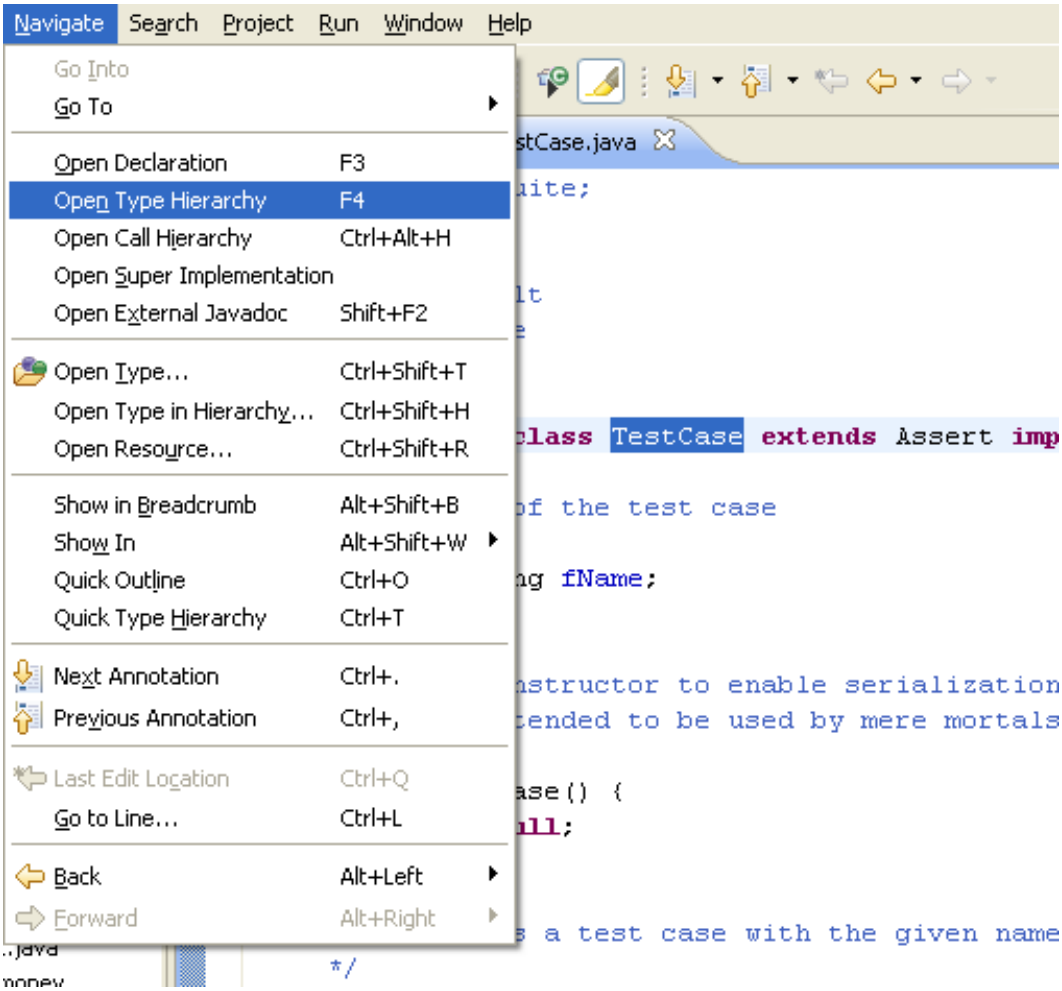


The TestCase class opens in the editor area and is also represented in the Outline view.

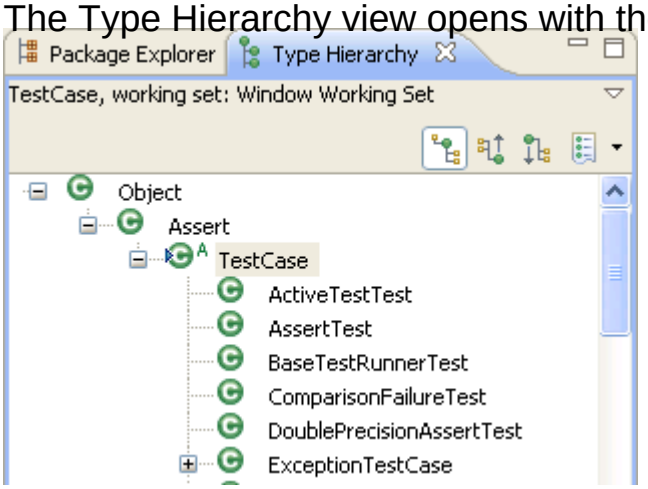
Note: This command also works on methods and fields.



3. With the *TestCase.java* editor open and the class declaration selected:
   - from the menu bar select **Navigate > Open Type Hierarchy** or
   - press F4.



4. The Type Hierarchy view opens with the TestCase class displayed.



Note: You can also open editors on types and methods in the Type Hierarchy view.

Next Section: Viewing the type hierarchy

Related tasks

Opening an editor for a selected element

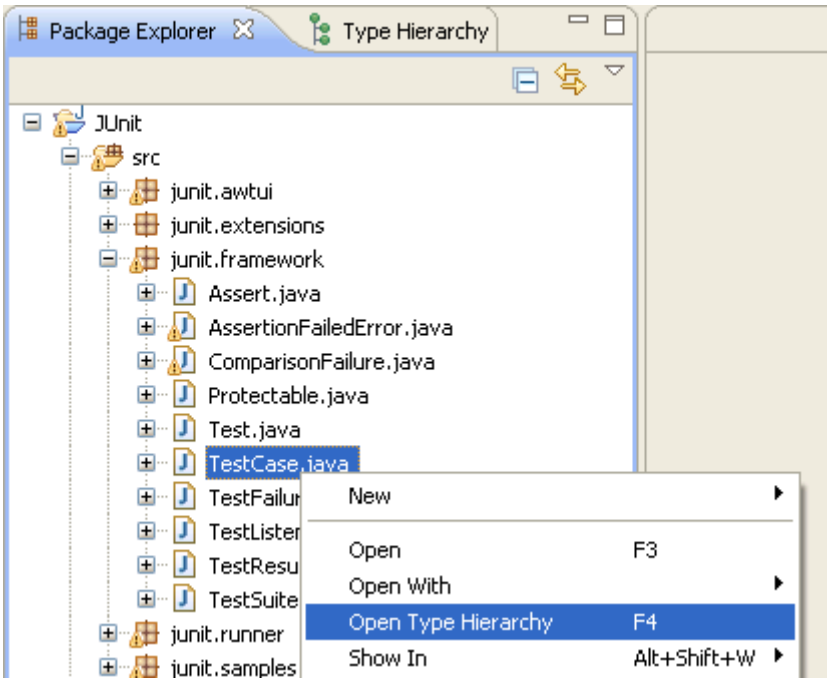
Related reference

Type Hierarchy View

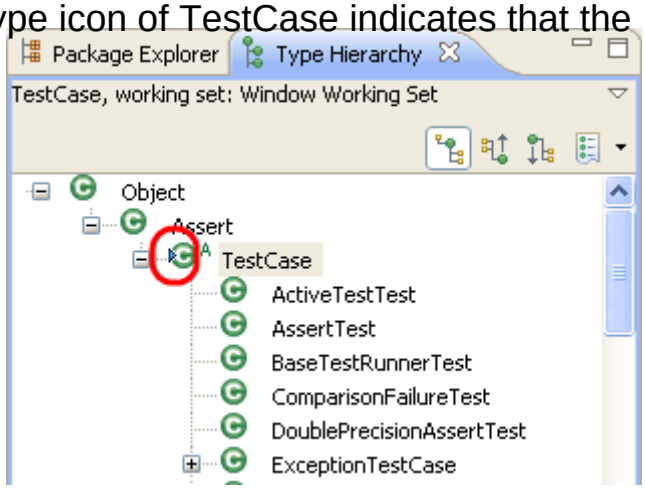
Viewing the type hierarchy

In this section, you will learn about using the Type Hierarchy view by viewing classes and members in a variety of different ways.

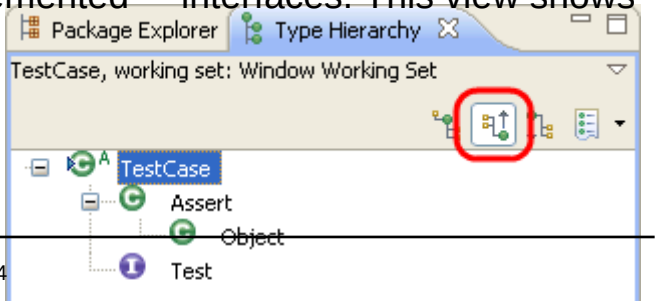
1. In the Package Explorer view, find *junit.framework/TestCase.java*. From its context menu, select [Image: /topic/org.eclipse.help/command_link.png]**Open Type Hierarchy**. You can also open type hierarchy view:
   - from the menu bar by selecting **Navigate > Open Type Hierarchy**.
   - from the keyboard by pressing F4 after selecting *TestCase.java*.



2. The buttons in the view tool bar control which part of the hierarchy is shown. Click the **Show the Type Hierarchy** button to see the class hierarchy, including the base classes and subclasses. The small arrow on the left side of the type icon of TestCase indicates that the hierarchy was opened on this type.

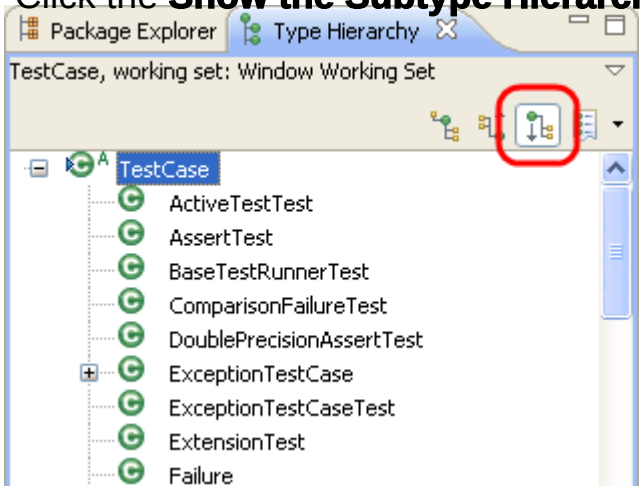


3. Click the **Show the Supertype Hierarchy** button to see a hierarchy showing the type's parent elements including implemented interfaces. This view shows

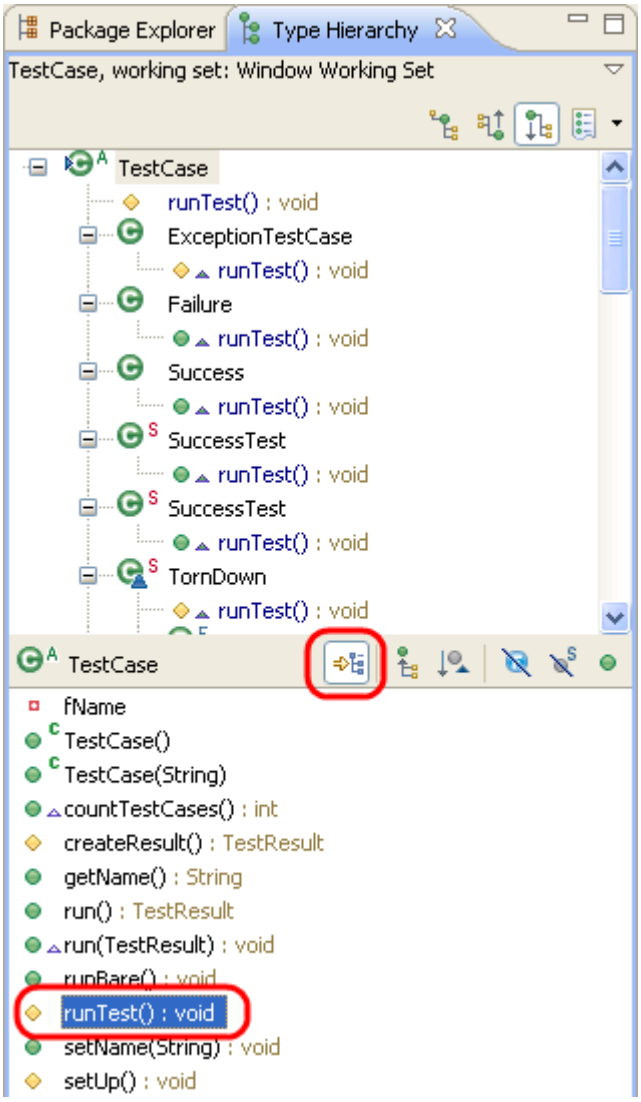


In this "reversed hierarchy" view, you can see that TestCase implements the Test interface.

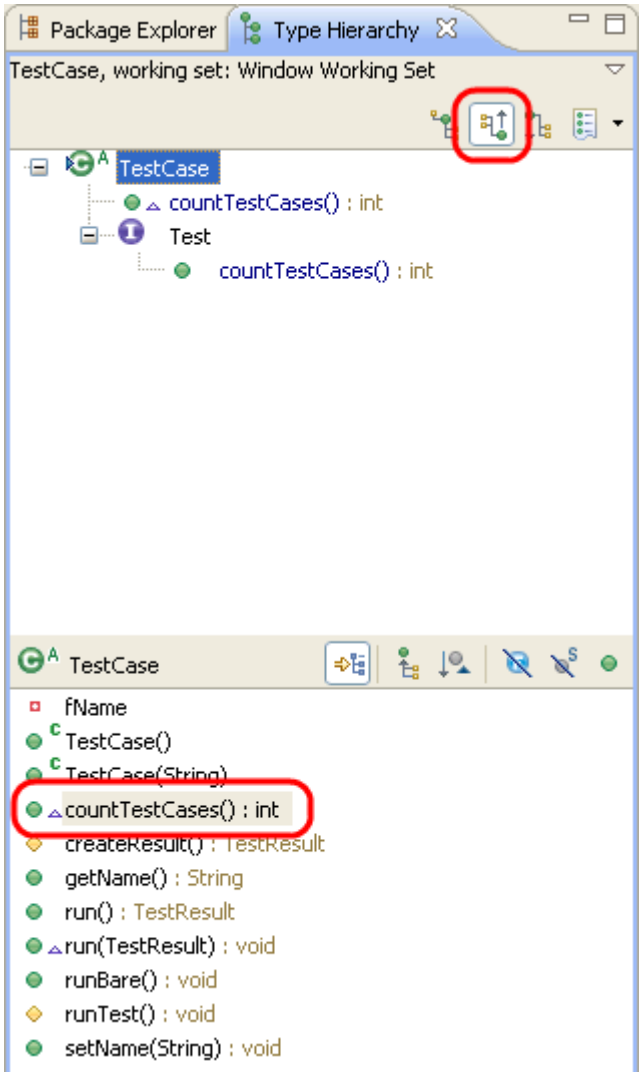
4. Click the **Show the Subtype Hierarchy** button in the view toolbar.



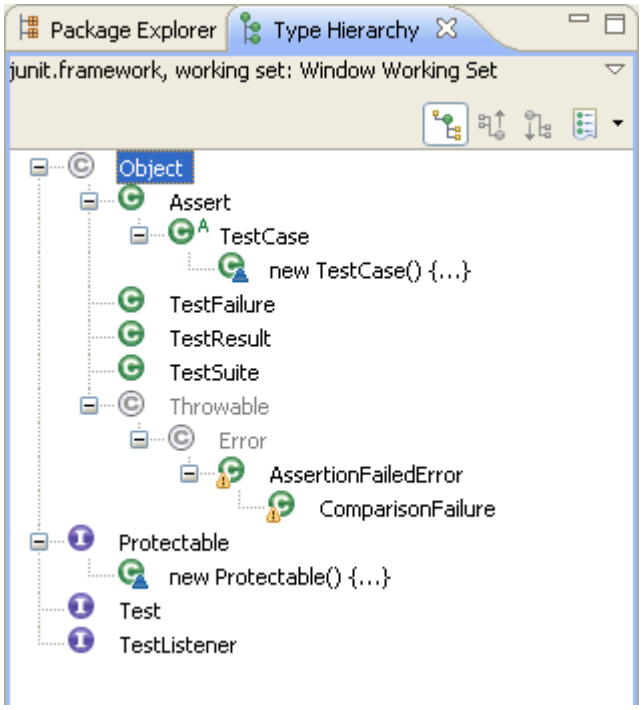
5. Click the **Lock View and Show Members in Hierarchy** button in the toolbar of the member pane, then select the runTest() method in the member pane. The view will now show all the types implementing runTest().



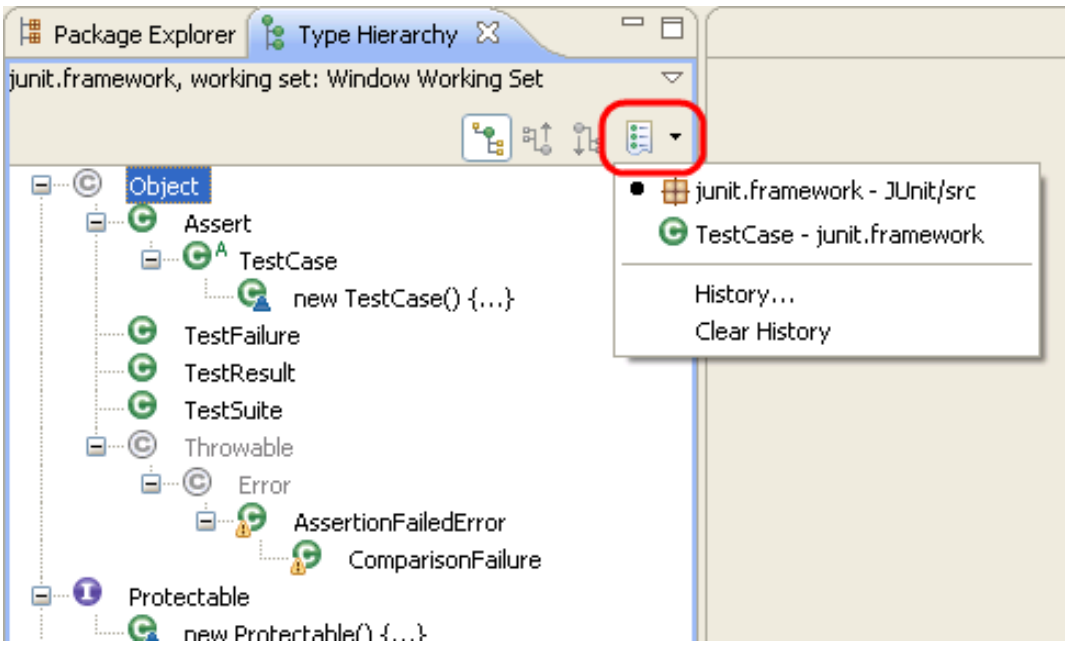
6. In the Type Hierarchy view, click the **Show the Supertype Hierarchy** button. Then on the member pane, select `countTestCases()` to display the places where this method is declared.



7. In the Type Hierarchy view select the *Test* element and select **Focus On 'Test'** from its context menu. *Test* is presented in the Type Hierarchy view.
8. Activate the Package Explorer view and select the package junit.framework. Use **Open Type Hierarchy** from its context menu. A hierarchy is opened containing all classes of the package. For completion of the tree, the hierarchy also shows some classes from other packages. These types are shown by a type icon with a white fill.

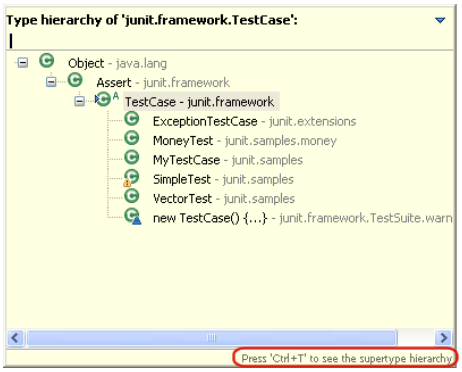


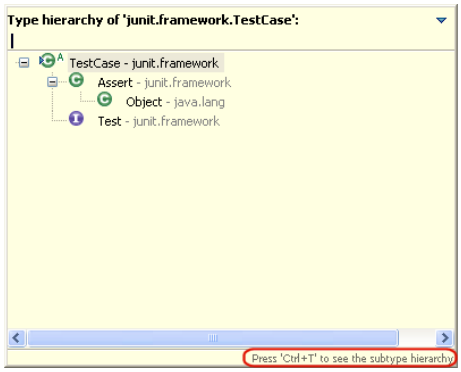
9. Use **Previous Type Hierarchies** to go back to a previously opened element. Click on the arrow next to the button to see a list of elements or click on the button to edit the history list.



If you are working in the editor and only want to do a quick lookup for a hierarchy you can use the **Quick Type Hierarchy**:

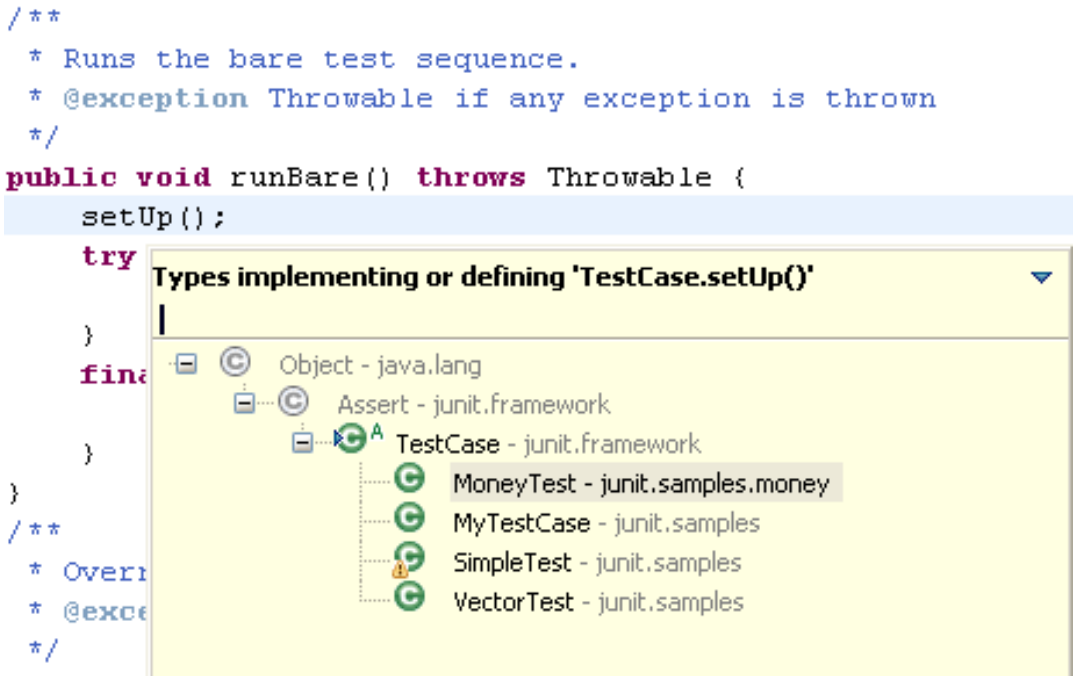
1. Open *junit.framework.TestCase.java* file in the Java editor if you do not already have it open.
2. Select the type name in the Java editor
3. Press `Ctrl+T` or invoke **Navigate > Quick Type Hierarchy** and the in-place type hierarchy view is shown.
4. Pressing `Ctrl+T` while the type hierarchy view is shown will toggle between supertype hierarchy and subtype hierarchy.





To see where a virtual method call can resolve to:

1. In the body of *runBare()* select the invocation of *setUp()*
2. Press `Ctrl+T` or invoke **Navigate > Quick Type Hierarchy** and the in-place type hierarchy view is shown.
3. You can see that *setUp()* is implemented in 3 more classes. *Object* and *Assert* are only shown with a white filled images as are only required to complete the hierarchy but do not implement *setUp()*
4. Select a type to navigate to its implementation of *setUp()*



Next Section: Searching the workbench

Related concepts

Java views

Related reference

Type Hierarchy view

Java Base preference page

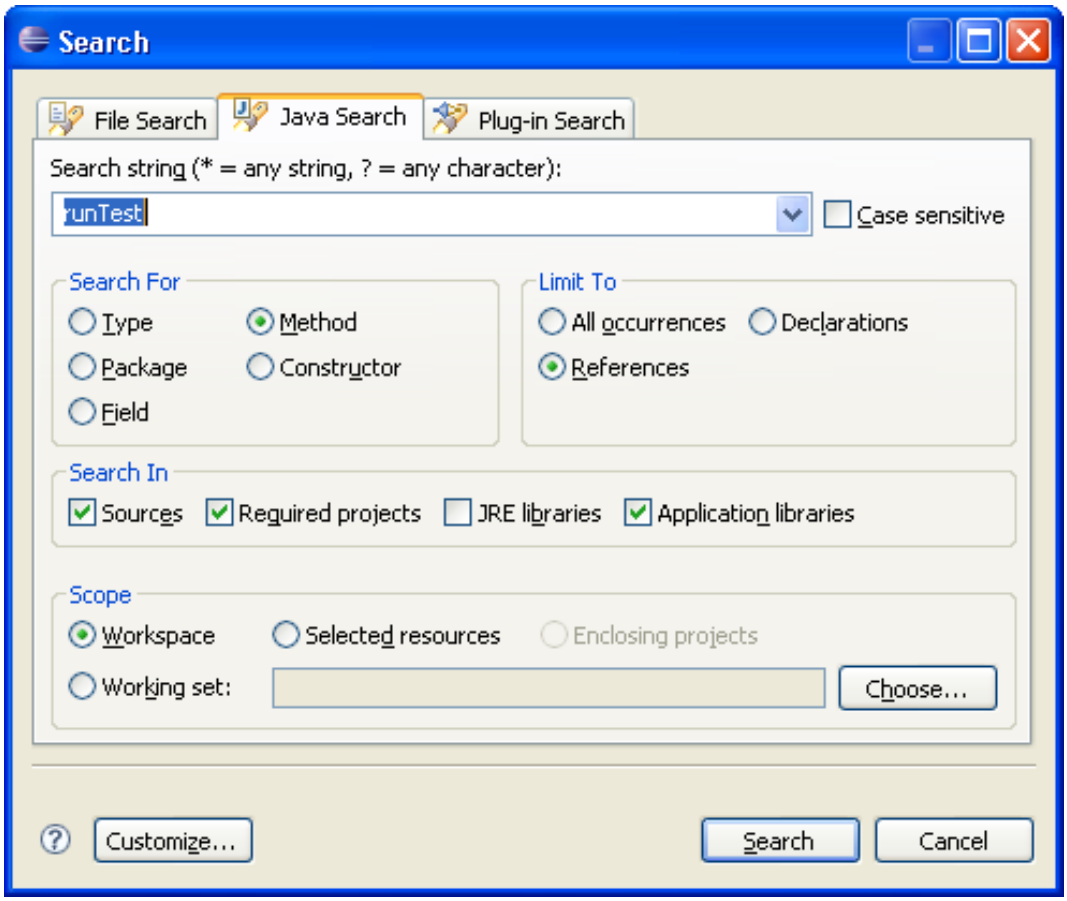
Searching the workbench

In this section, you will search the workbench for Java elements.

In the Search dialog, you can perform file, text or Java searches. Java searches operate on the structure of the code. File searches operate on the files by name and/or text content. Java searches are faster, since there is an underlying indexing structure for the code structure. Text searches allow you to find matches inside comments and strings.

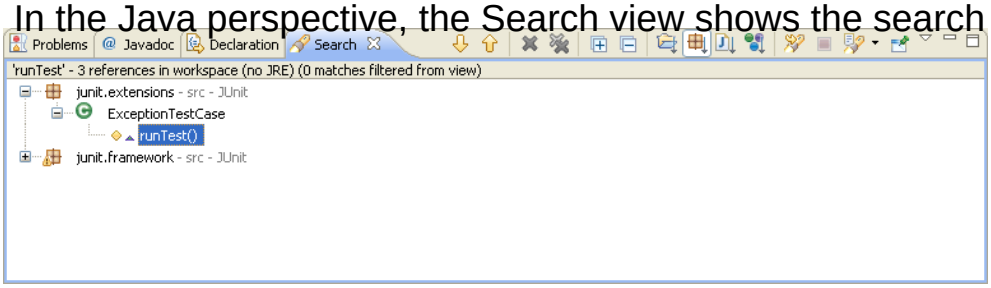
Performing a Java search from the workbench

1. In the Java perspective, click the **Search** (🔍) button in the workbench toolbar or use **Search > Java...** from the menu bar.
2. If it is not already selected, select the **Java Search** tab.
3. In the **Search string** field, type *runTest*. In the *Search For* area, select **Method**, and in the *Limit To* area, select **References**.
   Verify that the Scope is set to **Workspace**.



Then click **Search**. While searching you may click **Cancel** at any time to stop the search. Partial results will be shown.

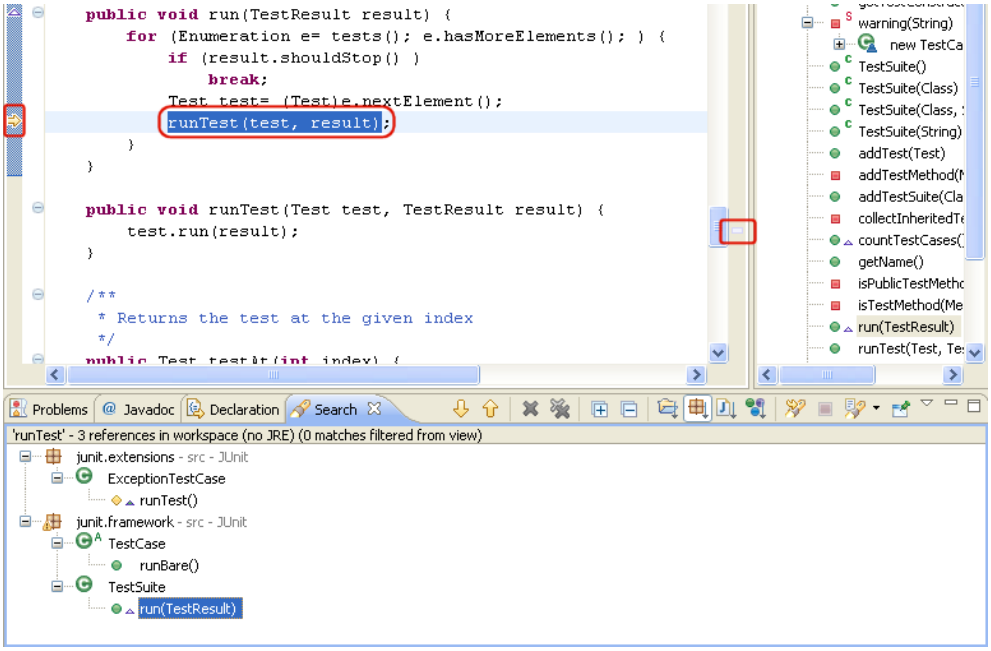
4. In the Java perspective, the Search view shows the search results.



Use the **Show Next Match** (⇩) and **Show Previous Match** (⇧) buttons

to navigate to each match. If the file in which the match was found is not currently open, it is opened in an editor.

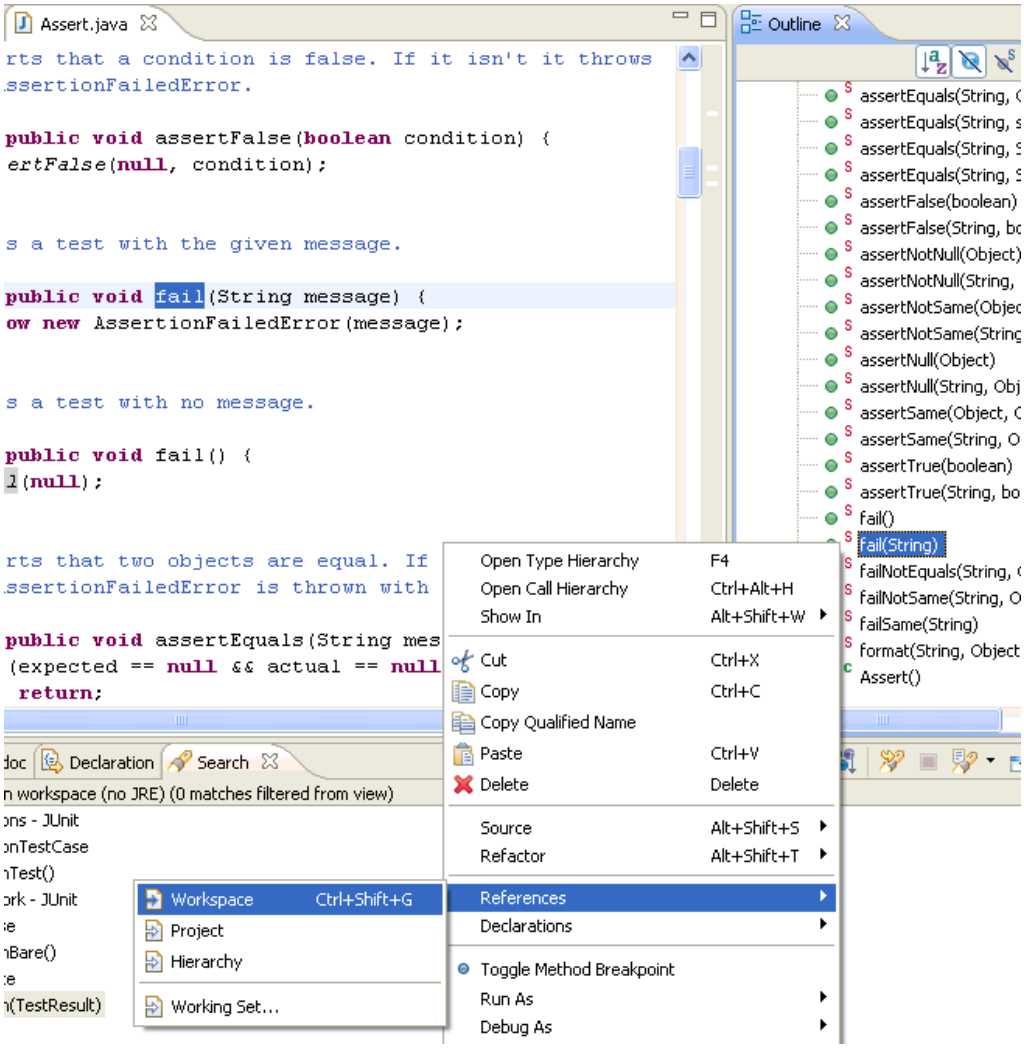
5. When you navigate to a search match using the Search view buttons, the file opens in the editor at the position of the match. Search matches are tagged with a search marker in the rulers.



Searching from a Java view

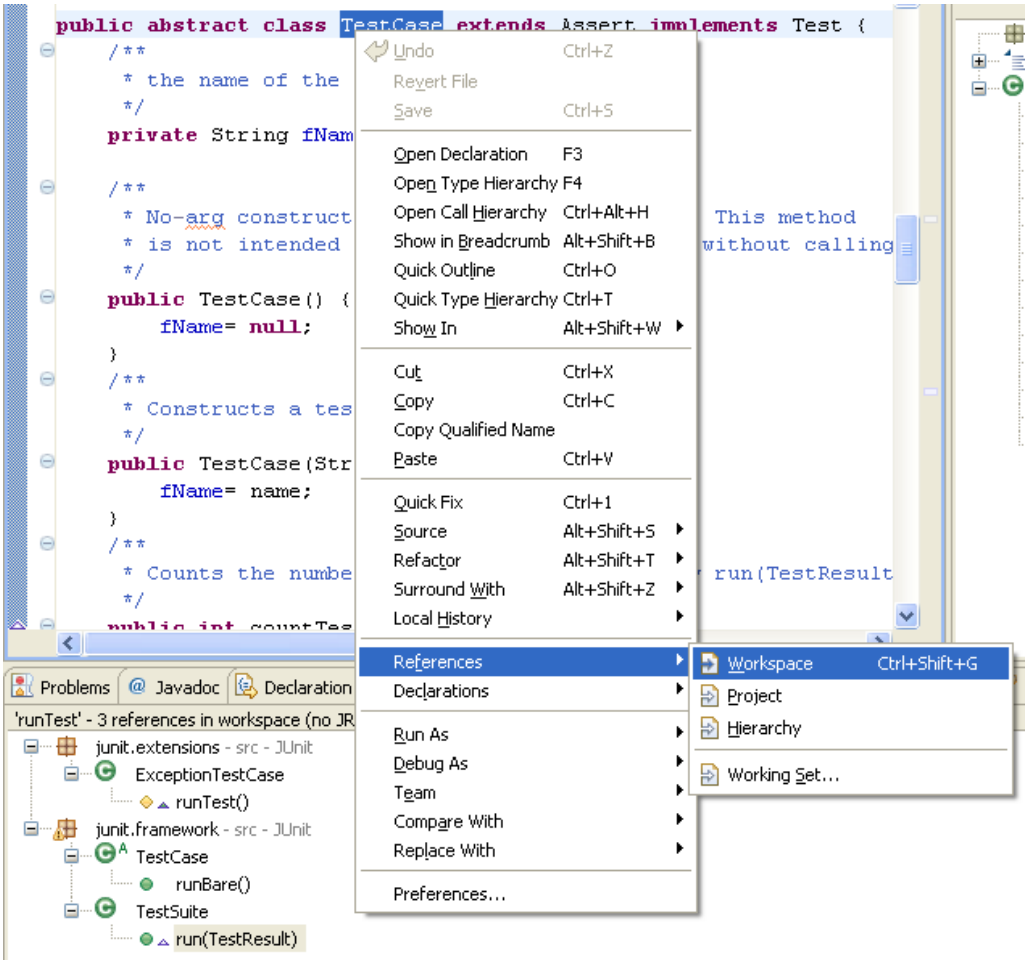
Java searches can also be performed from specific views, including the Outline, Type Hierarchy view and the Package Explorer view.

1. In the Package Explorer view, double-click *junit.framework.Assert.java* to open it in an editor.
2. In the Outline view, select the fail(String) method, and from its context menu, select **References > Workspace**.



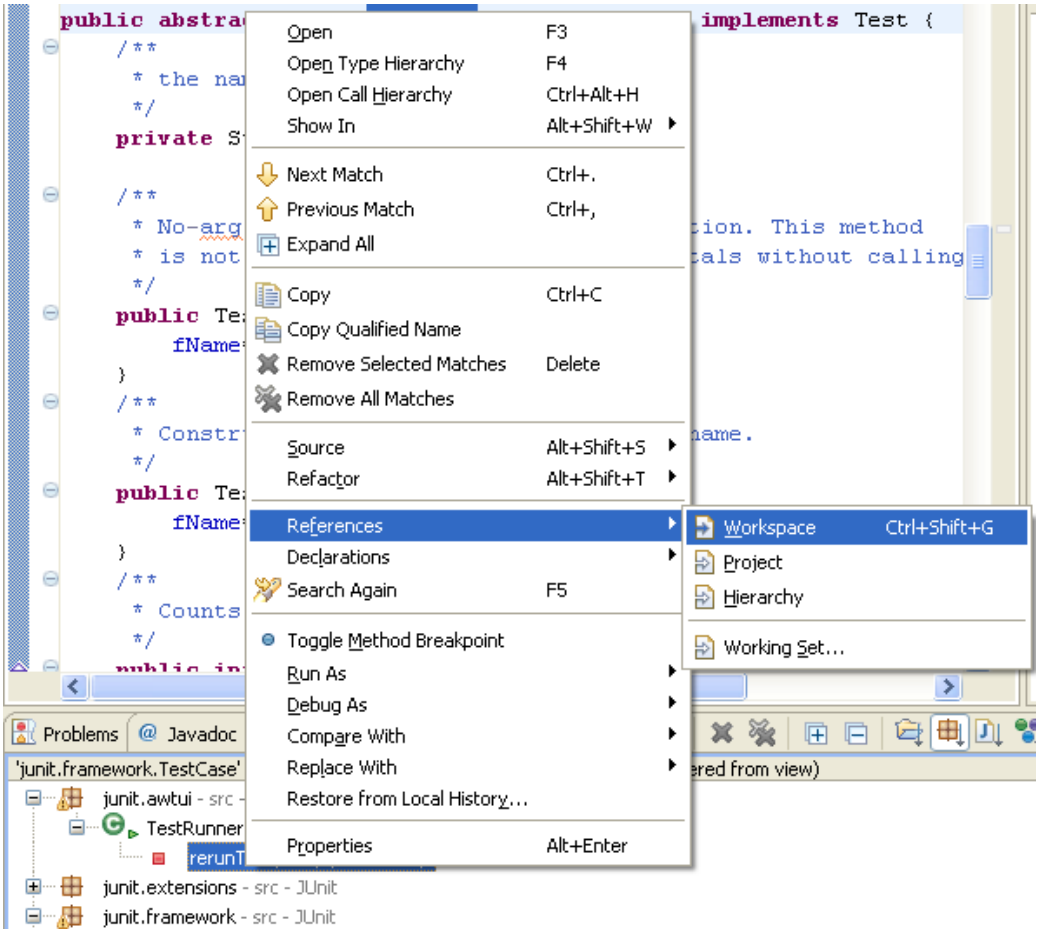
Searching from an editor

From the Package Explorer view, open *junit.framework.TestCase.java*. In the editor, select the class name *TestCase* and from the context menu, select **References > Workspace**.



Continuing a search from the search view

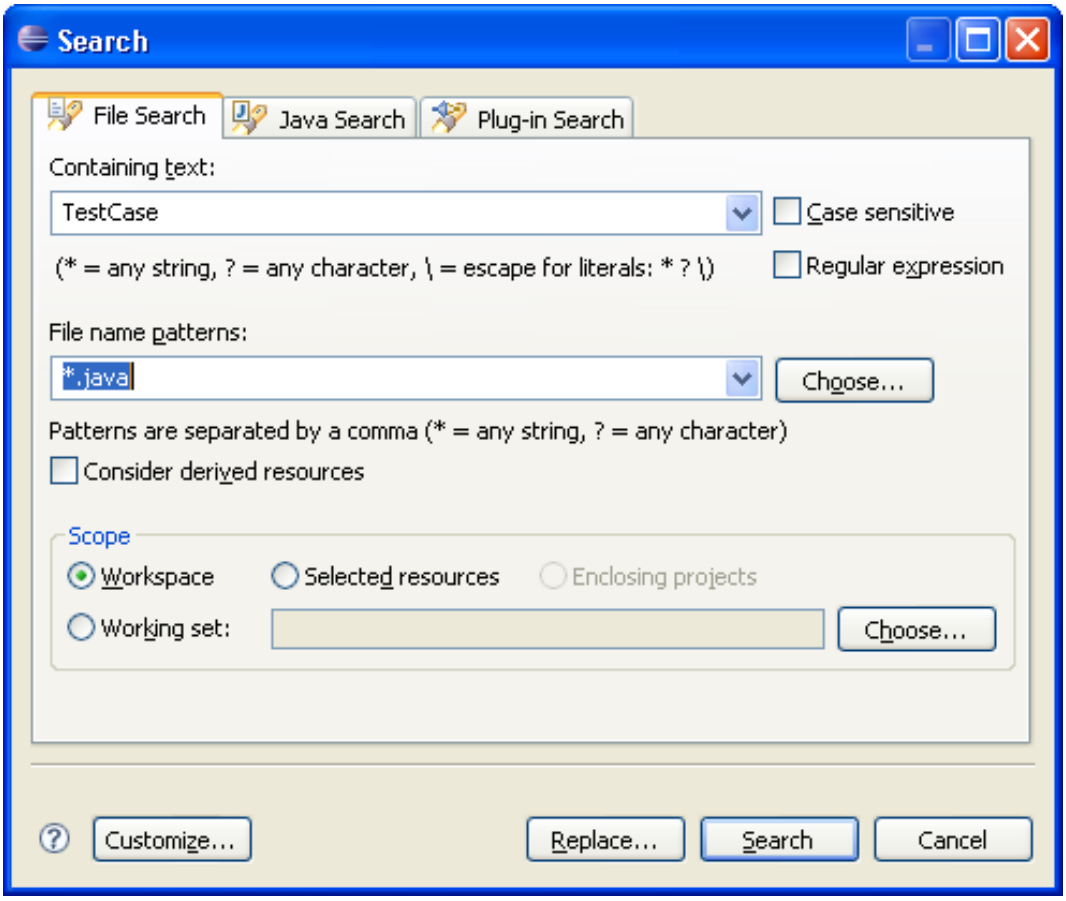
The Search Results view shows the results for the TestCase search. Select a search result and open the context menu. You can continue searching the selected element's references and declarations.



If you want to follow multiple levels of method calls, you can also use **Navigate > Open Call Hierarchy**.

Performing a file search

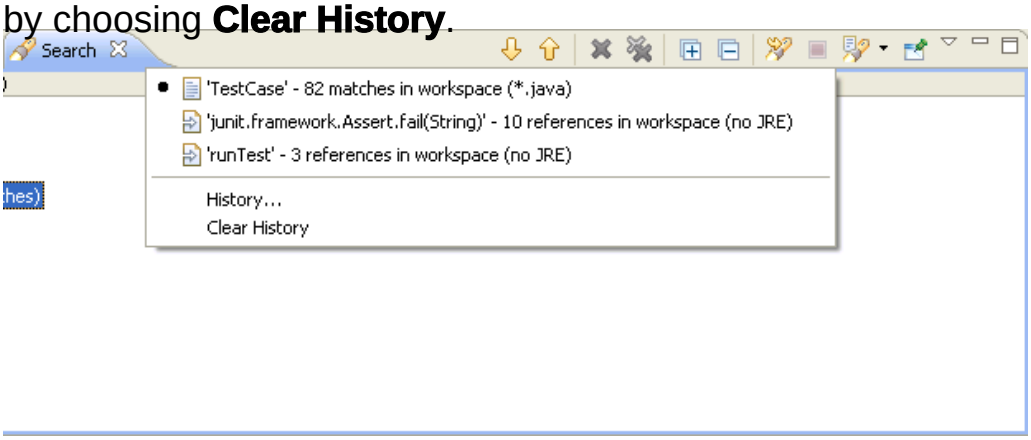
1. In the Java perspective, click the **Search** button in the workbench toolbar or select **Search > File** from the menu bar.
2. If it is not already selected, select the **File Search** tab.
3. In the **Containing text** field, type *TestCase*. Make sure that the **File name patterns** field is set to **.java*. The Scope should be set to *Workspace*. Then click **Search**.



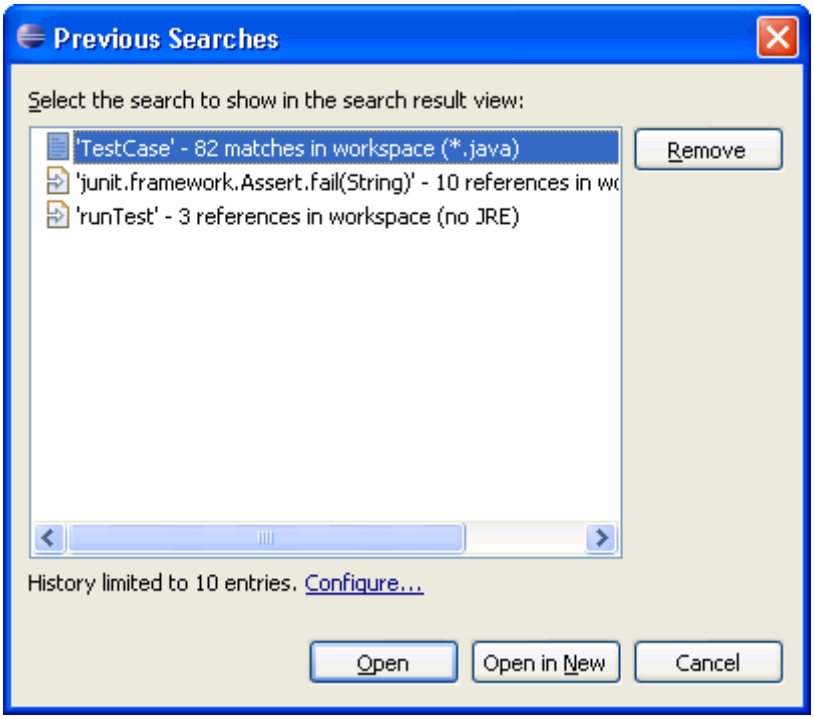
4. To find all files of a given file name pattern, leave the Containing Text field empty.

Viewing previous search results

In the Search Results view, click the arrow next to the **Previous Search Results** toolbar button to see a menu containing the list of the most recent searches. You can choose items from this menu to view previous searches. The list can be cleared by choosing **Clear History**.



The **Previous Search Results** button will display a dialog with the list of all previous searches from the current session.



Selecting a previous search from this dialog will let you view that search.

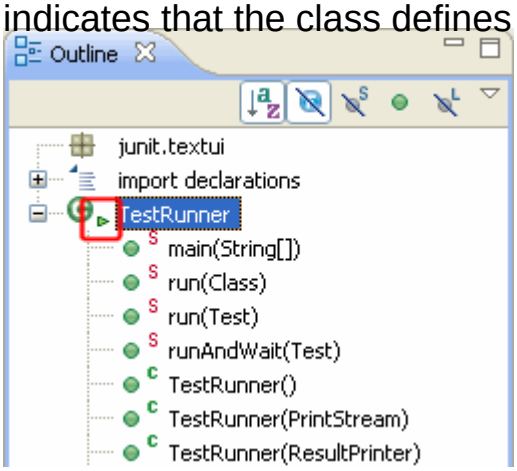
Next Section: Running your programs

- Related concepts
  - Java search
- Related reference
  - Refactoring actions
  - Refactoring wizard
  - Java preferences

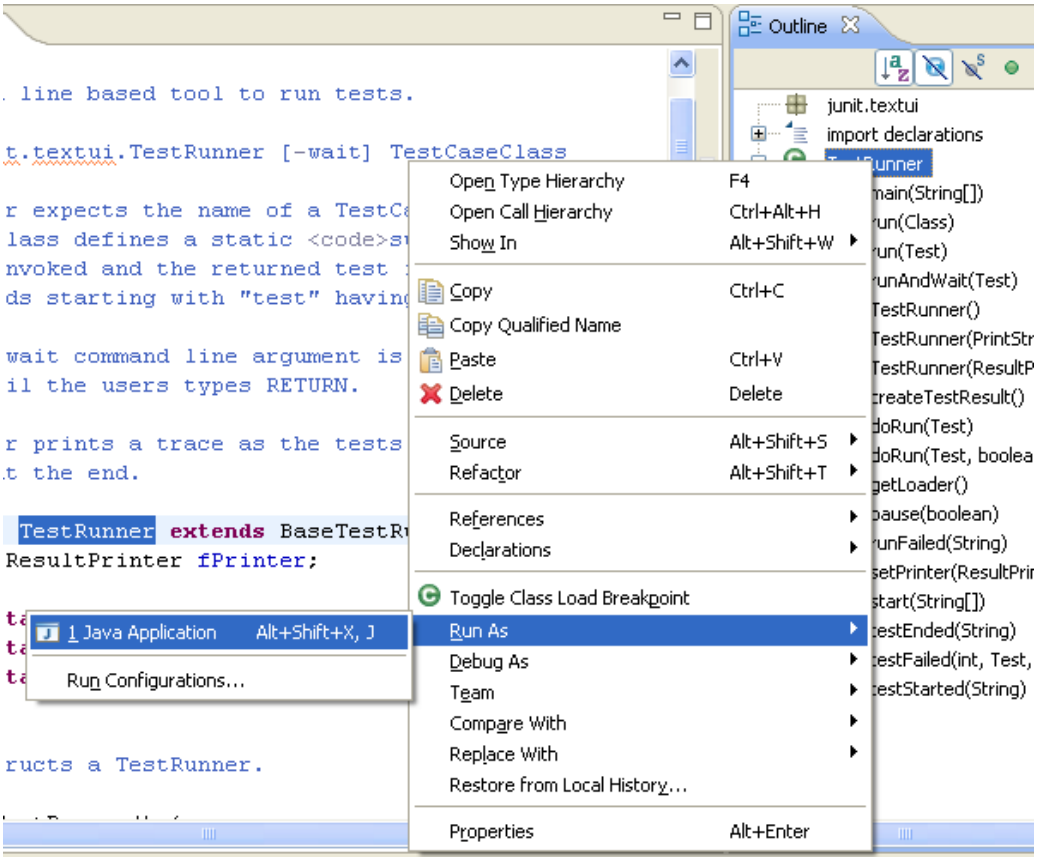
Running your programs

In this section, you will learn more about running Java programs in the workbench.

1. In the Package Explorer view, find *junit.textui/TestRunner.java* and double-click it to open it in an editor.
2. In the Outline view, notice that the TestRunner class has an icon which indicates that the class defines a `main` method.

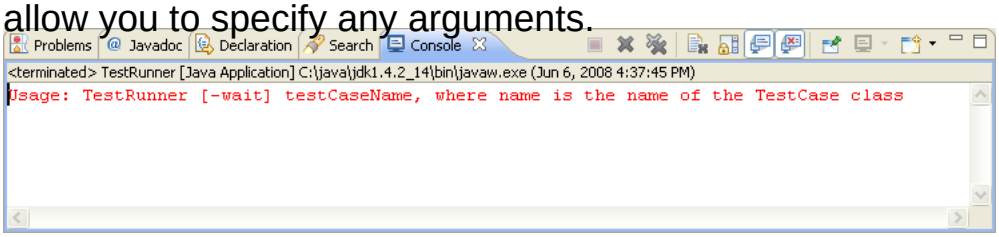


3. Right click on TestRunner.java in the Package Explorer and select **Run As** > **Java Application**. This will launch the selected class as a local Java application. The **Run As** context menu item is also available in other places, such as the Outline view.

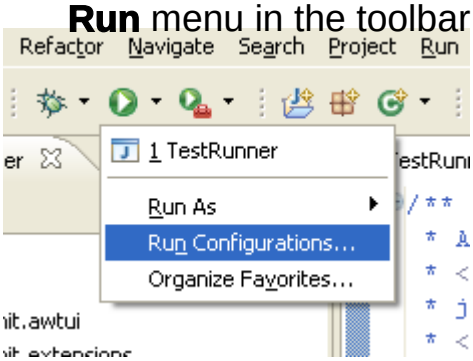


4. Notice that the program has finished running and the following message appears in the Console view telling you that the program needs an execution argument. Running class from the Package Explorer as a Java Application

uses the default settings for launching the selected class and does not allow you to specify any arguments.

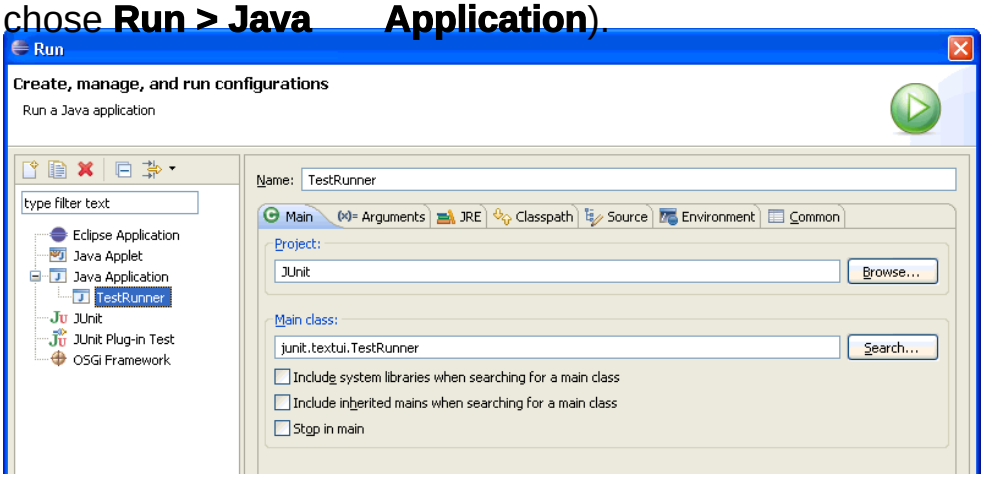


5. To specify arguments, use the drop-down **Run** menu in the toolbar and select **Run Configurations...**.

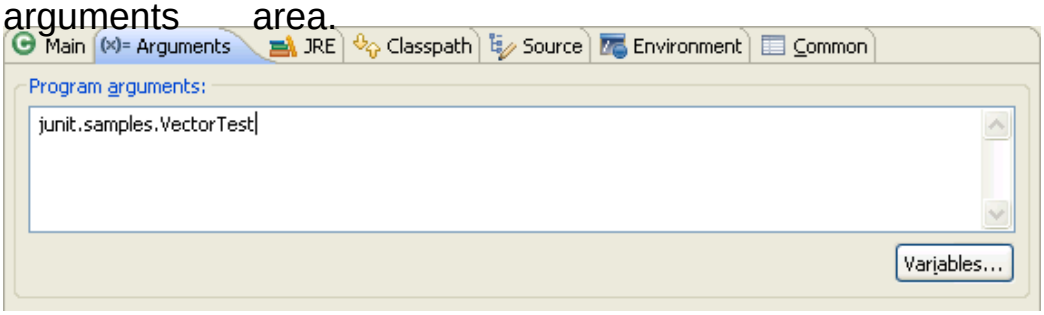


You can also Ctrl+Click a configuration in the drop-down menu to start editing that configuration.

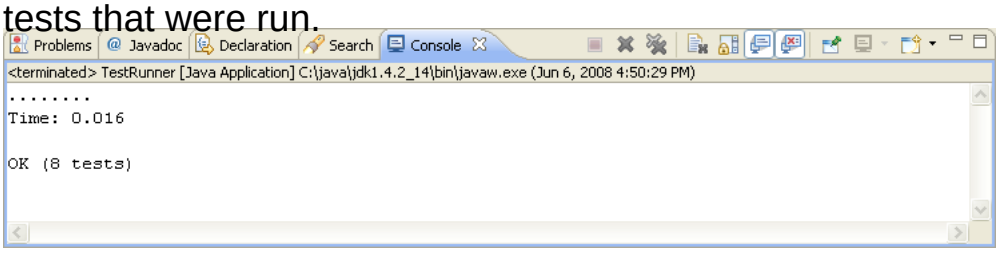
6. This time, the Launch Configurations dialog opens with the TestRunner launch configuration selected. A launch configuration allows you to configure how a program is launched, including its arguments, classpath, and other options. (A default launch configuration was created for you when you chose **Run > Java Application**).



7. Select the Arguments tab and type *junit.samples.VectorTest* in the Program arguments area.

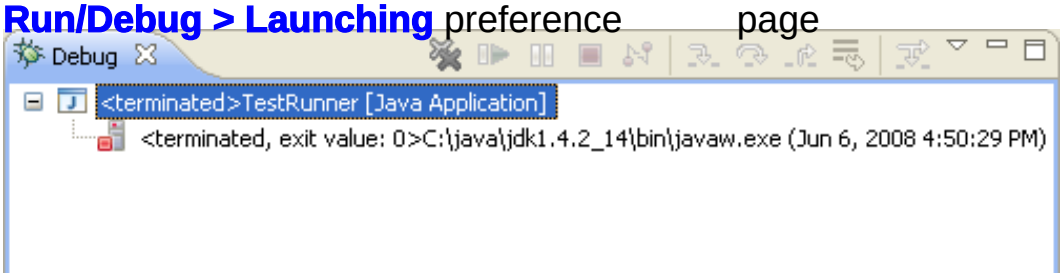


8. Click **Run**. This time the program runs correctly, indicating the number of tests that were run.



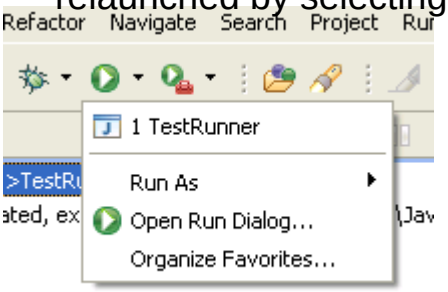
9. Switch to the Debug perspective. In the Debug view, notice that a process for the last program launch was registered when the program was run.

   By default, the Debug view automatically removes any terminated launches when a new launch is created. This preference can be configured on the [Image: /topic/org.eclipse.help/command_link.png] **Run/Debug > Launching** preference page

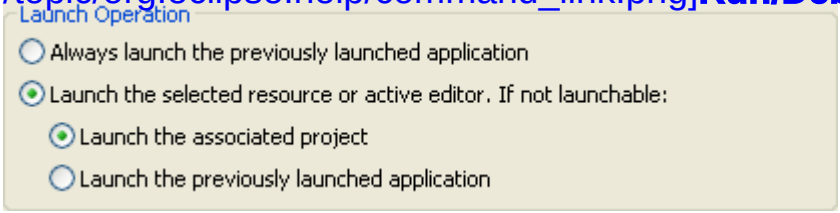


   Note: You can relaunch a terminated process by selecting ***Relaunch*** *from its context menu.*

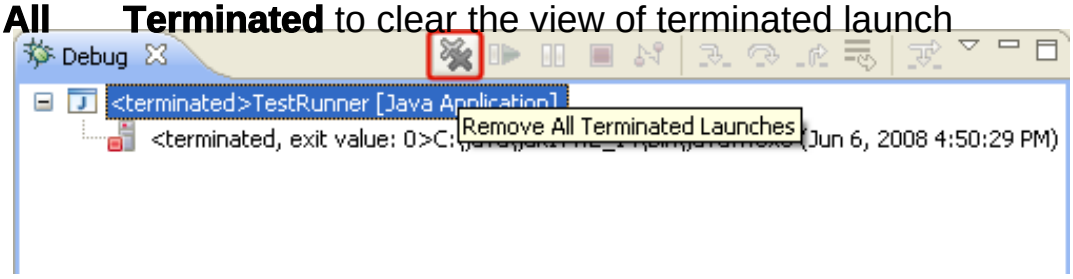
10. Select the drop-down menu from the **Run** button in the workbench toolbar. This list contains the previously launched programs. These programs can be relaunched by selecting them in the history list.



11. By default the currently selected resource or active editor is launched when the run button is hit. If none of these is launchable the current project will be launched. You can configure this behavior under [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Launching**.



12. From the context menu in the Debug view (or the equivalent toolbar button), select **Remove All Terminated** to clear the view of terminated launch processes.



Next Section: Debugging your programs

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM

Launching a Java program
Running and debugging

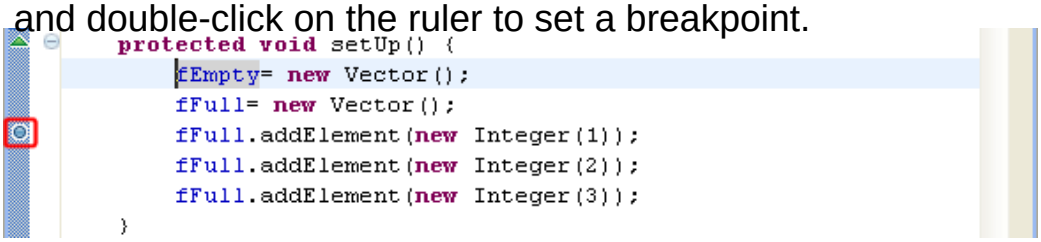
Related reference

Debug view
Run menu actions
Run and debug toolbar actions

Debugging your programs

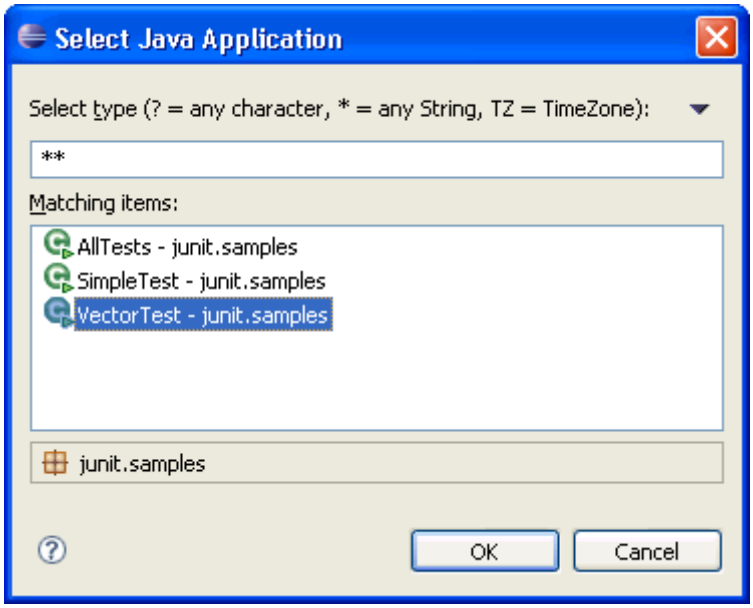
In this section, you will debug a Java program.

1. In the Package Explorer view in the Java perspective, double-click *junit.samples/VectorTest.java* to open it in an editor.
2. Place your cursor on the vertical ruler along the left edge of the editor area on the following line in the setUp() method: `fFull.addElement (new Integer(1));` and double-click on the ruler to set a breakpoint.

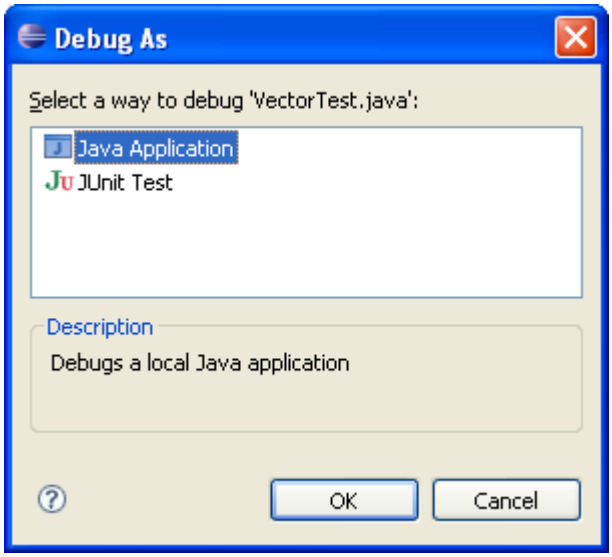


The breakpoint icon indicates the status of the breakpoint. The plain blue breakpoint icon indicates that the breakpoint has been set, but not yet installed. *Note: Once the class is loaded by the Java VM, the breakpoint will be installed and a checkmark overlay will be displayed on the breakpoint icon.*

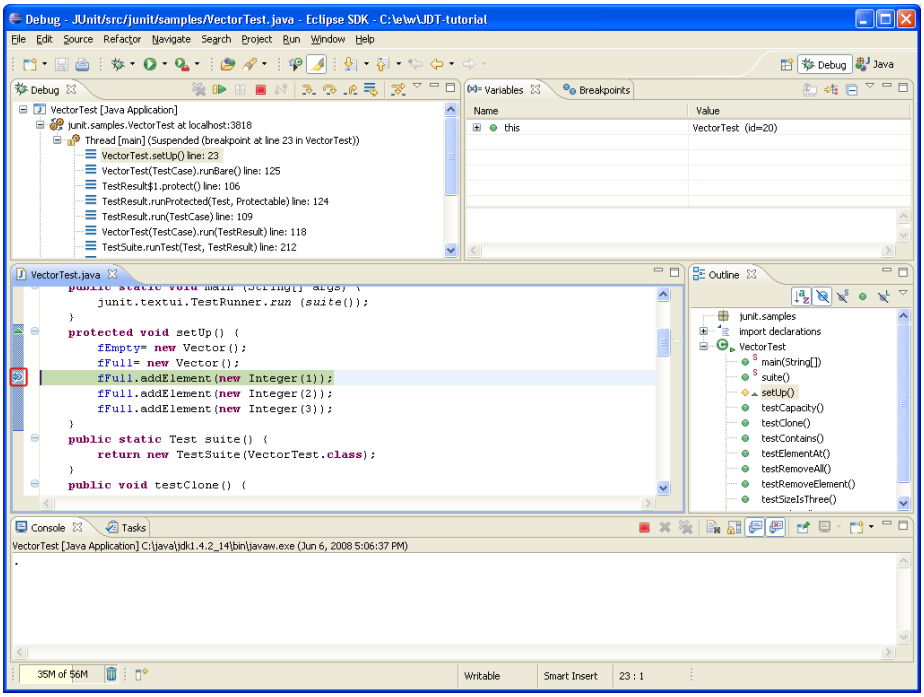
3. In the Package Explorer view, select the *junit.samples* package and select **Debug As**, and then **Java Application**. When you run a program from a package, you will be prompted to choose a type from all classes in the package that define a `main` method.
4. Select the *VectorTest* item in the dialog, then click **OK**.



Note: You can also simply hit the debug button  *which will launch the currently selected resource or active editor. Select* ***Java Application*** *when you are prompted to select a way to debug VectorTest.*

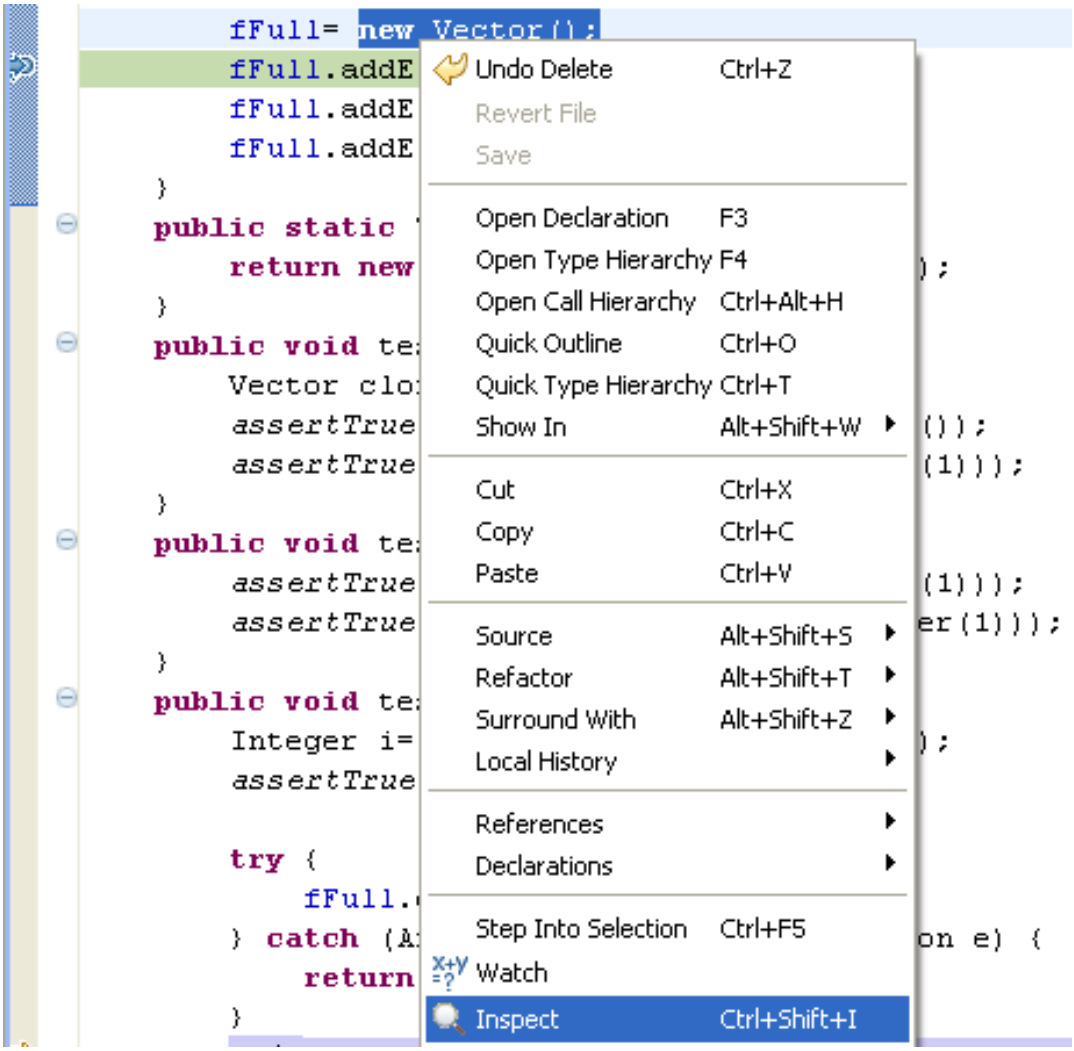


5. The program will run until the breakpoint is reached. When the breakpoint is hit, execution is suspended, and you are asked whether to open the Debug perspective. Click **Yes**. Notice that the process is still active (not terminated) in the Debug view. Other threads might still be running.

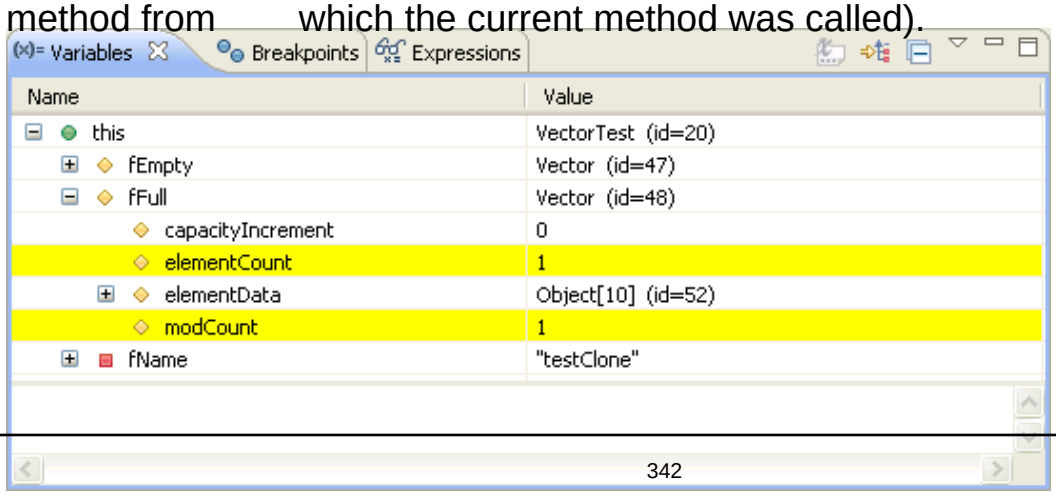


Note: The breakpoint now has a checkmark overlay since the class VectorTest was loaded in the Java VM.

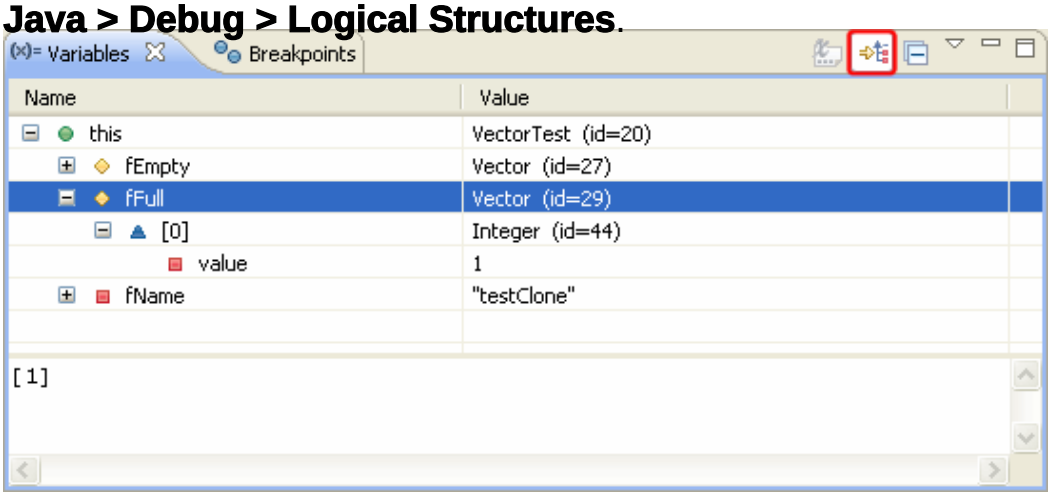
6. In the editor in the Debug perspective, select `new Vector()` from the line above where the breakpoint is set, and from its context menu, select **Inspect**.



7. The expression is evaluated in the context of the current stack frame, and a pop-up appears which displays the results. You can send a result to the Expressions view by pressing the key binding displayed in the pop-up.
8. Expressions that you evaluate while debugging a program will be listed in this view. To delete an expression after working with it, select the expression and choose **Remove** from its context menu.
9. The Variables view (available on a tab along with the Expressions view) displays the values of the variables in the selected stack frame. Expand the this.fFull tree in the Variables view until you can see elementCount.
10. The variables (e.g., elementCount) in the Variables view will change when you step through VectorTest in the Debug view. To step through the code, click the **Step Over** (↷) button. Execution will continue at the next line in the same method (or, if you are at the end of a method, it will continue in the method from which the current method was called).



11. In the variables view you can choose to see certain types as logical structures. This hides the implementation details of a type and simply shows it as arrays or fields. You can define logical structures by yourself in the preference page **Java > Debug > Logical Structures**.



12. Try some other step buttons (**Step Into**, **Step Return**) to step through the code. Note the differences in stepping techniques.
13. You can end a debugging session by allowing the program to run to completion or by terminating it.
    - You can continue to step over the code with the **Step** buttons until the program completes.
    - You can click the **Resume** button to allow the program to run until the next breakpoint is encountered or until the program is completed.
    - You can select **Terminate** from the context menu of the program's process in the Debug view to terminate the program.

Next Section: Evaluating expressions

Related concepts

Breakpoints
Remote debugging
Local debugging

Related tasks

Adding breakpoints
Resuming the execution of suspended threads
Running and debugging
Suspending threads

Related reference

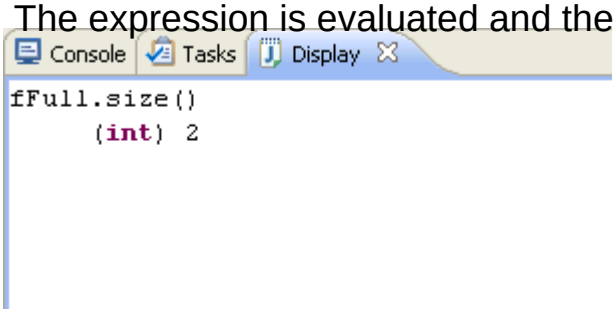
Debug preferences
Debug view
Run menu actions
Run and debug toolbar actions
Breakpoints view
Console view
Display view
Expressions view
Variables view

Evaluating expressions

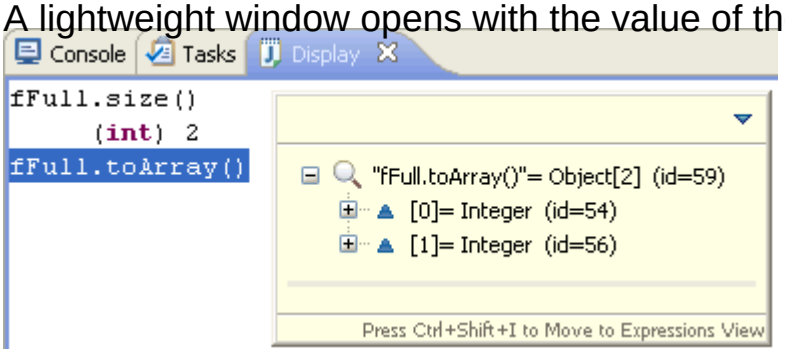
In this section, you will evaluate expressions in the context of your running Java program.

1. Debug *junit.samples.VectorTest.java* to the breakpoint in the `setUp()` method and select **Step Over** twice to populate `fFull`. (See the Debugging your Programs section for full details.)
2. Open the Display view by selecting **Window > Show View > Display** and type the following line in the view: `fFull.size()`
3. Select the text you just typed, and from its context menu, select **Display**. (You can also choose **Display Result of Evaluating Selected Text** () from the Display view toolbar.)

4. The expression is evaluated and the result is displayed in the Display view.



5. On a new line in the Display view, type the following line: `fFull.toArray()`
6. Select this line, and select **Inspect** from the context menu. (You can also choose **Inspect Result of Evaluating Selected Text** () from the Display view toolbar.)
7. A lightweight window opens with the value of the evaluated expression.



Next Section: Evaluating snippets

Related concepts

Debugger

Related tasks

Evaluating expressions
Displaying the result of evaluating an expression
Inspecting the result of evaluating an expression

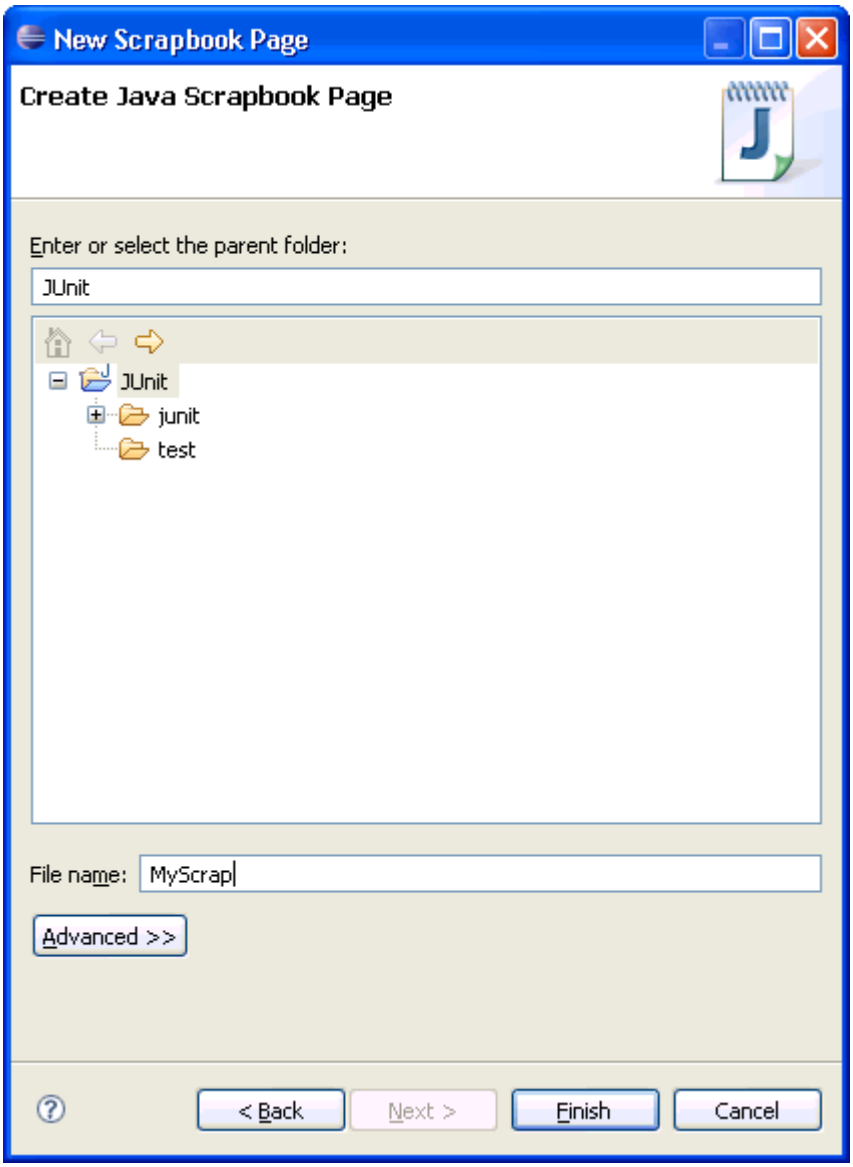
Related reference

Expressions view

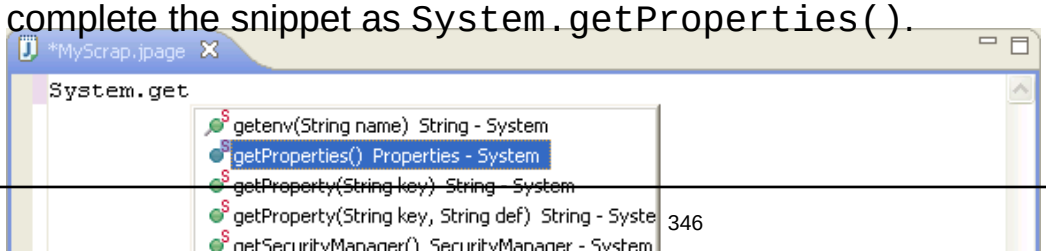
Evaluating snippets

In this section, you will evaluate Java expressions using the Java scrapbook. Java scrapbook pages allow you to experiment with Java code fragments before putting them in your program.

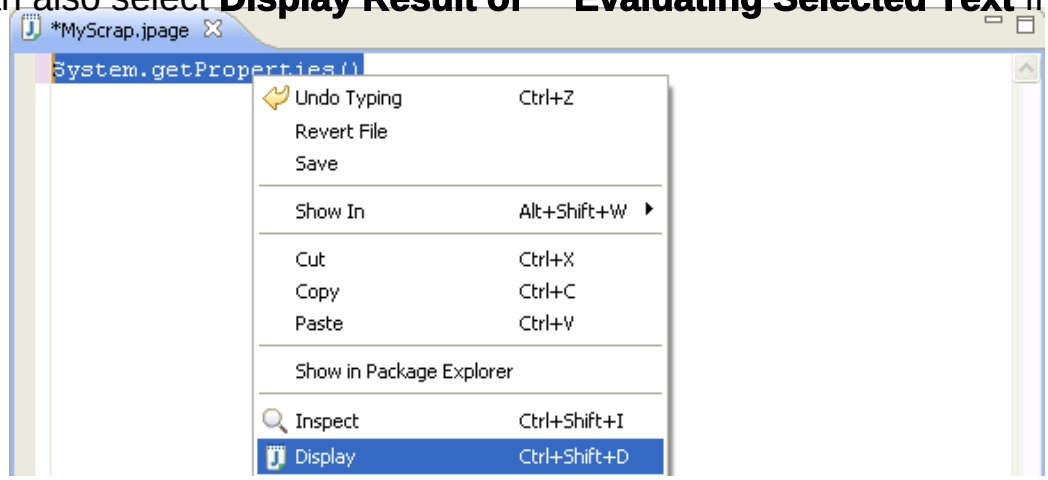
1. In the File menu select **New > Other > Java > Java Run/Debug > Scrapbook Page**. You will be prompted for a folder destination for the page.



2. In the **Enter or select the folder** field, type or browse below to select the *JUnit* project root directory.
3. In the **File name** field, type *MyScrap*.
4. Click **Finish** when you are done. A scrapbook page resource is created for you with the *jpage* file extension. (The *jpage* file extension will be added automatically if you do not enter it yourself.) The scrapbook page opens automatically in an editor.
5. In the editor, type `System.get` and then use content assist (`Ctrl+Space`) to complete the snippet as `System.getProperties()`.



6. Select the entire line you just typed and select **Display** from the context menu. You can also select **Display Result of Evaluating Selected Text** from the toolbar.



7. When the evaluation completes, the result of the evaluation is displayed and highlighted in the scrapbook page.
8. You can inspect the result of an evaluation by selecting text and choosing **Inspect** from the context menu (or selecting **Inspect Result of Evaluating Selected Text** from the toolbar.)
9. When you are finished evaluating code snippets in the scrapbook page, you can close the editor. Save the changes in the page if you want to keep the snippets for future use.

Next Section: Using the Java browsing perspective

Related concepts

Debugger

Related tasks

Creating a Java scrapbook page
Displaying the result of evaluating an expression
Inspecting the result of evaluating an expression

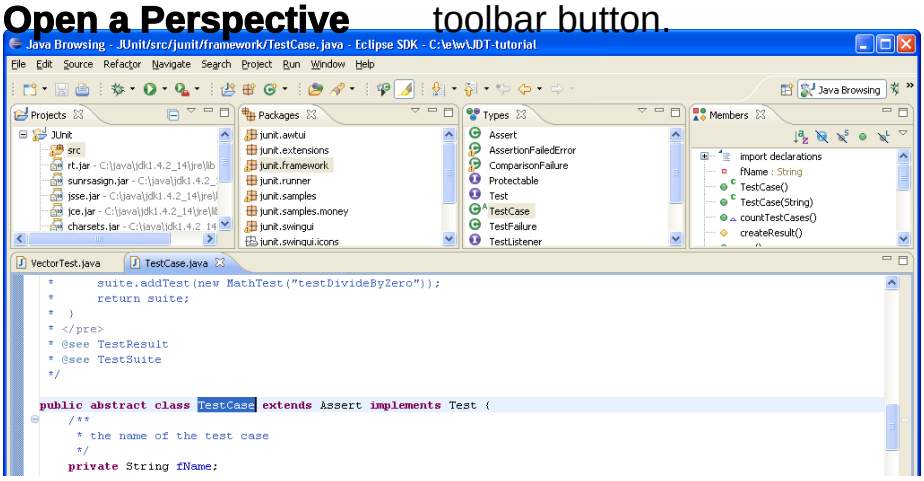
Related reference

New Java Scrapbook Page wizard
Java scrapbook page
Expressions view

Using the Java browsing perspective

In this section you will use the Java browsing perspective to browse and manipulate your code. Browsing Java elements with the Package Explorer gives an overview of using the Package Explorer to browse elements. In contrast to the Package Explorer, which organizes all Java elements in a tree, consisting of projects, packages, compilation units, types, etc., the browsing perspective uses distinct views to present the same information. Selecting an element in one view, will show its content in another view.

To open a browsing perspective activate [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Java Browsing** from within the Java perspective or use the context menu of the **Open a Perspective** toolbar button.



The views of the perspective are connected to each other in the following ways:

- Selecting an element in the **Projects** views shows its packages in the **Packages** view.
- The **Types** view shows the types contained in the package selected in the **Packages** view.
- The **Members** view shows the members of a selected type. Functionally, the **Members** view is comparable to the **Outline** view used in the normal Java perspective.
- Selecting an element in the **Members** view reveals the element in the editor. If there isn't an editor open for the element, double-clicking on the element will open a corresponding editor.

All four views are by default linked to the active editor. This means that the views will adjust their content and their selection according to the file presented in the active editor. The following steps illustrate this behavior:

1. Select *junit.extensions* in the **Packages** view.
2. Open type *TestSetup* in the editor by double-clicking it in the **Types view**.
3. Now give back focus to the editor opened on file *TestCase.java* by clicking on the editor tab. The **Packages**, **Types** and **Members** view adjust their content and selections to reflect the active editor. The **Packages** view's selection is set to *junit.framework* and the **Types** view shows the content of the *junit.framework* packages. In addition, the type *TestCase* is selected.

Functionally, the Java browsing perspective is fully comparable to the Java perspective. The context menus for projects, packages, types, etc. and the global menu and tool bar are the same. Therefore activating these functions is analogous to activating them in the Java perspective.

Next Section: Writing and running JUnit tests

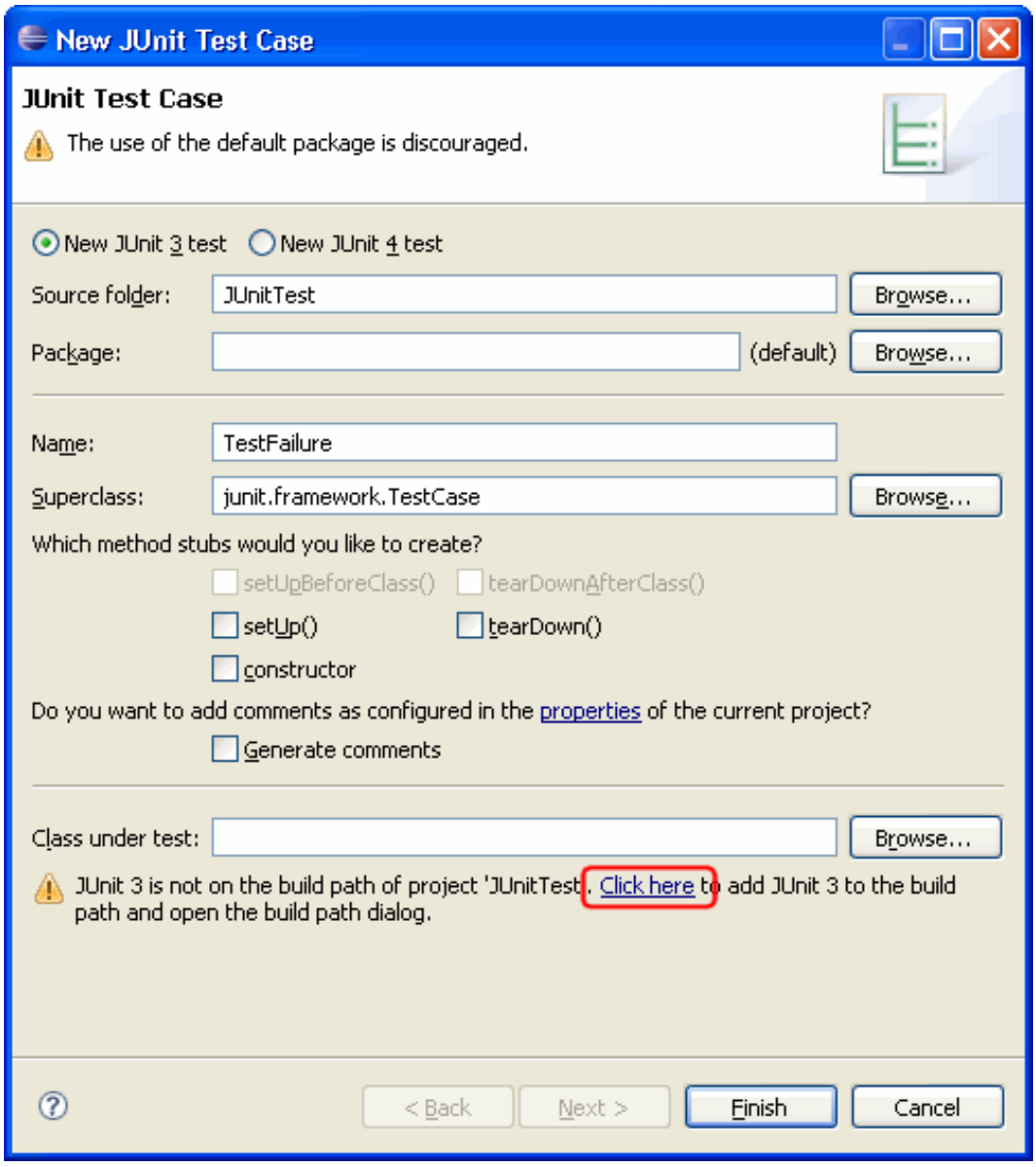
Writing and running JUnit tests

In this section, you will be using the JUnit testing framework to write and run tests. To get started with JUnit you can refer to the JUnit Cookbook.

Writing Tests

Create a project "JUnitTest". Now you can write your first test. You implement the test in a subclass of **TestCase**. You can do so either using the standard Class wizard or the specialized **Test Case** wizard:

1. Open the New wizard (**File > New > JUnit Test Case**).
2. Select **New JUnit 3 test** and enter "*TestFailure*" as the name of your test class:



Note: If you want to use JUnit 4 tests you have to make sure that your compiler compliance is set to 1.5.

3. You will see a warning message asking you to add the junit library to the build path. Use the **Click here** link to add the junit library automatically.
4. Click **Finish** to create the test class.

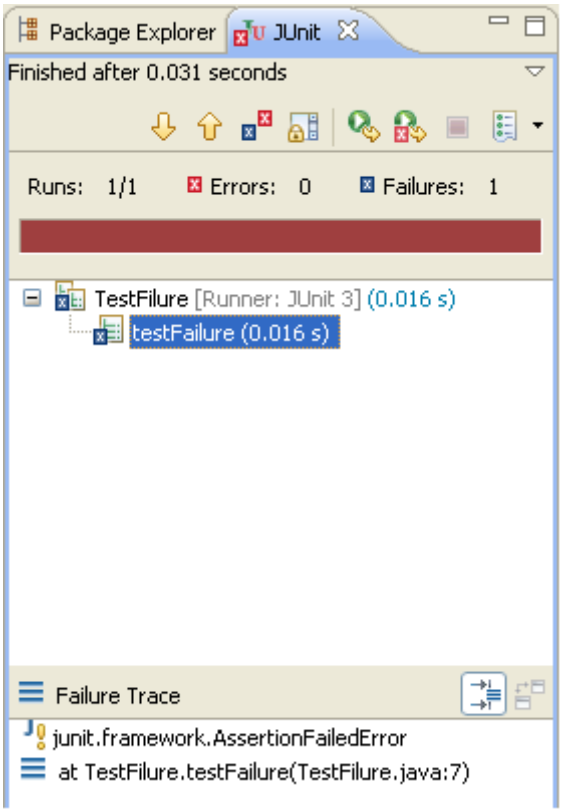
Add a test method that fails to the class *TestFailure*. A quick way to enter a test method is with the *test* template. To do so, place the cursor inside the class declaration. Type "test" followed by `Ctrl+Space` to activate code assist and select the "test" template. Change the name of the created method to *testFailure* and invoke the *fail()* method.

```
public void testFailure() throws Exception {

        fail();

  }
```

Now you are ready to run your first test.

Running Tests

To run TestFailure hit the run button in the toolbar. It will automatically run as JUnit Test. You can inspect the test results in the *JUnit* view. This view shows you the test run progress and status:



The view is shown in the current perspective whenever you start a test run. A convenient arrangement for the JUnit view is to dock it as a fast view. The JUnit view has two tabs: one shows you a list of failures and the other shows you the full test suite as a tree. You can navigate from a failure to the corresponding source by double clicking the corresponding line in the failure trace.

Dock the JUnit view as a fast view, remove the *fail()* statement in the method *testFailure()* so that the test passes and rerun the test again. You can rerun a test either by clicking the **Rerun** button in the view's tool bar or you can re-run the program that was last launched by activating the **Run** drop down. This time the test should succeed. Because the test was successful, the JUnit view doesn't pop up, but the success indicator shows on the JUnit view icon and the

status line shows the test result. As a reminder to rerun your tests the view icon is decorated by a "*" whenever you change the workspace contents after the run.

- A successful test run

- A successful test run, but the workspace contents have changed since the last test run.

In addition to running a test case as described above you can also:

- Run all tests inside a project, source folder, or package -

Select a project, package or source folder and run all the included tests with **Run as > JUnit Test**. This command finds all tests inside a project, source folder or package and executes them.

- Run a single test method -

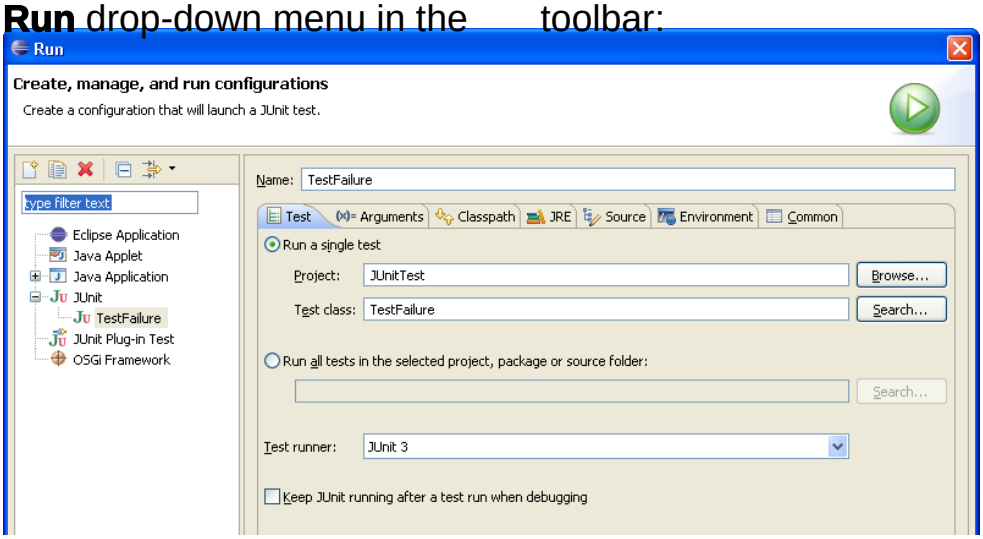
Select a test method in the Outline or Package Explorer and with **Run as > JUnit Test** the selected test method will be run.

- Rerun a single test -

Select a test in the JUnit view and execute **Run** from the context menu.

Customizing a Test Configuration

When you want to pass parameters or customize the settings for a test run you open the Launch Configuration Dialog. Select **Open Run Dialog...**.in the **Run** drop-down menu in the toolbar:



In this dialog you can specify the test to be run, its arguments, its run-time class path, and the Java run-time environment.

Debugging a Test Failure

In the case of a test failure you can follow these steps to debug it:

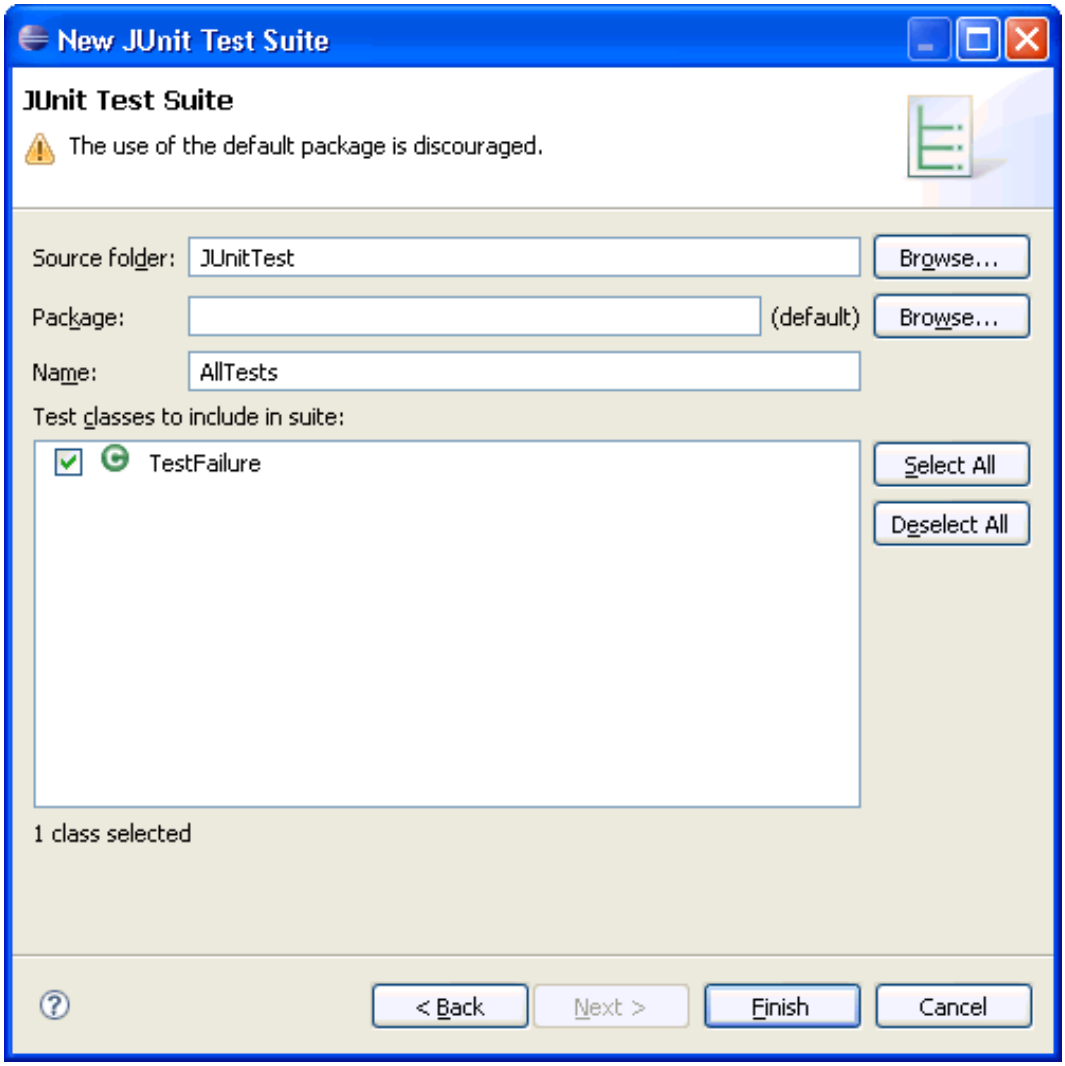
1. Double click the failure entry from the Failures tab in the JUnit view to open the corresponding file in the editor.
2. Set a breakpoint at the beginning of the test method.
3. Select the test case and execute **Debug As>JUnit Test** from the **Debug** drop down.

A JUnit launch configuration has a "keep alive" option. If your Java virtual machine supports "hot code replacement" you can fix the code and rerun the test without restarting the full test run. To enable this option select the **Keep JUnit running after a test run when debugging** checkbox in the JUnit launch configuration.

Creating a Test Suite

The JUnit **TestSuite** wizard helps you with the creation of a test suite. You can select the set of classes that should belong to a suite.

1. Open the New wizard
2. Select **Java > JUnit > JUnit Test Suite** and click **Next**.
3. Enter a name for your test suite class (the convention is to use "AllTests" which appears by default).



4. Select the classes that should be included in the suite. We currently have a single test class only, but you can add to the suite later.

You can add or remove test classes from the test suite in two ways:

- Manually by editing the test suite file
- By re-running the wizard and selecting the new set of test classes.

Note: the wizard puts 2 markers, `//$JUnit-BEGIN$` and `//$JUnit-END$`, into the created Test suite class, which allows the wizard to update existing test suite classes. Editing code between the markers is not recommended.

Next Section: Project configuration tutorial

Project configuration tutorial

In this section, you will create and configure a new Java project to use source folders and to match some existing layout on the file system. Some typical layouts have been identified. Choose the sub-section that matches your layout.

Next Section: Detecting existing layout

Related concepts

Java projects
Java views

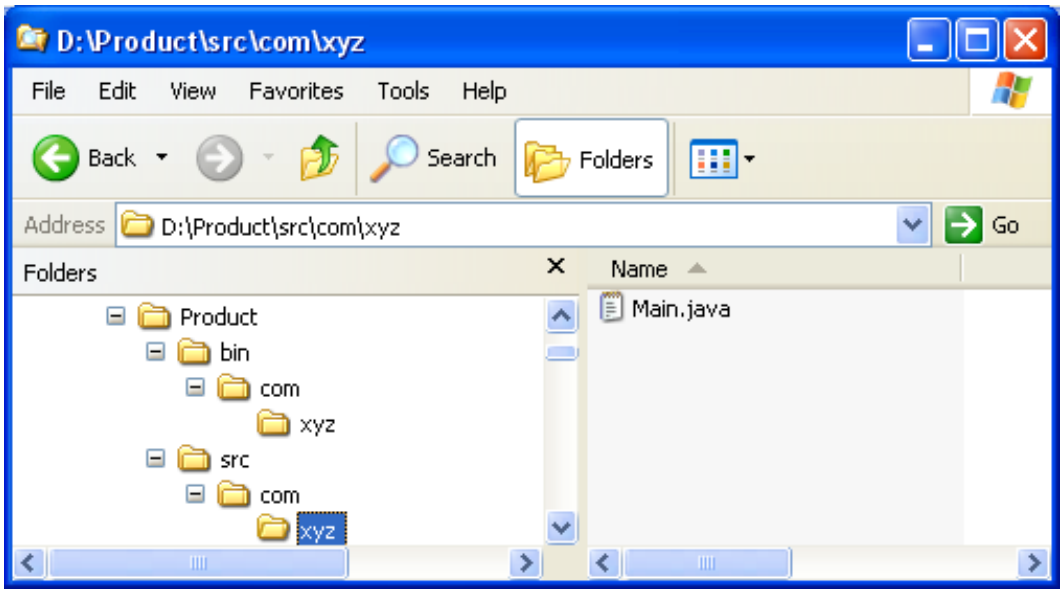
Related reference

New Java Project Wizard
Package Explorer View

Detecting existing layout

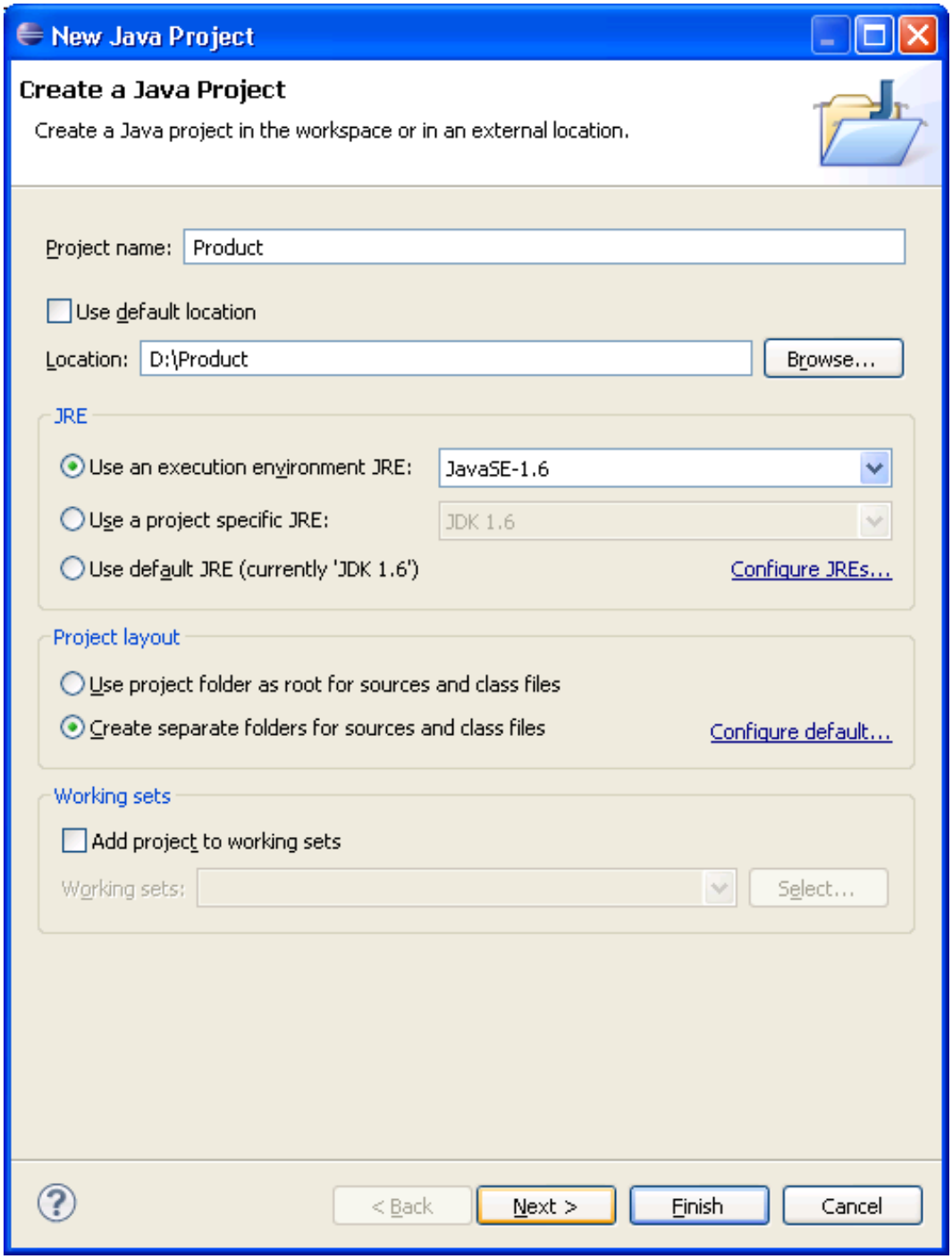
Layout on file system

- The source files for a product are laid out in one directory "src".
- The class files are in another directory "bin".

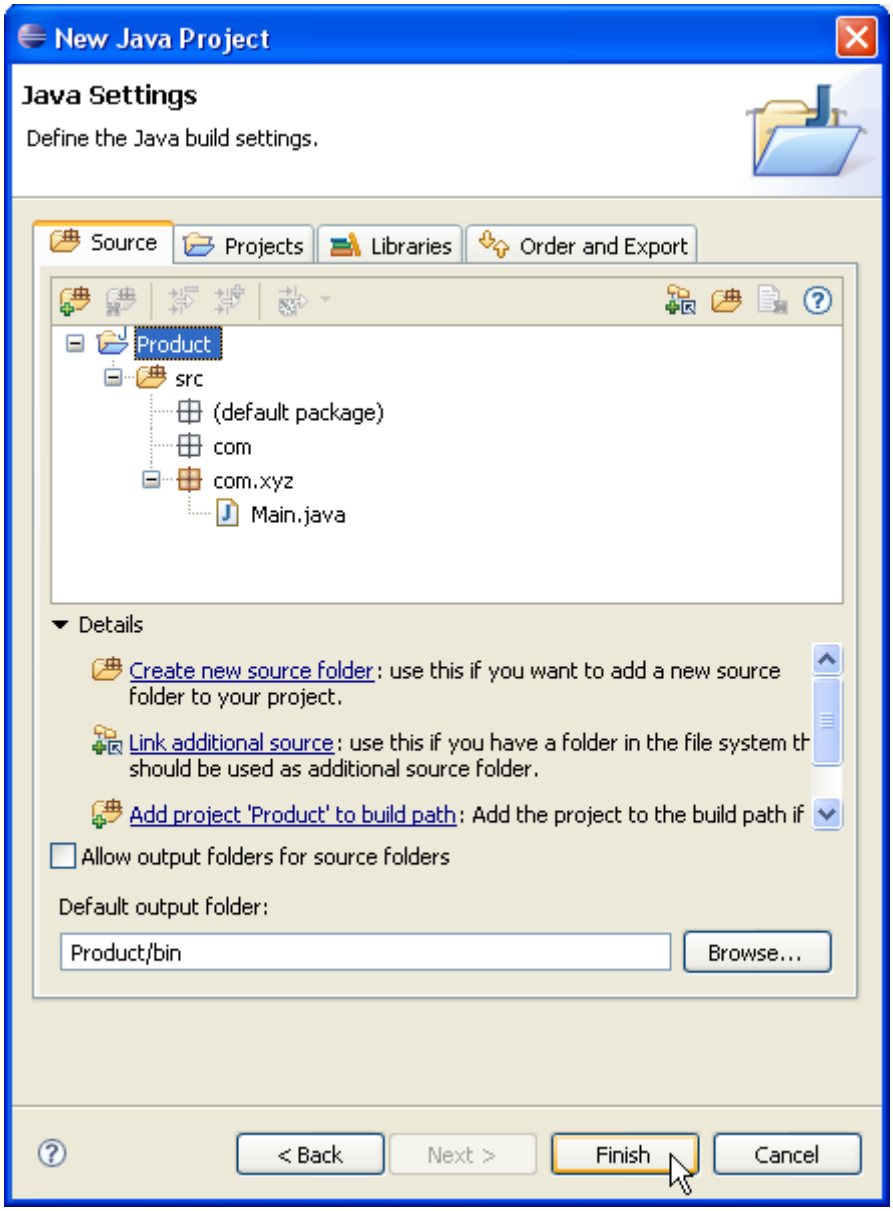


Steps for defining a corresponding project

1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Product" in the **Project name** field.
4. In **Project layout** group, change selection to **Create separate source and output folders**.
   Deselect **Use default location**.
   Click **Browse...** and choose the *Product* directory on drive *D:*.

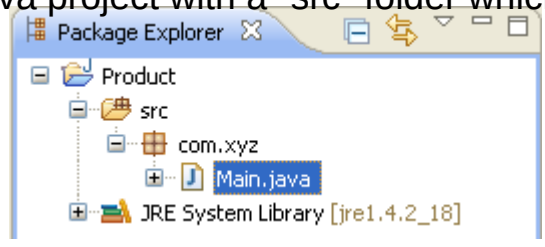


5. Click **Next**.
   Ensure that the source and output folders are detected.



Warning: If the **Scrub output folders when cleaning projects** preference in the **Output folder** section of the [Image: /topic/org.eclipse.help/command_link.png] **Java > Compiler > Building** preference page is checked, clicking **Finish** will scrub the "bin" directory in the file system before generating the class files.

6. Click **Finish**.
7. You now have a Java project with a "src" folder which contains the sources of the "Product" directory.



Note: This solution creates a ".project" file and a ".classpath" file in the "Product" directory. If you do not wish to have these files in the "Product" directory, you should use linked folders as shown in the Sibling products in a common source tree section.

Next Section: Organizing sources

Related concepts

Java projects

Java views

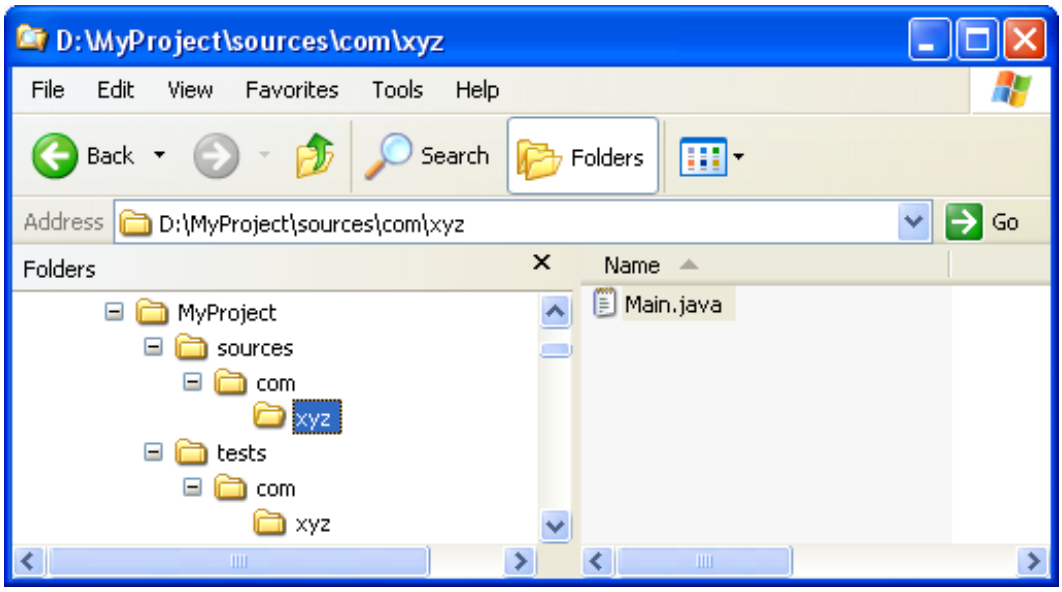
Related reference

New Java Project Wizard
Package Explorer View

Organizing sources

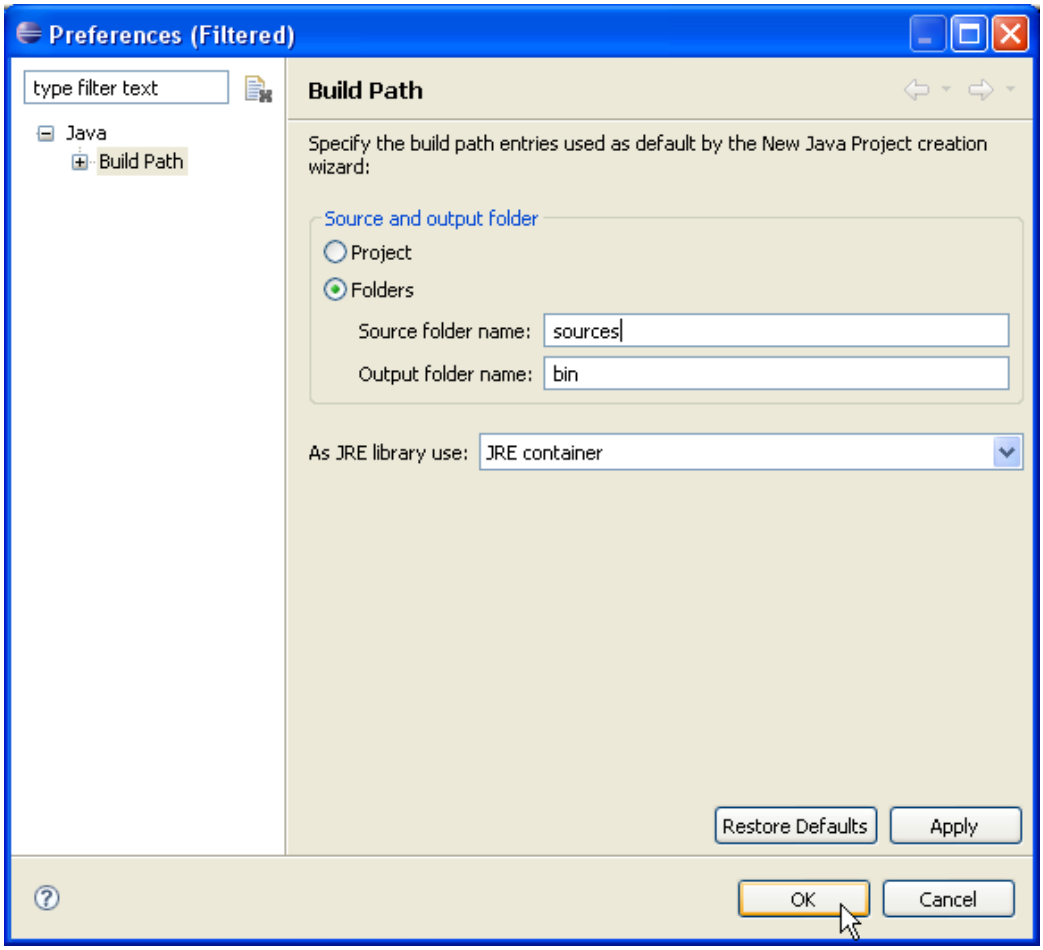
Layout on file system

- In this section, you will create a new Java project and organize your sources in separate folders. This will prepare you for handling more complex layouts.
- Let's assume you want to put your sources in one folder and your tests in another folder:

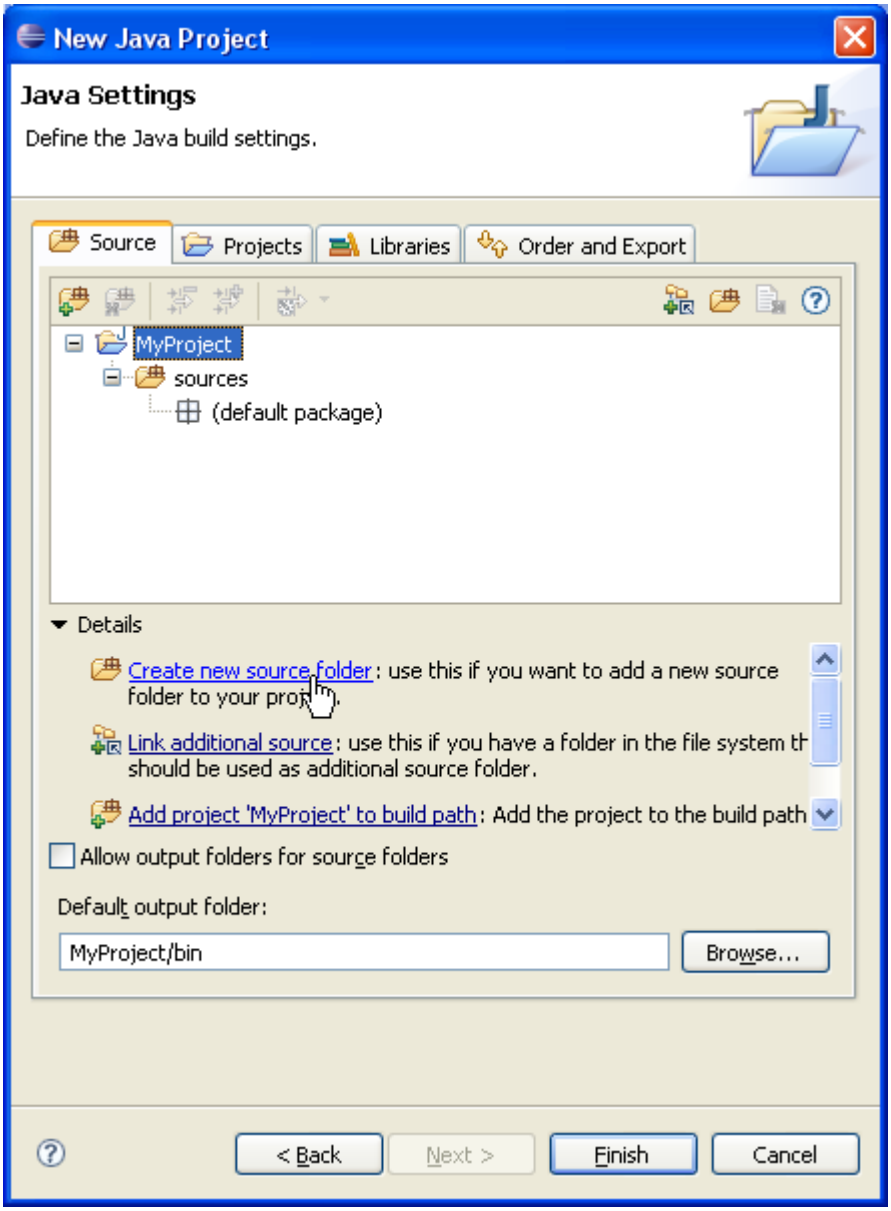


Steps for defining a corresponding project

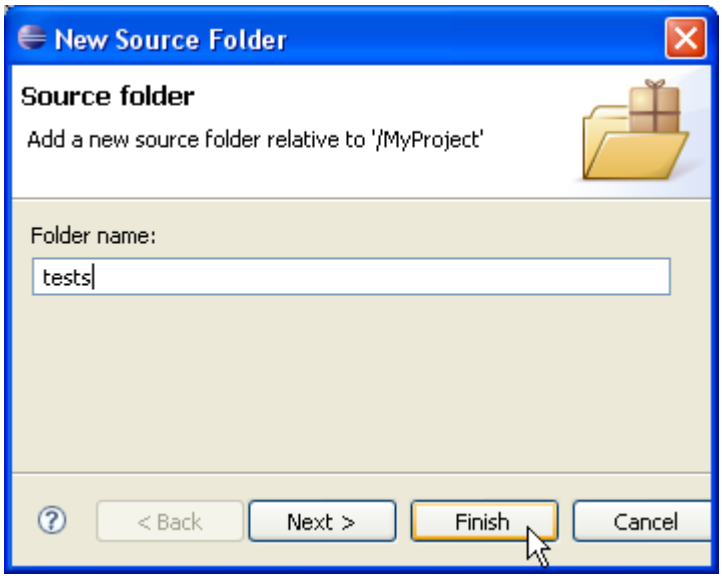
1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "MyProject" in the **Project name** field.
4. In **Project layout** group, change selection to **Create separate source and output folders** and edit **Configure default...** to modify **Source folder name** from "src" to "sources".



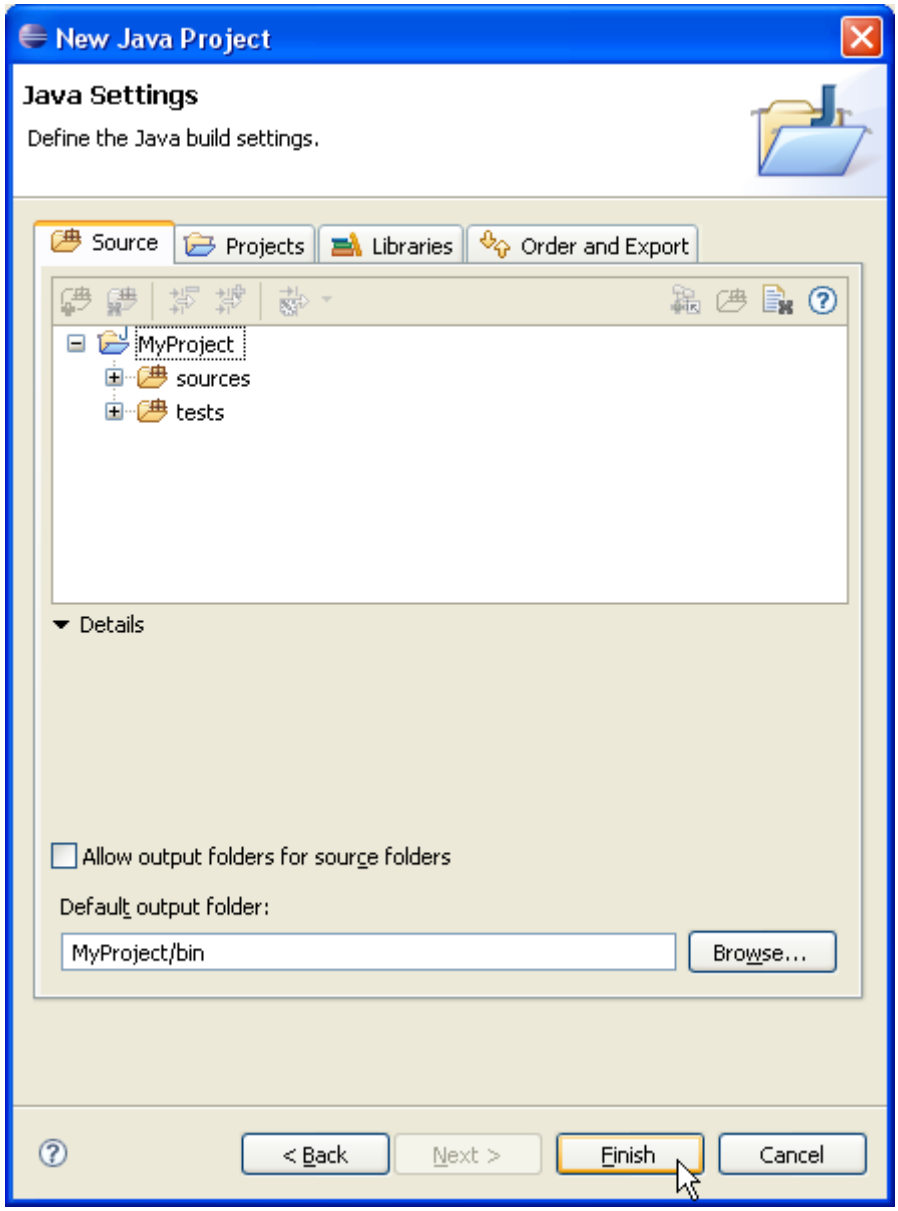
5. Click **OK** to return on New Java Project wizard and then click **Next**.



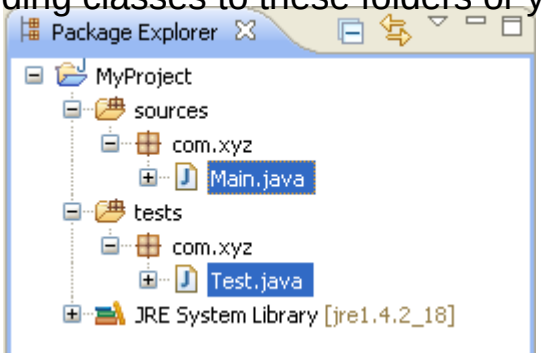
6. To add another source folder to your project, click **Create new source folder** link in **Details** pane or button in view bar.
7. In **New Source Folder** dialog, type "tests" in the **Folder name** field.



8. Click **Finish** to validate and close the dialog.
9. Your project setup now looks as follows:



10. Click **Finish**
11. You now have a Java project with two source folders: *sources* and *tests*. You can start adding classes to these folders or you can copy them using drag and drop.



Next Section: Sibling products in a common source tree

Related concepts

Java projects
Java views

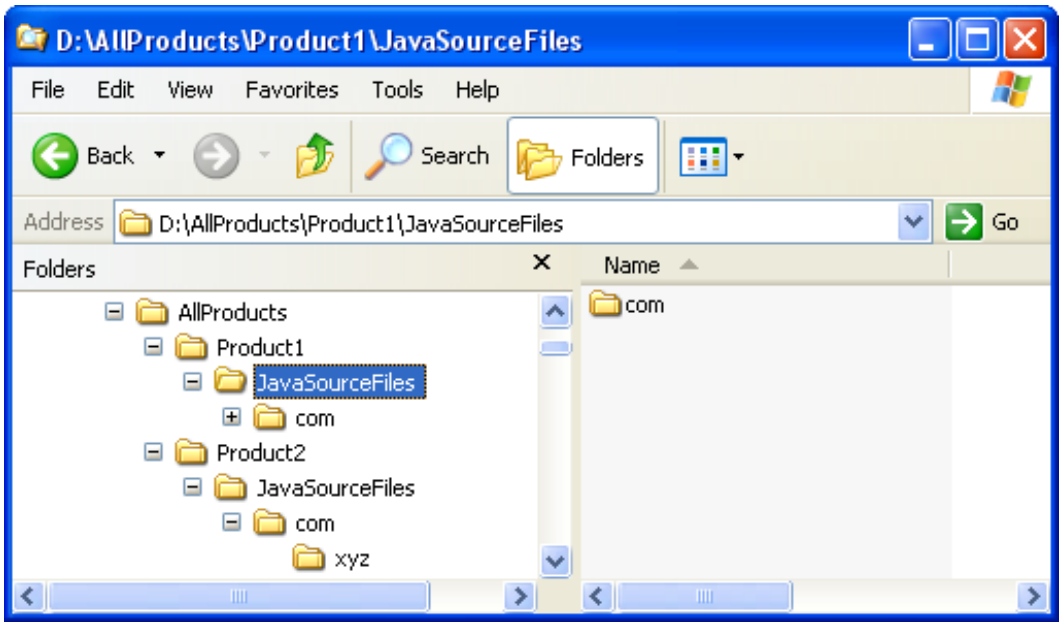
Related reference

New Java Project Wizard
Package Explorer View

Sibling products in a common source tree

Layout on file system

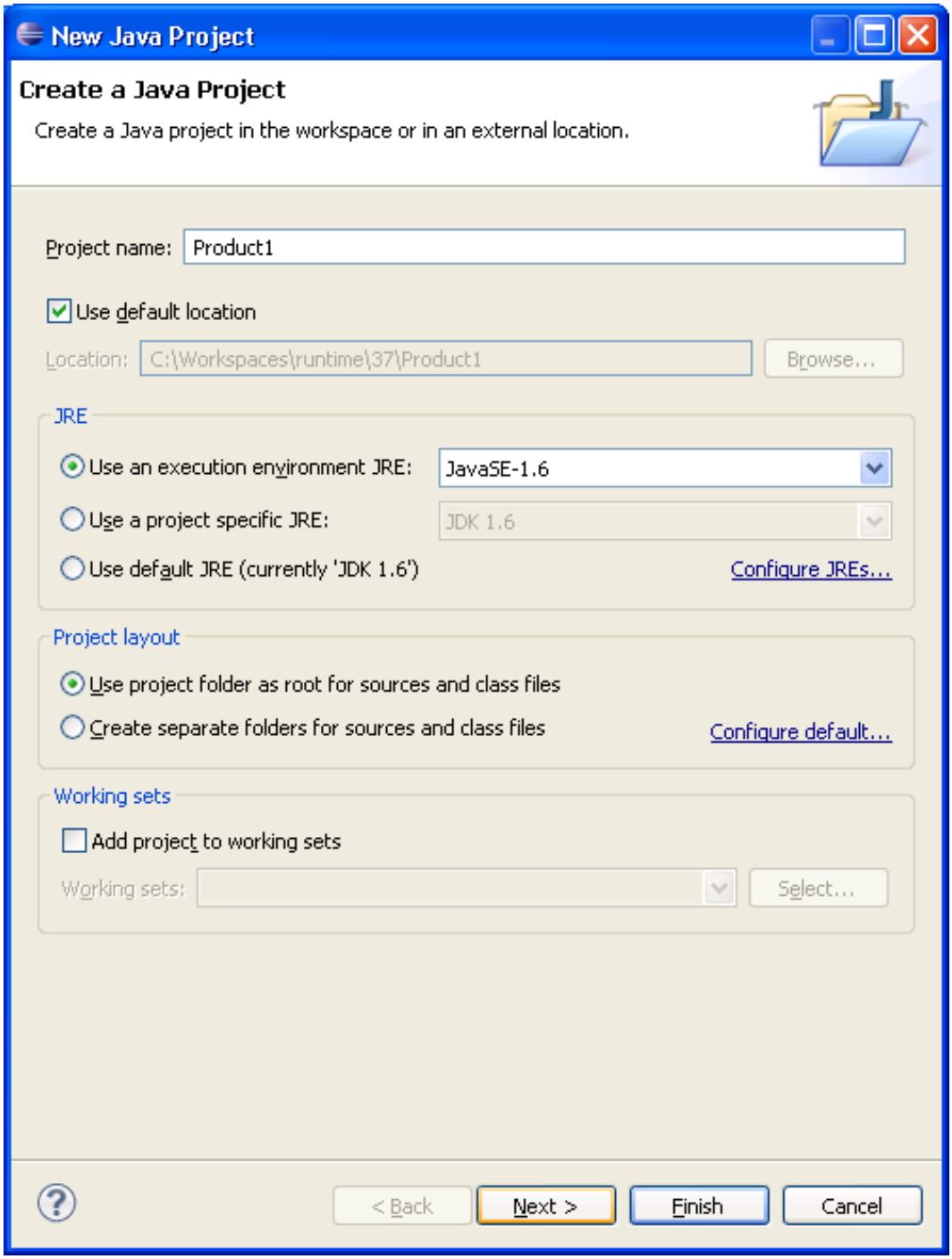
- The source files for products are laid out in one big directory that is version and configuration managed outside Eclipse.
- The source directory contains two siblings directories *Product1* and *Product2*.



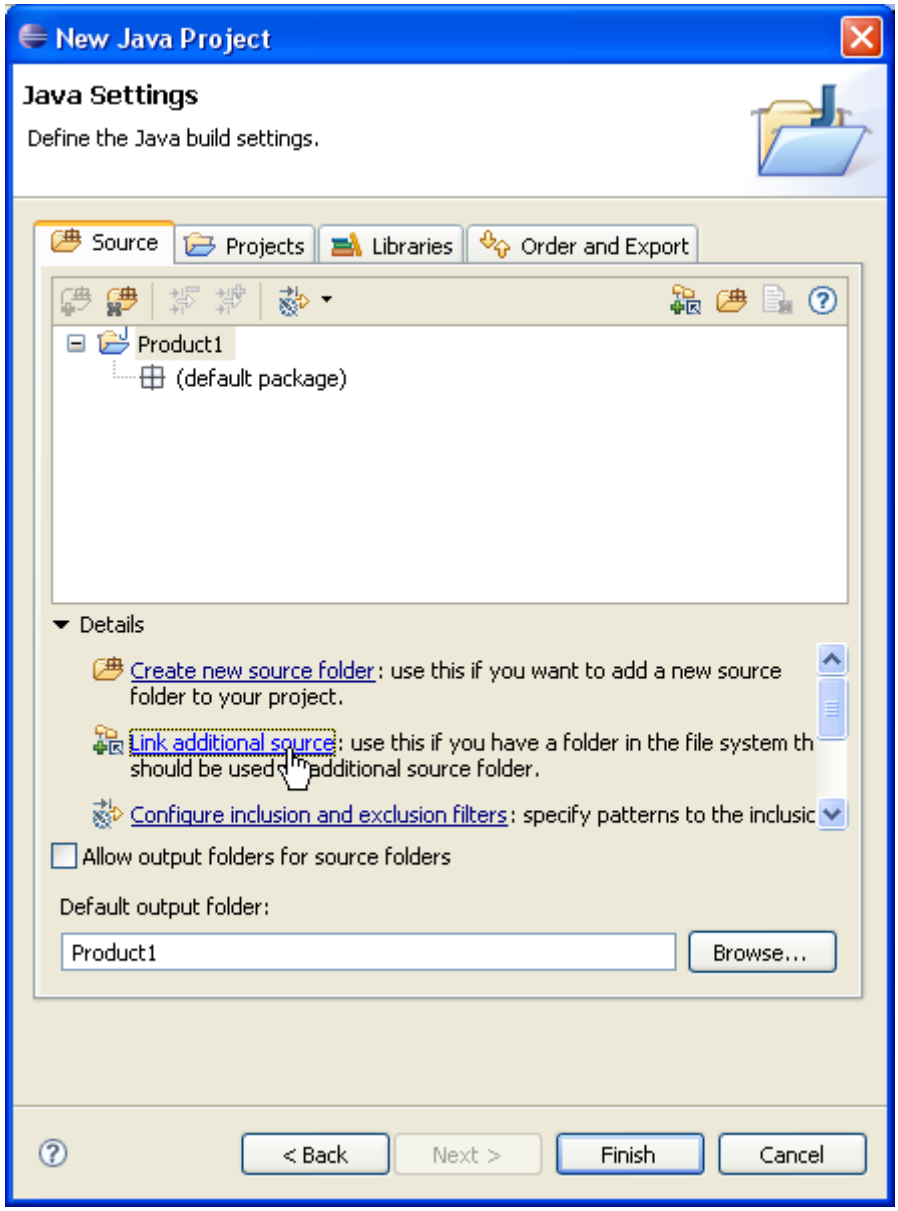
Steps for defining corresponding projects

1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.

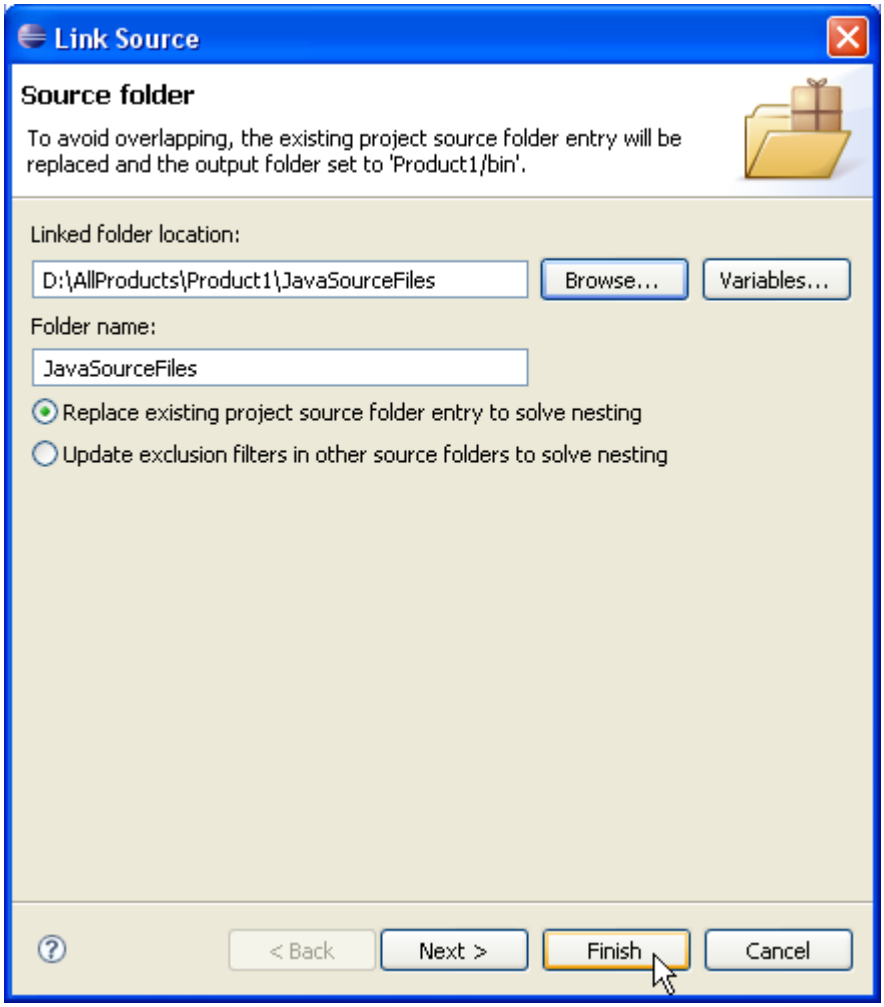
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Product1" in the **Project name** field. Click **Next**.



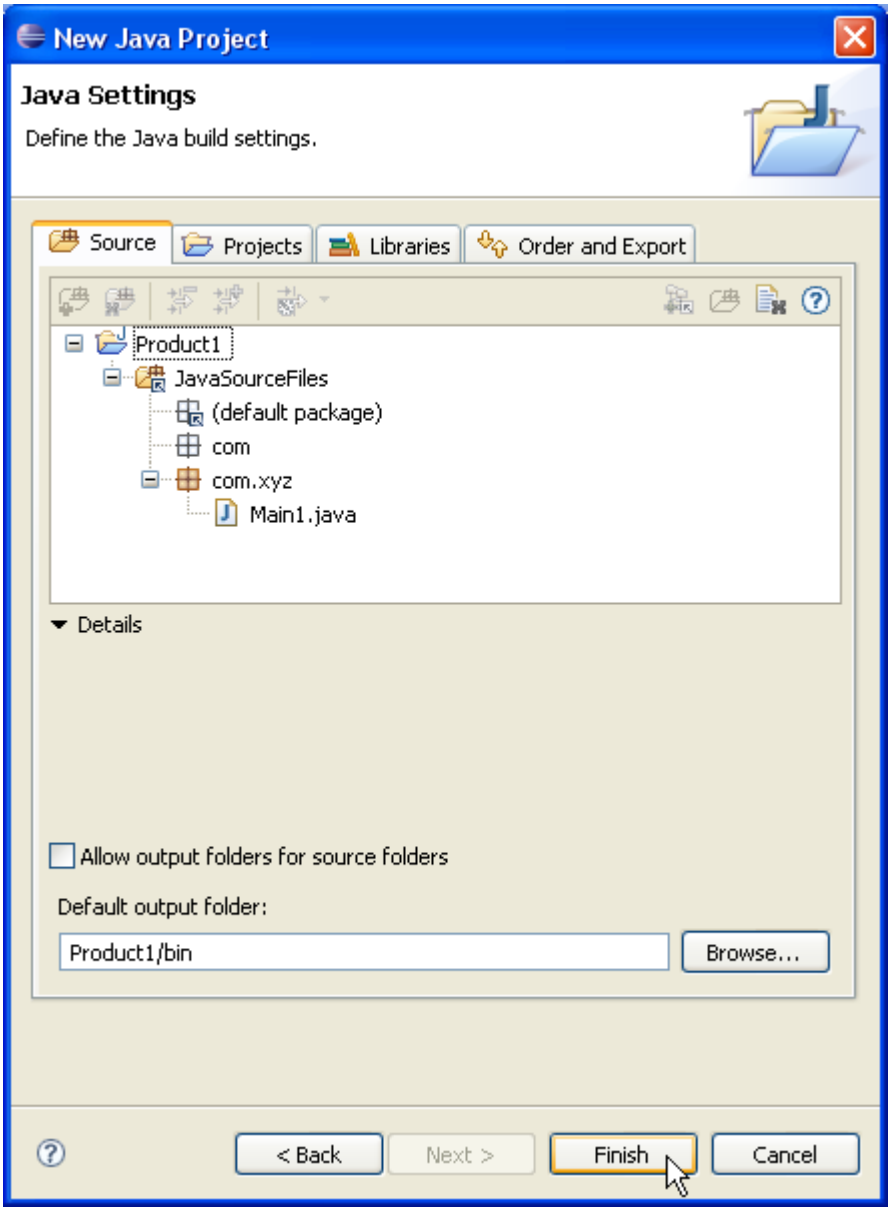
4. On the next page, Select "Product1" source folder.
Click **Link additional source** link in **Details** pane or button in view bar.



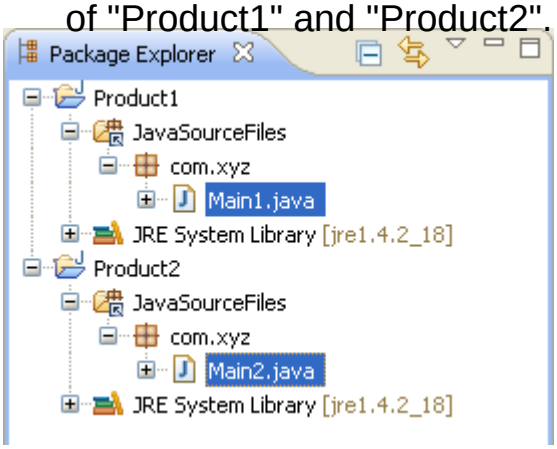
5. In **Link Source** dialog click **Browse....** and choose the *D:\AllProducts\Product1\JavaSourceFiles* directory.



6. Click **Finish** to validate and close the dialog.
7. Your project source setup now looks as follows:



8. Click **Finish**.
9. Repeat steps 2 to 8 for "Product2".
10. You now have two Java projects which respectively contain the sources of "Product1" and "Product2".



Next Section: Overlapping products in a common source tree

Related concepts

Java projects
Java views

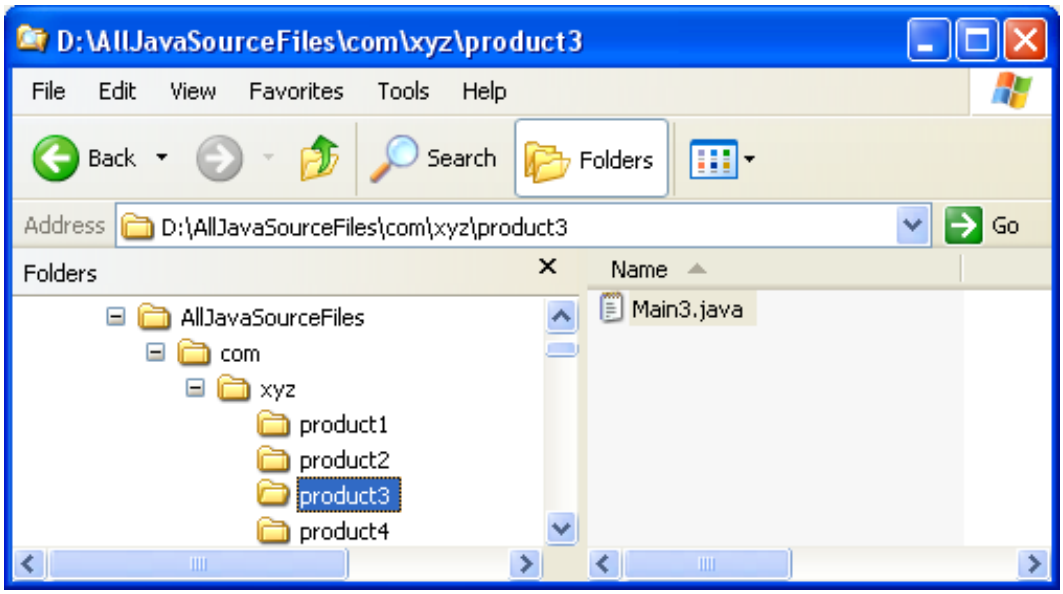
Related reference

New Java Project Wizard
Package Explorer View

Overlapping products in a common source tree

Layout on file system

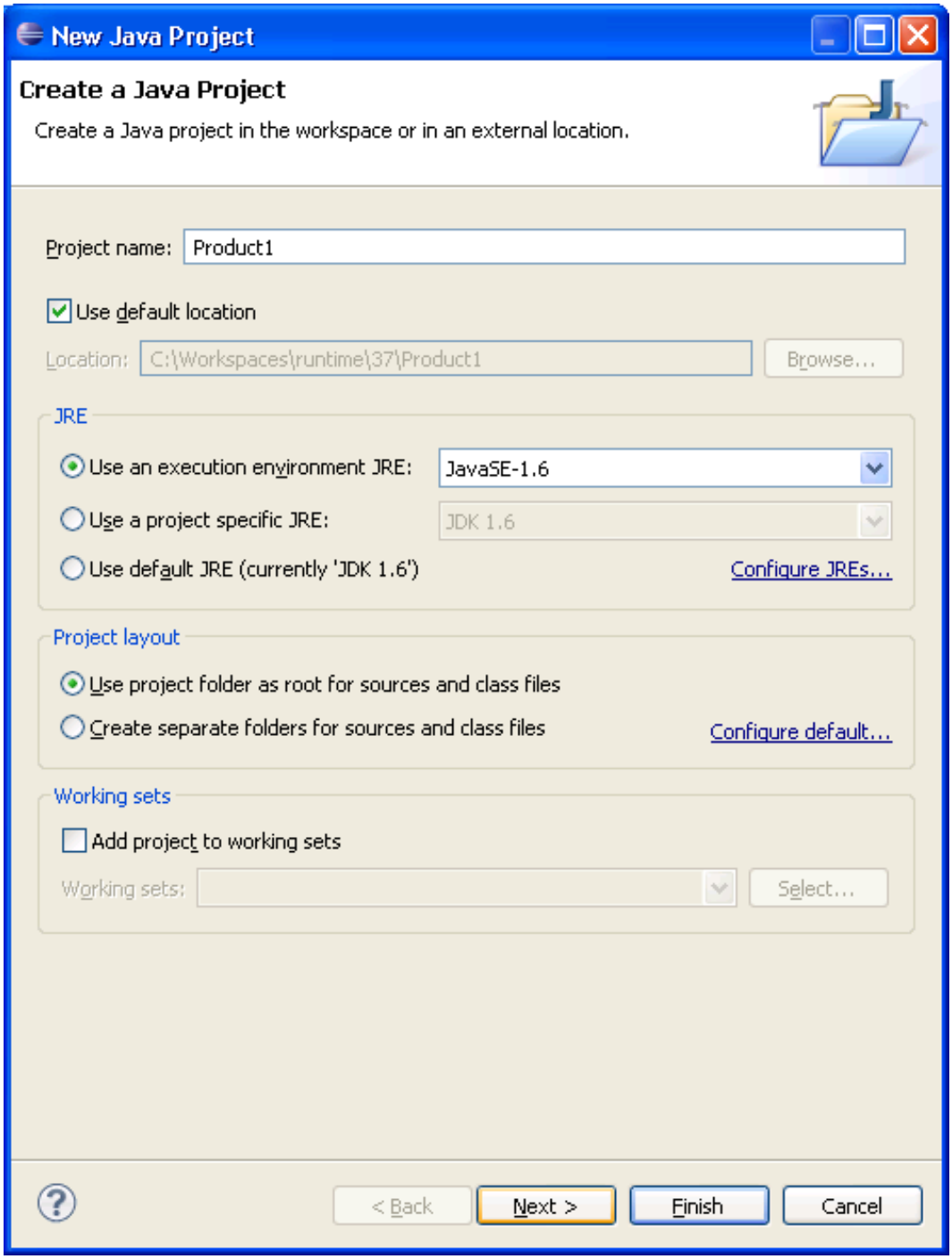
- The Java source files for products are all held in a single main directory.
- Products are separated into four siblings packages *Product1*, *Product2*, *Product3* and *Product4*.



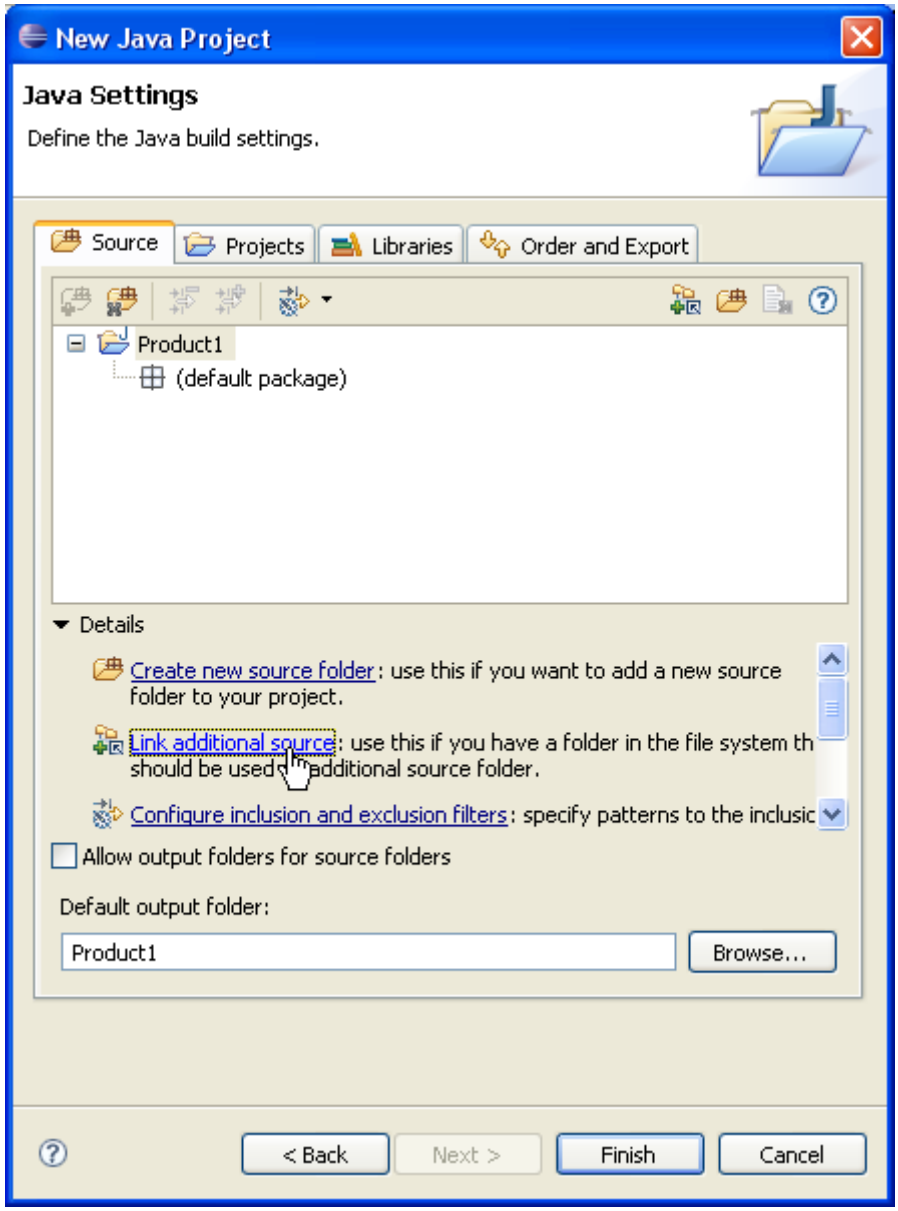
Steps for defining corresponding "Product1" and "Product2" projects

1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.

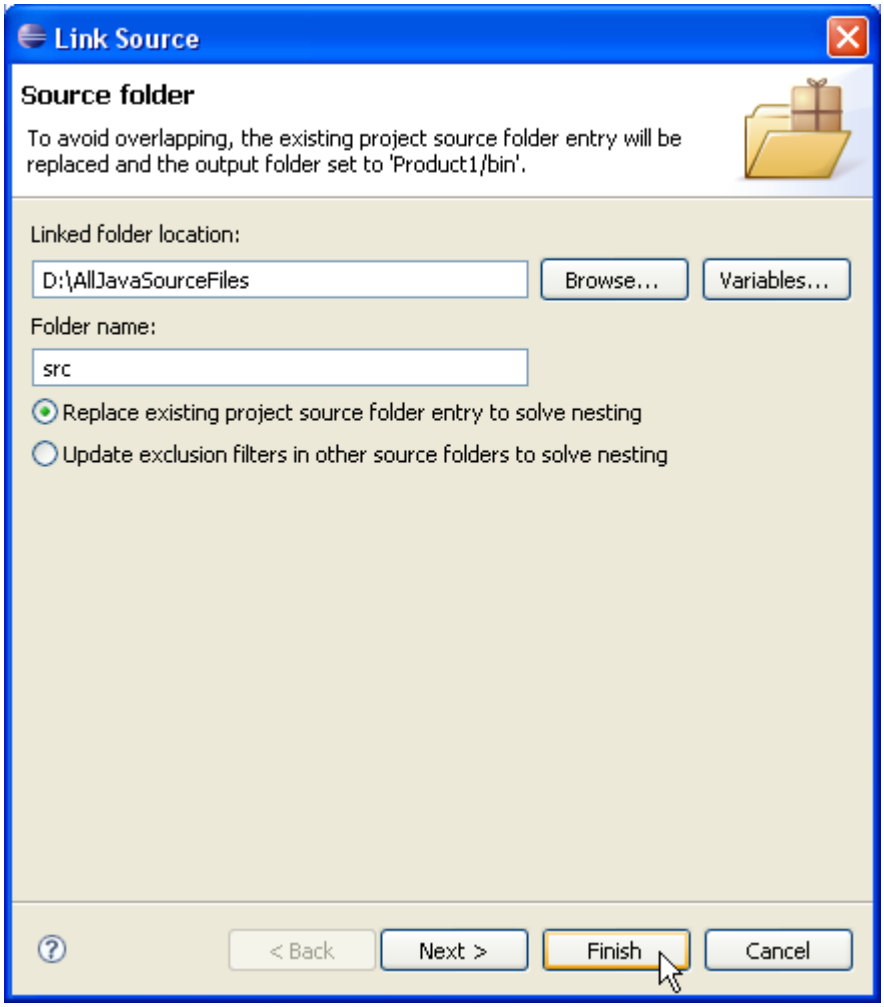
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Product1" in the **Project name** field. Click **Next**.



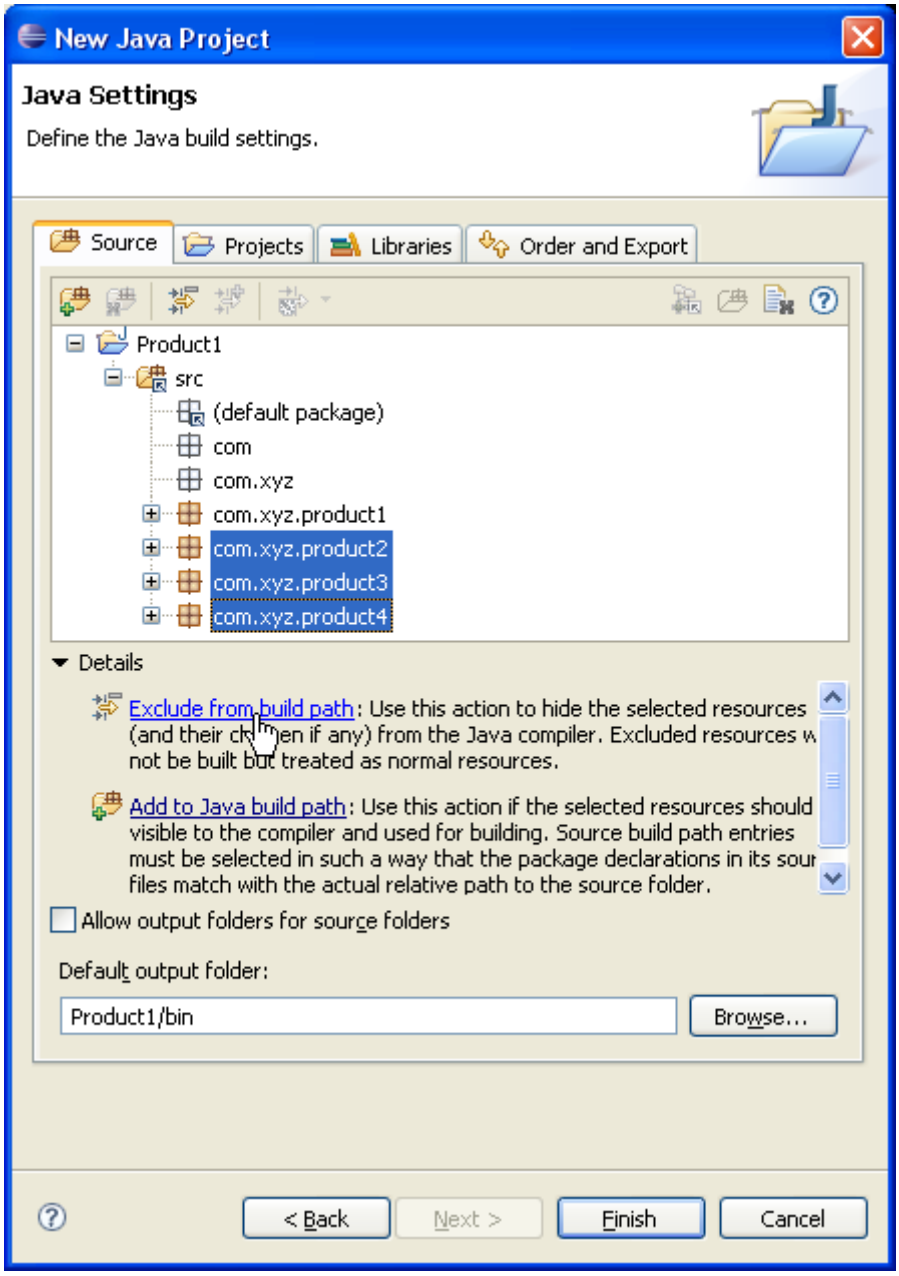
4. On the next page, Select "Product1" source folder.
Click **Link additional source** link in **Details** pane or button in view bar.



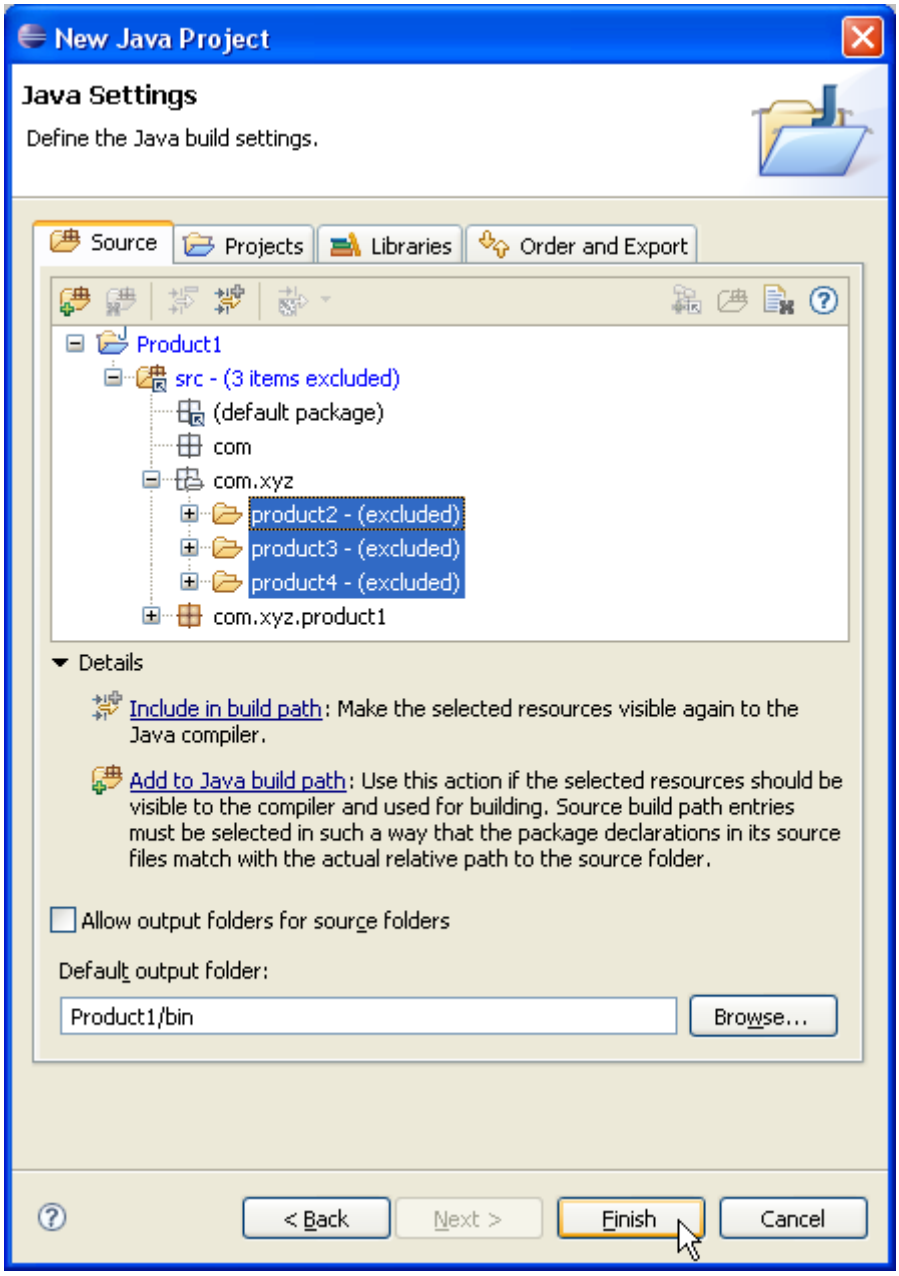
5. In **Link Source** click **Browse....** and choose the *D:\AllJavaSourceFiles* directory.
Type "src" in **Folder name**.



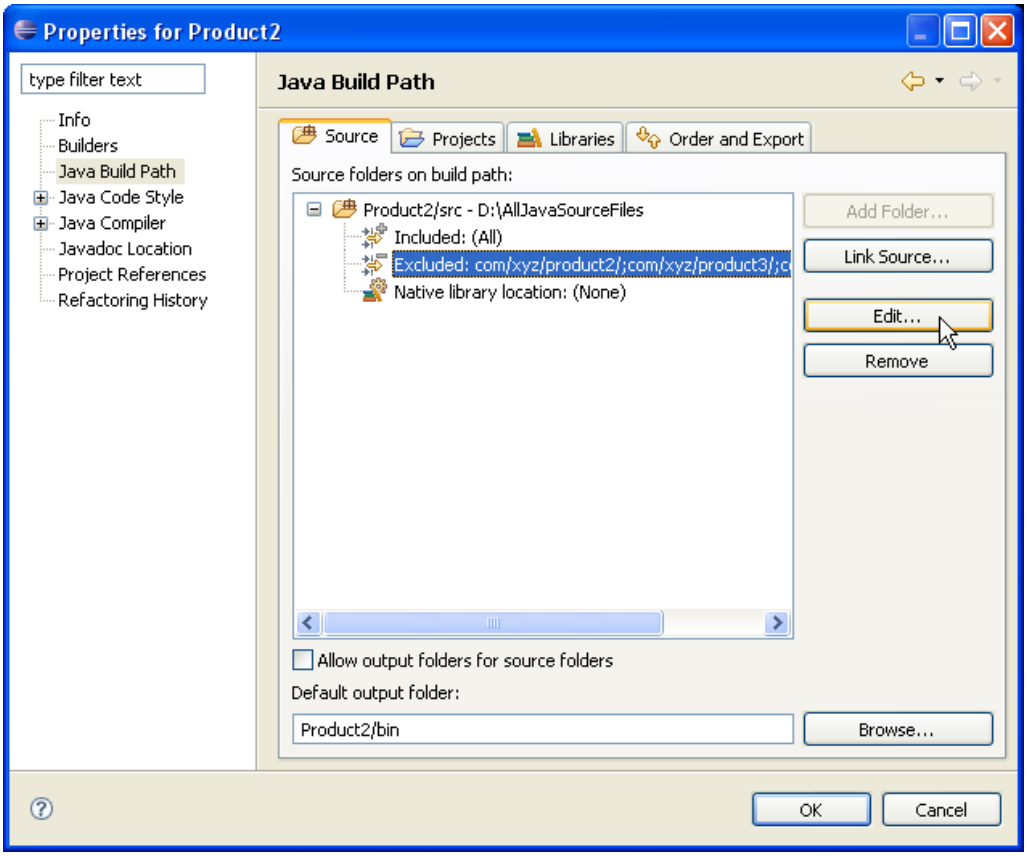
6. Click **Finish** to validate and close the dialog.
7. Expand the "src" source folder. Select the three last packages and exclude them from build path using either **Exclude from build path** link or **Exclude** popup-menu item.



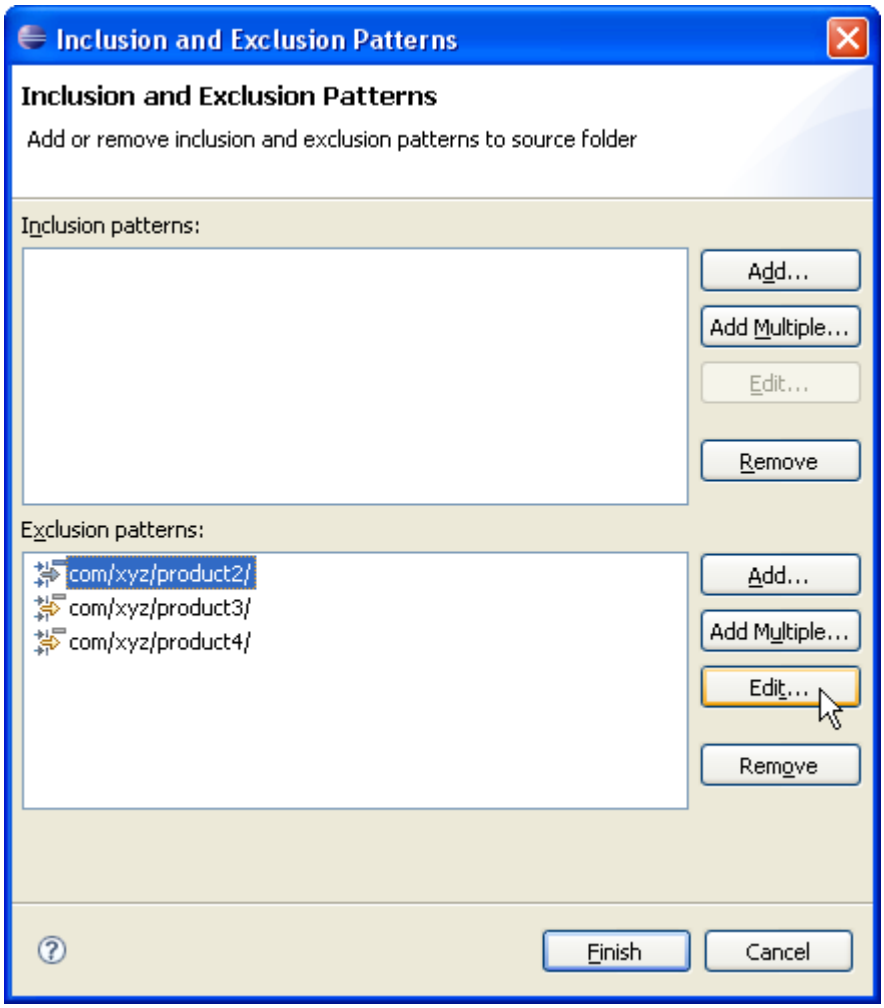
8. Your project source setup now looks as follows:



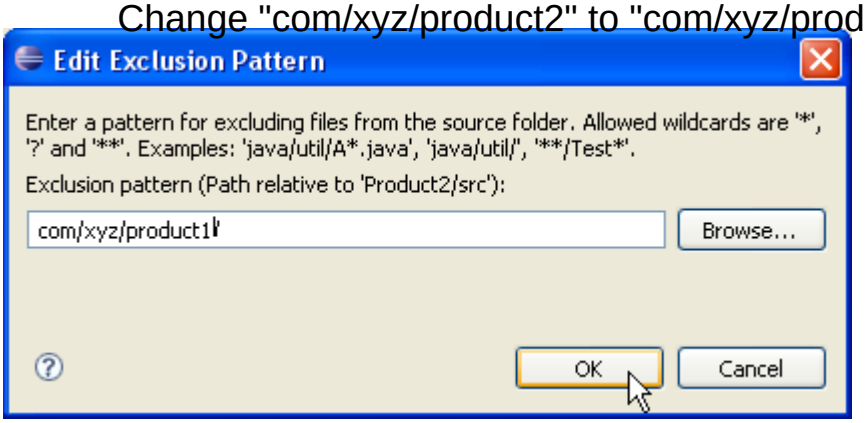
9. Click **Finish**.
10. Copy "Product1" project and paste it as "Product2".
    Edit "Product2" project properties and go on **Java Build Path** page.
    In **Source** tab, expand "Product2/src" source folder, select **Excluded** and click **Edit...**.



11. In **Inclusion and Exclusion Patterns**, select "com/xyz/product2" and click **Edit...**.

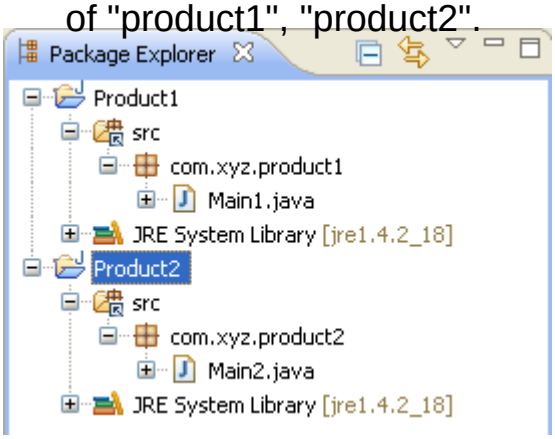


12. Change "com/xyz/product2" to "com/xyz/product1".



13. Click **OK** to enter the change.
Click **Finish** to validate and close **Inclusion and Exclusion Patterns** dialog.
Click **OK** again to validate "Product2" project properties changes.

14. You now have two Java projects which respectively contain the sources of "product1", "product2".



Next Section: Product with nested tests

Related concepts

Java projects
Java views

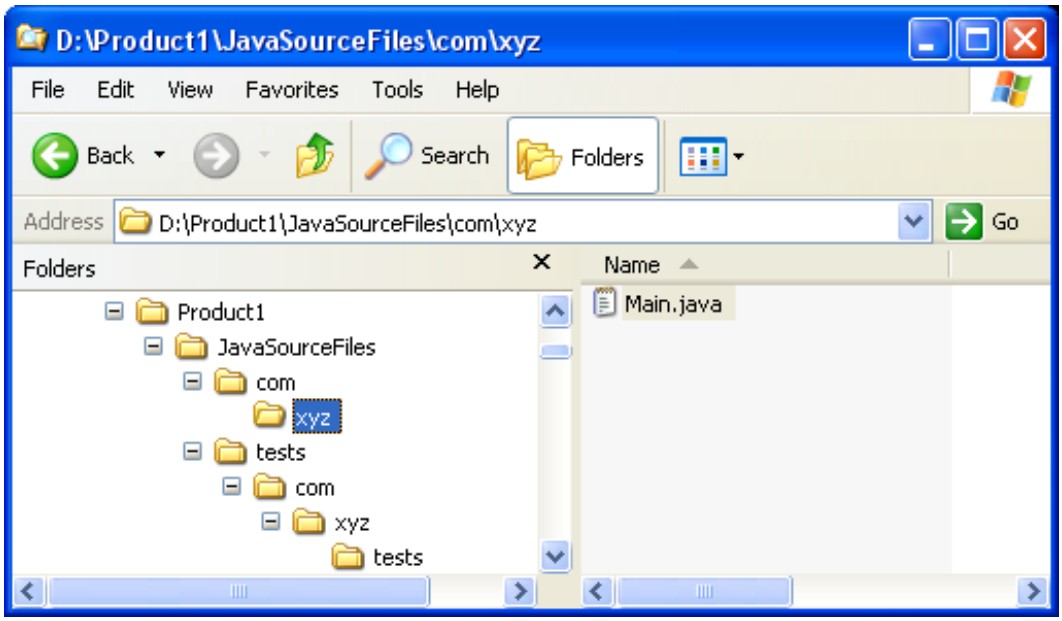
Related reference

New Java Project Wizard
Package Explorer View

Product with nested tests

Layout on file system

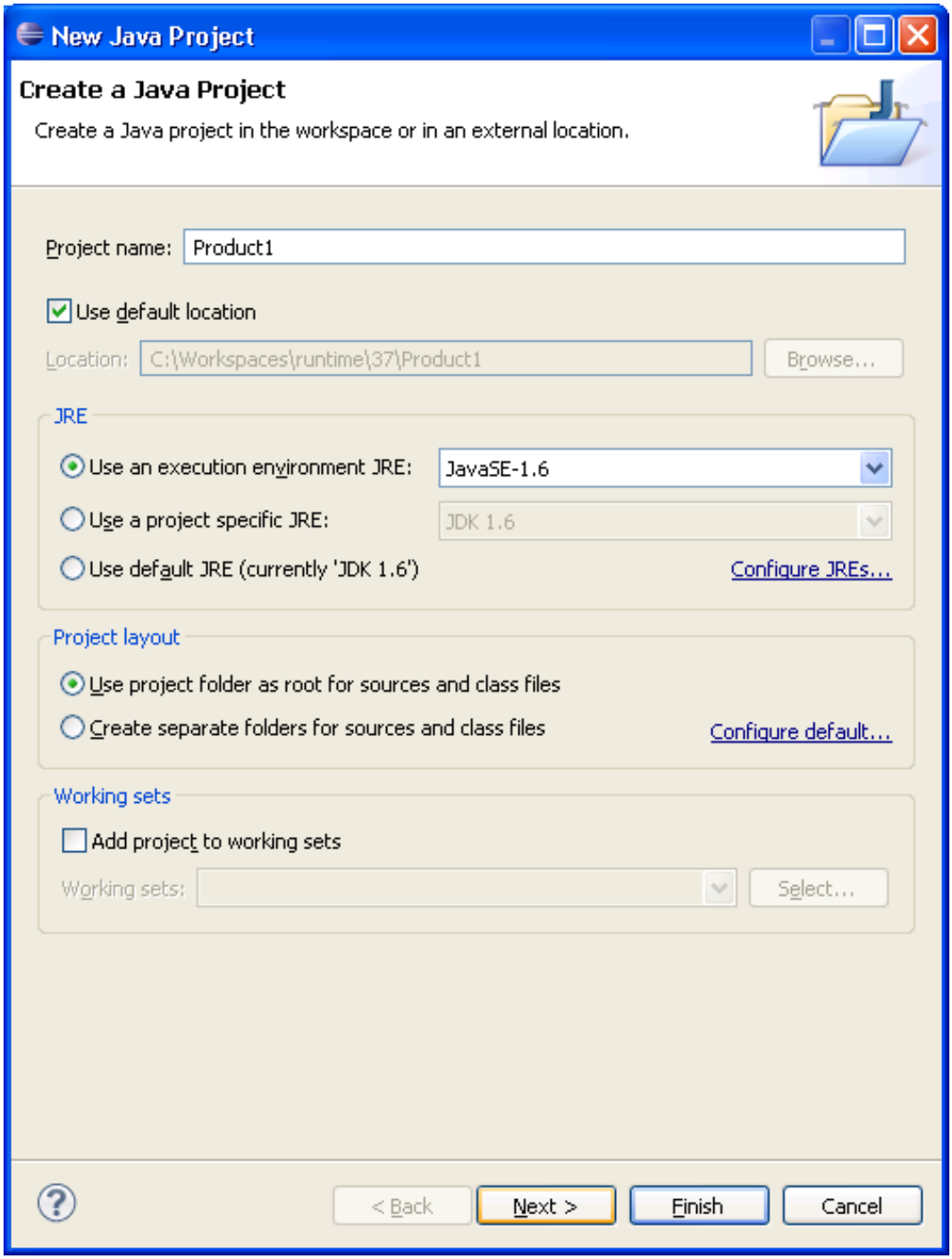
- The Java source files for a product are laid out in a package directory.
- Source files of tests are laid out in a nested package directory.



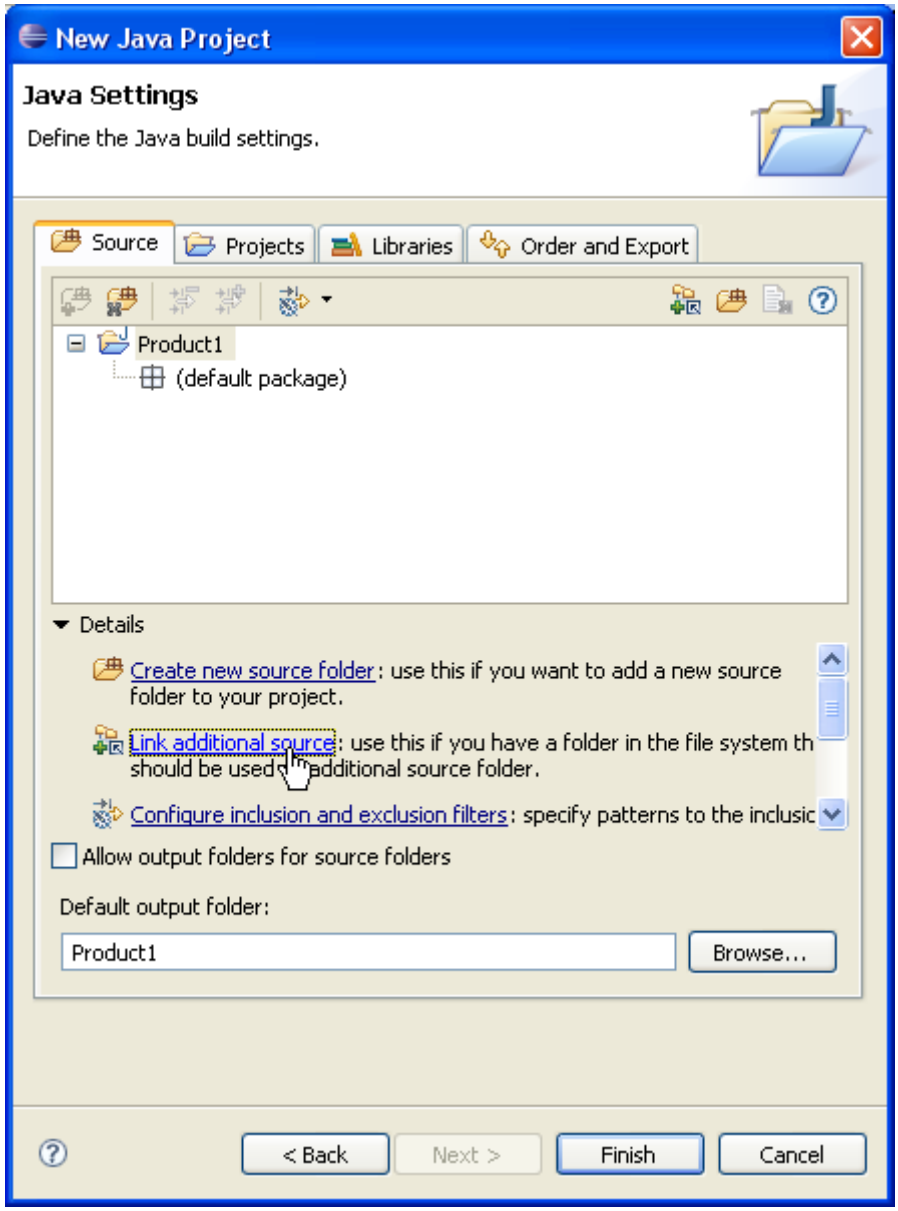
Steps for defining a corresponding project

1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.

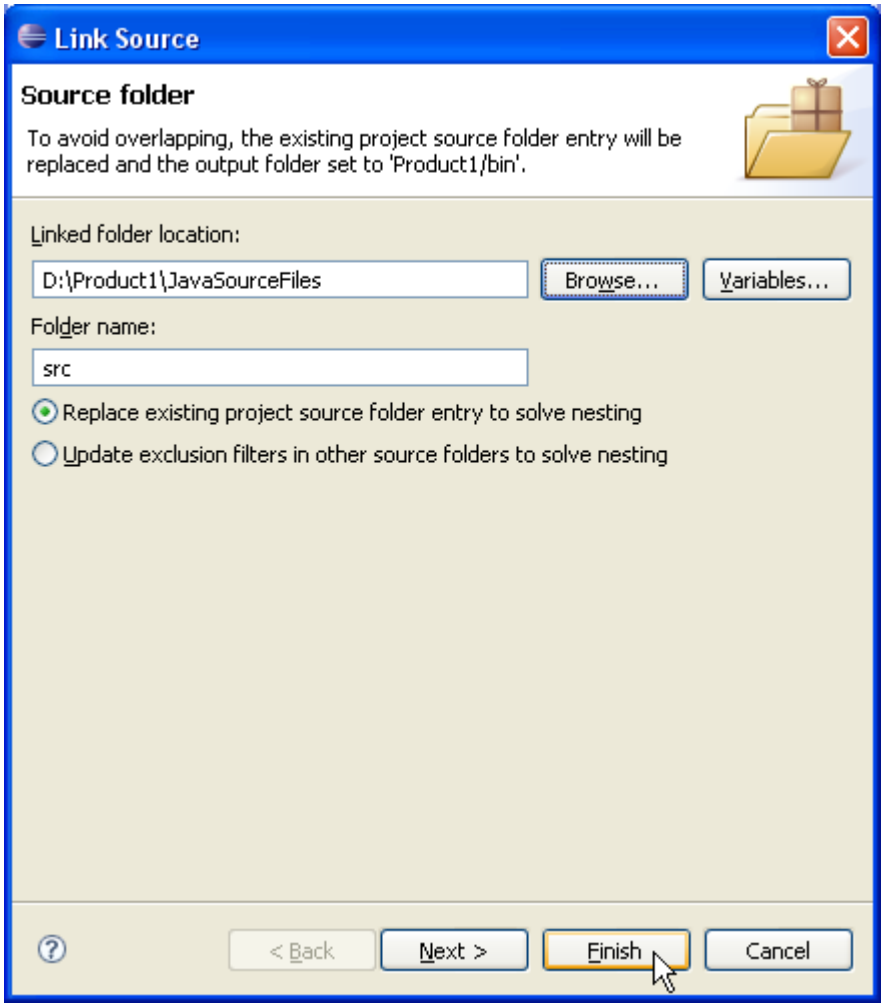
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Product1" in the **Project name** field. Click **Next**.



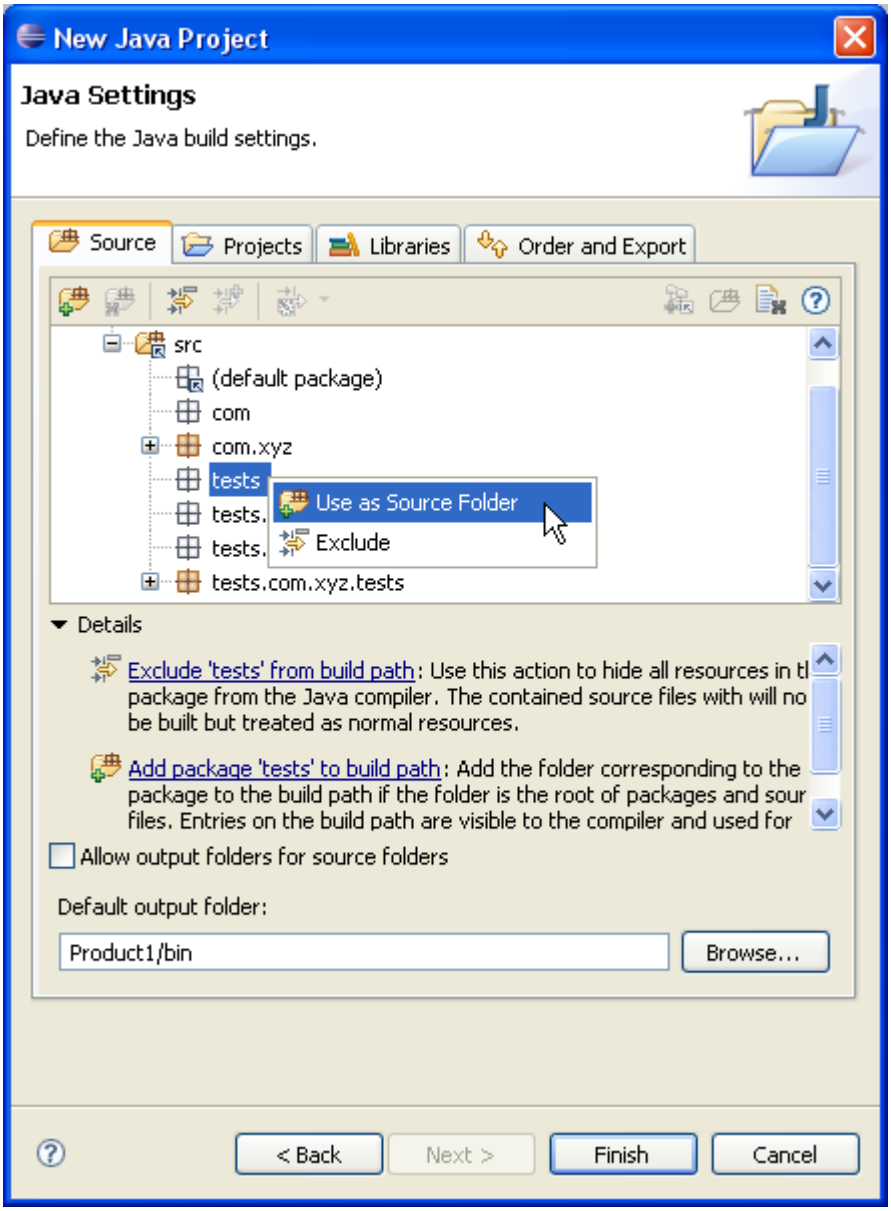
4. On the next page, Select "Product1" source folder.
Click **Link additional source** link in **Details** pane or button in view bar.



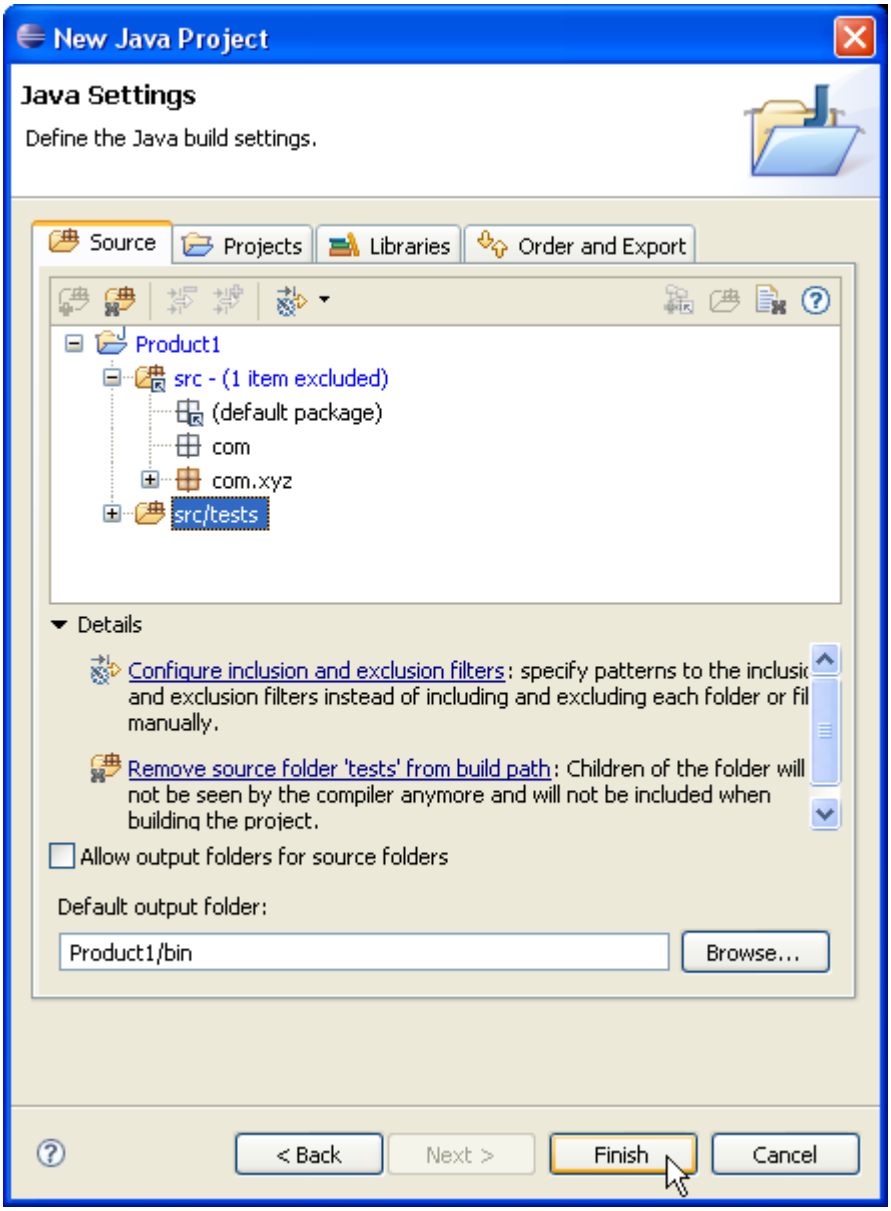
5. In **Link Source** click **Browse....** and choose the *D:\Product1\JavaSourceFiles* directory.
Type "src" in the **Folder name** field.



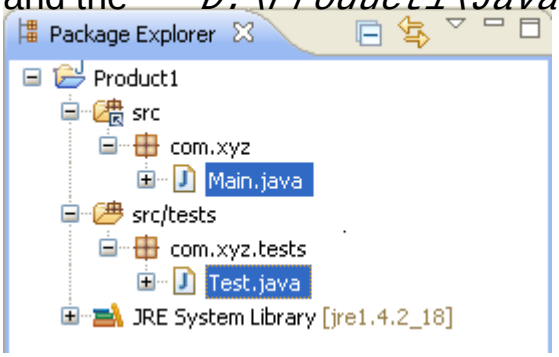
6. Click **Finish** to validate and close the dialog.
7. Expand the "src" source folder. Select the empty package "tests" and set it as source folder using either **Use as Source Folder** popup-menu item or button in view bar.



8. Your project source setup now looks as follows:



9. Click **Finish**.
10. You now have a Java project with two source folders: "src" and "src/tests" which contain respectively the *D:\Product1\JavaSourceFiles* directory and the *D:\Product1\JavaSourceFiles\tests* directory.



Next Section: Products sharing a common source framework

Related concepts

Java projects
Java views

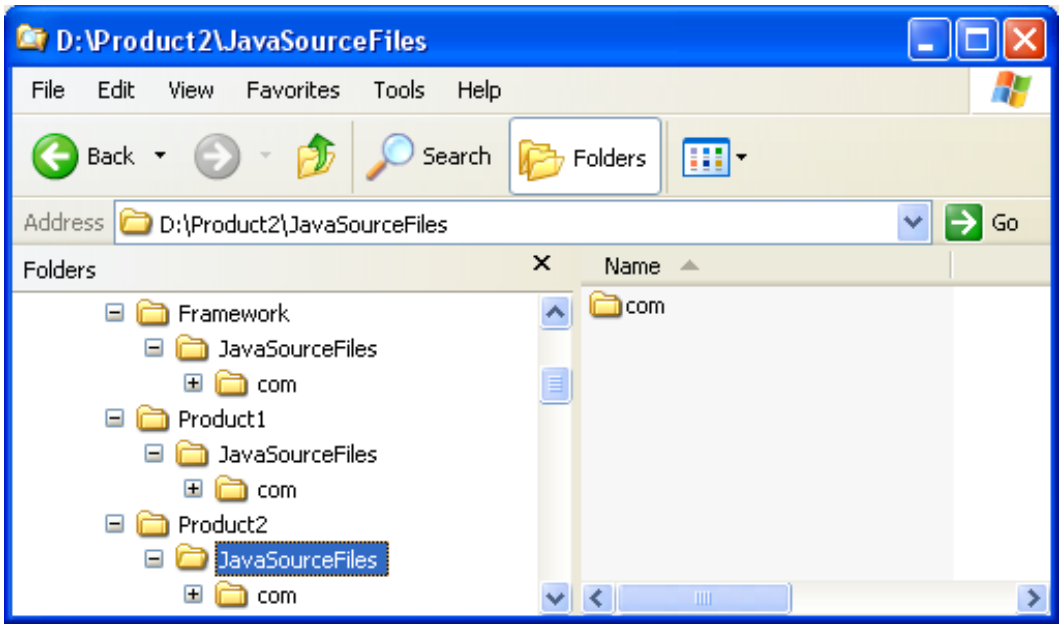
Related reference

New Java Project Wizard
Package Explorer View

Products sharing a common source framework

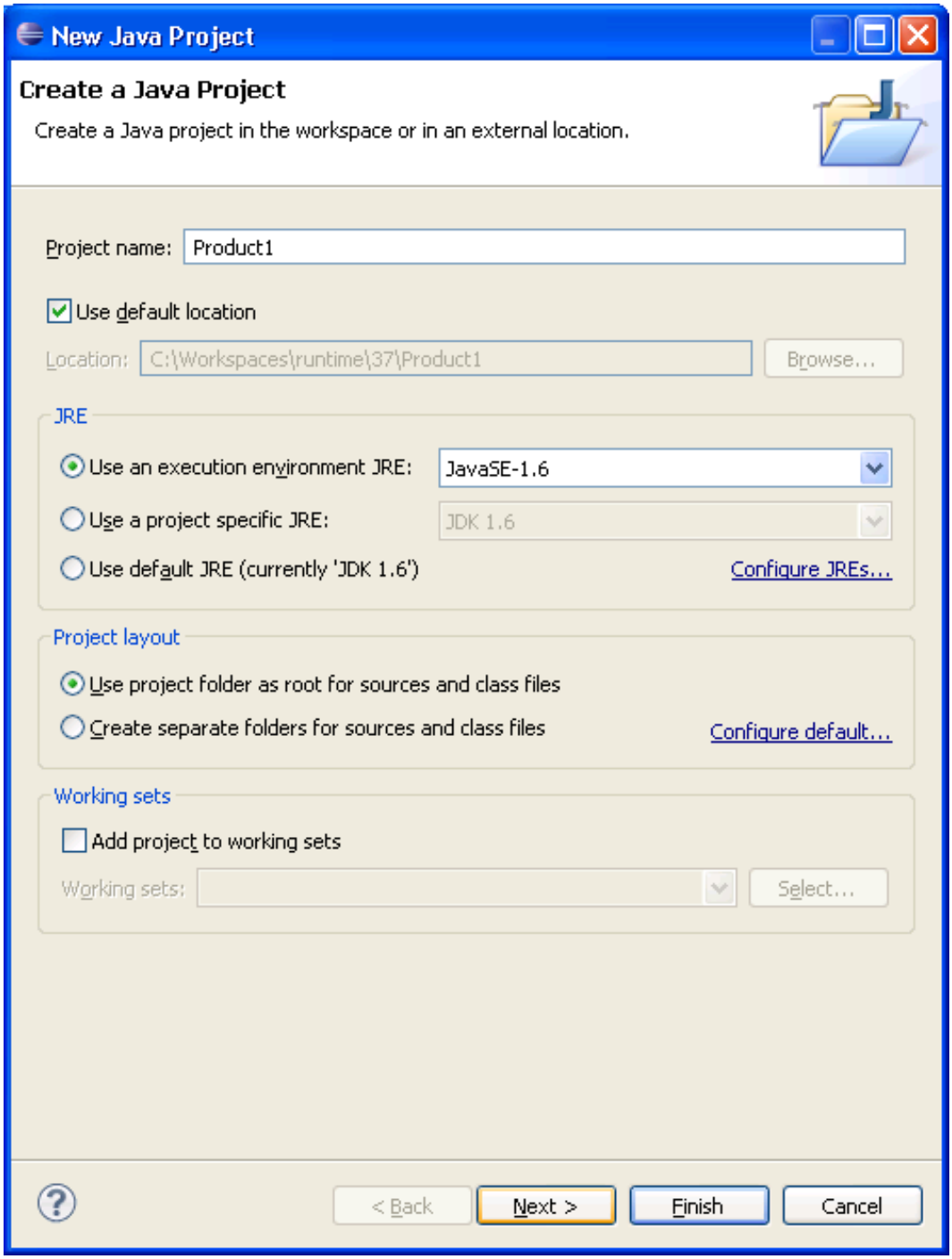
Layout on file system

- The Java source files for two products require a common framework.
- Projects and common framework are in separate directories which have their own source and output folders.

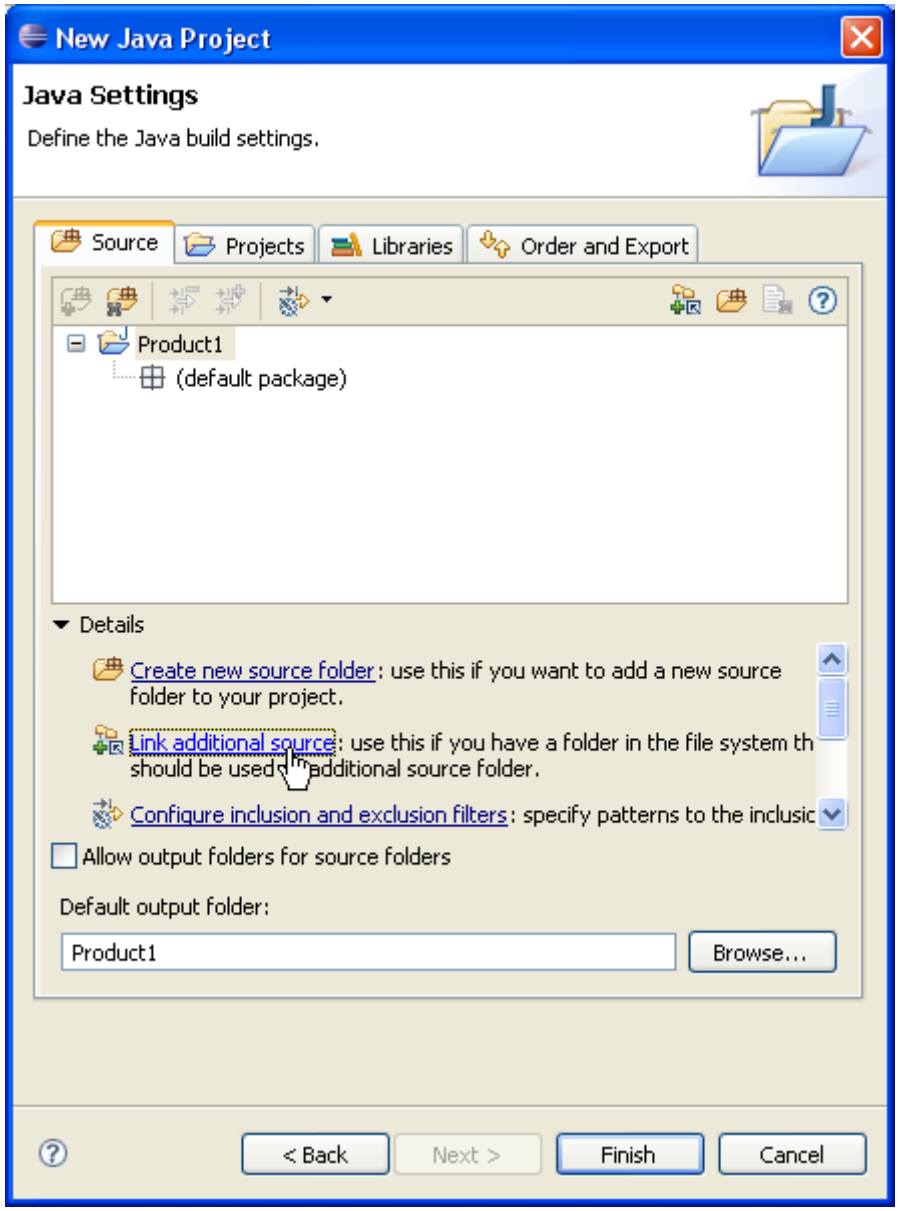


Steps for defining corresponding projects

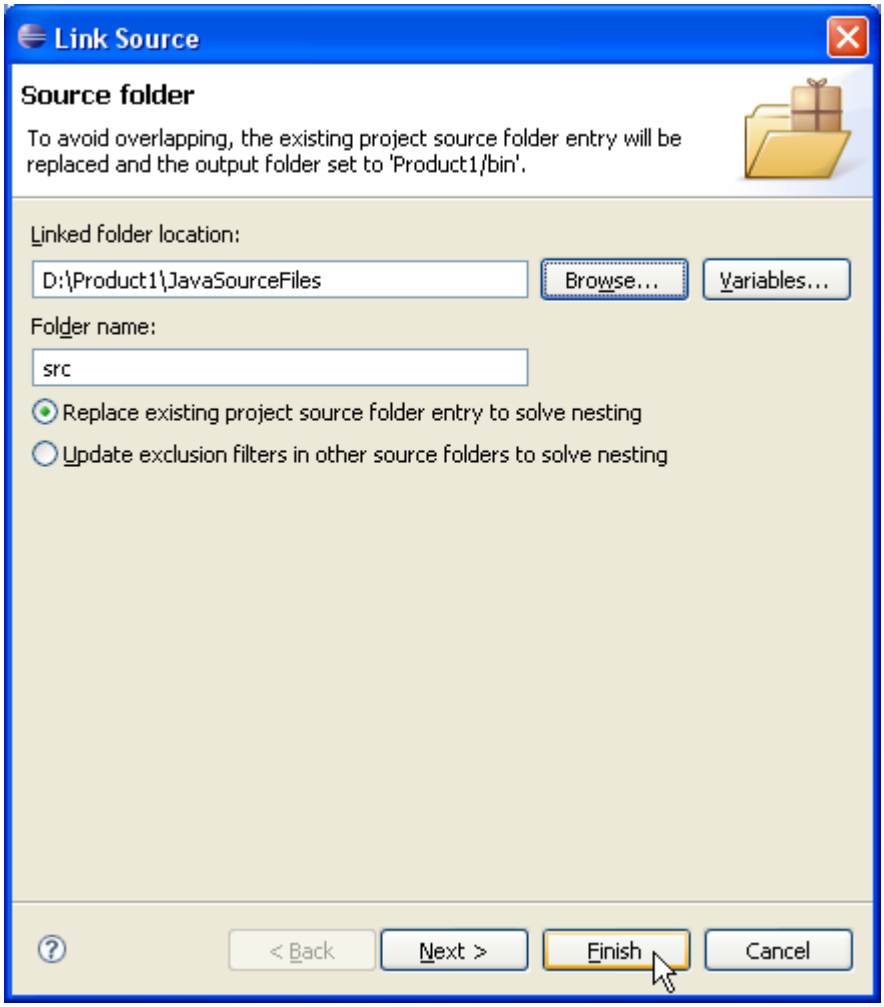
1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Product1" in the **Project name** field. Click **Next**.



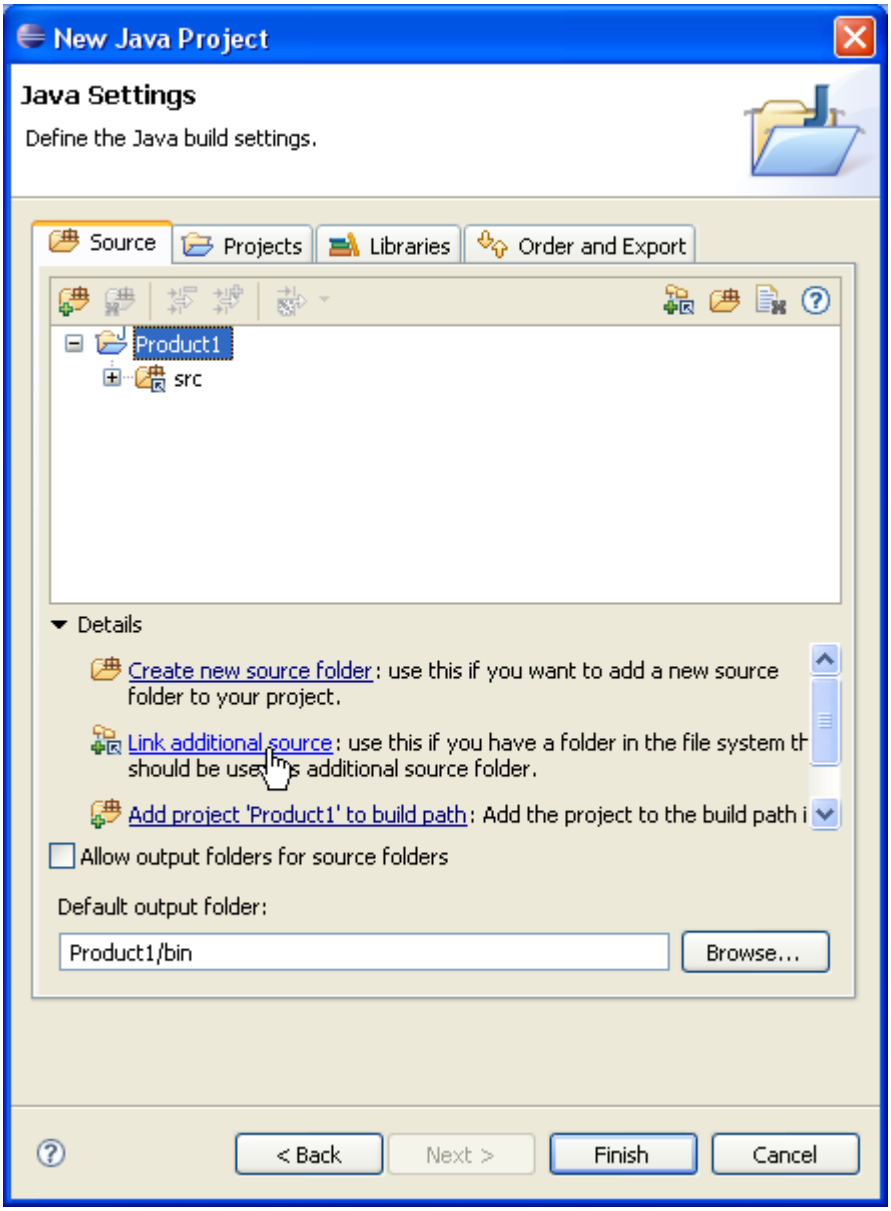
4. On the next page, Select "Product1" source folder.
Click **Link additional source** link in **Details** pane or button in view bar.



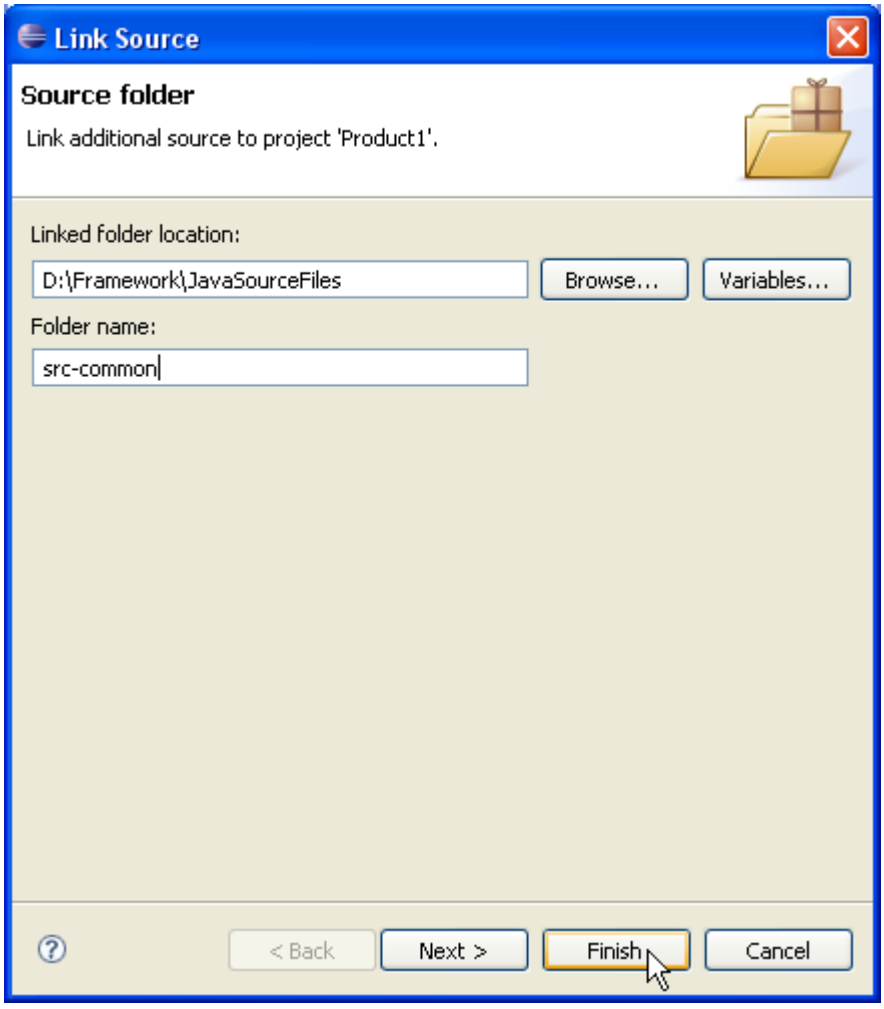
5. In **Link Source** click **Browse....** and choose the *D:\Product1\JavaSourceFiles* directory.
Type "src" in the **Folder name** field.



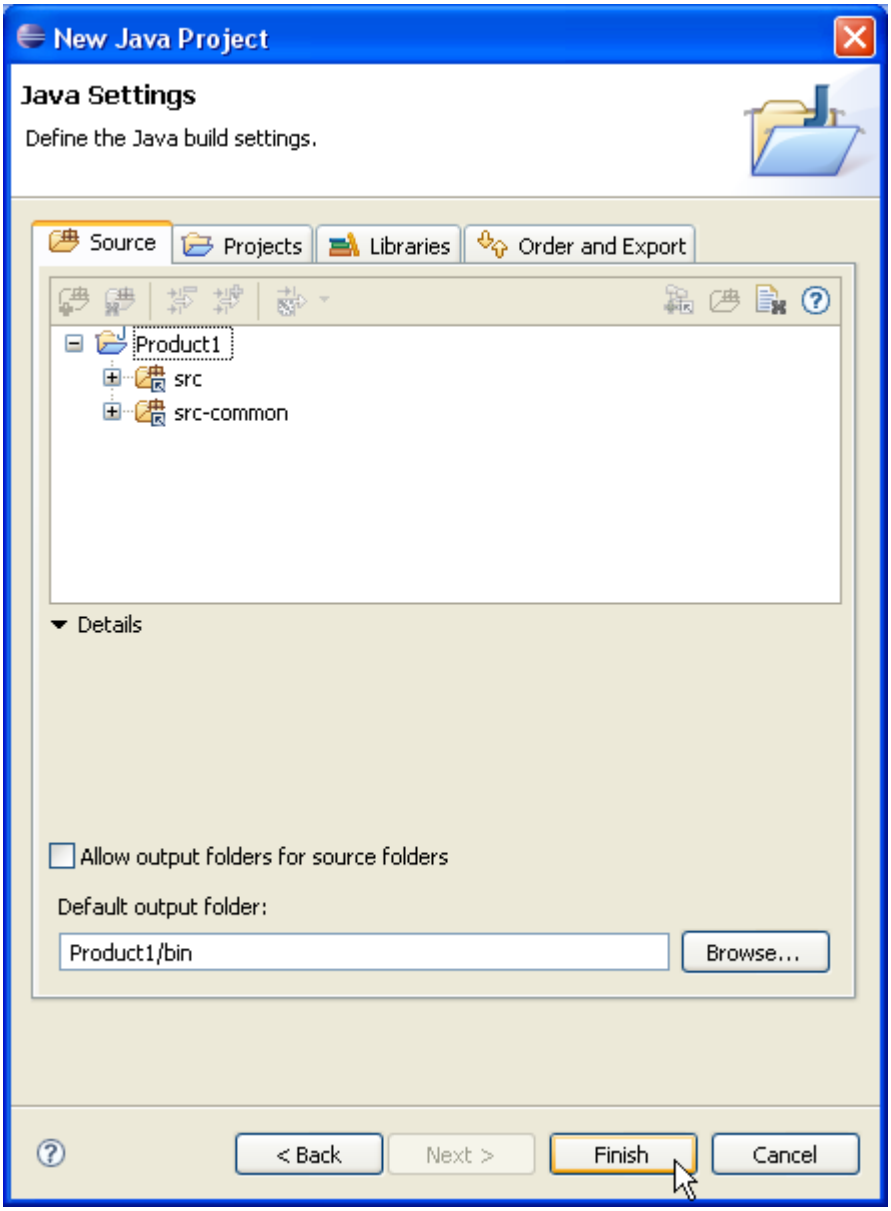
6. Click **Finish** to validate and close the dialog.
7. Again, Select "Product1" and click **Link additional source** link in **Details** pane or button in view bar.



8. In **Link Source** click **Browse....** and choose the *D:\Framework\JavaSourceFiles* directory.
Type "src-common" in the **Folder name** field.



9. Click **Finish** to validate and close the dialog.
Your project source setup now looks as follows:

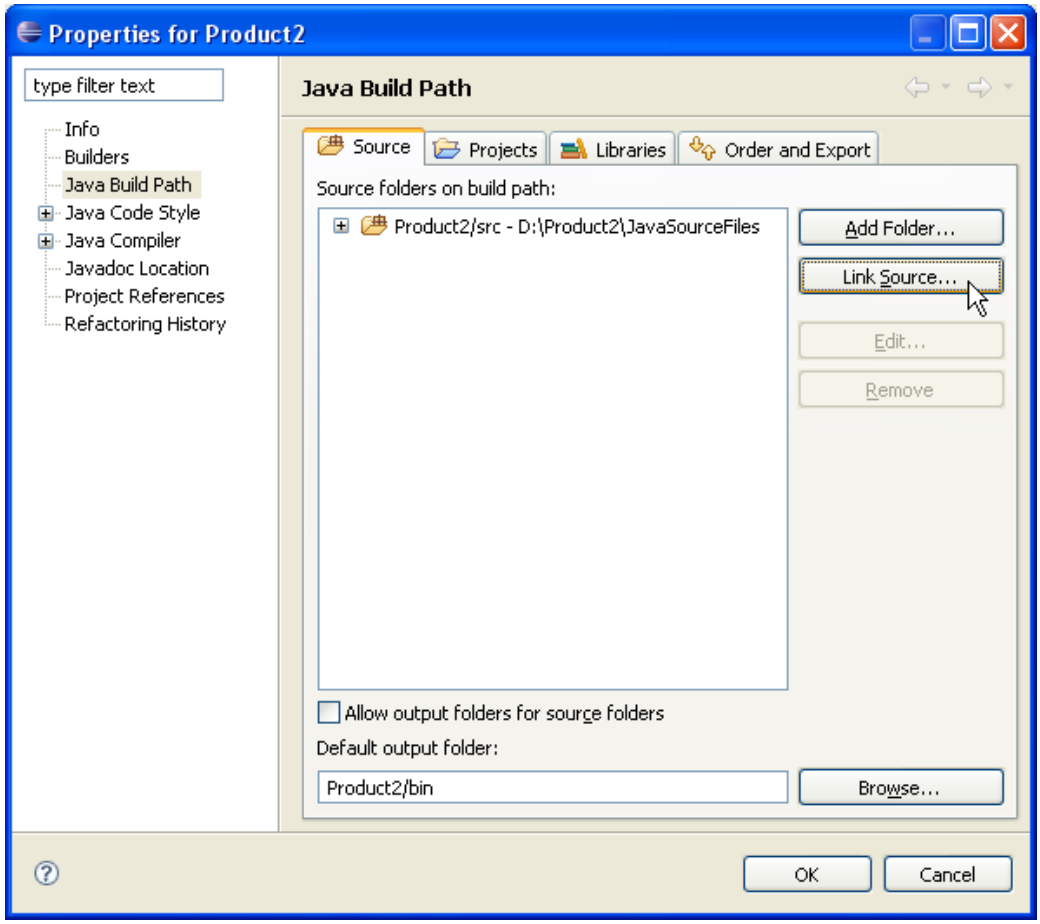


10. Click **Finish**.
11. Create project "Product2" repeating steps 2 to 5 choosing *D:\Product2\JavaSourceFiles* directory for source folder instead.

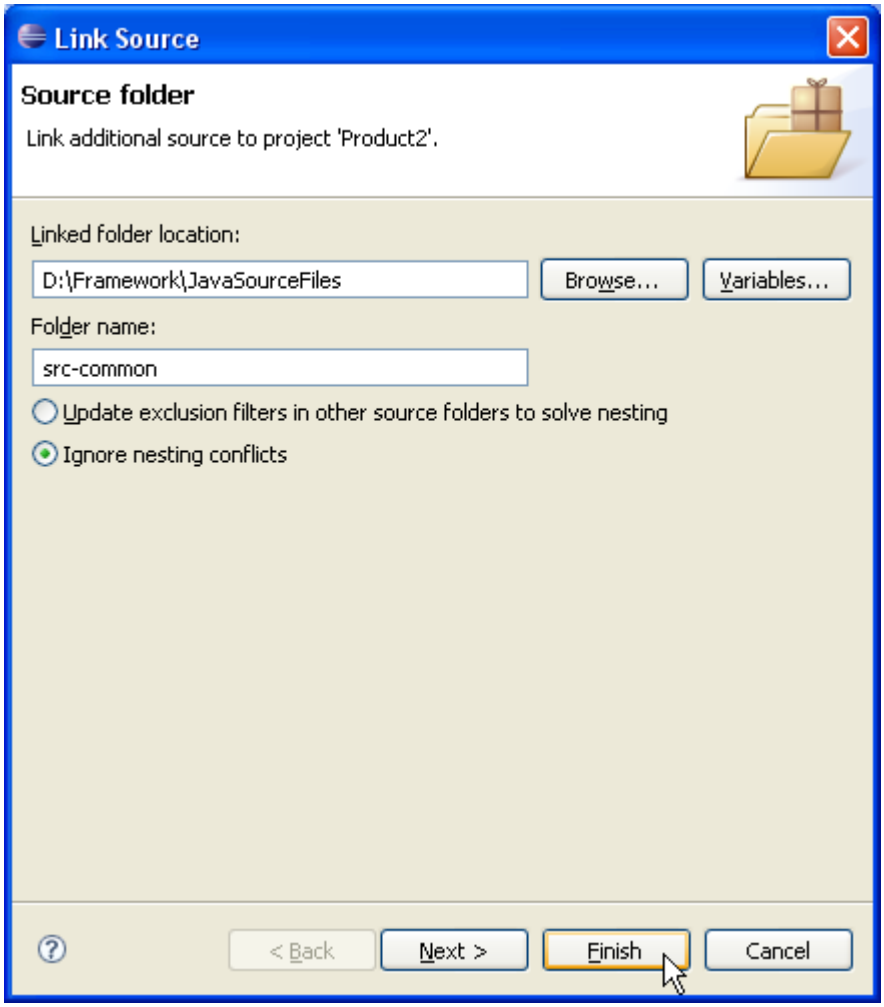
12. Click on **Finish** to create the project immediately.
13. Now, we'll see how to add a linked source folder when project is already created in workspace...

    Edit project "Product2" properties and select **Java Builder Path** page.

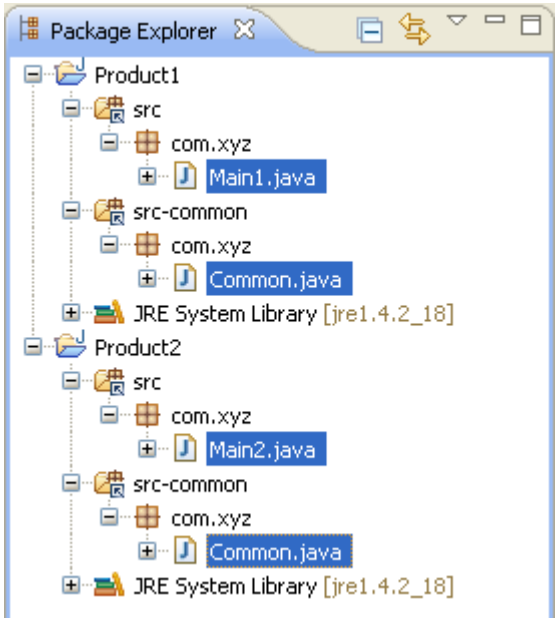
    On **Source** tab, click **Link Source...**.



14. In **Link Source** click **Browse....** and choose the *D:\Framework\JavaSourceFiles* directory.
Type "src-common" in the **Folder name** field.



15. Click **Finish** to validate and close the dialog.
    Click **OK** to apply project "Product2" properties changes.
16. You now have two Java projects which respectively contain the sources of "Product1" and "Product2" and which are using the sources of "Framework".



Note: Files in "src-common" are shared. So editing "Common.java" in "Product1" will modify "Common.java" in "Product2". However they are compiled in the

context of their respective projects. Two "Common.class" files will be generated; one for each project. If the two projects have different compiler options, then different errors could be reported on each "Common.java" file. **Next Section: Product nesting resources in output directory**

Related concepts

Java projects
Java views

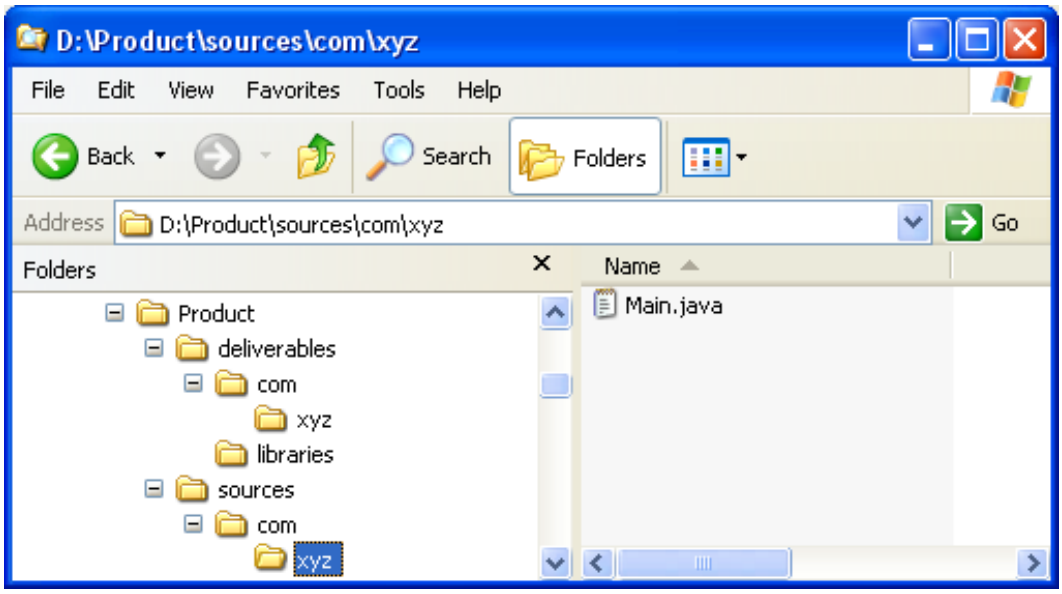
Related reference

New Java Project Wizard
Package Explorer View

Nesting resources in output directory

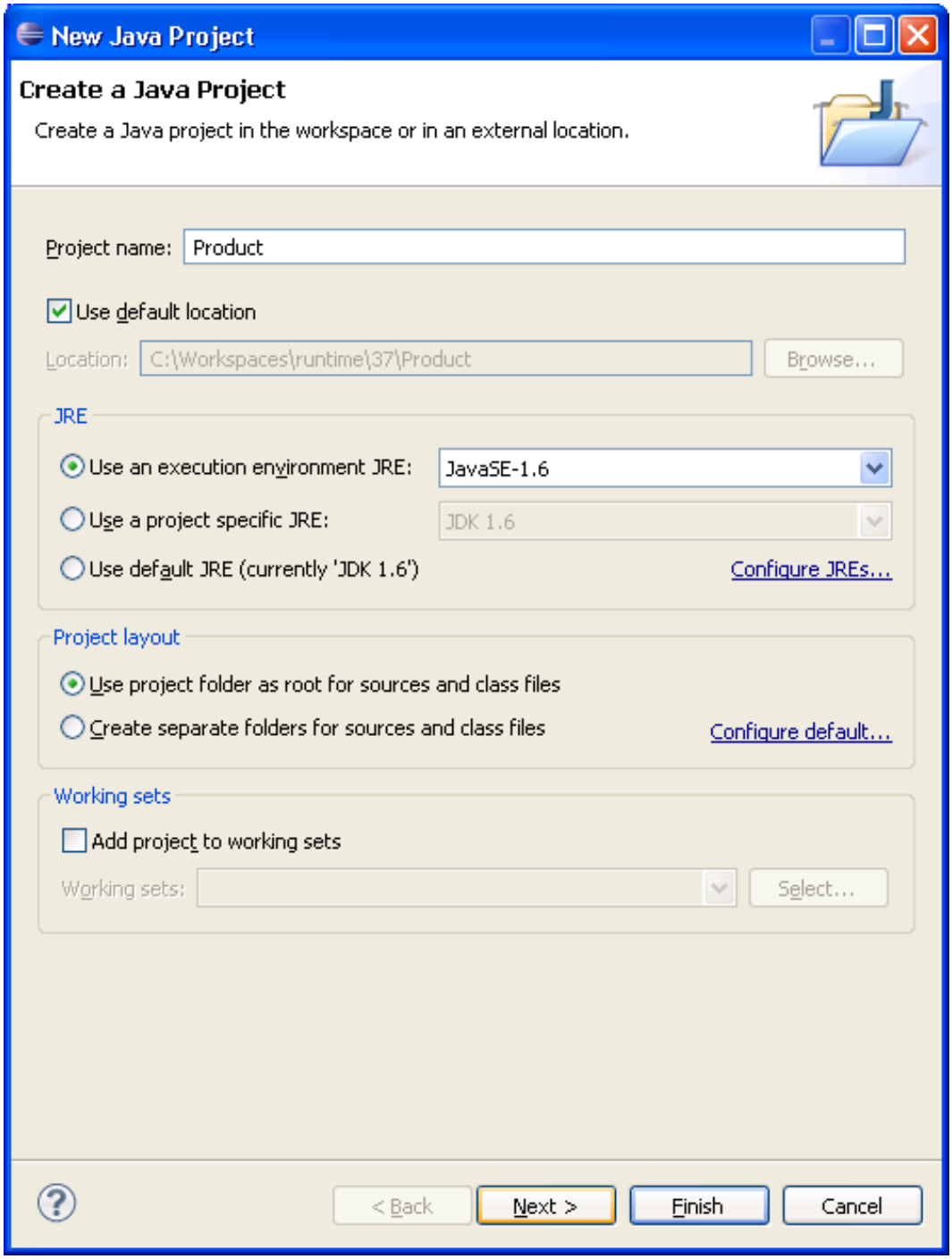
Layout on file system

- The Java source files for a product are laid out both in *sources* and *deliverables* directories.
- All Java class files are laid out in *deliverables* directory.
- Project needs to use some libraries located in *deliverables/libraries* directory:

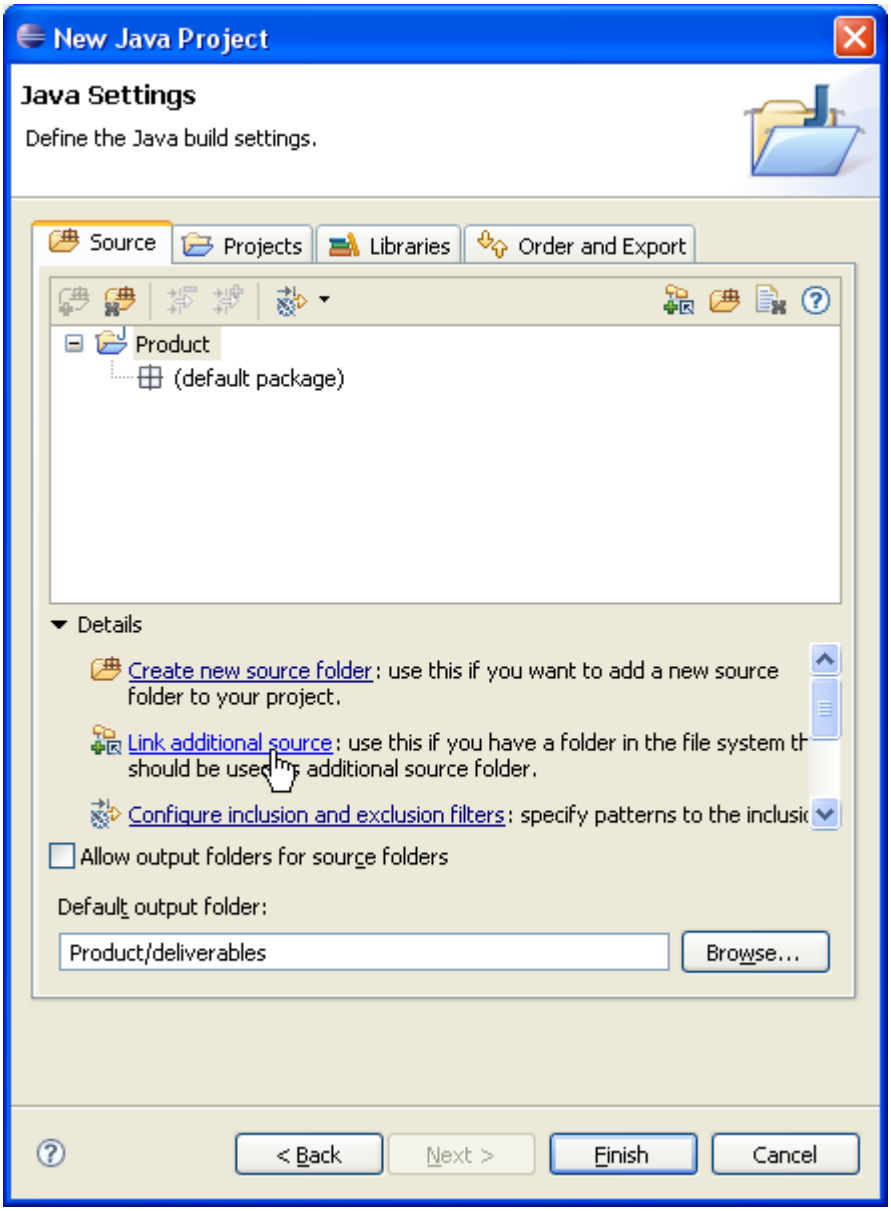


Steps for defining a corresponding project

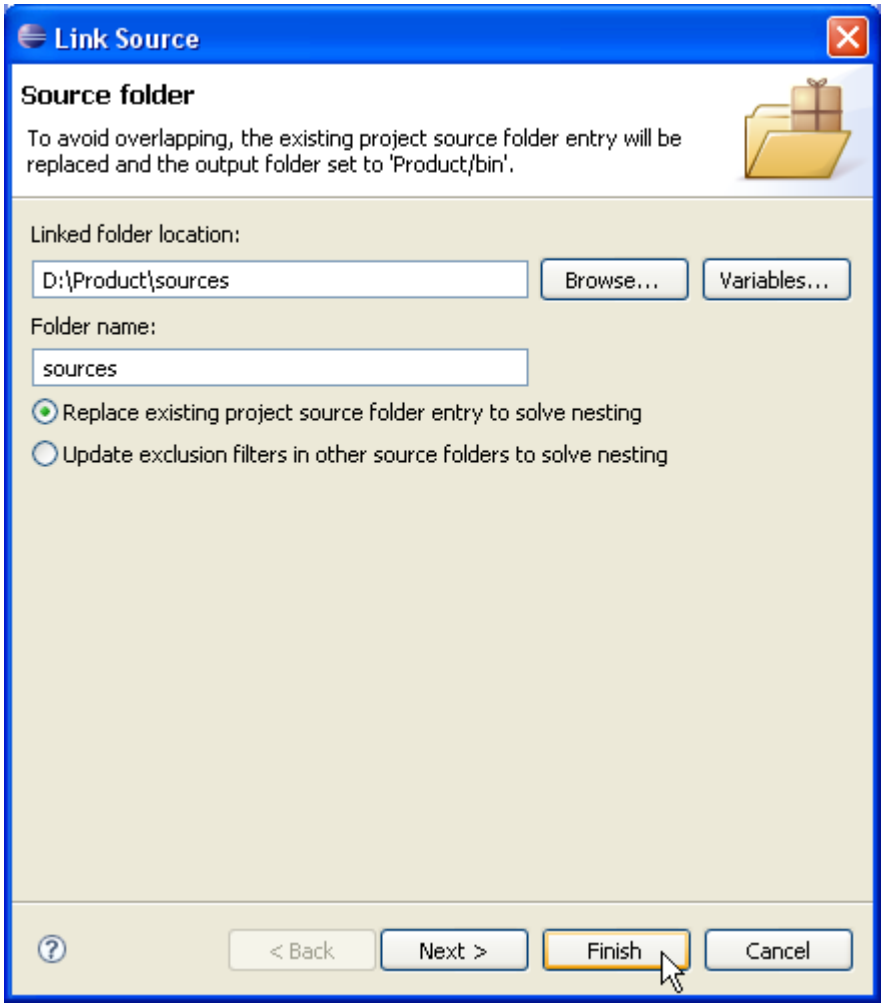
1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Product" in the **Project name** field. Click **Next**.



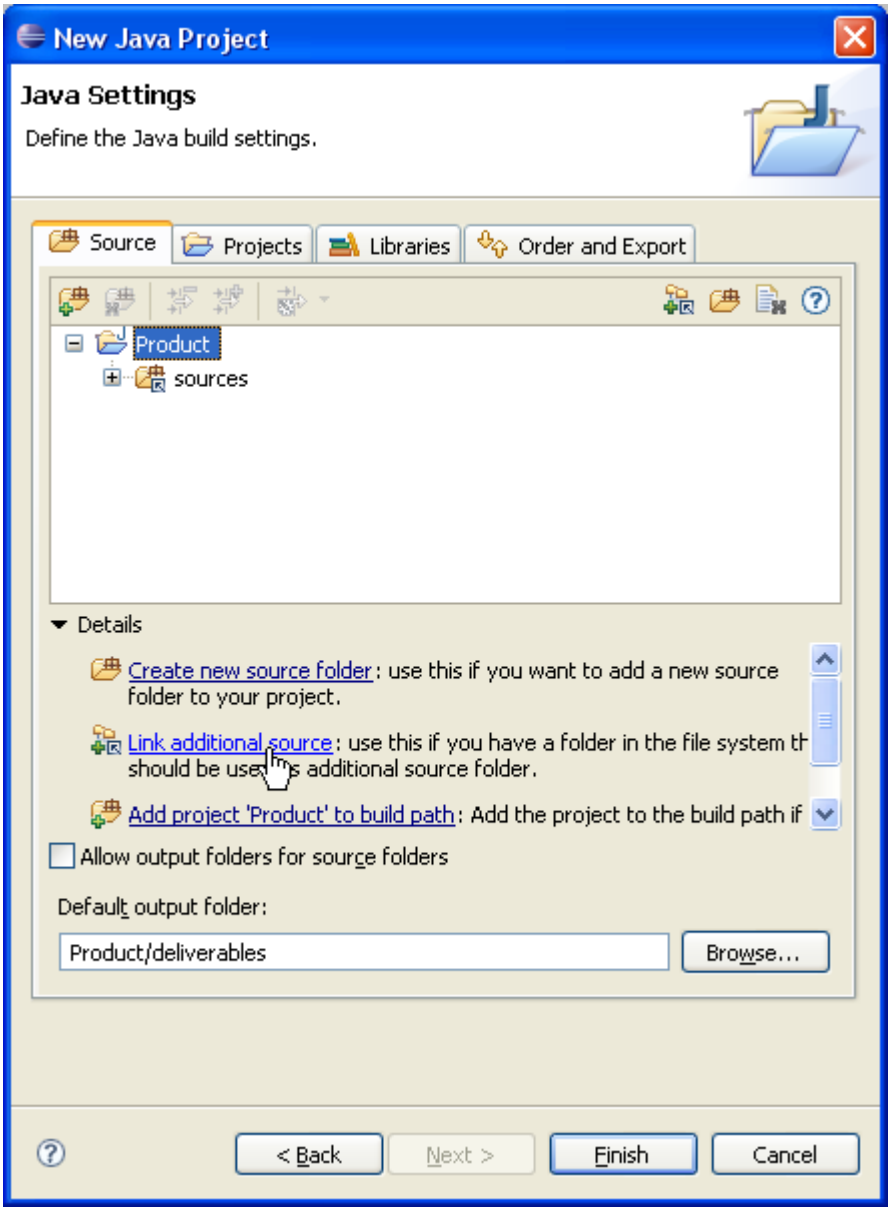
4. On the next page, Type "Product/deliverables" in **Default output folder** field.
   Select "Product" source folder.
   Click **Link additional source** link in **Details** pane or button in view bar.



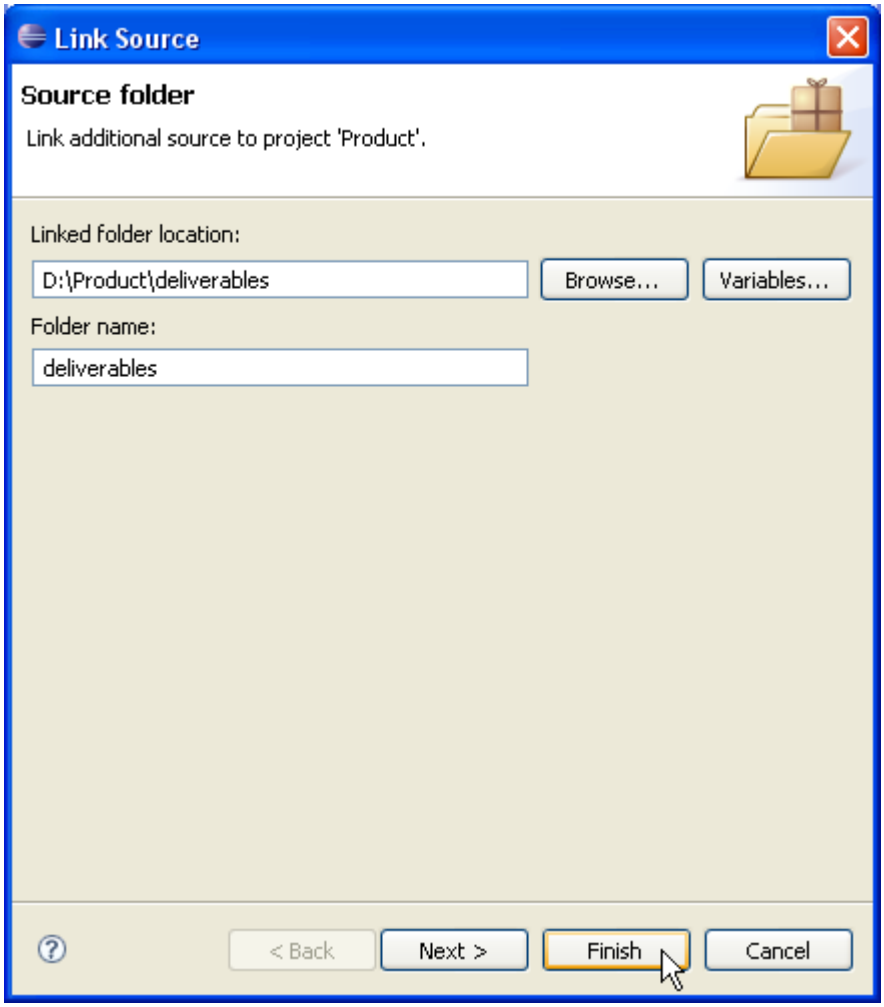
5. In **Link Source** click **Browse....** and choose the *D:\Product\sources* directory.



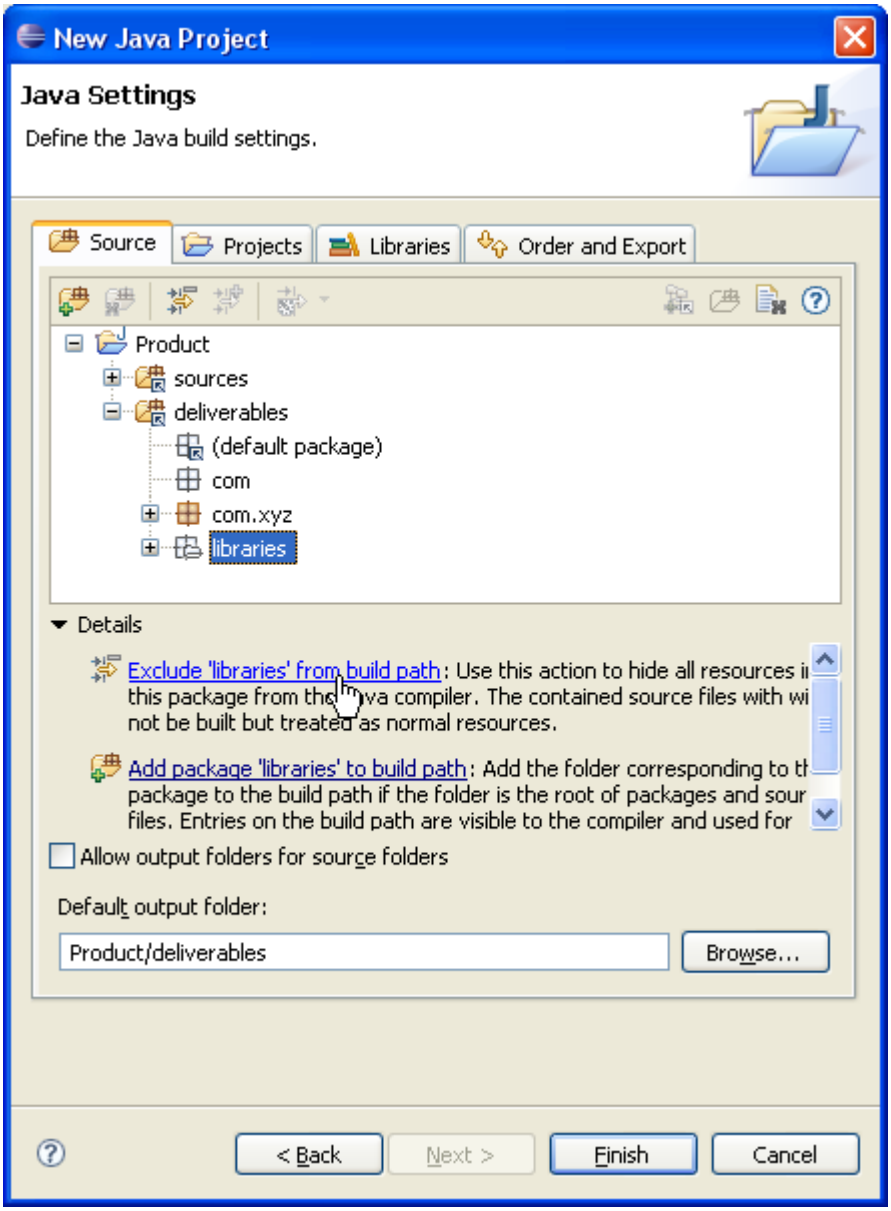
6. Click **Finish** to validate and close the dialog.
7. Again, Select "Product" and click **Link additional source** link in **Details** pane or button in view bar.



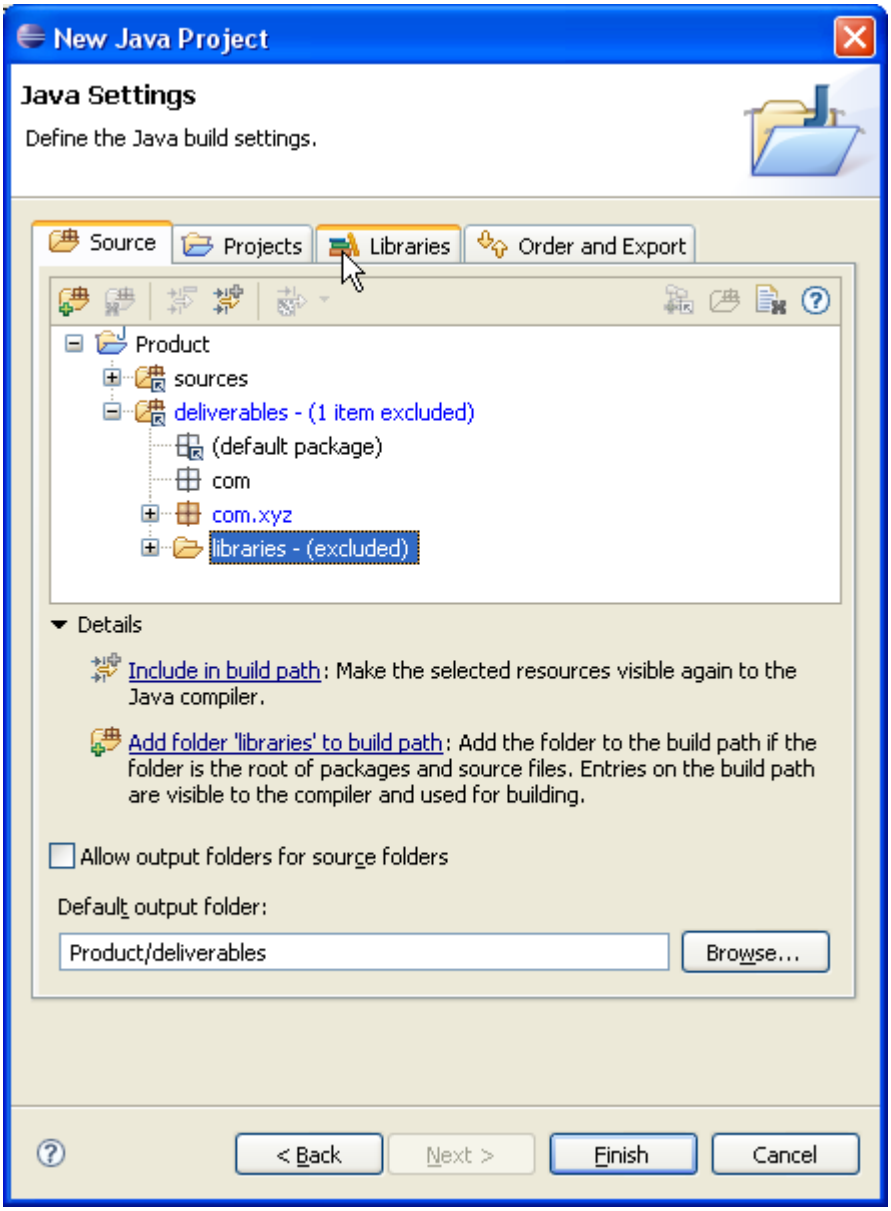
8. In **Link Source** click **Browse....** and choose the *D:\Product\deliverables* directory.



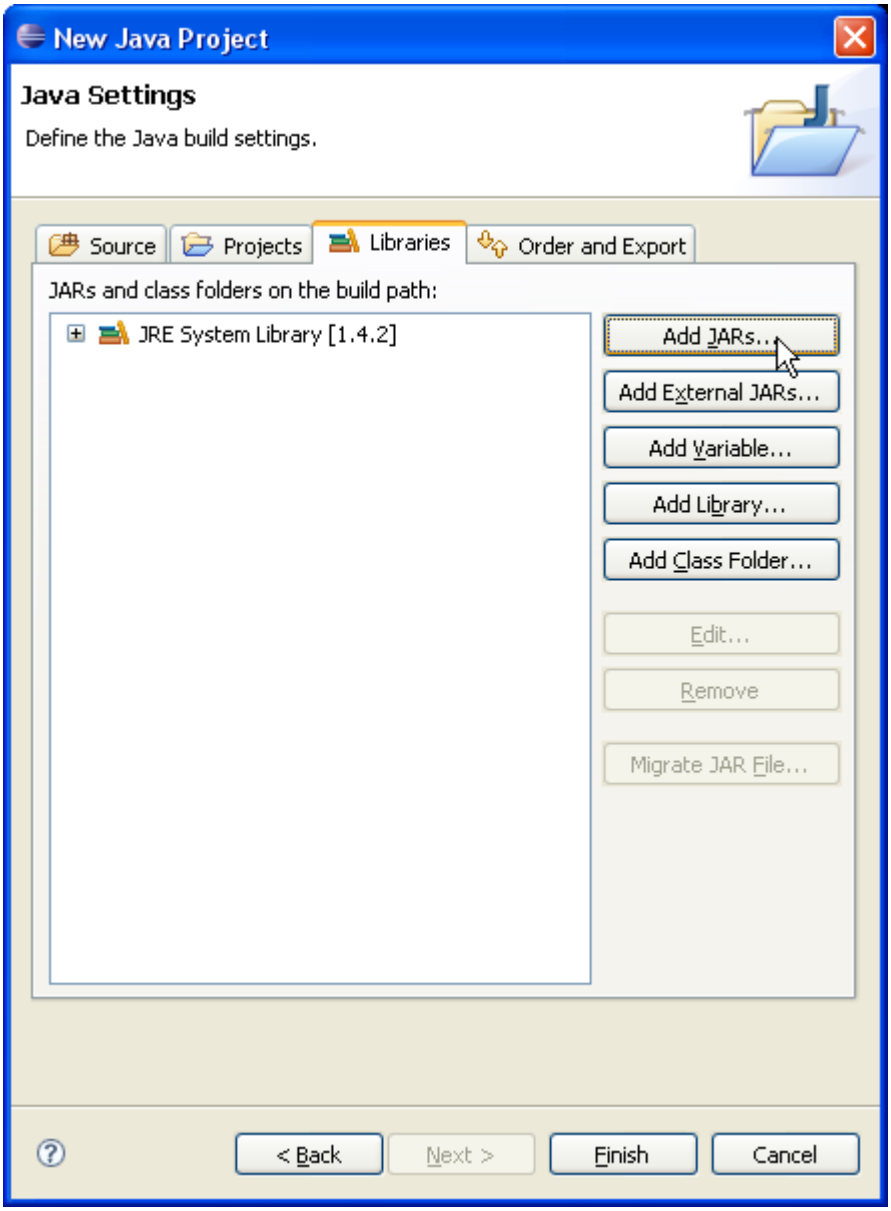
9. Expand the "Product/deliverables" source folder. Select the "libraries" package and exclude it from build path using either **Exclude 'libraries' from build path** link or **Exclude** popup-menu item.



10. Your project source setup now looks as follows:

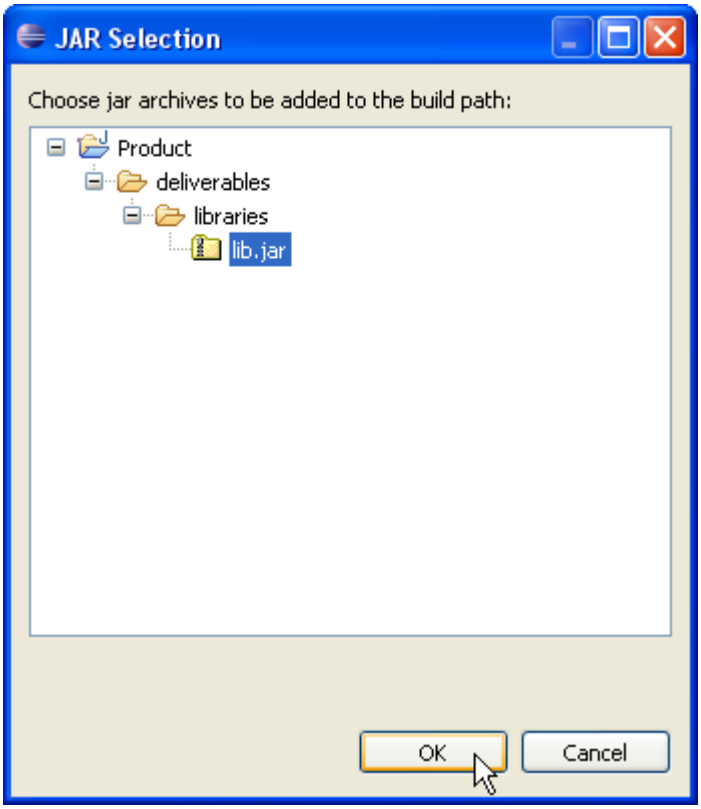


11. Select **Libraries** tab.
Click on **Add JARs...**.

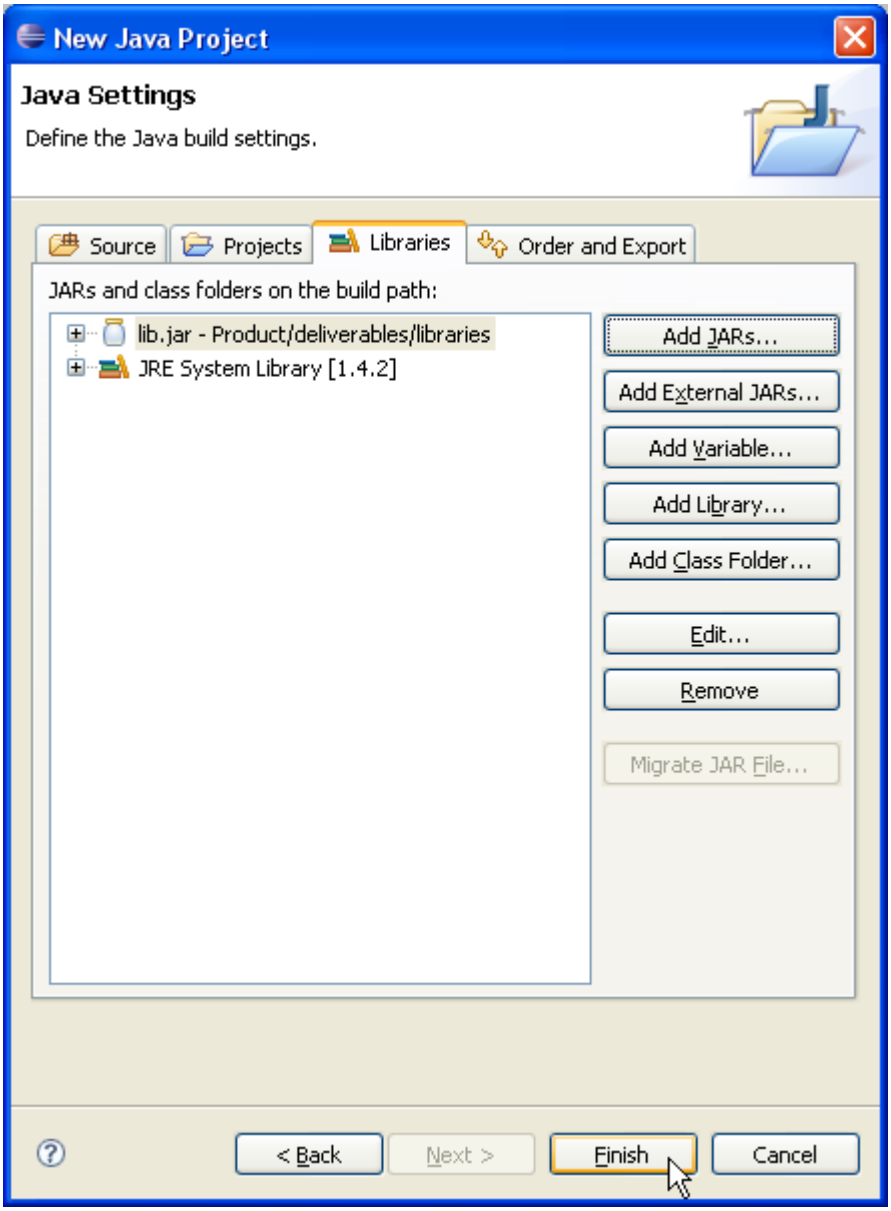


12. Expand "Product" hierarchy to select jar files in "libraries" directory

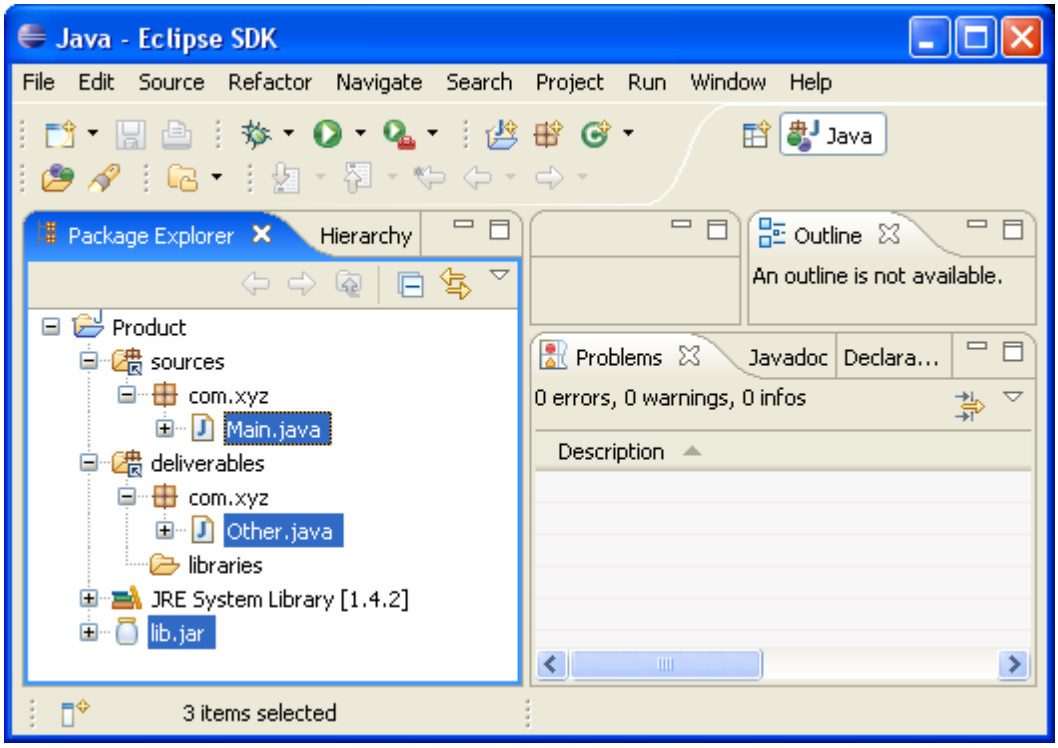
Click **OK**.



13. Click **Finish** to finalize project creation.



14. You now have a Java project with a "sources" folder and an output folder which contains nested library resources.



Next Section: Project using a source framework with restricted access

Related concepts

Java projects
Java views

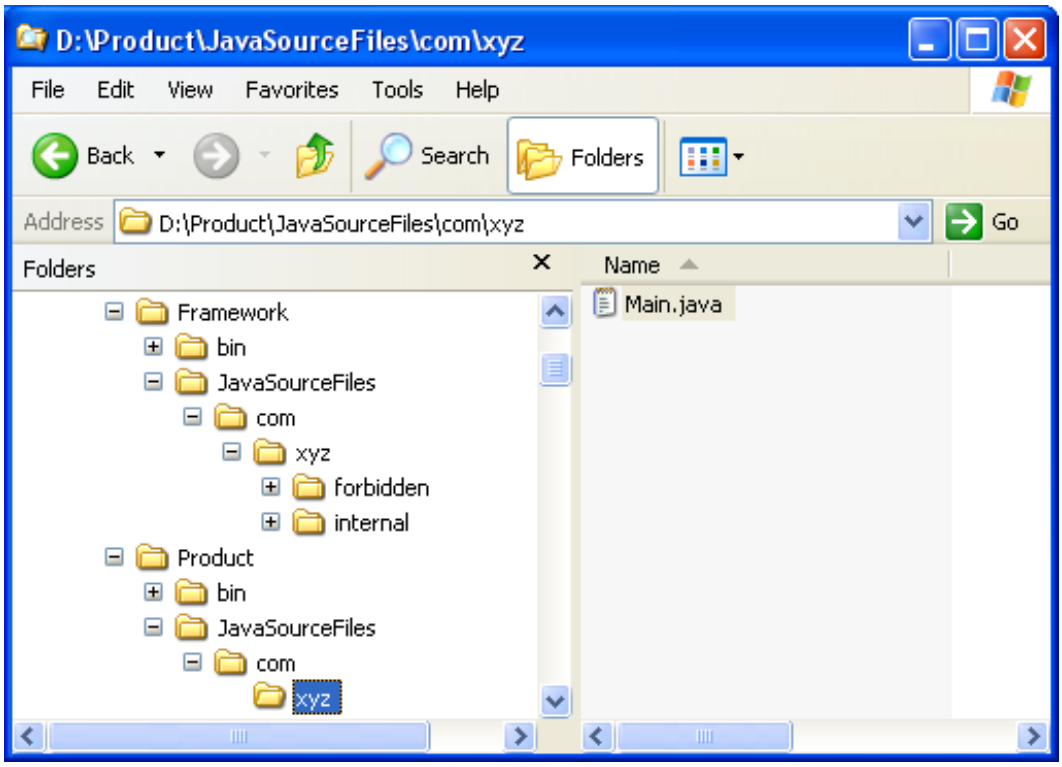
Related reference

New Java Project Wizard
Package Explorer View

Project using a source framework with restricted access

Layout on file system

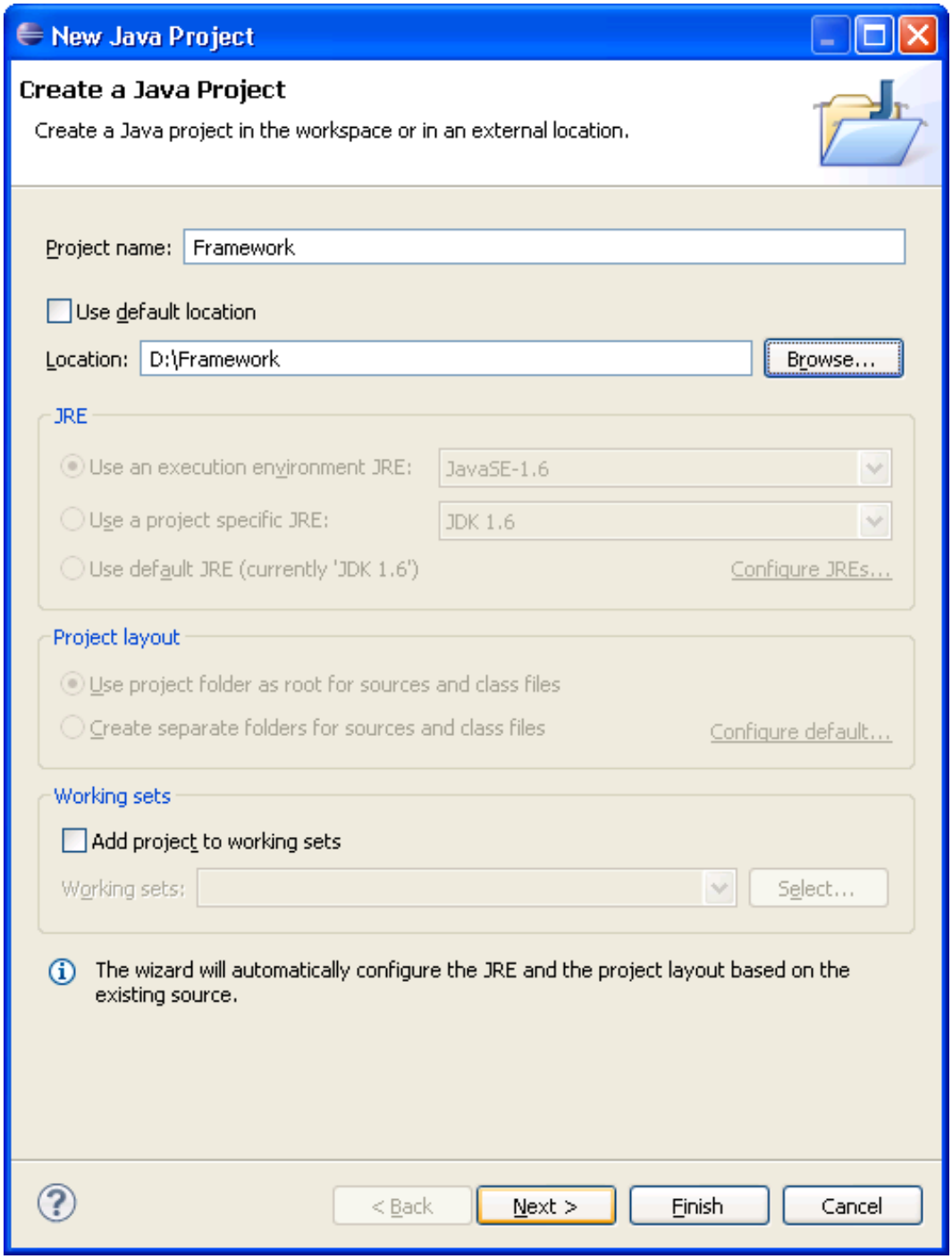
- The Java source files for a product requires a source framework.
- "Product" and "Framework" are in separate directories which have their own source and output folders.



Steps for defining corresponding projects

1. Click [Image: /topic/org.eclipse.help/command_link.png]**Window > Open Perspective > Other... > Java** to change to the Java perspective.

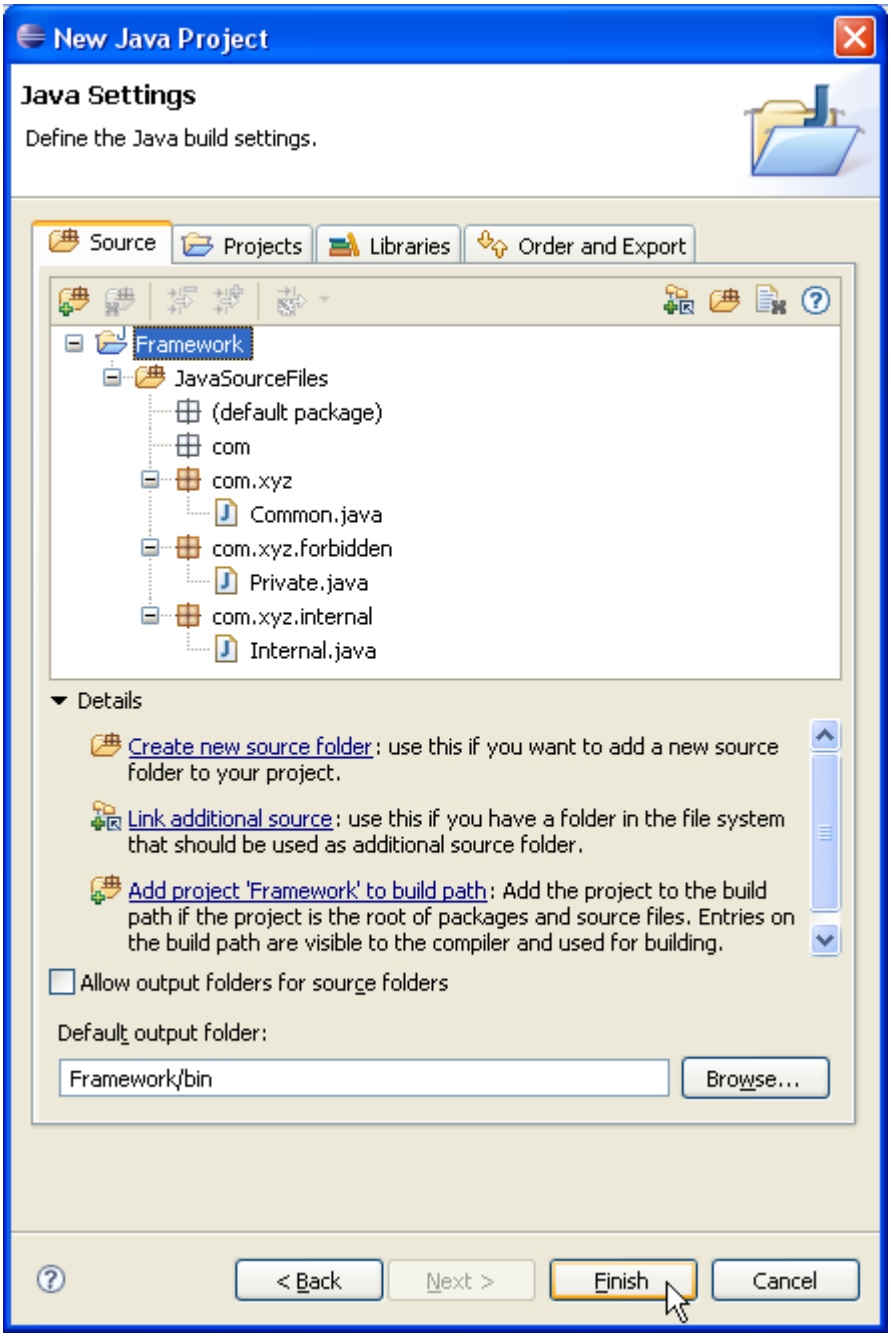
2. Click [Image: /topic/org.eclipse.help/command_link.png]**File > New > Other... > Java Project** to open the **New Java Project** wizard.
3. Type "Framework" in the **Project name** field.
4. Deselect **Use default location**.
   Click **Browse...** and choose the *D:\Framework* directory.



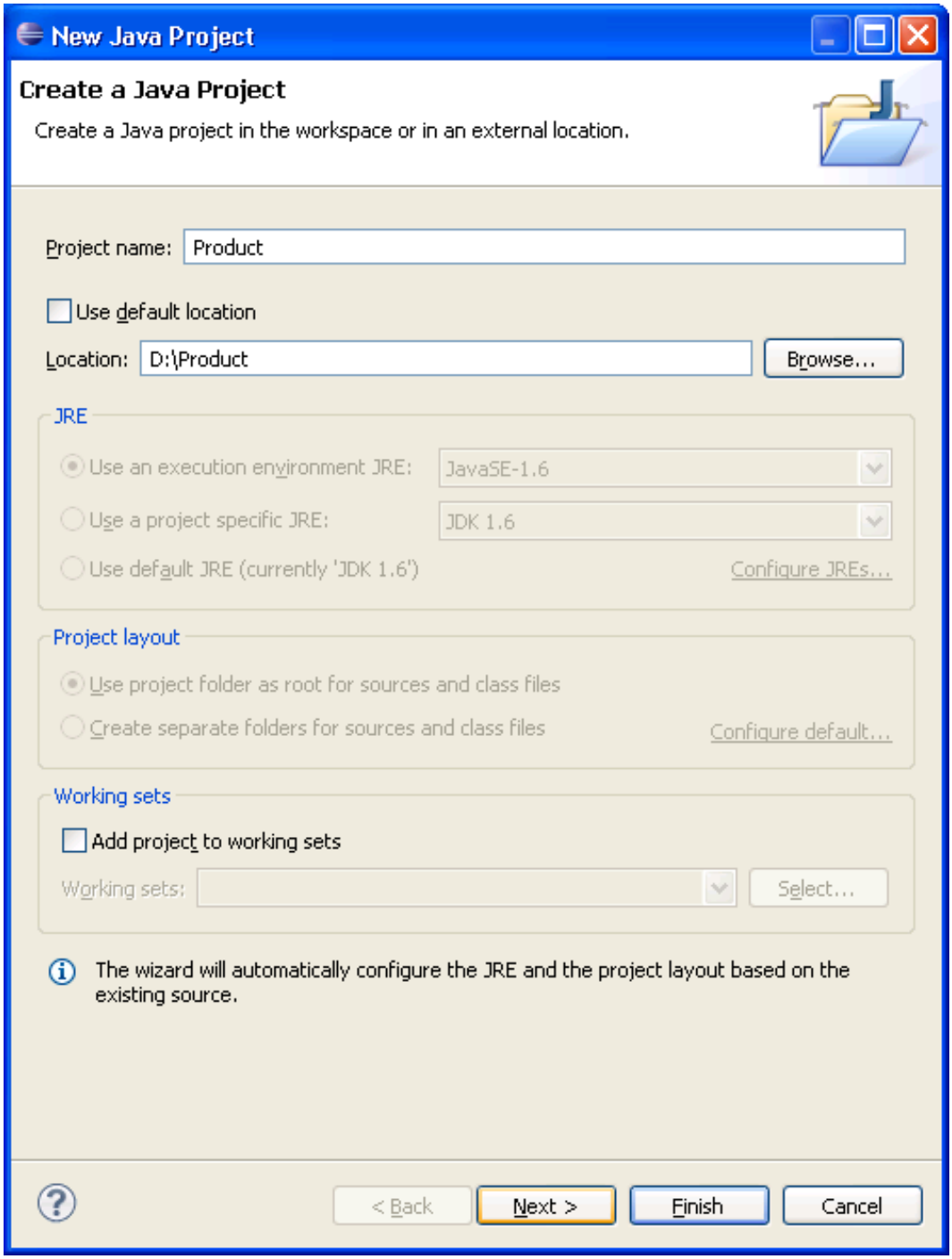
Click **Next**.

5. On the next page, verify that directory *JavaSourceFiles* has been automatically added as source folder.

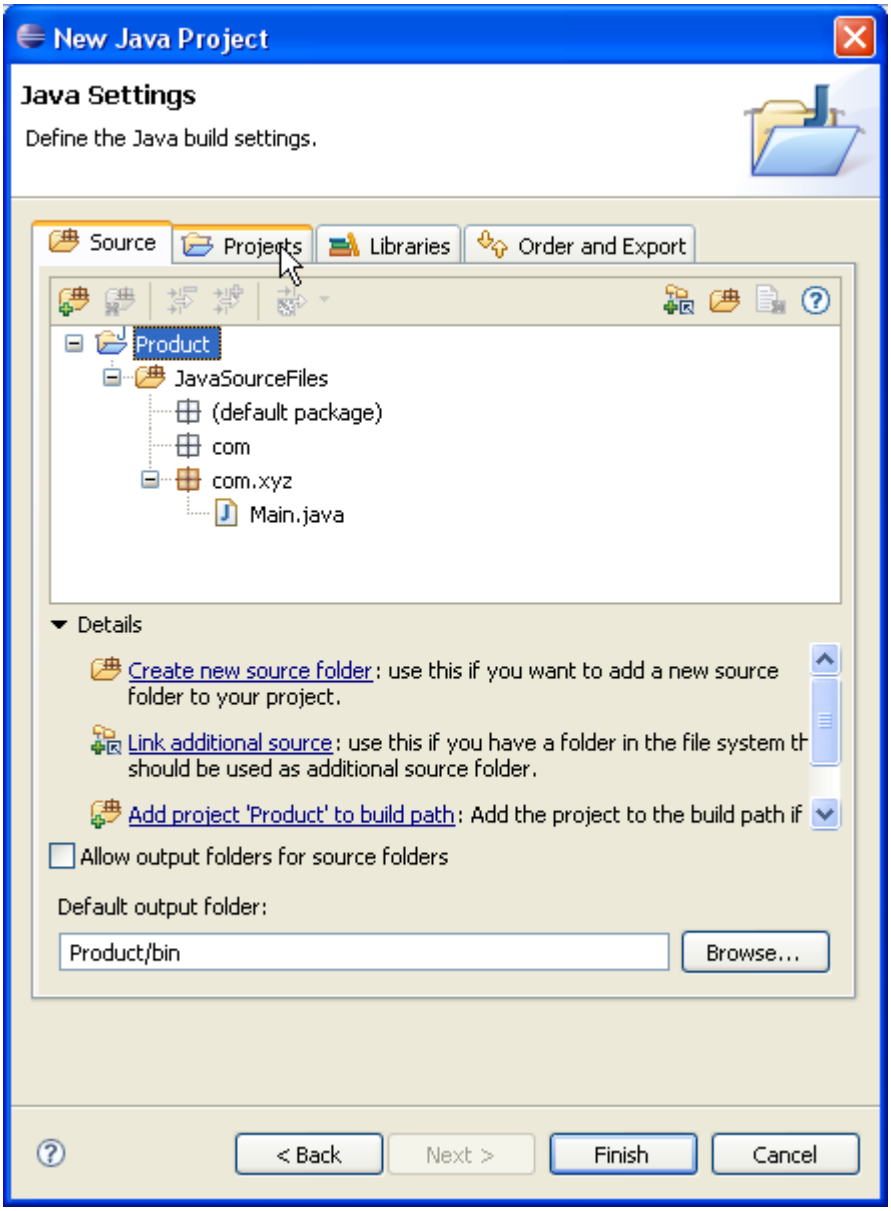
Expand it to preview your project source folder contents:



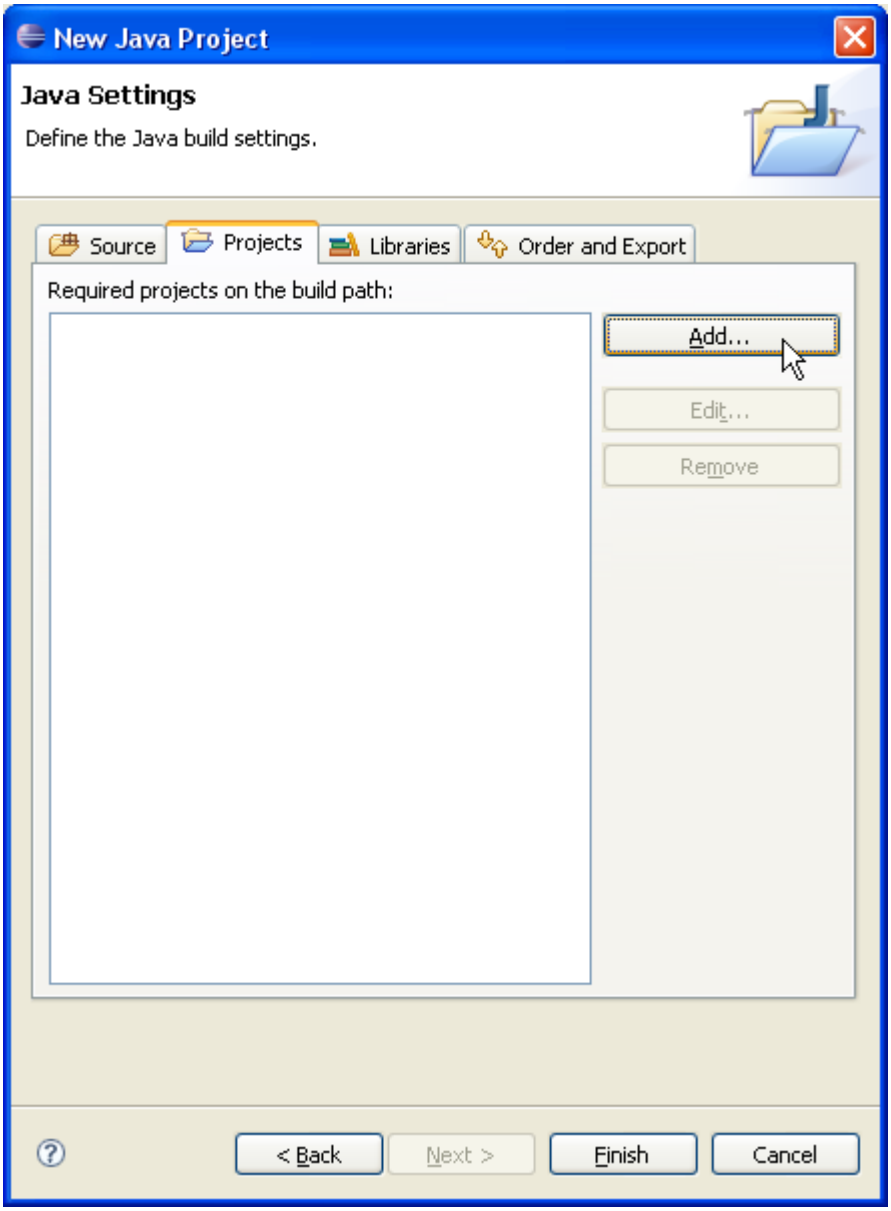
6. Click **Finish**.
7. In Java perspective, type **Ctrl+N** to open **New** wizards dialog. Select **Java project** in the list of wizards and click **Next**.
8. On the next page, type "Product" in the **Project name** field.
9. Deselect **Use default location**. Click **Browse...** and choose the *D:\Product* directory.



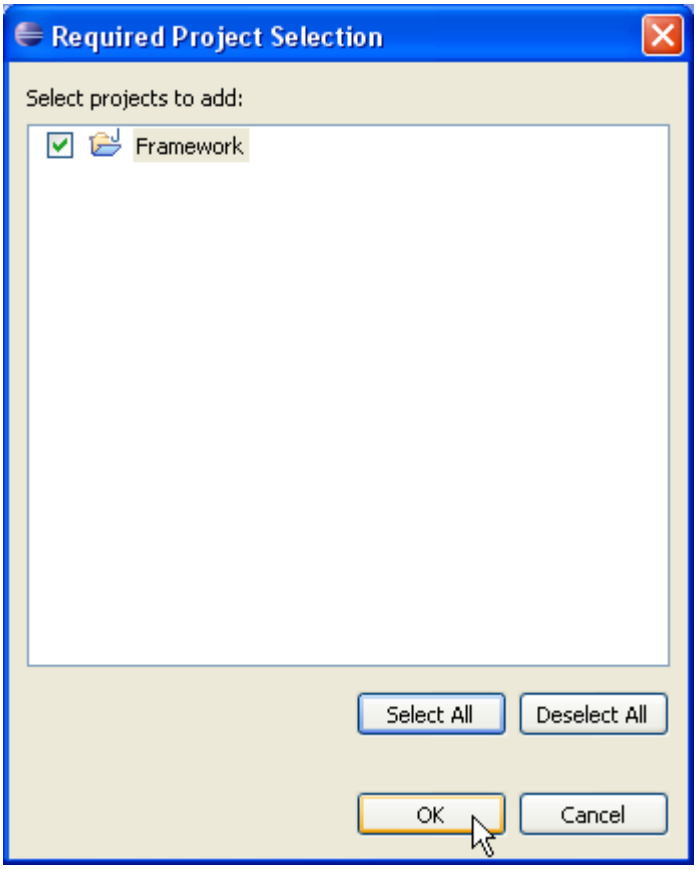
10. Click **Next**.
11. On the next page, verify that directory *JavaSourceFiles* has been automatically added as source folder.
    Expand it to preview your project source folder contents:



12. Select **Projects** tab.
Click **Add...**to add a dependency to source framework project...



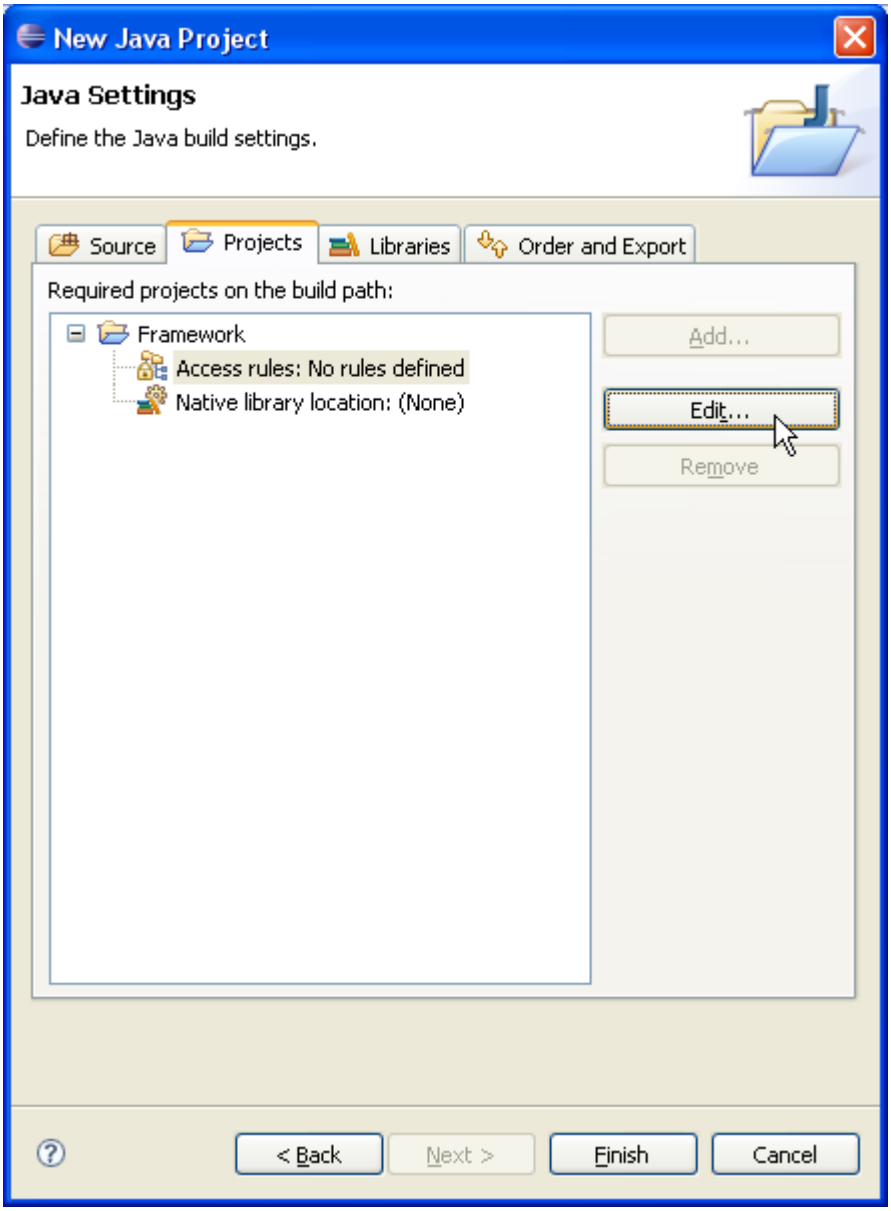
13. In **Required Project Selection**, check "Framework".



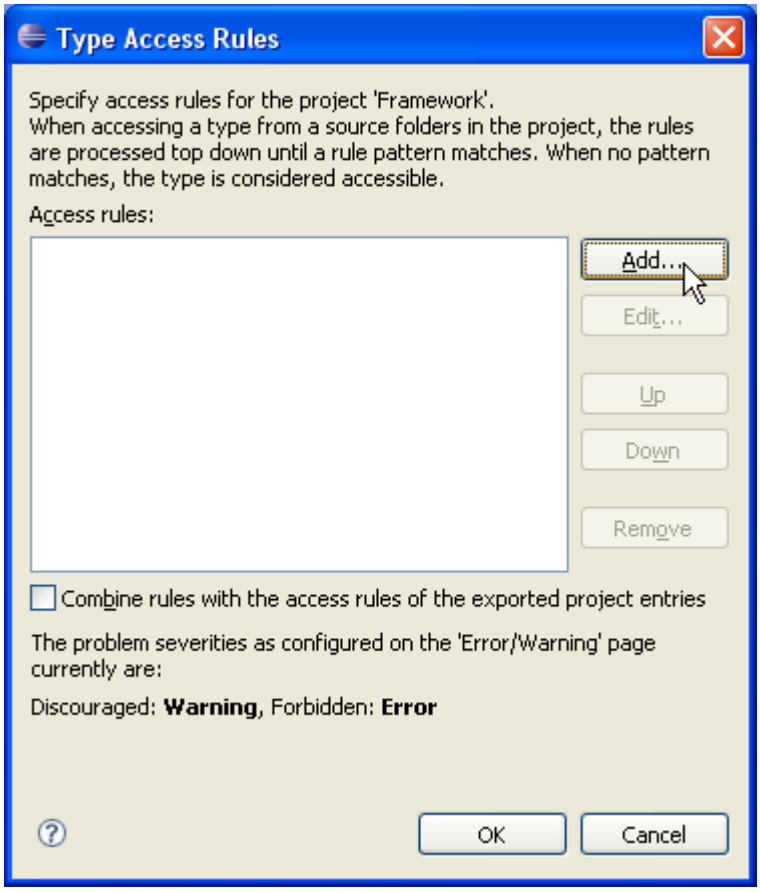
14. Click **OK** to validate and close dialog.
15. Now, let's put access rules on source framework content to authorize, discourage or forbid access to "Framework" source folders, package and classes...

    In **Projects** tab, select "Access rules" of "Framework" depending project.

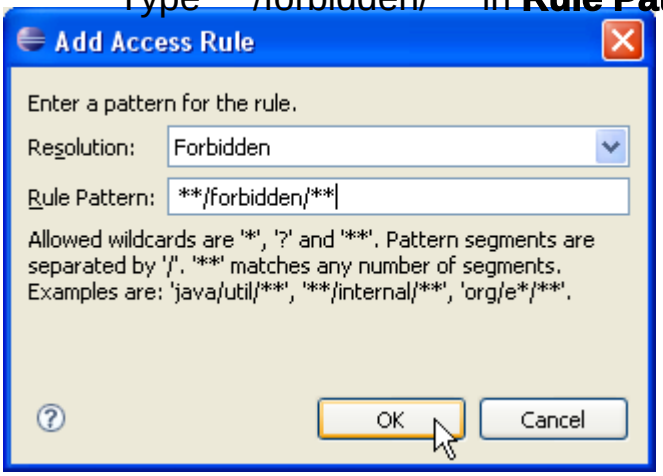
    Click **Edit...**.



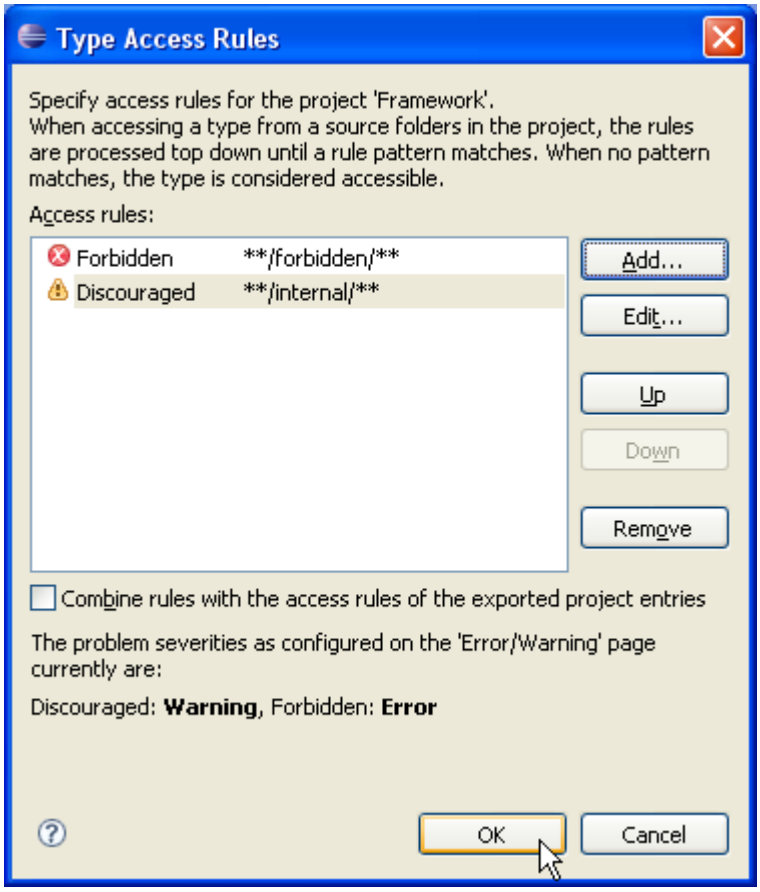
16. In **Type Access Rules**, click **Add...**.



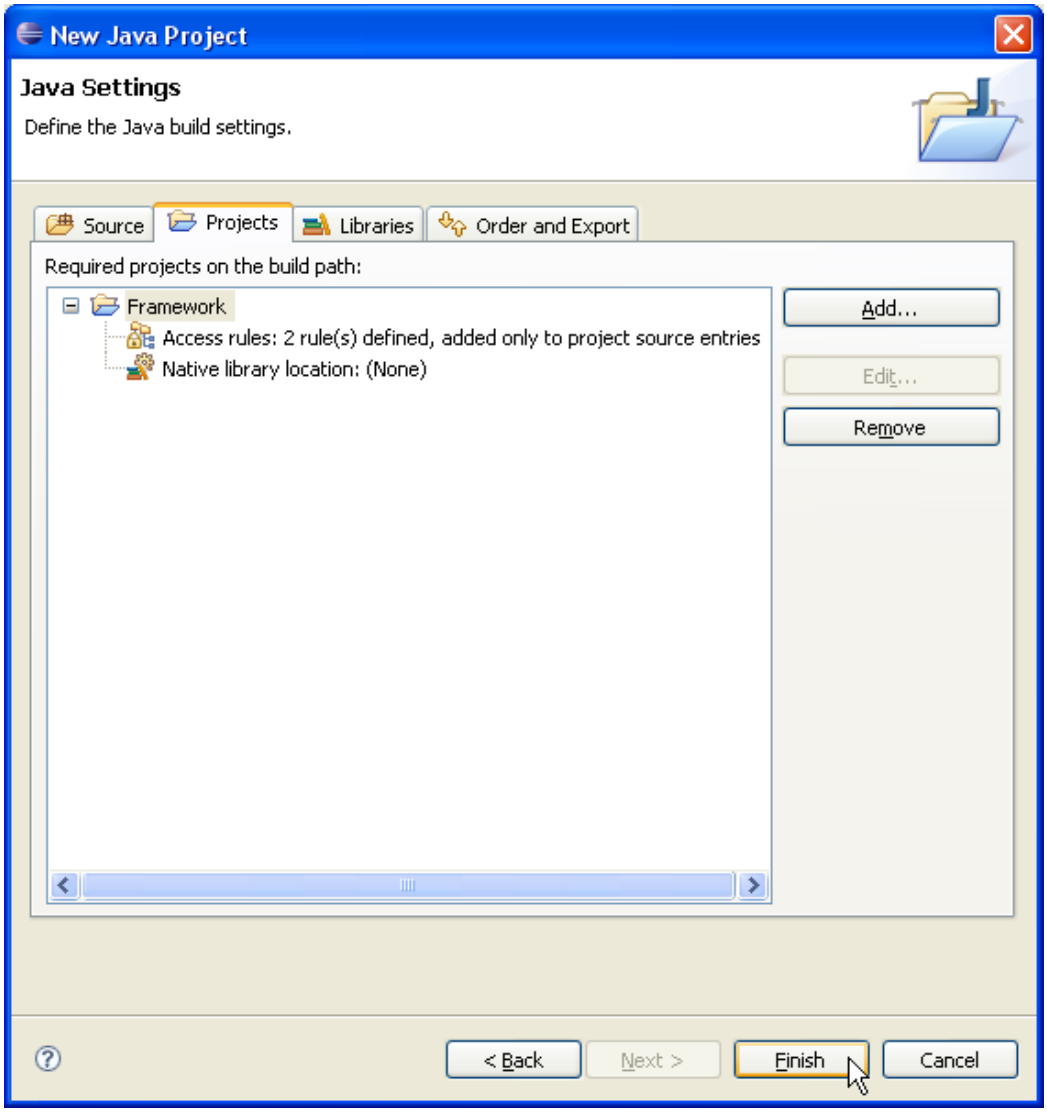
17. In **Add Access Rule**, select "Forbidden" for **Resolution**. Type "**/forbidden/**" in **Rule Pattern** field.



18. Click **OK** to validate access rule and close dialog.
19. Add another access rule: **Resolution**: "Discouraged" and **Rule Pattern**: "**/internal/**".
20. Your access rules now look as follows:

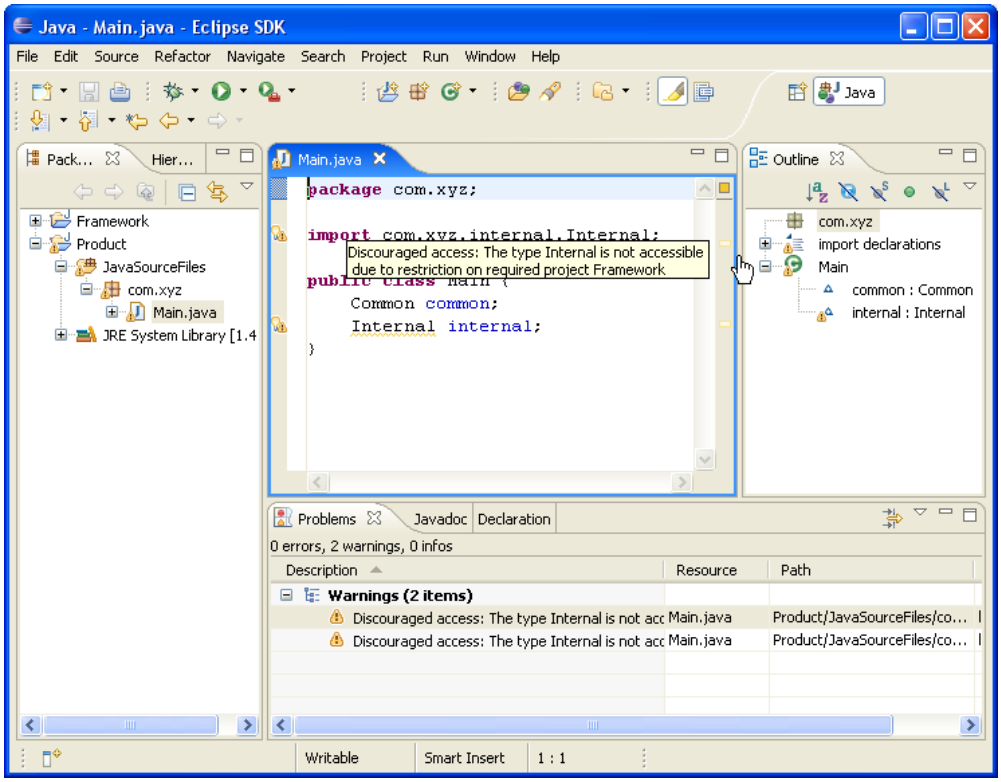


21. Click **OK** to validate these new rules and close dialog.
22. Dependent project has now 2 access rules set.



23. Click **Finish**.
24. You now have a Java project which contains the source of "Product" and which is using the source of "Framework".

    Some packages of the project "Framework" are restricted and if you try to import them, compiler displays either warnings or errors depending on your restriction level:



Next Section: Eclipse and J2SE 5.0

Related concepts

Java projects

Related reference

New Java Project Wizard
Package Explorer View

Eclipse and J2SE 5.0

Since release 3.1, Eclipse includes full support for the new Java language features of J2SE 5.0. One of the most important consequences of this support is that you may not notice it at all--everything you expect to work for J2SE 1.4, including editing, compiling, debugging, quick fixes, refactorings, source actions, searching, etc., will work seamlessly with J2SE 5.0's new types and syntax. In this document, we will introduce some of the more interesting capabilities Eclipse users will find when working with J2SE 5.0.

Note that both version numbers '1.5' and '5.0' are used to identify the release of the Java 2 Platform Standard Edition. Version '5.0' is the product version, while '1.5' is the developer version and also used for the compliance level.

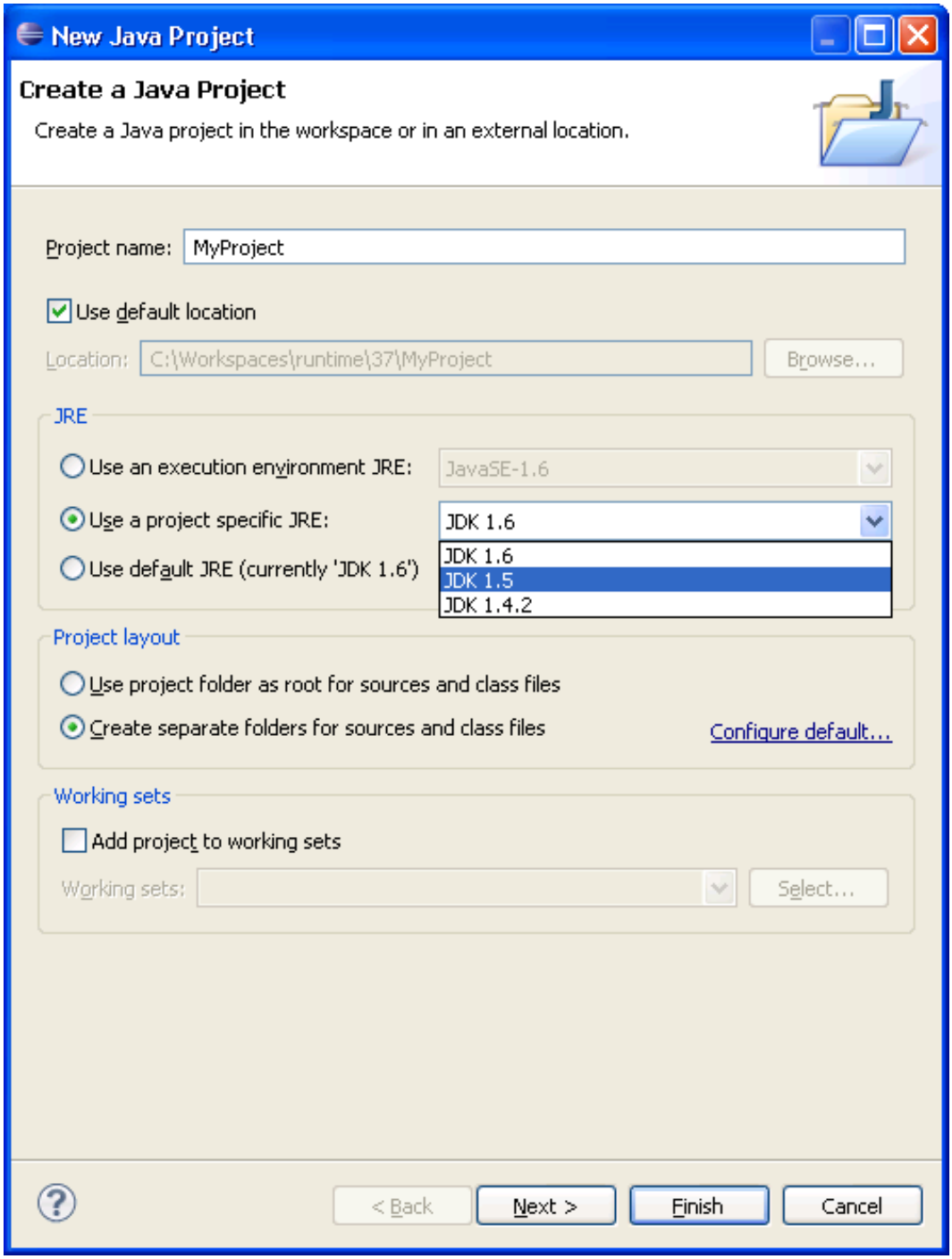
Prerequisites

In order to develop code compliant with J2SE 5.0, you will need a J2SE 5.0 or J2SE 6.0 Java Runtime Environment (JRE). If you start Eclipse for the first time using a J2SE 5.0 JRE, then it will use it by default. Otherwise, you will need to use the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs** preference page to register it with Eclipse.

This document introduces some of the new language features in J2SE 5.0 very briefly, but it is not a proper tutorial for these features. See here for more information.

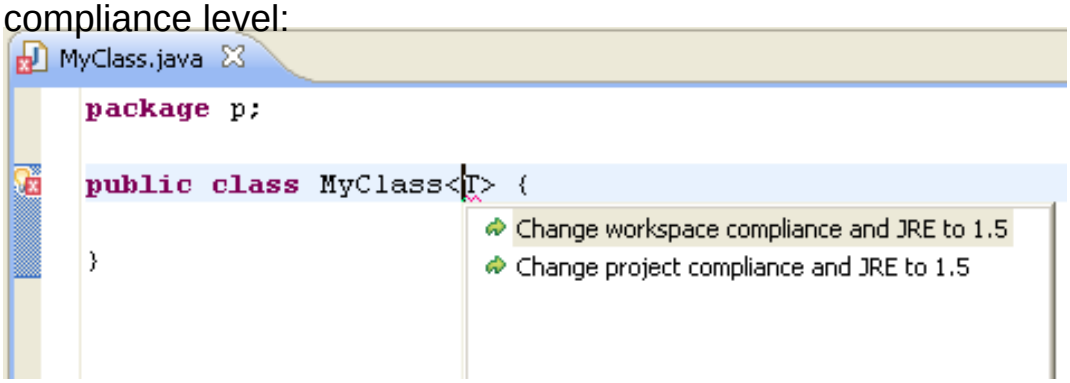
Compiler Compliance Level

To use the new J2SE 5.0 features, you must be working on a project that has a 1.5 compliance level enabled and has a 5.0 JRE. New projects will automatically get 1.5-compliance when choosing a 5.0 JRE on the first page of the [Image: /topic/org.eclipse.help/command_link.png]**New Java Project wizard**:

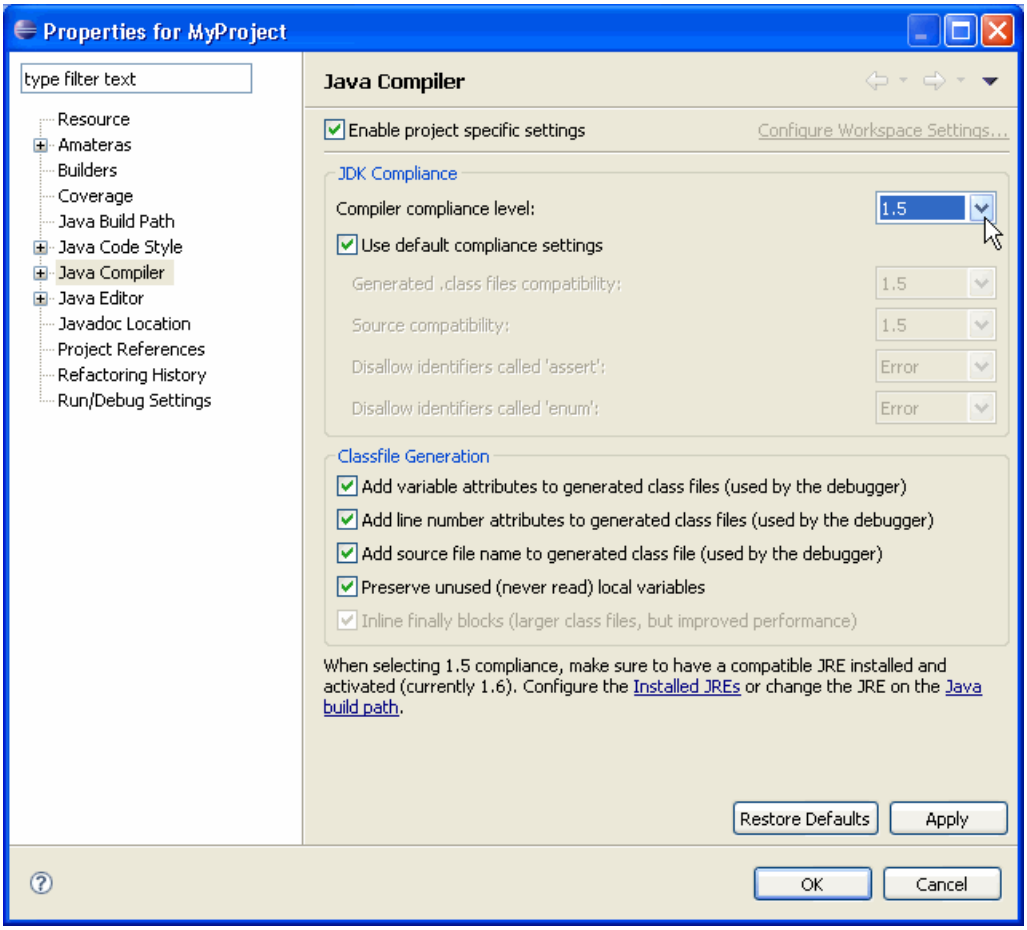


To convert an existing J2SE 1.4 project to J2SE 5.0, you can simply:

1. Make sure you have a J2SE 5.0 JRE installed.
2. Start using the 5.0 features in your code.
3. When a compiler error is flagged, use Quick Fix to update the project's compliance level:



4. For more fine-tuned control, the compiler compliance level can be set globally for a workspace (with the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler** preference page) or individually for each project (from the project's context menu, choose **Properties > Java Compiler**).



Projects with different compliance levels can co-exist in the workspace, and depend on each other. You can also fine-tune the kinds of compiler warnings and errors produced for each project using **Properties > Java Compiler > Errors/Warnings**. The **Generic Types** and the **Annotations** section contain options added for J2SE 5.0.

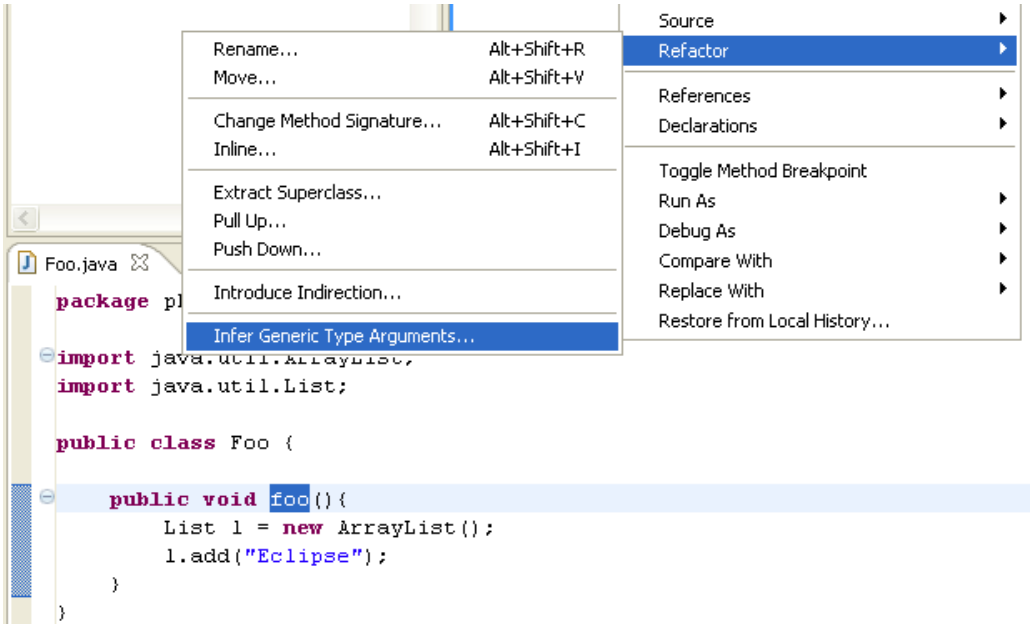
Generic Types

Generic types allow objects of the same class to safely operate on objects of different types. For example, they allow compile-time assurances that a `List<String>` always contains `Strings`, and a `List<Integer>` always contains `Integers`.

Anywhere that Eclipse handles a non-generic type, it can handle a generic type:

- Generic types can be safely renamed.
- Type variables can be safely renamed.
- Generic methods can be safely extracted from / inlined into generic code.
- Code assist can automatically insert appropriate type parameters in parameterized types.

In addition, a new refactoring has been added: **Infer Generic Type Arguments** can infer type parameters for every type reference in a class, a package, or an entire project.



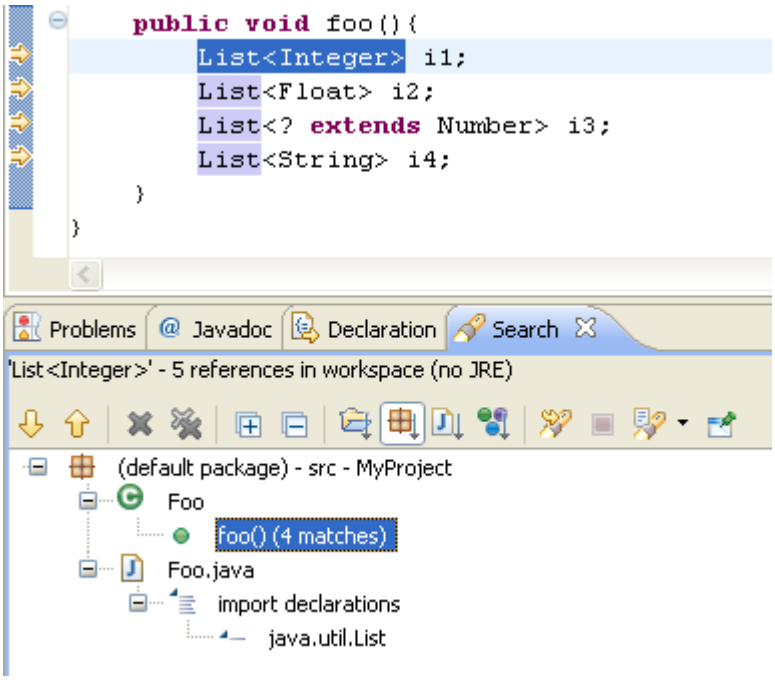
Invoking the refactoring produces:

```
    public void foo() {
        List<String> l= new ArrayList<String>();
        l.add("Eclipse");
    }
```

Eclipse provides new options when searching for references to generic types. Consider this example:

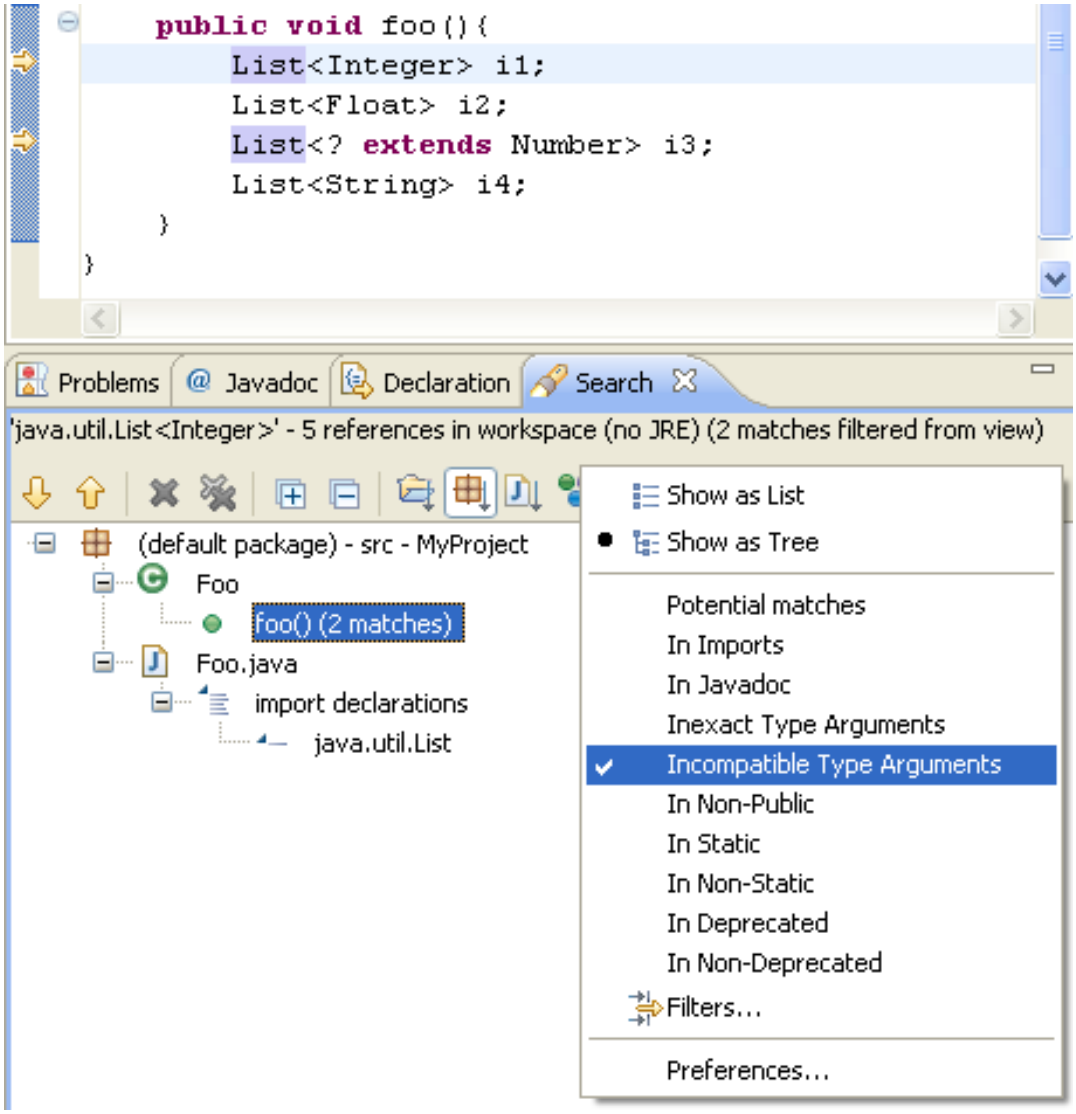
```
    void foo() {
        List<Integer> i1;
        List<Float> i2;
        List<? extends Number> i3;
        List<String> i4;
    }
}
```

Selecting the reference to `List<Integer>` and using **Search > References > Project** from the context menu will highlight the List types on all four lines:

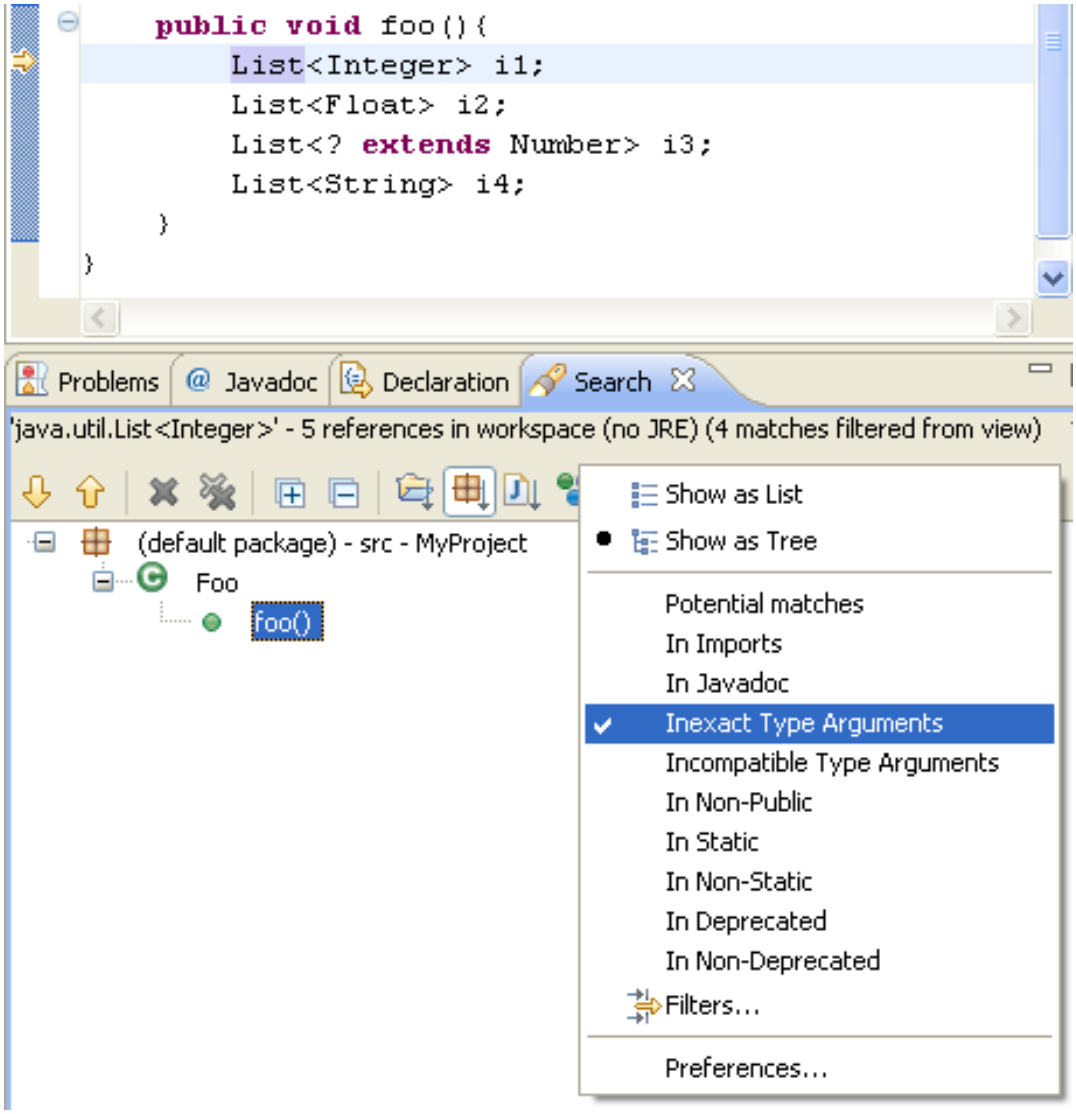


Using the Search View menu, the results can be filtered: Filter **Incompatible**

Type Arguments leaves only references to types that are assignment-compatible with the selected type:



Filter **Inexact Type Arguments** leaves only type references with the exact same signature:



Annotations

Annotations attach metadata about how Java types and methods are used and documented to the Java source and can then affect compilation or be queried at run-time. For example, `@Override` will trigger a compiler warning if the annotated method does not override a method in a superclass:

```
@Override public String toString() {return "a";}
```

java.lang.Override

Indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message.

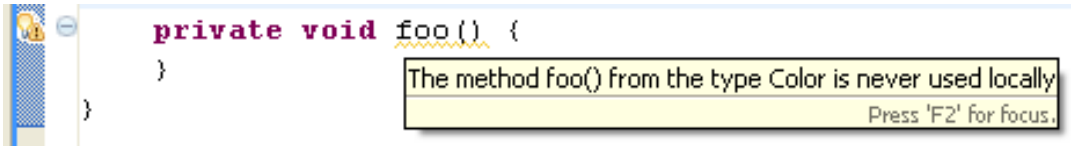
Author:
Joshua Bloch
Since:

Press 'F2' for focus.

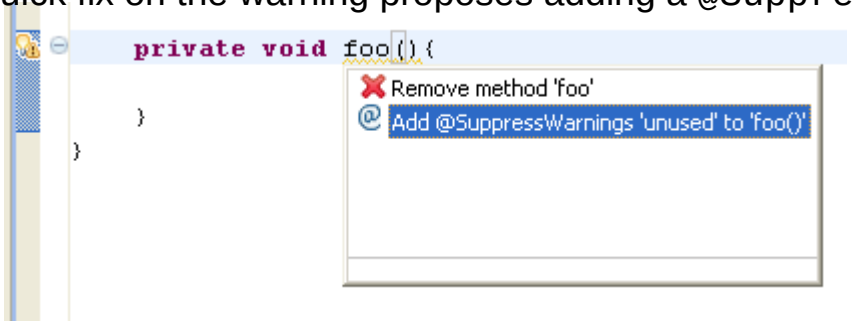
Everything you can do with a Java type, you can do with an annotation:

- Create new annotations using **New > Annotation**
- Refactor: rename, move, change signatures of members, etc.
- Search for occurrences
- Use code assist to fill in names and values

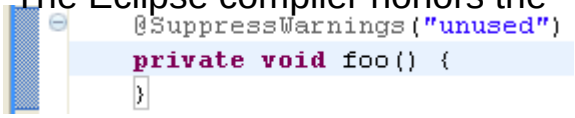
A very useful annotation with full support in Eclipse is `@SuppressWarnings`. For example, consider a private method that is currently unused, but you'd rather not delete:



Invoking quick fix on the warning proposes adding a `@SuppressWarnings` annotation:



Selecting the quick fix adds the annotation. The Eclipse compiler honors the annotation by removing the warning on `foo`:



Enumerations

Enumerations are types that are instantiated at runtime by a known, finite set of objects:

```
public enum Color {
    RED, GREEN, BLUE;
}
```

Again, anything you can do to a Java class can be done to an enumeration:

- Create new enumerations using **New > Enum**
- Refactor: rename, move, rename constants, etc.
- Search for occurrences
- Use code assist to fill in constants

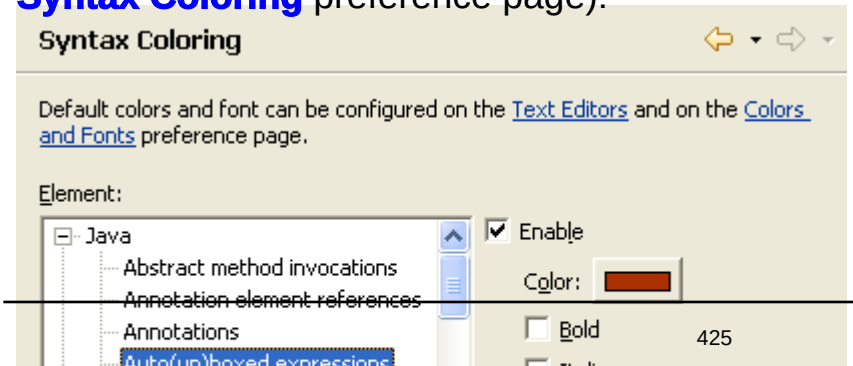
Autoboxing

Autoboxing and auto unboxing allow for elegant syntax when primitive types are assigned to or retrieved from Object references:

```
public void foo(Integer i) {
    foo(10);
}
```

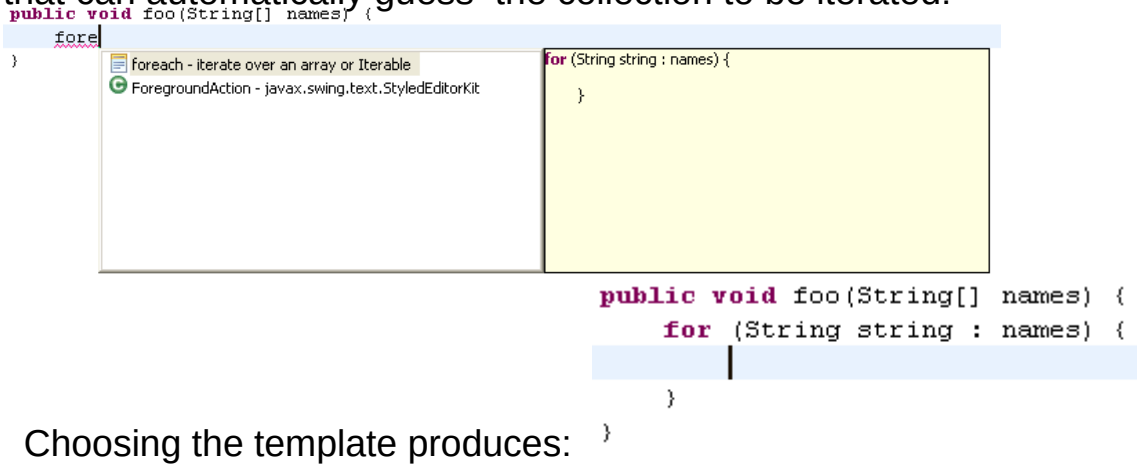
Eclipse's source manipulation features handle autoboxing seamlessly, giving the correct types to new local variables and correct code assists. For code understanding, it is also possible to flag instances of autoboxing or autounboxing conversions:

- mark them as compiler warnings or errors (**Boxing and unboxing conversions** in the **Potential Programming Problems** section of the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page)
- highlight them using syntax coloring (via the **Java > Auto(un)boxed expressions** section of the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Syntax Coloring** preference page):



Enhanced for loop

For the common case of operating on each element of an array or collection in turn, J2SE 5.0 allows a new, cleaner syntax. Eclipse provides a "foreach" code template that can automatically guess the collection to be iterated:




```
public void foo(String[] names) {
    for (String string : names) {
        |
    }
}
```

Choosing the template produces:

Eclipse also provides a "Convert to enhanced for loop" quick-assist to upgrade 1.4-style `for` loops where possible.

Static Imports

Static imports allow you to use static fields and methods from other classes without qualification.

Content assist in the Java editor can suggest such static members and adds a static import when required. To get such proposals, configure the static import favorites on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist > Favorites** preference page.

Converting J2SE 1.4 code to 5.0

Eclipse offers the following tools to help you bring J2SE 1.4 code to 5.0:

- **Clean Up** action
  - Add missing `@Override` and `@Deprecated` annotations
  - Convert `for` loops to enhanced
- **Infer Generic Type Arguments** refactoring

Happy coding!

Java Projects

A Java project contains source code and related files for building a Java program. It has an associated Java builder that can incrementally compile Java source files as they are changed.

A Java project also maintains a model of its contents. This model includes information about the type hierarchy, references and declarations of Java elements. This information is constantly updated as the user changes the Java source code. The updating of the internal Java project model is independent of the Java builder; in particular, when performing code modifications, if auto-build is turned off, the model will still reflect the present project contents.

You can organize Java projects in two different ways:

- Using the project as the source container. This is the recommended organization for simple projects.
- Using source folders inside the project as the source container. This is the recommended organization for more complex projects. It allows you to subdivide packages into groups.

Related concepts

Java builder

Related reference

New Java project wizard

Java Builder

The Java builder builds Java programs using its own compiler (the Eclipse Compiler for Java) that implements the Java Language Specification. The Java builder can build programs incrementally as individual Java files are saved. Note that the Eclipse Compiler for Java can also be invoked using Ant as described in the **Using the ant javac adapter** section.

Problems detected by the compiler are classified as either warnings or errors. The existence of a warning does not affect the execution of the program; the code executes as if it were written correctly. Compile-time errors (as specified by the Java Language Specification) are always reported as errors by the Java compiler. For some other types of problems you can, however, specify if you want the Java compiler to report them as warnings, errors or to ignore them. To change the default settings, use the [Image: /topic/org.eclipse.help/command_link.png]Java > Compiler > Errors/Warnings preference page.

The Java compiler can create CLASS files even in presence of compilation errors. However, in the case of serious errors (for example, references to inconsistent binaries, most likely related to an invalid build path), the Java builder does not produce any CLASS files.

Related concepts

Build classpath

Related tasks

Using the batch compiler
Using the ant javac adapter
Excluding warnings using SuppressWarnings

Related reference

Java build path properties
Java compiler preferences

Build Classpath

The build classpath specifies which Java source files and resource files in a project are considered by the Java builder and specifies how to find types outside of the project. The Java builder compiles the Java source files into the output folder and also copies the resources into it. The build classpath is specified for each project. In the project properties, it is referred to as the **Java Build Path**

Related concepts

Java builder
Classpath variables
Inclusion and exclusion patterns
Access rules

Related reference

Classpath variables preferences
Java build path properties

Inclusion and Exclusion Patterns

Inclusion and Exclusion patterns can be configured for source folders. They define which files are considered by the Java builder and other Java tooling. This set of files consists of all files and folders where the path relative to the source folder matches an inclusion pattern but not an exclusion pattern.

By default, all files and folders contained in a source folder are included. By defining an inclusion pattern, the set of included resources will be limited to the resources matching the inclusion pattern. Using exclusion patterns, some of these resources can be excluded again.

Exclusion patterns are required when nesting source folders. The nested folder must be excluded from the outer source folder.

The patterns have the same format as ANT patterns:

- '*' matches zero or more characters, '?' matches one character.
- '/' is used to separate folders: This means the first segment in the pattern is matched against the most outer folder name in the path to match, the second segment with the second, and so on.
- '**' matches any number of folders

Related concepts

Java builder
Classpath variables
Access rules

Related reference

Java Build Path properties

Access Rules

Access rules can be added to build classpath entries to specify which types in the given entry can be accessed and which not. If the compiler detects a type access to a type that should not be accessed, it will create a problem marker.

- Non-accessible rules define types that must not be referenced. The compiler typically creates an error marker for accesses to these types.
- Discouraged rules define types that should not be referenced. The compiler typically creates a warning marker for accesses to these types.
- Accessible rules define types that can be referenced.

Each rule consist of a pattern (same format as ANT patterns) and one of the rule types listed above.

Each classpath entry can have any number of rules defined. The compiler will process the list in the order defined and take the first matching rule.

The severity of the problem marker generated for accesses to 'Non-accessible' and 'Discouraged' type can be configured on the Java compiler's Error/Warnings preference page.

Related concepts

Combine Access Rules
Java builder
Classpath variable
Inclusion and exclusion patterns

Related reference

Java build path properties
Error/Warnings preference page

Classpath Variables

The build path for a Java project can include source code files, other Java projects, folders containing class files and JAR files. JAR files can be specified using file system paths, or by using variables that refer to locations on the network.

Classpath variables allow you to avoid references to the location of a JAR file or folders on your local file system. By using a classpath variable, you can specify a JAR file or folder using only a variable name, such as JRE_LIB, rather than specifying the location on your workstation. In this way, you can share build paths across teams and define the variables to refer to the correct location for your particular computer.

Related concepts

Java builder
Classpath variable
Inclusion and exclusion patterns
Access rules

Related reference

Classpath variables preferences
Java build path properties

Java Perspectives

The Java development tools contribute the following perspectives to the workbench:

Java

A perspective designed for working with Java projects. It consists of an editor area and the following views:

- Package Explorer
- Outline
- Problems
- Javadoc
- Declaration

Java Browsing

A perspective designed for browsing the structure of Java projects. It consists of an editor area and the following views:

- Projects
- Packages
- Types
- Members

Java Type Hierarchy

A perspective designed for exploring a type hierarchy. It can be opened on types, compilation units, packages, projects or source folders and consists of the Type Hierarchy view and an editor.

Debug

A perspective designed for debugging your Java program. It includes an editor area and the following views.

- Debug
- Breakpoints
- Variables
- Expressions
- Outline
- Console
- Tasks

Related concepts

Java views

Related reference

Breakpoints view
Console view
Debug view
Display view
Expressions view
Java outline
Package Explorer view
Type Hierarchy view
Variables view

Java Views

The Java development tools contribute the following views to the workbench:

Java

- ### Package Explorer View

  The Package Explorer view shows a Java-specific view of resources inside Java projects in your workbench. The Java element hierarchy is defined by the project's build class paths and consists of elements like
  - source folders and libraries
  - packages
  - source and class files
  - types, methods, fields and initializers

- ### Type Hierarchy View

  The Type Hierarchy view allows you to look at the complete hierarchy for a type, only its subtypes, or only its supertypes. You can also open the hierarchy on a project, source folder or package.

- ### Call Hierarchy View

  The Call Hierarchy view allows you to look at the callers and callees for a selected Java member.

- ### Java Outline View

  The Java outline view displays an outline of the structure of the currently-active Java file in the editor area.

- ### JUnit View

  The JUnit view shows you the JUnit test run progress and status, and allows you to rerun tests.

- ### Javadoc View

  The Javadoc view shows the Javadoc of the element selected in the Java editor or in a Java view.

- ### Declaration View

  The Declaration view shows the source of the element selected in the Java editor or in a Java view.

Java Browsing

- ### Projects View

  The Projects view shows Java projects, source folders, external and internal libraries.

  Note: source folders and libraries (both internal and external) presented in this view are not expandable. When they are selected, their contents are shown in the Packages view.

- ### Packages View

  The Packages view shows a list of Java packages from the currently selected Java projects, source folders or libraries. Typically, the Projects view is used to make this selection.

- ### Types View

  The Types view shows a list of Java types from the currently selected packages. Typically, the Packages view is used to make this selection.

- **Members View**

The Members view shows the content of a type, compilation unit or CLASS file. Typically, the Types view is used to make this selection.

Debug

- **Debug View**

The Debug view allows you to manage the debugging or running of a program in the workbench. It displays the stack frame for the suspended threads for each target you are debugging. Each thread in your program appears as a node in the tree. It displays the process for each target you are running.

- **Breakpoints View**

The Breakpoints view lists all the breakpoints you currently have set in your workspace. You can also enable or disable breakpoints, delete them, add new ones, group them by working set, or configure attributes of a selected breakpoint.

- **Expressions View**

Data can be inspected in the Expressions view. You can inspect data from a scrapbook page, a stack frame of a suspended thread, and other places. The Expressions View opens automatically when an item is added to the view. Entries in the Expressions View can be selected to have more detailed information be displayed in the Detail Pane. When debugging a Java program, data that contains variables can be expanded to show the variables and the fields the variables contain.

- **Variables View**

The Variables view displays information about the variables associated with the stack frame selected in the Debug View. When debugging a Java program, variables can be selected to have more detailed information be displayed in the Detail Pane. In addition, Java objects can be expanded to show the fields that variable contains.

- **Console View**

The Console view displays a variety of console types depending on the type of development and the current set of user settings. The consoles that are provided by default with the Eclipse Platform are:

- The Process Console
- The Stacktrace Console
- The CVS Console
- The OSGi Console

- **Display View**

The Display view displays the result of evaluating an expression in the context of the current stack frame. You can evaluate and display a selection either from the editor or directly from the Display View.

View Customization

Both the appearance and behavior of the Java views can be customized.

- Filtering
- Sorting
- Java element decorations

- Presentation options

Related concepts

Java perspectives

Related tasks

Changing the appearance of the Console view

Related reference

Views and editors

Filtering in Java Views

Applying View Filters

All Java views allow to filter the displayed Java elements. The set of available filters depends on the view.

To filter elements:

1. On the view toolbar, click the **Menu** button and choose **Filters...**.
2. Select or clear the filters that you want to apply to the view (read **Filter description** to learn about the selected filter's functionality).
3. Optionally, you can select patterns for filtering the view:
   - Select the **Name filter patterns** checkbox at the top of the dialog.
   - In the text field below, specify the desired patterns (names matching one of the patterns will be hidden).

Filtering Members

Several Java views (e.g. **Outline**, **Type Hierarchy**, **Members**) offer filtering of members (fields, types and methods). The filters are available as toolbar buttons or as view menu items, depending on the view. There are 3 member filters:

- Hide Fields: when activated, this filter causes all fields to be removed from the view.
- Hide Static Fields and Methods: when activated, this filter causes all static members to be removed from the view.
- Hide Non-Public Members: when activated, this filter causes all non-public members to be removed from the view.

Additionally, the **Package Explorer** view can display or hide all elements inside compilation units. To show or hide members in the **Package Explorer** view, select or clear the Show members in Package Explorer checkbox in the [Image: /topic/org.eclipse.help/command_link.png]Java > Appearance preference page.

Filtering Working Sets

Several Java views (e.g. **Package Explorer**, **Type Hierarchy**) offer to filter by one or more working sets. The filters are available in the view menu. Only the elements that are contained in the selected working sets are shown in the view.

Related concepts

Java projects
Working sets

Related reference

Package Explorer

Sorting in Java Views

The **Members** and **Outline** views can present members sorted or in the order of declaration in the compilation unit.

To sort members:

- Toggle on the Sort toolbar button in the view toolbar.
- The sorting order can be configured on the on the [Image: /topic/org.eclipse.help/command_link.png]Java > Appearance > Members Sort Order preference page.
- After the above sorting is performed, members in each group are sorted alphabetically.

To present members in the order of declaration in the compilation unit.

- Toggle off the Sort toolbar button in the view toolbar.

Related reference

Java toolbar actions

Java Element Decorations

Override Indicators

Java views can show special decoration icons to indicate methods that override or implement methods from supertypes.

To show or hide the override indicators, select or clear the Java Method Override Indicator checkbox in the [Image: /topic/org.eclipse.help/command_link.png]General > Appearance > Label Decorations preference page.

Java Type Indicator

The package explorer can indicate the kind of Java type (Enum, Interface, Annotation) contained in a Java file.

Note:

- If a Java file contains more than one top-level type, it is decorated according to its primary type.
- No decoration is shown for classes, except for abstract classes.

To show or hide the Java type indicators, select or clear the Java Type Indicator checkbox in the [Image: /topic/org.eclipse.help/command_link.png] General > Appearance > Label Decorations preference page.

Related reference

Appearance preference page
Package Explorer

Presentation Options for Java Views

Showing and Hiding Method Return Types

Several Java views (e.g. **Outline**, **Members**) present methods and can also show their return types and type parameters.

To show method return types in Java views select or clear the Show method return types or Show method type parameters checkbox in the [Image: /topic/org.eclipse.help/command_link.png]Java > Appearance preference page.

Showing Full or Compressed Package Names

The **Package Explorer** and **Packages** views can show full or compressed package names.

To show full package names:

- Clear the Compress all package name segments, except the final segment checkbox on the [Image: /topic/org.eclipse.help/command_link.png]Java > Appearance preference page.

To show compressed package names:

- Select the Compress all package name segments, except the final segment checkbox on the [Image: /topic/org.eclipse.help/command_link.png]Java > Appearance preference page.

  Compression patterns control how many characters of each package name segment are displayed. The last segment of a package name is always displayed.

  A compression pattern of "." indicates that only the separating periods are shown to represent a segment. A digit (n) in a compression pattern represents the first n characters of a package name segment. Examples are the best way to understand compression patterns. The package org.eclipse.jdt would be displayed as follows using the example compression patterns:

.	..jdt
0	jdt
2~	or~.ec~.jdt
3~	org.ecl~.jdt

Related reference

Java appearance preference page
Package Explorer

Java Editor

The Java editor provides specialized features for editing Java code.

Associated with the editor is a Java-specific Outline view, which shows the structure of the active Java compilation unit. It is updated as the user edits the compilation unit.

The editor can also show a breadcrumb navigation bar. The breadcrumb shows the parent chain of the element at the current cursor position. It is also updated when the user edits the compilation unit or changes the selection.

The Java editor can be opened on binary CLASS files. If a JAR file containing the CLASS files has a source attachment, then the editor shows the corresponding source.

The editor includes the following features:

- Syntax highlighting
- Content/code assist
- Code formatting
- Import assistance
- Quick fix
- Integrated debugging features

The Java editor can be configured to either show an entire compilation unit or a single Java element only. To change the setting, use the toolbar button Show Source of Selected Element Only.

The most common way to invoke the Java editor is to open a Java file from the Package Explorer using pop-up menus or by clicking the file (single or double-click depending on the user preferences). You can also open the editor by opening Java elements, such as types, methods, or fields, from other views.

Related concepts

Quick fix and assist
Templates

Related tasks

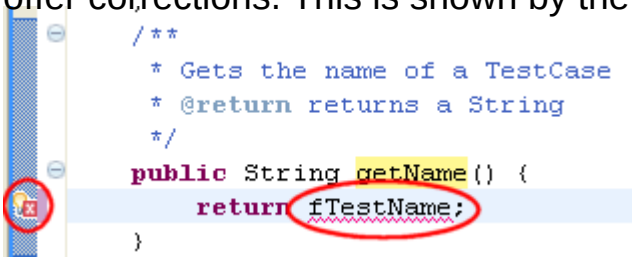
Opening an editor for a selected element
Viewing runtime exceptions
Evaluating expressions

Related reference

Quick assists
Quick fixes
Java editor actions
Java editor preferences
Java outline
Java Editor Breadcrumb
Views and editors

Quick Fix and Quick Assist

For most problems underlined with a problem highlight line, the Java editor can offer corrections. This is shown by the light bulb shown in the editor marker bar.



To see the correction proposals use the Quick Fix action

- Set the cursor inside or near the highlight range, and select **Quick Fix** from the Edit menu or the context menu.
- Set the cursor inside or near the highlight range, and press Ctrl + 1
- Click on the light bulb

Quick fix is also available on a problem entry in the Problems view. The Quick Fix action will open a dialog to select the correction.

Note that the light bulb is only a hint. It is possible that even with the light bulb shown, it turns out that no corrections can be offered.
A overview of quick fixes available in the Java editor can be found here.

Quick Assists are proposals available even if there is no problem or warning. They use the same short cut and action like Quick Fix. Quick assists are used for local code manipulations. See the Quick assist reference for more information.

Related concepts

Java editor

Related reference

Quick assists
Quick fix
Java editor preferences
Edit menu

Editor Templates

Templates are a structured description of coding patterns that reoccur in source code. The Java editor supports the use of templates to fill in commonly used source patterns. Templates are inserted using content assist (**Ctrl+Space**).

For example, a common coding pattern is to iterate over the elements of an array using a for loop that indexes into the array. By using a template for this pattern, you can avoid typing in the complete code for the loop. Invoking content assist after typing the word `for` will present you with a list of possible templates for a for loop. You can choose the appropriate template by name (`iterate over array`). Selecting this template will insert the code into the editor and position your cursor so that you can edit the details.

Templates can contain template variables. Variables mark the editable locations. They can be resolves to a concrete value when the template is evaluated in its context. They can also provide a list of alternative proposals valid at the given location.

Many common templates are already defined. These can be viewed with the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Templates** preference page. You can also create your own templates or edit the existing ones.

Related reference

Templates preferences
Template variables
Template editing
Edit menu
Java content assist

Java Search

The Java searching support allows you to find declarations, references and occurrences of Java elements (packages, types, methods, fields). Searching is supported by an index that is kept up to date in the background as the resources corresponding to Java elements are changed. The Java search operates on workspaces independent of their build state. For example, searches can be conducted when auto-build is turned off.

The following searches can be initiated from the pop-up menus of Java elements or from the Java search dialog:

Command	Description
References	Finds all references to the selected Java element
Declarations	Finds all declarations of the selected Java element
Implementors	Finds all implementors of the selected Java interface
Read access	Finds all read accesses to the selected Java field
Write access	Finds all write accesses to the selected Java field
Match locations for type references	Finds all type references at specified locations (in casts, as field type, ...)
Occurrences in File	Finds all occurrences of the selected Java element in its file

The scope of the search is defined as:

Workspace - all projects and files in the workspace are included in this search

Enclosing Projects - the projects enclosing the currently selected elements

Hierarchy - only the type hierarchy of the selected element is included in this search

Working Set - only resources that belong to the chosen working set are included in this search

Related reference

Search actions

Java search tab

Refactoring Support

The goal of Java program refactoring is to make system-wide code changes without affecting the behavior of the program. The Java tools provide assistance in easily refactoring code.

The refactoring tools support a number of transformations described in Martin Fowler's book *Refactoring: Improving the Design of Existing Code*, Addison Wesley 1999, such as *Extract Method*, *Inline Local Variable, etc.*.

To get an overview of all offered refactorings look at the **Refactor menu**. Refactoring commands are also available from the context menus in many views or appear as **quick assists**.

When performing a refactoring operation, you can optionally preview all of the changes resulting from a refactoring action before you choose to carry them out. When previewing a refactoring operation, you will be notified of potential problems and will be presented with a list of the changes the refactoring action will perform. If you do not preview a refactoring operation, the change will be made in its entirety and any resultant problems will be shown. If a problem is detected that does not allow the refactoring to continue, the operation will be halted and a list of problems will be displayed.

Refactoring commands are available from the context menus of several Java views (e.g. Package Explorer, Outline) and editors. Many "apparently simple" commands, such as **Move** and **Rename**, are actually refactoring operations, since moving and renaming Java elements often require changes in dependent files.

Refactorings can not only be performed interactively, but also from refactoring scripts. Most refactorings available in the **Refactor** menu are stored in the workspace refactoring history in order to be used in refactoring scripts afterwards. The refactoring tools support the creation of refactoring scripts based on refactorings in the workspace refactoring history. Refactoring scripts can then be applied to an arbitrary workspace. Applying a refactoring script launches a refactoring wizard which is able to replay the refactorings as if they had been initiated by the user which originally had created them.

Related to refactoring scripts, the refactoring tools offer a refactoring to migrate a JAR File to a newer version, using refactoring information to avoid breaking changes in your workspace after the migration.

Related reference

Refactoring actions
Refactoring wizard
Java preferences

Debugger

The Java development toolkit (JDT) includes a debugger that enables you to detect and diagnose errors in your programs running either locally or remotely.
The debugger allows you to control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.
The debugger has a client/server design so you can debug programs running remotely on other systems in the network as well as programs running locally on your workstation. The debug client runs inside the workbench on your workstation. The debugger server runs on the same system as the program you want to debug. This could be a program launched on your workstation (local debugging) or a program started on a computer that is accessible through a network (remote debugging).

- Related concepts

Java development tools (JDT)
Breakpoints
Remote debugging
Local debugging

- Related tasks

Adding breakpoints
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Evaluating expressions
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Suspending threads

- Related reference

Debug preferences
Debug view
Run menu actions
Run and Debug toolbar actions

Scrapbook

The Java development toolkit (JDT) contributes a scrapbook facility that can be used to experiment and evaluate Java code snippets before building a complete Java program. Snippets are edited and evaluated in the Scrapbook page editor, with resultant problems reported in the editor.

From a Java scrapbook editor, you can select a code snippet, evaluate it, and display the result as a string. You can also show the object that results from evaluating a code snippet in the debuggers' **Expressions View**.

- Related concepts

Java development tools (JDT)
Debugger

- Related tasks

Creating a Java scrapbook page
Displaying the result of evaluating an expression
Inspecting the result of evaluating an expression

- Related reference

New Java Scrapbook Page wizard
Java Scrapbook page
Expressions view

Local Debugging

The Java debugger has a client/server design so that it can be used to debug programs that run locally (on the same workstation as the debugger) or remotely (on another computer on the network).

Local debugging is the simplest and most common kind of debugging. After you have finished editing and building your Java program, you can launch the program on your workstation using the [Image: /topic/org.eclipse.help/command_link.png]**Run > Debug Configurations...** menu item on the workbench. Launching the program in this way will establish a connection between the debugger client and the Java program that you are launching. You may then use breakpoints, stepping, or expression evaluation to debug your program.

- Related concepts

Breakpoints

- Related tasks

Adding breakpoints
Resuming the execution of suspended threads
Running and debugging
Suspending threads

- Related reference

Debug preferences
Debug view
Run menu actions
Run and Debug toolbar actions

Remote Debugging

The client/server design of the Java debugger allows you to launch a Java program from computer on your network and debug it from the workstation running the platform. This is particularly useful when you are developing a program for a device that cannot host the development platform. It is also useful when debugging programs on dedicated machines such as web servers.

Note: To use remote debugging, you must be using a Java VM that supports this feature.

To debug a program remotely, you must be able to launch the program in debug mode on the remote machine so that it will wait for a connection from your debugger. The specific technique for launching the program and connecting the debugger are VM-specific. The basic steps are as follows:

1. Ensure that you are building your Java program with available debug information. (You can control these attributes from the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler** preference page).
2. After you build your Java program, install it to the target computer. This involves copying the .CLASS files or .JAR files to the appropriate location on the remote computer.
3. Invoke the Java program on the remote computer using the appropriate VM arguments to specify debug mode and a communication port for the debugger.
4. Start the debugger using a remote launch configuration and specify the address and port of the remote computer.

When setting up the remote launch configuration there are a few items to take note of:

- You must ensure you enter the correct hostname i.e. the name of the remote computer where you are currently running your code. The hostname can also be the IP address of the remote machine, for example using ***127.0.0.1*** instead of ***localhost***.
- The port number must be the number of the port on the ***remote machine***.

Certain Java VMs support another way of starting a remote debug session. In this alternative connection type, the debugger is launched first. The debugger then waits and listens for the VM to connect to it. To debug this way, you must properly configure and launch a remote debug session, then launch the VM, specifying the location of your waiting debugger. For more information, see Using the remote Java application launch configuration.

Related tasks

Using the remote Java application launch configuration
Disconnecting from a VM

Breakpoints

A breakpoint suspends the execution of a program at the location where the breakpoint is set.

Breakpoints can be enabled and disabled via the context menu in the **Breakpoints View**, or via the context menu in the **Java Editor** ruler.

- An enabled breakpoint causes a thread to suspend whenever the breakpoint is encountered. Enabled breakpoints are drawn with a blue circle [ ] and have a checkmark overlay once successfully installed. A breakpoint can only be installed when the class the breakpoint is located in has been loaded by the VM.
- A disabled breakpoint will not cause threads to suspend. Disabled breakpoints are drawn with a white circle [ ].

Breakpoints are displayed in the vertical editor ruler and in the Breakpoints view.

Related tasks

Adding breakpoints
Resuming the execution of suspended threads
Running and debugging
Suspending threads

Related reference

Debug preferences
Debug view
Run menu actions
Run and debug toolbar actions
Breakpoints view

String Externalization

The Java tools help you to develop applications that can be run on international platforms. An important facet of designing a program for use in different countries is the localization, or externalization, of text that is displayed by the program. By externalizing strings, the text can be translated for different countries and languages without rebuilding the Java program.

The JDT provides the following support for internationalization and string externalization:

- A compiler option lets you mark non-externalized strings as compile-time warnings or errors.
  - See the **Non-externalized strings** option in the **Code style** section of the [Image: /topic/org.eclipse.help/command_link.png]Java > Compiler > Errors/Warnings preference page.
- Tools that allow you to find strings that have not been externalized.
- A wizard that will guide you through externalizing the strings.
- Tools that help you to find unused and incorrectly used keys for strings located in property files.

Comments can be used to denote strings that should not be externalized and should not result in compile-time warnings or errors. These comments are of form `//$NON-NLS-n$` where n is the 1-based index of the string in a line of code.

Additional information about internationalized applications can be found in the following documents:

- http://eclipse.org/articles/Article-Internationalization/how2I18n.html
- http://java.sun.com/docs/books/tutorial/i18n/intro/index.html

Related tasks

Finding strings to externalize
Finding unused and incorrectly used keys in property files

Related reference

Source menu
Externalize strings wizard
Java compiler preferences

Changing the active perspective when launching

You can control which perspective becomes active when a program is launched and when it suspends. The setting is configurable for each launch configuration type, for each of the launch modes it supports. To activate a particular perspective when a program is launched, do the following:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Perspectives** preference page.
2. Select the **Always** option for the **Open the associated perspective when launching** preference. This will cause the perspective associated with a program to become active whenever it is launched.

To activate a particular perspective when a program is suspends, do the following:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Perspectives** preference page.
2. Select the **Always** option for the **Open the associated perspective when an application suspends** preference. This will cause the perspective associated with a program to become active whenever a program suspends.

To associate a particular perspective with a program, do the following:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Perspectives** preference page.
2. Select the type of application that you would like to associate a perspective with (for example, **Java Application**).
3. For each launch mode, select the desired perspective using the combo box. This will cause the perspective you choose to become active based on your preference settings (i.e. when a program is launched and/or when it suspends).

If the specified perspective is not open at the time it needs to be activated, that perspective is created.

Related concepts

Debugger
Remote debugging
Local debugging
Java perspectives

Related tasks

Running and debugging
Setting execution arguments
Launching a Java program

Related reference

Console view
Debug preferences
Debug view

Run Menu

Changing the appearance of the console view

To set the types of output (and their colors) in the Console view:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Console** preference page.
2. Checking the **Show when program writes to standard out** checkbox will make the Console view visible each time new output is written to the console from the program's standard output stream. If there is no Console view in the current perspective, one will be created.
3. Checking the **Show when program writes to standard err** checkbox will make the Console view visible each time new output is written to the console from the program's standard error stream. If there is no Console view in the current perspective, one will be created.
4. Click any of the color buttons to change the color for the corresponding text stream.

To set the fonts used in the Console view:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**General > Appearance > Colors and Fonts** preference page.
2. Select **Console font** from the **Debug** category and use the **Change...** button to change the font. (The **Detail Pane Text Font** can be used to change the font of the debugger's default **Detail Pane**).

Related concepts

Debugger
Java views

Related reference

Console view
Views and editors

Creating JAR Files

You can create and regenerate JAR files in the workbench.

- Related tasks

Creating a new JAR file
Regenerating a JAR file

- Related reference

JAR file exporter

Creating a New JAR File

To create a new JAR file in the workbench:

1. In the Package Explorer, you can optionally pre-select one or more Java elements to export. (These will be automatically selected in the [Image: /topic/org.eclipse.help/command_link.png]**JAR Package Specification** wizard page, described in Step 4.)
2. Either from the context menu or from the menu bar's **File** menu, select **Export**.
3. Expand the **Java** node and select **JAR file**. Click **Next**.
4. In the **JAR File Specification** page, select the resources that you want to export in the **Select the resources to export** field.
5. Select the appropriate checkbox to specify whether you want to **Export generated class files and resources** or **Export Java source files and resources**. **Note**: Selected resources are exported in both cases.
6. In the **Select the export destination** field, either type or click **Browse** to select a location for the JAR file.
7. Select or clear the **Compress the contents of the JAR file** checkbox.
8. Select or clear the **Overwrite existing files without warning** checkbox. If you clear this checkbox, then you will be prompted to confirm the replacement of each file that will be overwritten.
9. **Note**: The overwrite option is applied when writing the JAR file, the JAR description, and the manifest file.
10. You have two options:
    - Click **Finish** to create the JAR file immediately.
    - Click **Next** to use the JAR Packaging Options page to set advanced options, create a JAR description, or change the default manifest.

Related tasks

Setting advanced options
Defining the JAR file's manifest
Regenerating a JAR file

Related reference

JAR file exporter
Package Explorer

Setting Advanced Options

1. Follow the procedure for creating a JAR file, but click **Next** in the last step to go to the JAR Packaging Options page.
2. If you want to save the JAR file description, select the **Save the description of this JAR in the workspace** checkbox. A JAR file description can be used to regenerate a JAR file without using the wizard.
3. The compiler is able to generate CLASS files even when source contains errors. You have the option to exclude CLASS (but not source) files with compile errors. These files will be reported at the end, if reporting is enabled.
4. You can choose to exclude CLASS (but not source) files that have compile warnings. These files will be reported at the end.
   Note: This option does not automatically exclude class files with compile errors.
5. You can choose to include the source folder path by selecting the **Create source folder structure** checkbox.
6. Select the **Build projects if not built automatically** checkbox if you want the export to perform a build before creating the JAR file.
7. Click **Finish** to create the JAR file immediately or **Next** if you want to change the default manifest.

Related tasks

Creating a new JAR file
Defining the JAR file's manifest
Regenerating a JAR file

Related reference

JAR file exporter

Defining the JAR File's Manifest

You can either define the important parts of the JAR file manifest directly in the wizard or choose to use a manifest file that already exists in your workbench.

Creating a new manifest

1. Follow the procedure for creating a JAR file, but click **Next** in the last step to go to the JAR Packaging Options page.
2. Set any advanced options that you want to set, and then click **Next** again to go to the JAR Manifest Specification page.
3. If it is not already selected, click the **Generate the manifest file** button.
4. You can now choose to save the manifest in the workbench. This will save the manifest for later use. Click **Save the manifest in the workspace**, then click **Browse** next to the **Manifest file** field to specify a path and file name for the manifest.
5. If you decided to save the manifest file in the previous step and you chose to save the JAR description on the previous wizard page, then you can choose to reuse it in the JAR description (by selecting the **Reuse and save the manifest in the workspace** checkbox). This means that the saved file will be used when the JAR file is recreated from the JAR description.This option is useful if you want to modify or replace the manifest file before recreating the JAR file from the description.
6. You can choose to seal the JAR and optionally exclude some packages from being sealed or specify a list with sealed packages. By default, nothing is sealed.
7. Click the **Browse** button next to the **Main class** field to specify the entry point for your applications.
   Note: If your class is not in the list, then you forgot to select it at the beginning.
8. Click **Finish**. This will create the JAR, and optionally a JAR description and a manifest file.

Using an existing manifest

You can use an existing manifest file that already exists in your workbench.

1. Follow the procedure for creating a JAR file, but click **Next** in the last step to go to the JAR Packaging Options page.
2. Set any advanced options that you want to set, and the click **Next** again to go to the JAR Manifest Specification page.
3. Click the **Use existing manifest from workspace** radio button.
4. Click the **Browse** button to choose a manifest file from the workbench.
5. Click **Finish**. This will create the JAR and optionally a JAR description.

Related tasks

Creating a new JAR file
Setting advanced options
Regenerating a JAR file

Related reference

JAR file exporter

Regenerating a JAR File

You can use a JAR file description to regenerate a previously created JAR file.

1. Select one or more JAR file descriptions in your workbench.
2. From the selection's pop-up menu, select **Create JAR**. The JAR file(s) are regenerated.

Related tasks

Creating a new JAR file
Defining the JAR file's manifest

Related reference

JAR file exporter

Using the Local History

The JDT extends the workbench concept of a local history in three ways:

- A file can be replaced with an edition from the local history in the Package Explorer view.
- The JDT allows you to compare and/or replace individual Java elements (types and their members) with editions from the local history.
- The JDT allows you to restore Java elements (and files) deleted from the workbench that have been kept in the local history.

Note: Files and Java elements such as types and their members change in time. A 'snapshot' of what they look like a point in time (as saved in the local history) is called an edition.

Related concepts

Java development tools (JDT)
Java views

Related tasks

Replacing a Java element with a local history edition
Comparing a Java element with a local history edition
Restoring a deleted workbench element

Related reference

Package Explorer

Comparing a Java Element With a Local History Edition

1. Make sure that a Java view is visible.
2. Open a Java editor for the Java file in which you want to compare a Java element with an edition from the local history.
3. Activate the editor by clicking its tab in the editor area. The Outline view also displays the Java file. *Note: The Package Explorer can be configured to show or not show Java elements in files. Use the* ***Show Members in Package Explorer*** *checkbox on the* *[Image: /topic/org.eclipse.help/command_link.png]* ***Java > Appearance*** *preference page to set your preference.*
4. Select the element that you want to compare in the Outline or the Package Explorer.
5. From the element's pop-up menu, select **Compare With > Element from Local History**.
6. The **History** view opens and shows all editions of the selected element available in the local history.
7. Select an edition in the history view to see the differences between the selected edition and the edition in the workbench in a comparison editor.
8. If you are done with the comparison, click close the comparison editor.

Related concepts

Java views
Java editor

Related tasks

Using the local history

Related reference

Java outline

Replacing a Java Element With a Local History Edition

1. Make sure that a Java view is visible.
2. Open a Java editor for the Java file in which you want to replace a Java element with an edition from the local history.
3. Activate the editor by clicking its tab in the editor area. The Outline view also displays the Java file. *Note: The Package Explorer can be configured to show or not show Java elements in files. Use the* ***Show Members in Package Explorer*** *checkbox on the* *[Image: /topic/org.eclipse.help/command_link.png]* ***Java > Appearance*** *preference page to set your preference.*
4. Select the element that you want to replace in the Outline or the Package Explorer.
5. From the element's pop-up menu, select **Replace With > Element from Local History**.
6. In the upper pane of the resulting dialog, all available editions of the selected element in the local history are displayed.
7. Select an edition in the upper pane to view the differences between the selected edition and the edition in the workbench.
8. When you have identified the edition with which you want to replace the existing Java element, click **Replace**.
9. The local history edition replaces the current one in the editor. *Note: The changed compilation unit has not yet been saved at this point.*

Related concepts

Java views
Java editor

Related tasks

Using the local history

Related reference

Java outline

Restoring a Deleted Workbench Element

1. Ensure that a Java view that show Java elements inside files (such as the Outline view) is visible.
2. Open the compilation unit to which you want to add a previously removed Java element from the local history.
3. Activate the editor by clicking its tab in the editor area, and the Java view shows the content of the Java file.
4. In the Java view, select the element to whose container type you want to restore the deleted element.
5. From the type's pop-up menu in the Java view, select **Restore from Local History**.
6. In the upper left pane of the resulting dialog, all available editions of the selected element in the local history are displayed.
7. In the left pane check all elements that you want to replace.
8. For every checked element select an edition in the right hand pane and view its content in the bottom pane.
9. When you have identified the edition that you want to restore, press **Restore**. The local history editions are loaded into the editor.

- Related concepts

Java editor

- Related tasks

Using the local history

- Related reference

Java outline

Externalizing Strings

Contents

Finding Strings to Externalize

To find strings to externalize:

- In a Java view (e.g. Package Explorer), select a set of packages, source folders or projects.
- From the menu bar, select Source > Externalize Strings...
- A dialog comes up with a list of all compilation units that have some non-externalized strings
- In the dialog, you can double click on a listed compilation unit or press the **Externalize** button to open the Externalize Strings wizard

Related concepts

String externalization

Related tasks

Finding unused and incorrectly used keys in property files

Related reference

Externalize strings wizard
Source menu

Finding Unused and Incorrectly Used Keys in Property Files

Finding unused and incorrectly used keys in a property file:

- in a Java view (e.g. Package Explorer), select a set of packages, source folders, projects, message property files, or resource bundle accessor classes
- from the menu bar, select Source > Find Broken Externalized Strings

After the search is finished, the Search Result view displays a list of unused keys in the properties files and all incorrect references to non-existing keys.

Related concepts

String externalization

Related tasks

Externalizing strings
Finding strings to externalize

Related reference

Externalize strings wizard
Source menu

Opening an Editor for a Selected Element

You can select the name of a type, method, or field in the Java source editor or in the scrapbook and open an editor on the definition of the element.

1. In the Java editor, select the name of a type, method, or field. You can also just click into the name once.
2. Do one of the following:
   - From the menu bar, select **Navigate** > **Open Declaration**
   - From the editor's pop-up menu, select **Open Declaration**
   - Press **F3**

   or

1. Hold down the **Ctrl** key.
2. In the Java editor, move the mouse over the name of a type, method, or field until the name becomes underlined.
3. Click the hyperlink once.

If there are multiple definitions of the same name, a dialog is shown, and you can select one definition that you want to open. An editor opens containing the selected element.

Related concepts

Java editor

Related reference

Navigate menu
Views and editors

Showing an Element in the Package Explorer View

You can reveal an element's location in the Package Explorer view

- Select a Java element or activate a Java editor.
- From the menu bar, select **Navigate** > **Show In** > **Package Explorer**. If the Package Explorer is not already open, then it opens in the current perspective. The workbench navigates to the selected element (or the edited compilation unit).
- From the Java editor's pop-up menu, select **Show In >** **Package Explorer**. The currently edited compilation unit will be revealed.

Note: The element might not be revealed if Package Explorer filters are active or the **Show Members in Package Explorer** preference is cleared on the [Image: /topic/org.eclipse.help/command_link.png]**Java >** **Appearance** preference page.

Related concepts

Java views

Related tasks

Setting execution arguments

Related reference

Java preference page
Package Explorer

Opening a Type in the Package Explorer View

You can open the Package Explorer on any type that is included on a project's class path.

1. From the menu bar, select **Navigate > Go To > Type.** The Go to Type dialog opens.
2. In the **Choose a type** field, begin typing an expression to narrow the list of available types, using wildcards as needed. As you type, the list is filtered to display only types that match the current expression.
3. In the **Matching items** list, select a type. Hint: you can press the **Down** key to move to the first type.
4. Click **OK** when you are done. The selected type is displayed in the Package Explorer.

Note: The Goto Type dialog maintains a history of recently opened types. These are shown when the dialog is opened and stay above a separator line when you start to type a filter expression.

Note: Revealing may not be possible if Package Explorer filters are applied.

Related concepts

Java development tools (JDT)

Related tasks

Showing a type's compilation unit in the Package Explorer

Related reference

Navigate actions
Package Explorer

Opening an Editor on a Type

You can open an editor on any type in the workbench.

1. Press **Ctrl+Shift+T** or, select **Navigate** > **Open Type** from the menu bar. The Open Type dialog opens.
2. In the **Enter type name prefix or pattern** field, begin typing an expression to narrow the list of available types, using wildcards as needed. As you type, the list is filtered to display only types that match the current expression. CamelCase notation is also supported; that means you only need to enter the capital letters of the type name.
3. In the **Matching items** list, select a type. Hint: you can press the **Down** key to move to the first type.
4. Click **OK** when you are done. An editor opens on the selected type.

Note: The Open Type dialog maintains a history of recently opened types. These are shown when the dialog is opened and stay above a separator line when you start to type a filter expression.

Note: If you open a type from a CLASS or JAR file, you will see a special editor showing only method signatures unless you have attached source to it.

Related concepts

Java editor

Related tasks

Opening an editor for a selected element

Related reference

Open type dialog
Navigate actions
Views and editors

Opening a Package

To reveal a package in the tree of the Package Explorer:

1. Select **Navigate** > **Go To** > **Package** from the menu bar. The Go to Package dialog opens.
2. Type a package name in the **Choose a package** field to narrow the list of available packages, using wildcards as needed. As you type, the list is filtered to display only packages that match the current expression.
3. Select a package from the list, then click **OK**. The selected package is displayed in the Package Explorer.

Related concepts

Java views

Related reference

Navigate actions

Package Explorer

Working with JREs

You can install as many different Java Runtime Environments (JREs) as you like. JREs are used to run and debug java programs. Your JREs are managed on the Installed JREs preference page.

A JRE definition consists of:

- The type of the JRE (e.g. Standard VM or Standard 1.x.x VM)
- A name
- The location where the JRE is installed
- The system libraries containing the Java system classes (like java.lang.Object). Optionally, the system libraries can be associated with the source file containing the source for the classes in the JRE's CLASS files and a javadoc location (URL).
- Other information needed by the VM to build, run and debug applications.

You can switch the default JRE for the workbench. The default JRE is used by default when building, running, and debugging applications. Alternatively, projects may specify a specific JRE that they should be built and run with.

Related concepts

Java development tools (JDT)

Related tasks

Adding a new JRE definition
Deleting a JRE definition
Choosing a default JRE
Choosing a JRE for launching a project

Related reference

Installed JREs preference page

Assigning the default JRE for the workbench

The default JRE is used for compiling and launching Java programs in all projects unless you specifically override the default JRE on a project's build path or on a launch configuration.

Here is how you can change the default JRE:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs** preference page.
2. Check the box on the line for the JRE that you want to assign as the default JRE in your workbench. If the JRE you want to assign as the default does not appear in the list, you must add it.
3. Click **OK**.

Note: Changing the default JRE may cause a build to occur if you have auto build enabled (**Project > Build Automatically** or in the [Image: /topic/org.eclipse.help/command_link.png]**General > Workspace** preference page).

Related concepts

Java development tools (JDT)

Related tasks

Working with JREs
Adding a new JRE definition
Deleting a JRE definition
Choosing a JRE for launching a project

Related reference

Installed JREs preference page

Adding a new JRE definition

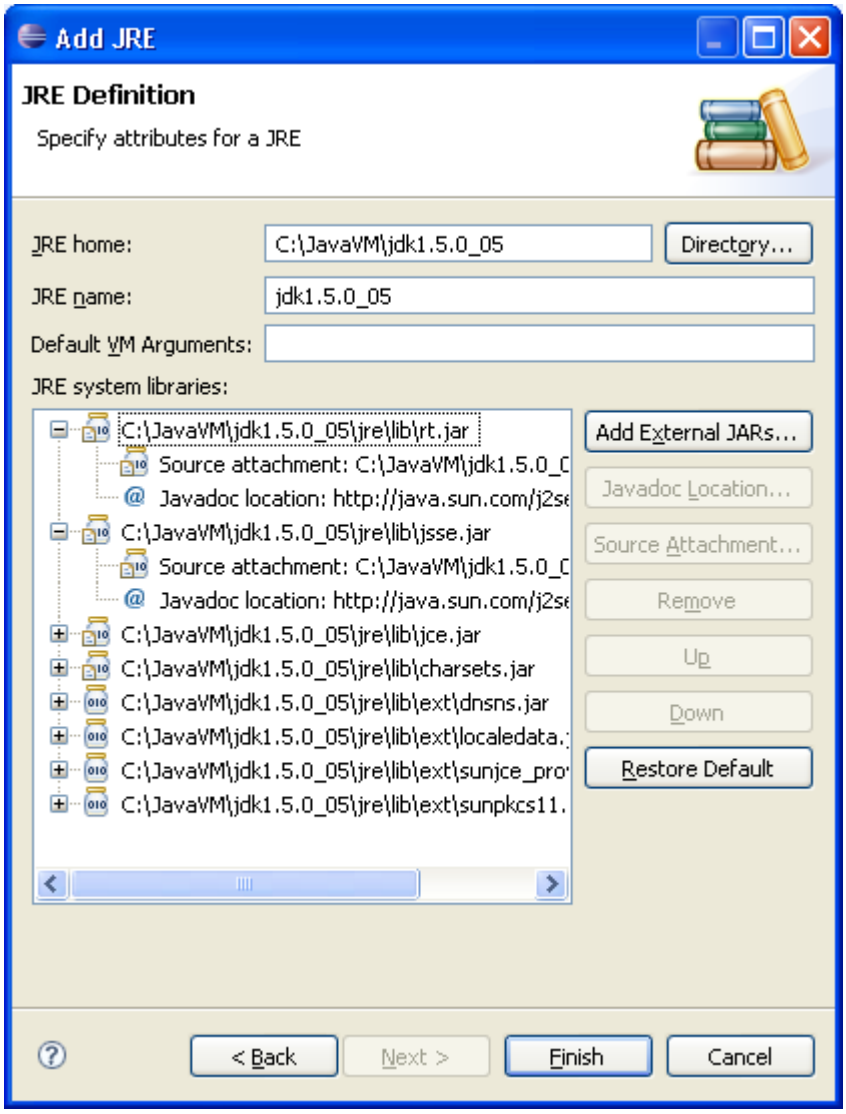
In your workspace you can have any number of JRE definitions. You can also add as many JRE definitions as you want.

To add a new JRE definition:

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs** preference page.
2. Select the **Add...** button. The Add JRE wizard opens.
3. You will have to select the type of JRE you would like to create:
   - **Standard VM** - You will select a folder where the JRE is installed. The details of the JRE will be determined by scanning the location. You can further customize the JRE install, specifying vm arguments, source attachments, etc.
   - **Standard VM 1.x.x** - Same options as a standard VM install, but the install is customized to handle the 1.x.x install
   - **Execution Environment Description** - You will select an execution environment (EE) file that describes all the details required to set up the JRE. You will be able to customize some aspects of the install.

Standard VM:

1. In the **JRE home** field, click **Browse...** to select a path to the root directory of a JRE installation (usually the directory containing the *bin* and *lib* directories for the JRE). You may also type a folder name into this field.
2. In the **JRE name** field, type a name for the new JRE definition. All JREs of the same type must have a unique name.
3. In the **Default VM Arguments** field, you can add/edit the default arguments that will be passed to the VM when launching.
4. The default libraries appear for the JRE in the JRE system libraries list. You can modify the libraries as desired.
   - Source and javadoc locations can be specified for the referenced JARs. The Javadoc location is used by the Javadoc export wizard as a default value and by the 'Open External Javadoc' action.
   - Jars can be added, removed, and reordered.
5. Click **Finish** when you are done.



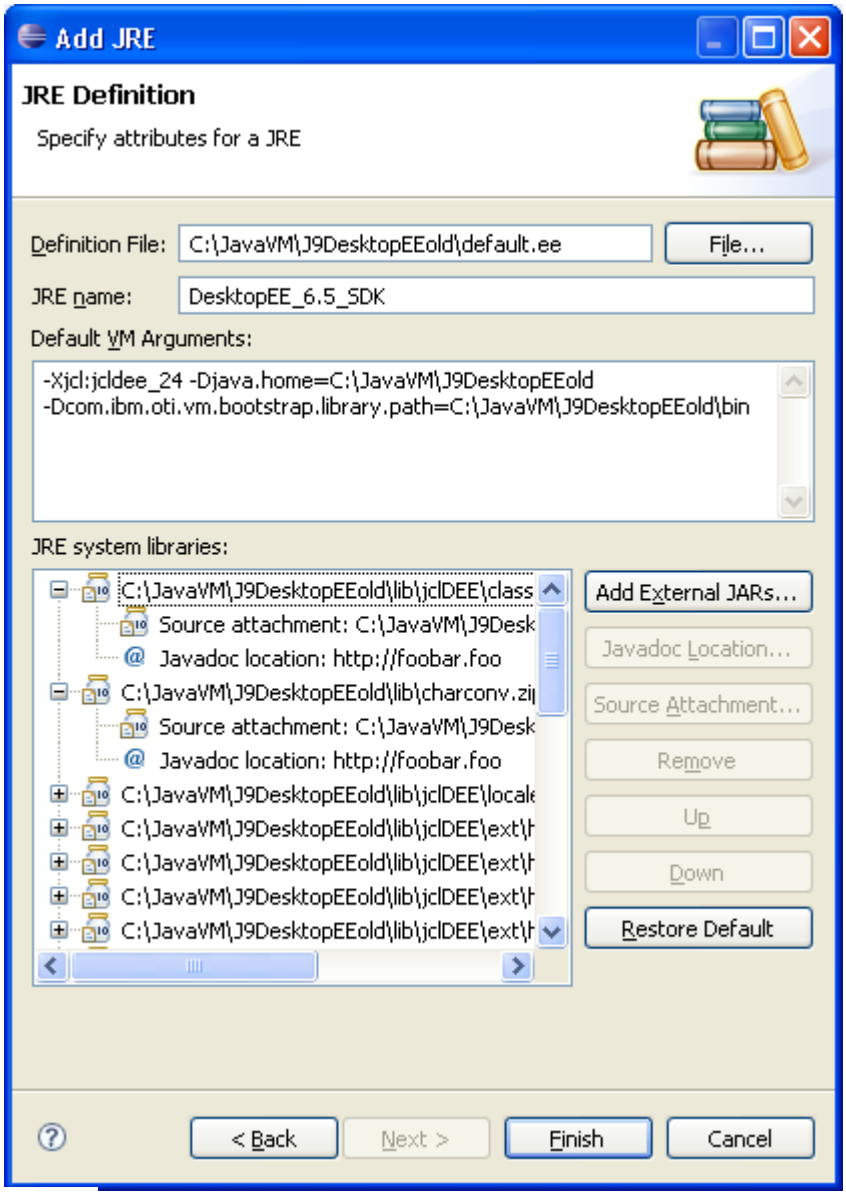
Standard 1.1.x VM:

Follow the same steps as for **Standard VM** JREs.

Execution Environment Files:

For more information on EE File structure see the Wiki Page.

1. In the **Definition home directory** field, click **File...** to select a execution environment description file to load the JRE installation from. You may also type a file path into this field.
2. In the **JRE name** field, type a name for the new JRE definition. All JREs of the same type must have a unique name.
3. In the **Default VM Arguments** field, you can add/edit the default arguments that will be passed to the VM when launching.
4. The default libraries appear for the JRE in the JRE system libraries list. You can modify the libraries as desired.
   - Source and javadoc locations can be specified for the referenced JARs. The Javadoc location is used by the Javadoc export wizard as a default value and by the 'Open External Javadoc' action.
   - Jars can be added, removed, and reordered.
5. Click **Finish** when you are done.



Related concepts

Java development tools (JDT)

Related tasks

Working with JREs
Deleting a JRE definition
Choosing a default JRE
Choosing a JRE for launching a project

Related reference

Installed JREs preference page

Choosing a JRE for a launch configuration

Instead of using the default JRE for running and debugging all **Java Application** launch configurations, you can specifically assign a JRE for launching an individual configuration.

1. With a Java Application configuration selected in the [Image: /topic/org.eclipse.help/command_link.png]**Launch Configuration Dialog**, select the JRE tab.
2. In the list of available JREs, select the JRE you want to use to launch this configuration and click **Apply**, then **Run** or **Debug**.

Note: Changing the JRE used for running does not affect the way Java source is compiled. You can adjust the build path to compile against custom libraries.

Related concepts

Java development tools (JDT)

Related tasks

Working with JREs
Adding a new JRE definition
Deleting a JRE definition
Choosing a default JRE

Related reference

Installed JREs preference page

Deleting a JRE definition

You can delete Java runtime environment definitions that are available for executing Java programs in the workbench.

1. Open the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs** preference page.
2. Select the definition you want to delete and click **Remove**.
3. Check the box for the definition that you want to use as the default JRE for the workbench.

Related concepts

Java development tools (JDT)

Related tasks

Working with JREs
Adding a new JRE definition
Choosing a default JRE
Choosing a JRE for launching a project

Related reference

Installed JREs preference page

Running and Debugging

You may launch your Java programs from the workbench. The programs may be launched in either run or debug mode.

- In run mode, the program executes, but the execution may not be suspended or examined.
- In debug mode, execution may be suspended and resumed, variables may be inspected, and expressions may be evaluated.

Related concepts

Debugger
Remote debugging
Local debugging

Related tasks

Changing debugger launch options
Choosing a JRE for launching a project
Creating a Java scrapbook page
Disconnecting from a VM
Launching a Java program
Local debugging
Preparing to debug
Re-launching a program
Remote debugging
Resuming the execution of suspended threads
Setting execution arguments
Stepping through the execution of a program
Suspending threads

Related reference

Run Menu
Debug view
Debug Preferences
Console Preferences

Adding Line Breakpoints

Line breakpoints are set on an executable line of a program.

1. In the editor area, open the file where you want to add the breakpoint.
2. Directly to the left of the line where you want to add the breakpoint, open the marker bar (vertical ruler) pop-up menu and select **Toggle Breakpoint.** You can also double-click on the marker bar next to the source code line. A new breakpoint marker appears on the marker bar, directly to the left of the line where you added the breakpoint.

Also, the new breakpoint appears in the **Breakpoints View** list.

While the breakpoint is enabled, thread execution suspends before that line of code is executed. The debugger selects the thread that has suspended and displays the thread's stack frames. The line where the breakpoint was set is highlighted in the editor in the **Debug Perspective**.

Related concepts

Debugger
Java perspectives
Java editor

Related tasks

Applying hit counts
Catching Java exceptions
Removing breakpoints
Enabling and disabling breakpoints
Managing conditional breakpoints
Setting method breakpoints
Stepping through the execution of a program

Related reference

Breakpoints view

Removing Line Breakpoints

Breakpoints can be easily removed when you no longer need them.

1. In the editor area, open the file where you want to remove the breakpoint.

2. Directly to the left of the line where you want to remove the breakpoint, open the marker bar pop-up menu and select **Toggle Breakpoint**. The breakpoint is removed from the workbench. You can also double-click directly on the breakpoint icon to remove it.

Breakpoints can also be removed in the **Breakpoints View**. Select the breakpoint(s) to be removed and from the context menu select **Remove**.

All breakpoints can be removed from the workbench using the **Remove All** action in the context menu of the **Breakpoints View**. If you find yourself frequently adding and removing a breakpoint in the same place, consider disabling the breakpoint when you don't need it (using **Disable Breakpoint** in the breakpoint context menu or the **Breakpoints View**) and enabling it when needed again.

Related concepts

Debugger
Java perspectives
Java editor

Related tasks

Adding breakpoints
Enabling and disabling breakpoints
Applying hit counts
Catching Java exceptions
Managing conditional breakpoints
Setting method breakpoints
Stepping through the execution of a program

Related reference

Breakpoints view

Enabling and disabling breakpoints

Breakpoints can be enabled and disabled as needed. When a breakpoint is enabled, thread execution suspends before that line of code is executed. When a breakpoint is disabled, thread execution is not suspended by the presence of the breakpoint.

To disable a breakpoint in the **Breakpoints View**:

1. Open the breakpoint's context menu and select **Disable**, or deselect the breakpoint's checkbox.
2. The breakpoint image will change to a white circle and its checkbox will be empty.

To disable a breakpoint in the marker bar of an editor:

1. Open the breakpoint's context menu and select **Disable Breakpoint**.
2. The breakpoint image will change to a white circle.

To enable the breakpoint in the **Breakpoints View**:

1. Open the breakpoint's context menu and select **Enable**, or select the breakpoint's checkbox.
2. The breakpoint image will change back to a blue circle, and its checkbox will be checked.

To enable a breakpoint in the marker bar of an editor:

1. Open the breakpoint's context menu and select **Enable Breakpoint**.
2. The breakpoint image will change to a white circle.

Related concepts

Debugger
Java perspectives
Java editor

Related tasks

Applying hit counts
Catching Java exceptions
Removing breakpoints
Setting method breakpoints
Managing conditional breakpoints
Stepping through the execution of a program

Related reference

Breakpoints view

Setting Method Breakpoints

Method breakpoints are used when working with types that have no source code (binary types).

1. Open the class in the **Outline View**, and select the method where you want to add a method breakpoint.
2. From the method's pop-up menu, select **Toggle Method Breakpoint**.
3. A breakpoint appears in the **Breakpoints View**. If source exists for the class, then a breakpoint also appears in the marker bar in the file's editor for the method that was selected.
4. While the breakpoint is enabled, thread execution suspends when the method is entered, before any line in the method is executed.

Method breakpoints can also be setup to break on method exit. In the Breakpoints view, select the breakpoint and toggle the **Exit** item in its context menu.
Method breakpoints can be removed, enabled, and disabled just like line breakpoints.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Enabling and disabling breakpoints
Applying hit counts
Catching Java exceptions

Related reference

Breakpoints view
Java outline

Applying Hit Counts

A hit count can be applied to line breakpoints, exception breakpoints, watchpoints and method breakpoints. When a hit count is applied to a breakpoint, the breakpoint suspends execution of a thread the *n*th time it is hit, but never again, until it is re-enabled or the hit count is changed or disabled.

To set a hit count on a breakpoint:

1. Select the breakpoint to which a hit count is to be added.
2. From the breakpoint's pop-up menu, select **Hit Count**.
3. In the **Enter the new hit count for the breakpoint** field, type the number of times you want to hit the breakpoint before suspending execution.

Note: When the breakpoint is hit for the *n*th time, the thread that hit the breakpoint suspends. The breakpoint is disabled until either it is re-enabled or its hit count is changed.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Enabling and disabling breakpoints
Setting method breakpoints

Related reference

Breakpoints view

Managing conditional breakpoints

A conditional expression can be applied to a line breakpoint such that the breakpoint suspends execution of a thread in one of these cases:
- when the result of the expression is true
- when the result of the expression changes

A conditional expression can contain arbitrary Java code and may contain more than one statement, allowing breakpoint conditions to implement features like tracing. For example, a condition can execute a print statement and then return a hard coded value to never suspend ("`System.out.println(...); return false;`").

To set a condition on a breakpoint:

1. Find the breakpoint to which an enabling condition is to be applied (in the **Breakpoints View** or in the editor marker bar).
2. From the breakpoint's pop-up menu, select **Breakpoint Properties...**. The Breakpoint properties dialog will open.
3. In the properties dialog, check the **Enable Condition** checkbox.
4. In the **Condition** field enter the expression for the breakpoint condition.
5. Do one of the following:
   - If you want the breakpoint to stop every time the condition evaluates to *true*, select the **condition is 'true'** option. The expression provided must be a boolean expression.
   - If you want the breakpoint to stop only when the result of the condition changes, select the **value of condition changes** option.
6. Select **OK** to close the dialog and commit the changes. While the breakpoint is enabled, thread execution suspends before that line of code is executed if the breakpoint condition evaluates to *true*.

A conditional breakpoint has a question mark overlay on the breakpoint icon.

Related concepts

Debugger
Java perspectives
Java editor

Related tasks

Adding breakpoints
Applying hit counts
Catching Java exceptions
Removing breakpoints
Setting method breakpoints
Stepping through the execution of a program

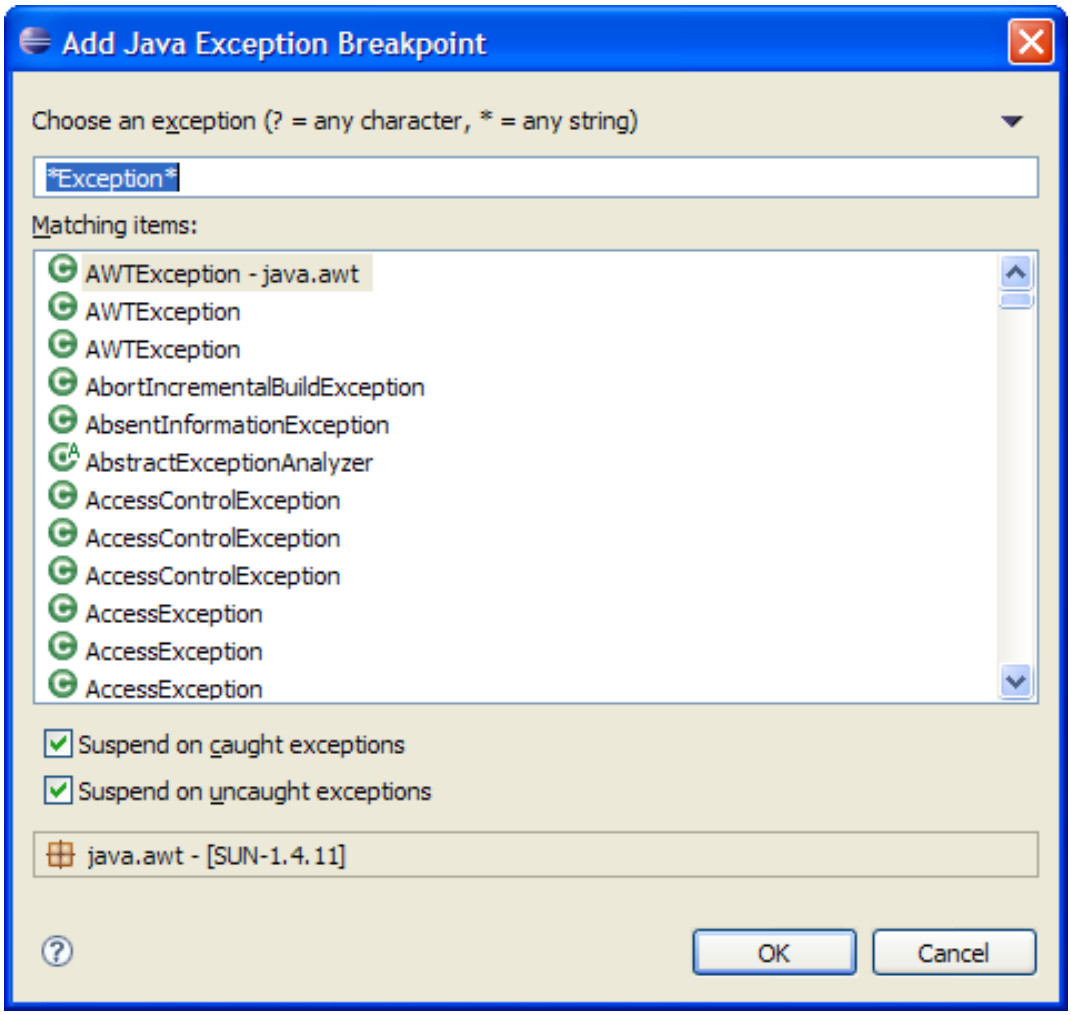
Related reference

Breakpoints view

Catching Java Exceptions

It is possible to suspend the execution of thread when an exception is thrown by specifying an exception breakpoint. Execution can be suspended at locations where the exception is uncaught, caught, or both.

1. Choose **Add Java Exception Breakpoint** from the **Breakpoints View** or the workbench **Run** menu.
2. A dialog listing all of the available exceptions is presented (see the following figure).



3. Type the name of the exception you want to catch or select it from the list.

4. At the bottom of the page, use the checkboxes to specify how you want execution to suspend at locations where the exception is thrown.
   - Select **Caught** if you want execution to suspend at locations where the exception is thrown and caught by a `catch` clause.
   - Select **Uncaught** if you want execution to suspend at locations where the exception is uncaught.

Note: Exception breakpoints can be enabled and disabled and have hit counts just like regular breakpoints.

Related concepts

Java development tools (JDT)
Breakpoints

- Related tasks

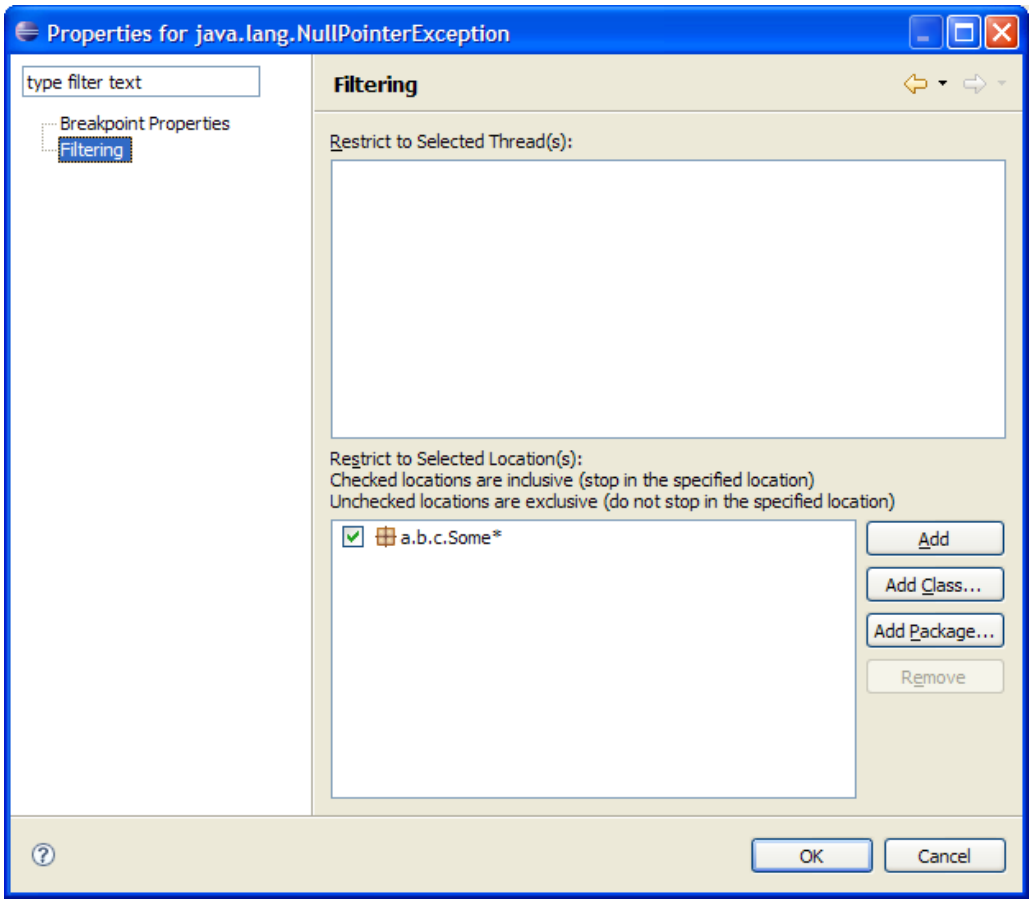
Creating Exception Breakpoint Filters
Suspending threads
Adding breakpoints
Removing breakpoints
Enabling and disabling breakpoints
Setting method breakpoints

- Related reference

Breakpoints view
Add Java exception breakpoint

Create Exception Breakpoint Filtering

Type name and package name filtering can be set up for Java exception breakpoints. This can be done from an exception breakpoints' **Filtering** properties page, as shown in the following figure.

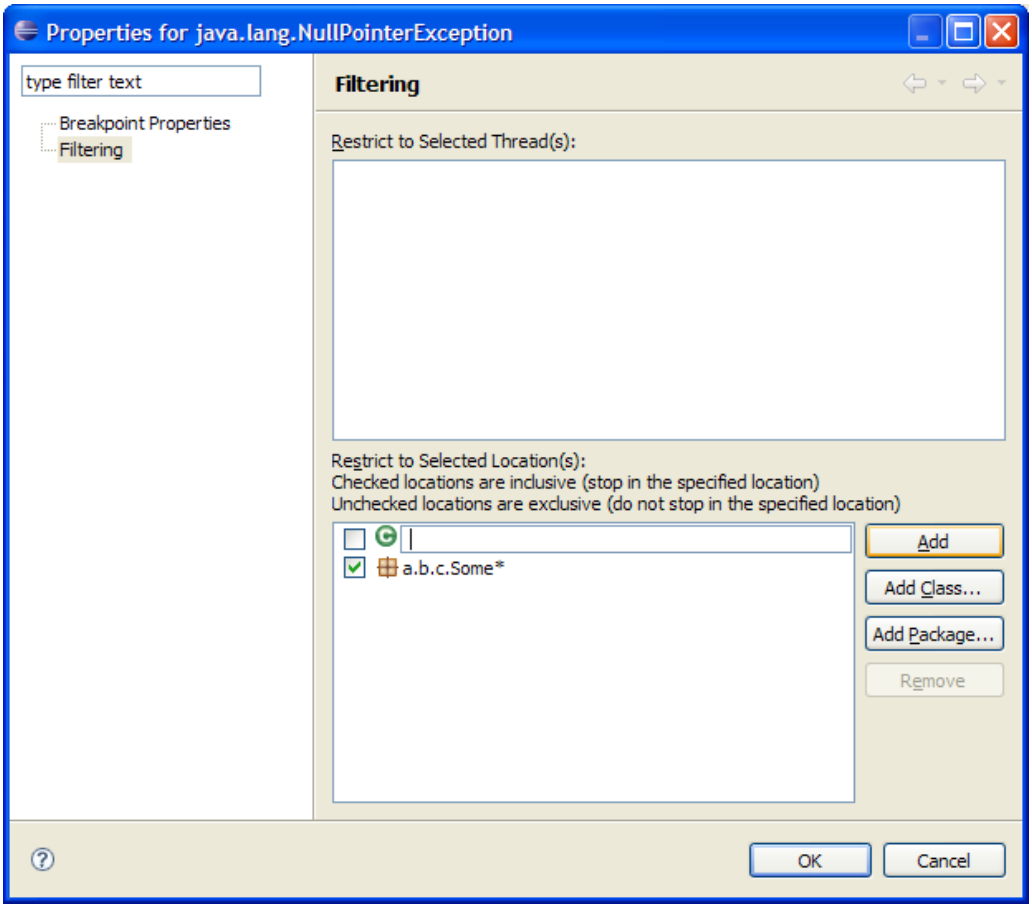


Using the Filtering properties page there are four ways to add filtering for a Java exception breakpoint:

1. You can select specific threads to restrict the breakpoint to (which must be done while in a debug session)
2. You can use the **Add Class** button to select a class via the Type Selection Dialog
3. You can use the **Add Package** button to select a package using a filtered Type Selection Dialog
4. You can use the **Add** button to define your own pattern to match as a class and/or package filter

Defining Your Own Filter Pattern

Using the Add button mentioned above, you can define your own pattern to be used as a filter for a Java exception breakpoint. Once pressed, you can then enter any expression in the new space in the **Selected Locations** list, as shown in the following figure.



There are some rules to follow when creating your own pattern.
1. Your pattern can only contain a '*' at the end
2. Your pattern must be fully qualified, E.g. *a.b.c.MyClass*
3. Your pattern cannot have spaces in it

Examples

- *a.b.c** - would match everything in the package a.b.c
- *a.b.c.My** - would match anything in the a.b.c package that started with 'My'
- *My** - would match anything in the default package that started with 'My'

Preparing to Debug

You can make your programs easier to debug by following these guidelines:

- Where possible, do not put multiple statements on a single line, because some debugger features operate on a line basis. For example, you cannot step over or set line breakpoints on more than one statement on the same line.
- Attach source code to JAR files if you have the source code.

Related concepts

Debugger
Remote debugging
Local debugging

Related tasks

Changing debugger launch options
Running and debugging

Related reference

Debug preferences
Debug view
Run Menu

Launching a Java program in debug mode

Launching a program in debug mode allows you to suspend and resume the program, inspect variables, and evaluate expressions using the debugger.

To launch a Java program in debug mode,

1. In the **Package Explorer**, select the Java compilation unit or class file with the main method you want to launch.
2. Press the **Debug** [] button in the workbench toolbar or select **Run > Debug** from the workbench menu bar. Alternatively you can select **Run > Debug As > Java Application** from the workbench menu bar, or select **Debug As > Java Application** in the **Package Explorer** pop-up menu or in the drop-down menu on the **Debug** tool bar button.
3. Your program is now launched and the launched process appears in the Debug view.

If you want your program to stop in the *main* method so that you can step through its complete execution, create a **Java Application** launch configuration and check the **Stop in main** checkbox on the **Main** tab.

You can also debug a Java program by selecting a project instead of the compilation unit or class file. You will be prompted to select a class from those classes that define a *main* method. (If only one class with a main method is found in the project, that class is launched as if you selected it.)

Related concepts

Java views
Java editor
Debugger

Related tasks

Connecting to a remote VM with the Java Remote Application launcher
Re-launching a program
Running and debugging
Setting execution arguments
Stepping through the execution of a program

Related reference

Debug view
Package Explorer

Suspending Threads

To suspend an executing thread:

1. Select the thread in the **Debug View**.
2. Click the **Suspend** button [ ] in the view toolbar.

The thread suspends its execution. The current call stack for the thread is displayed, and the current line of execution is highlighted in the editor in the **Debug Perspective**.

When a thread suspends, the top stack frame of the thread is automatically selected. The **Variables View** shows the stack frame's variables and their values. Complex variables can be further examined by expanding them to show the values of their members.

When a thread is suspended and the cursor is hovered over a variable in the Java editor, the value of that variable is displayed.

Related concepts

Debugger
Java editor
Java perspectives

Related tasks

Catching Java exceptions
Evaluating expressions
Resuming the execution of suspended threads
Stepping through the execution of a program

Related reference

Debug view
Resume command
Suspend command
Variables view

Resuming the execution of suspended threads

To resume the execution of a suspended threads:

1. Select the thread or its stack frame in the **Debug View**.
2. Click the **Resume** button [ ] in the view toolbar (or press the ***F8*** key).

The thread resumes its execution and stack frames are no longer displayed for the thread. The **Variables View** is also cleared.

Related concepts

Debugger

Related tasks

Evaluating expressions
Stepping through the execution of a program
Suspending threads

Related reference

Debug view
Resume command
Suspend command
Variables View

Stepping through the execution of a Java program

When a thread is suspended, the step controls can be used to step through the execution of the program line-by-line. If a breakpoint is encountered while performing a step operation, the execution will suspend at the breakpoint and the step operation is ended.

Step over

1. Select a stack frame in the **Debug View**. The current line of execution in that stack frame is highlighted in the editor in the **Debug Perspective**.
2. Click the **Step Over** button [ ] in the view toolbar, or press the ***F6*** key. The currently-selected line is executed and suspends on the next executable line.

Step into

1. Select a stack frame in the **Debug View**. The current line of execution in the selected frame is highlighted in the editor in the **Debug Perspective**.
2. Click the **Step Into** button [ ] in the view toolbar, or press the ***F5*** key. The next expression on the currently-selected line to be executed is invoked, and execution suspends at the next executable line in the method that is invoked.

Step into Selection

1. Select a stack frame in the **Debug View**. The current line of execution in the selected frame is highlighted in the editor in the **Debug Perspective**.
2. In the **Java Editor**, within the current line of execution, place the cursor on the name of a method that you would like to step into.
3. Click the **Step into Selection** action in the Run menu or Java editor context menu, or press the ***Ctrl-F5*** key. Execution resumes until the selected method is invoked.

Step with filters

1. Toggle the **Use Step Filters** button [ ] in the Debug view toolbar, or use ***Shift+F5***. When the action is toggled on, each of the step actions (over, into, return) will apply the set of step filters which are defined in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Step Filtering** preference page. When a step action is invoked, stepping will continue until an unfiltered location is reached or a breakpoint is encountered.

Step Return

1. Select a stack frame in the **Debug View**. The current line of execution in the selected frame is highlighted in the editor in the **Debug Perspective**.
2. Click the **Step Return** button [ ] in the view toolbar or press the ***F7*** key. Execution resumes until the next return statement in the current method is executed, and execution suspends on the next executable line.

Run to line

When a thread is suspended, it is possible to resume execution until a specified line is executed. This is a convenient way to suspend execution at a line without setting a breakpoint.

1. Place your cursor on the line at which you want the program to run.

2. Select the **Run to Line** command [] from the pop-up menu or use ***Ctrl+R***. Program execution is resumed and suspends just before the specified line is to be executed.

It is possible that the line will never be hit and that the program will not suspend. Breakpoints and exceptions can cause the thread to suspend before reaching the specified line.

Related concepts

Breakpoints
Java perspectives

Related tasks

Adding breakpoints
Launching a Java program
Resuming the execution of suspended threads
Running and debugging
Setting execution arguments
Suspending threads

Related reference

Debug view

Inspecting values

When a stack frame is selected, you can see the visible variables in that stack frame in the **Variables View**.

The Variables view shows the value of primitive types. Complex variables can be examined by expanding them to show their members.

Related reference

Inspecting Values in the Expressions View
Inspecting Values in the Variables View
Inspecting Values in other views

Evaluating expressions

When the VM suspends a thread (due to hitting a breakpoint or stepping through code), you can evaluate expressions in the context of a stack frame.

1. Select the stack frame in which an evaluation is to be performed. The evaluation context will be the currently selected variable in the view. If no variable is selected, the selected stack frame will be the context.
2. Expressions can be entered and evaluated in the following areas:
   - **Display View**
   - **Detail Pane** (in the **Expressions View** and **Variables View**)
   - **Java Editor** when it is displaying source and it is not read-only
3. Select the expression to be evaluated and select **Display**, **Inspect** or **Execute** from the context pop-up menu. The result of a **Display** or **Inspect** evaluation is shown in a popup window. Note that **Execute** does not display a result - the expression is just executed.
4. The result popup window can be dismissed by clicking outside of the popup window or by pressing ***Esc***. The result can be persisted to the Display view (if **Display** was chosen) or Expressions view (if **Inspect** was chosen) by pressing the key sequence shown at the bottom of the popup window. For example, to move the result of an **Inspect** evaluation to the Expressions view press ***Ctrl-Shift-I***. Note that when the **Display** action is used from the Display view the result is written to the Display view rather than a popup

Note: Evaluations cannot be performed in threads that have been manually suspended.

Related concepts

Debugger
Java editor

Related tasks

Suspending threads
Resuming the execution of suspended threads

Related reference

Detail Pane
Display view
Expressions view
Variables view

Using the remote Java application launch configuration

The **Remote Java Application** launch configuration should be used when debugging an application that is running on a remote VM. Since the application is started on the remote system, the launch configuration does not specify the usual information about the JRE, program arguments, or VM arguments. Instead, information about connecting to the application is supplied.

To create a **Remote Java Application** launch configuration, do the following:

1. Select [Image: /topic/org.eclipse.help/command_link.png]**Run > Debug Configurations...** from the workbench menu bar (or **Debug Configurations...** from the drop-down menu on the **Debug** tool bar button) to show the launch configuration dialog.
2. Select the **Remote Java Application** in the list of configuration types on the left.
3. Click the **New** toolbar button. A new remote launch configuration is created and three tabs are shown: **Connect**, **Source**, and **Common**.
4. In the **Project** field of the **Connect** tab, type or browse to select the project to use as a reference for the launch (for source lookup). A project does not need to be specified.
5. The **Connection Type** field of the **Connect** tab allows you to choose how you will connect to the virtual machine. In most cases, you will be attaching to the vm at a specific location, in which case select *Standard (Socket Attach)*. the rest of these instructions assume you have chosen this option. The *Standard (Socket Listen)* connection type creates a launch that will listen for incoming connections from a remote VM. You will need to specify a port that the launch will listen at.
6. In the **Host** field of the **Connect** tab, type the IP address or domain name of the host where the Java program is running.
   If the program is running on the same machine as the workbench, type *localhost*.
7. In the **Port** field of the **Connect** tab, type the port where the remote VM is accepting connections. Generally, this port is specified when the remote VM is launched.
8. The **Allow termination of remote VM** flag is a toggle that determines whether the **Terminate** command is enabled in the debugger. Select this option if you want to be able to terminate the VM to which you are connecting.
9. Click **Debug**. The launch attempts to connect to a VM at the specified address and port, and the result is displayed in the Debug view. If the launcher is unable to connect to a VM at the specified address, an error message appears.

Specific instructions for setting up the remote VM should be obtained from your VM provider.

Related concepts

Debugger
Remote Debugging

Related tasks

Launching a Java program
Disconnecting from a VM
Setting execution arguments
- Related reference

Debug view

Disconnecting from a VM

To disconnect from a VM that was connected to with a **Remote Java Application** launch configuration:

1. In the **Debug View**, select the launch.
2. Click the **Disconnect** button [] in the view toolbar.

Communication with the VM is terminated, and all threads in the remote VM are resumed. Although the remote VM continues to execute, the debug session is now terminated.

Related concepts

Debugger

Related tasks

Connecting to a remote VM with the Remote Java application launch configuration
Running and debugging

Related reference

Debug view

Creating a Java application launch configuration

When you choose **Run >Run As >Java Application** to launch your class, you are running your class using a generic **Java Application** launch configuration that derives most of the launch parameters from your Java project and your workbench preferences. In some cases, you will want to override the derived parameters or specify additional arguments.

You do this by creating your own **Java Application** launch configuration.

1. Select [Image: /topic/org.eclipse.help/command_link.png]**Run > Run Configurations...** or [Image: /topic/org.eclipse.help/command_link.png]**Run > Debug Configurations...** from the workbench menu bar. This opens a dialog that lets you create, modify, and delete launch configurations of different types.
2. Select **Java Application** in the left hand list of launch configuration types, and press the **New** button in the toolbar. This will create a new launch configuration for a Java application. The tabs on the right hand side allow you control specific aspects of the launch.

- The **Main** tab defines the class to be launched. Enter the name of the project containing the class to launch in the project field, and the fully qualified name of the main class in the Main class field. Check the **Stop in main** checkbox if you want the program to stop in the main method whenever the program is launched in debug mode.

  Note: You do not have to specify a project, but doing so allows a default classpath, source lookup path, and JRE to be chosen.
- The **Arguments** tab defines the arguments to be passed to the application and to the virtual machine (if any). You can also specify the working directory to be used by the launched application.
- The **JRE** tab defines the JRE used to run or debug the application. You can select a JRE from the already defined JREs, or define a new JRE.
- The **Classpath** tab defines the location of class files used when running or debugging an application. By default, the user and bootstrap class locations are derived from the associated project's build path. You may override these settings here.
- The **Source** tab defines the location of source files used to display source when debugging a Java application. By default, these settings are derived from the associated project's build path. You may override these settings here.
- The **Environment** tab defines the environment variable values to use when running or debugging a Java application. By default, the environment is inherited from the Eclipse runtime. You may override or append to the inherited environment. Variables specified in the tab always replace values in the underlying native environment. However, when "Append environment to native environment" is selected the launched environment is seeded with the native environment, after which variables in the tab replace (existing variables) or augment the set of environment variables. When "Replace

native environment with specified environment" is selected, the launched environment is comprised only of the variables specified in the tab.
- The **Common** tab defines general information about the launch configuration. You may choose to store the launch configuration in a specific file and specify which perspectives become active when the launch configuration is launched.

Related concepts

Debugger
Local debugging

Related tasks

Choosing a JRE for launching a project
Launching a Java program
Setting execution arguments
Changing debugger launch options

Related reference

Debug preferences
Debug view
Run Menu

Launching a Java Program

The simplest way to launch a Java program is to run it using a **Java Application** launch configuration. This launch configuration type uses information derived from the workbench preferences and your program's Java project to launch the program.

1. In the **Package Explorer**, select the Java compilation unit or class file with the main method you want to launch.
2. Press the **Run** [ ] button in the workbench toolbar or select **Run > Run** from the workbench menu bar. Alternatively, select **Run As > Java Application** in the **Package Explorer** pop-up menu, or select **Run > Run As > Java Application** in the workbench menu bar, or select **Run As > Java Application** in the drop-down menu on the **Run** tool bar button.
3. Your program is now launched, and text output is shown in the **Console View**.

You can also launch a Java program by selecting a project instead of the compilation unit or class file. You will be prompted to select a class from those classes that define a *main* method. (If only one class with a main method is found in the project, that class is launched as if you selected it.)

Related concepts

Java views
Java editor
Debugger

Related tasks

Connecting to a remote VM with the Java Remote Application launcher
Re-launching a program
Running and debugging
Setting execution arguments
Stepping through the execution of a program

Related reference

Console View
Debug view
Package Explorer

Launching a Java applet

If your Java program is structured as an applet, you can use the **Java Applet** launch configuration. This launch configuration uses information derived from the workbench preferences and your program's Java project to launch the program.

1. In the **Package Explorer**, select the Java compilation unit or class file containing the applet you want to launch.
2. Press the **Run** [ ] button in the workbench toolbar or select **Run > Run** from the workbench menu. Alternatively, select **Run As > Java Applet** from the **Package Explorer** pop-up menu, select **Run > Run As > Java Applet** in the workbench menu bar, or select **Run As > Java Applet** in the drop-down menu on the **Run** tool bar button.
3. Your program is now launched.

You can also launch a Java applet by selecting a project instead of the compilation unit or class file. You will be prompted to select a class from those classes that extend Applet. (If only one applet class is found in the project, that class is launched as if you selected it.)

Related concepts

Debugger

Related tasks

Re-launching a program
Running and debugging
Stepping through the execution of a program

Related reference

Debug view
Package Explorer

Setting execution arguments

If you want to specify execution arguments for your program, you must define a launch configuration that specifies the arguments.

1. Select [Image: /topic/org.eclipse.help/command_link.png]**Run > Run Configurations...** or [Image: /topic/org.eclipse.help/command_link.png]**Run > Debug Configurations...** from the workbench **Run** menu to open the list of launch configurations. Launch configurations for Java programs are shown underneath **Java Application** in this list.
2. Select an existing configuration or create a new launch configuration by pushing the **New** button after selecting **Java Application**.
3. On the **Arguments** tab for the configuration, you can specify the following fields as necessary:
   - **Program Arguments**: Application-specific values that your code is expecting (a user name or a URL for locating help files, for example).
   - **VM Arguments**: Values meant to change the behavior of the Java virtual machine (VM). For example, you may need to tell the VM whether to use a just-in-time (JIT) compiler, or you may need to specify the maximum heap size the VM should use. Refer to your VM's documentation for more information about the available VM arguments.
   - **Working Directory**: The working directory used for the launched process. To change from using the default working directory, select **Other** and specify the workspace or local directory to use for the working directory of the launched process.
4. Click **Apply** or **Close** when you are done. Every time you launch this configuration, these execution arguments will be used.

Related tasks

Creating a Java Application launch configuration
Launching a Java program

Re-launching a program

The workbench keeps a history of each launched and debugged program. To relaunch a program, do one of the following:

- Select a previous launch from **Run** or **Debug** button pull-down menus.
- From the menu bar, select **Run** > **Run History** or **Run** > **Debug History** and select a previous launch from these sub-menus.
- In the **Debug View**, select a process that you want to relaunch, and select **Relaunch** from the process's pop-up menu.

To relaunch the most recent launch, do one of the following:

- Click the **Run** or **Debug** buttons (without using the button pull-down menu).
- Select **Run** > **Run Last Launched** (`Ctrl+F11`), or **Run** > **Debug Last Launched** (`F11`) from the workbench menu bar.

Related tasks

Launching a Java program
Running and debugging

Related reference

Debug view
Relaunch command
Terminate and Relaunch command

Inspecting memory in the Memory view

With the Memory view, you can look at the contents of memory at a specific address. The address can be obtained from an expression that references a variable or a register. When you monitor memory, you can set the monitor to be rendered in various data formats.

The Memory view is comprised of two portions:

- The **Monitors** pane is located on the left-hand side of the view. This portion of the view contains a list of expressions, variables, and registers that you have added for monitoring.
- The **Renderings** pane is located on the right-hand side of the view (if the view is set to horizontal orientation). You use it to set the data format (or formats) that you want displayed for monitored memory.

When you monitor an address or expression from the **Monitors** pane, the **Renderings** pane becomes populated with a set of default renderings provided by your debugger. When you monitor an address or expression from the **Renderings** pane, the pane becomes populated with a list of renderings from which you can choose one to display.

You can monitor multiple variables, expressions, and registers in the Memory view - and you can add multiple renderings to the **Renderings** pane. In the **Monitors** pane, each variable, expression, or register that you have added is listed. In the **Renderings** pane, only the memory rendering(s) for the currently-selected monitor in the Memory view is displayed (multiple renderings are separated by tabs or a split pane).

You can set the two panes in the Memory view to display in a horizontal orientation (side-by-side) or in a vertical orientation (top-to-bottom). To set the layout of the view, click the Memory view down-arrow icon and select **Layout** from the menu. This will open a submenu, from which you can choose the orientation that you want to display.

Adding a variable, expression, or register to the Memory view

You can add a register, variable name, or expression (such as a raw address, for example, 0x00000234) to a memory monitor from the Memory view.

To add a new memory monitor from the Memory view:

1. Click the Memory view **Add Memory Monitor** push button (✚). This button is located in the **Monitors** pane.
2. In the Monitor Memory dialog box, enter the address or expression in the field (expressions must evaluate to an address). This entry does not need to be case-sensitive. Alternatively, if you have previously monitored the address or expression when debugging this application, choose it from the pull down list. Depending on the debug engine that the debugger user interface is connected to, examples of valid expressions include register names, variables, and HEX addresses. **Note:** When adding a register to a memory monitor, the Registers view is a convenient location to see the names of all registers in your application. You can make note of the name of a register from the view and then use it when adding a memory monitor.
3. Click **OK**.

After adding the new memory monitor you can choose the memory format that you want to display in the **Renderings** pane.

You can also add a new memory monitor from the Memory view **Renderings** pane:

1. Click the Memory view **Add Rendering(s)** push button (✚). This button is located in the **Renderings** pane.
2. In the Add Memory Rendering dialog box, select an existing memory monitor from the pull-down list - or add a new memory monitor by clicking **Add New**. If you add a new memory monitor, you will be prompted via the Monitor Memory dialog box to enter an address or expression to monitor. Click **OK** to return to the Add Memory Rendering dialog box.
3. Do one of the following:
   - Select the memory rendering that you want to display for the memory monitor and click **OK**.
   - Click **Cancel** and then choose the memory rendering that you want to display right from the **Renderings** pane.

Adding multiple memory renderings and removing renderings

When you add a variable, expression, or register to the Memory view, you can do so multiple times, each time adding a new (or the same) rendering. Alternatively, once you have added a memory monitor and rendering, you can go to the **Renderings** pane and click the **Add Rendering(s)** push button (). This will prompt you with a dialog to select the rendering that you want to add to the view. In this dialog, you can select more than one rendering by using the keyboard Shift or Ctrl keys - doing this will cause a rendering to be opened for each rendering format that is selected. When you add multiple memory renderings, they are separated by tabs.

You can also split the **Memory Renderings** pane by selecting the **Toggle Split Pane** push button (). When the **Renderings** pane is split, you can view two renderings side-by-side.

When you have multiple memory renderings for a memory monitor, you can set the renderings to be linked with one another. To do this, select the **Link Memory Rendering Panes** push button (). When renderings are linked, they are synchronized with each other (for example, if you change the column size in one rendering, the column size in the other rendering will also change - or if you scroll or move the cursor in one rendering, the other rendering will scroll or follow the same cursor movement). Linking memory renderings only applies to the current Memory view. If you have multiple Memory views open, they do not link to each other.

To remove a rendering, select it in the **Renderings** pane and click **Remove Rendering** (). When you remove all memory renderings for a monitored expression, variable, or register, the **Renderings** pane will be populated with the memory rendering selection list. From this list, you can select the data format that you want to use for the memory rendering and then click **Add Rendering(s)**.

Working with memory monitors

To view the contents of memory from the Memory view:

1. In the **Monitors** pane, select the memory monitor that contains the memory location that you want to view. Memory will appear in the **Renderings** pane, where you will perform all other steps. If you have added multiple renderings, select the tab that contains the rendering that you want to view.
2. If desired, split the **Renderings** pane by selecting the **Toggle Split Pane** push button (). By default, the Memory view only displays one rendering pane. When you click **Toggle Split Pane**, a second rendering opens and displays as a split pane.
3. If necessary, use the scroll bar in the rendering to view memory locations above or below the base address of the memory monitor being shown by the current rendering. Alternatively, you can right-click in the rendering and choose the **Go to Address** pop-up menu item or hit Ctrl+G. This will open a section at the bottom of the rendering, in which you can perform the following actions:
   - Select the **Go to Address** pull-down menu item and then enter an address that you want to jump to. The rendering will be positioned so that the address entered is visible and selected.
   - Select the **Go to Offset** pull-down menu item and then enter the offset. The rendering will be positioned so that the address of the expression (base address), plus the offset entered, is visible and selected. A negative value will position the rendering back from the base address.
   - Select the **Jump Memory Units** pull-down menu item. This function takes the currently-selected address and adds the number of memory units that you specify to it. The resulting address is selected. A negative value will position the rendering back from the current address.

   For all of these entries, you can input them as HEX by selecting the **Input as Hex** check box (if this check box is not selected, input will be decimal). Once you have made the entry in the field, hit Enter or click **OK** to go to the location in the rendering. To close this section, click **Cancel** or hit Ctrl+G.
   Note: Input is also treated as HEX if it is prefixed with `0x`.
4. If you want, change the width of any column by clicking the left or right side of its header cell and dragging it to alter the width of the column - or right-click inside the rendering and select **Resize to Fit** from the pop-up menu so that all columns are re-sized so that all text within them can be viewed. Alternatively, you can right-click inside the rendering and select **Format** from the pop-up menu. This will open the Format dialog box. In this dialog box, you can set the number of units per row and the number of units per column. As you make these settings, a **Preview** window in the dialog box displays the rendering layout that you are setting. To save these settings as the default layout, click **Save as Defaults**.
5. You can also hide elements of the Memory view for easier viewing:
   - You can hide the **Monitors** pane by deselecting the **Toggle Memory Monitors Pane** toggle.
   - You can hide the **Address** column by right-clicking inside the rendering and selecting **Hide Address Column**. To restore a the address column when it is hidden, right-click inside the rendering and select **Show Address Column** from

the pop-up menu.

If you are in a memory rendering and move away from the address that you originally set to monitor, choosing the **Reset to Base Address** pop-up menu item will position the cursor back to the base address of the memory monitor.

Alternatively, you can reset all renderings for a memory monitor by right-clicking the monitor and selecting **Reset** (or, you can select multiple monitors and choose this action). When you reset a monitor, by default, the visible renderings will be reset to the base address. To reset all renderings in the current Memory view to the base address, modify the Memory view preferences.

Changing the contents of a memory location

To change the contents of a memory location in a memory monitor in the Memory view:

1. In the **Monitors** pane, select the memory monitor that contains the memory location that you want to edit. Memory will appear in the **Renderings** pane, where you will perform all other steps. If you have added multiple renderings, select the tab that contains the rendering that you want to edit.
2. Scroll down to the memory location you want to change. Alternatively, right-click in the monitor and choose the **Go to Address** pop-up menu item. This will open a Go To Address section at the bottom of the rendering, in which you can enter an address that you want to jump to.
3. Select the row containing the value that you want to change and then double-click the value that you want to change. **Tip:** If the rendering is currently in focus, you do not need to double-click the value that you want to change to be able to edit it. Rather, you can simply start typing the change and the editor will activate.
4. Enter a valid value for that memory location.
5. Press **Enter** to submit the change. The debugger checks for a valid value.

Memory view preferences

You can set table rendering, codepage, and padded string preferences for memory renderings. In addition, you can modify the preferred behavior for resetting memory renderings.

Memory view preference dialog boxes are opened from the Memory view down-arrow icon menu. To open the Memory view Preferences dialog box, click the Memory view down-arrow icon and select **Preferences** from the menu. To open the Memory view table renderings Preferences dialog box, click the Memory view down-arrow icon and select **Table Renderings Preferences** from the menu.

To restore any changes that you make in the preferences to their default settings, click **Restore Defaults**.

Preferences: Reset Memory Monitor

You can reset a rendering to the base address if you have moved away from it. When you reset a rendering to the base address, you can set it to reset only the visible renderings - or you can set it to reset all renderings. If you choose to reset all renderings, performance of the reset operation can be negatively impacted. To set this preference, open the Preferences dialog box and then select the **Reset Memory Monitor** node. In the Reset Memory Monitor page, choose the appropriate radio button.

Preferences: Padded String

The padded string is the string that will appear in memory contents when memory cannot be retrieved. To set the padded string, open the Preferences dialog box and select the **Padded String** node. In the Padded String page, specify the string that you want to display when memory contents cannot be determined.

Preferences: Select Codepages

When monitoring ASCII and EBCDIC text-based renderings (and mapped memory, if it is available in the product that you installed this debugger with) in the **Renderings** pane, you can set the codepage in which you want the rendering to be displayed.

To set the codepage for rendering memory to ASCII/EBCDIC, open the Preferences dialog box and select the **Select Codepages** node. In the Select Codepages page, specify the codepage of the character set that you want to change (for ASCII renderings, EBCDIC renderings, or both).

Table Renderings Preferences

To set preferences for memory renderings that are displayed in a table, click the Memory view down-arrow icon and select **Table Renderings Preferences**. In the resulting preferences dialog box, there are two options:

- **Automatic** radio button: Select this if you want the debugger to automatically preload a buffer of memory before and after the visible region in the Memory view. When you select this setting, you can easily scroll away from the visible region of memory with scroll and page up/down actions. If the buffer size is large, the performance of refreshing the Memory view can be negatively impacted (the Memory view refreshes when there is a potential for memory change - so, for example, the Memory view refreshes when you step, suspend after running, change a variable or register, or modify memory). This is because a request for more memory must be made from the debug engine for each refresh. On the other

hand, if the buffer size is too small, the performance of scrolling can be negatively impacted. This is because a request for more memory must be made from the debug engine when the buffer is exhausted. Given this performance trade-off, you need to set this option to suit your needs.

- **Manual** radio button: If you select this setting, then the number of lines per page that you specify will be loaded into the **Renderings** pane. When using this setting, you have more control in choosing exactly what you want to monitor, however, you will not be able to scroll outside the buffer defined by this page size setting. Instead, to view memory from the next page or previous page, you must right-click to use the **Previous Page** and **Next Page** actions from the pop-up menu. If the buffer size is large, the performance of refreshing the Memory view can be negatively impacted (the Memory view refreshes when there is a potential for memory change - so, for example, the Memory view refreshes when you step, suspend after running, change a variable or register, or modify memory). This is because a request for more memory must be made from the debug engine for each refresh.

Working with multiple Memory views

You can add additional Memory views to the workbench. To do this, click **New Memory View** (). When you have multiple Memory views open, you cannot link their renderings to each other. However, you can pin the contents of a Memory view so that memory renderings that are added to one view do not affect the other view. To pin a memory monitor, ensure that the **Pin Memory Monitor** button () in the Memory view is toggled on. If you then go to another Memory view and add a memory monitor, it will show up in both Memory views, however, the memory rendering that is currently displayed in the pinned monitor will not change.
When you add a new Memory view, its **Renderings** pane will be populated with the memory rendering selection list. From this list, you can select the data format that you want to use for the memory rendering and then click **Add Rendering(s)**.

Removing memory monitors from the Memory view

To remove a memory monitor from the Memory view:

1. Select the memory monitor that you want to remove (by selecting it in the list in the **Monitors** pane).
2. Click the **Remove Memory Monitor** push button (✖).

To remove multiple memory monitors, select them using the keyboard Ctrl or Shift keys, and then click **Remove Memory Monitor**. To remove all memory monitors, click **Remove All**.

Note: If you have added multiple renderings for a memory monitor, all renderings will be removed when you choose to remove the monitor.

Creating a Java Scrapbook Page

The scrapbook allows Java expressions, to be run, inspected, and displayed under the control of the debugger. Breakpoints and exceptions behave as they do in a regular debug session.

Code is edited on a scrapbook page. A VM is launched for each scrapbook page in which expressions are being evaluated. The first time an expression is evaluated in a scrapbook page after it is opened, a VM is launched. The VM for a page will remain active until the page is closed, terminated explicitly (in the debugger or via the **Stop the Evaluation** button in the editor toolbar), or when a *System.exit()* is evaluated.

There are several ways to open the [Image: /topic/org.eclipse.help/command_link.png]**New Java Scrapbook Page** wizard.

- Create a file with a **.jpage** extension
- From the menu bar, select **File** > **New** > **Other...**. Then select **Java** > **Java Run/Debug** > **Scrapbook Page**. Then click **Next**.

Once you've opened the New Java Scrapbook Page wizard:.

1. In the **Enter or select the folder** field, type or click **Browse** to select the container for the new page.
2. In the **File name** field, type a name for the new page. The *.jpage* extension will be added automatically if you do not type it yourself.
3. Click **Finish** when you are done. The new scrapbook page opens in an editor.

Related concepts

Scrapbook
Java projects

Related tasks

Running and debugging

Related reference

Java scrapbook page

Inspecting the result of evaluating an expression

Inspecting shows the result of evaluating an expression in the Expressions view.

1. In the scrapbook page, either type an expression or highlight an existing expression to be inspected. For example: `System.getProperties();`
2. Click the **Inspect** button in the toolbar (or select **Inspect** from the selection's pop-up menu).
3. The result of the inspection appears in a pop-up.
4. The result can be inspected like a variable in the debugger (for example, children of the result can be expanded).

Related concepts

Scrapbook

Related tasks

Creating a Java scrapbook page
Displaying the result of evaluating an expression
Executing an expression
Viewing runtime exceptions

Related reference

Expressions view
Java scrapbook page

Displaying the result of evaluating an expression

Displaying shows the result of evaluating an expression in the scrapbook editor.

1. In the scrapbook page, either type an expression or highlight an existing expression to be displayed. For example: `System.getProperties();`
2. Click the **Display** button in the toolbar (or select **Display** from the selection's pop-up menu.)
3. The result of the evaluation appears highlighted in the scrapbook editor. The result displayed is either
   - the value obtained by sending `toString()` to the result of the evaluation, or
   - when evaluating a primitive data type (e.g., an *int*), the result is the simple value of the result.

   For example:

- Type and highlight `new java.util.Date()` in the editor, and click **Display**. A result such as `(java.util.Date) Tue Jun 12 14:03:17 CDT 2001` appears in the editor.
- As another example, type and highlight `3 + 4` in the editor, and press **Display**. The result `(int) 7` is displayed in the editor.

Related concepts

Scrapbook

Related tasks

Executing an expression
Inspecting the result of evaluating an expression
Viewing runtime exceptions

Related reference

Java scrapbook page

Executing an expression

Executing an expression evaluates an expression but does not display a result.

If you select the expression to execute and click the **Execute** button in the toolbar, no result is displayed, but the code is executed.

For example, if you type and highlight `System.out.println("Hello World")`, and click the **Execute** button, *Hello World* appears in the Console view, but no result is displayed in the scrapbook editor or the Expressions view.

Related concepts

Java views

Related tasks

Displaying the result of evaluating an expression
Inspecting the result of evaluating an expression
Viewing runtime exceptions

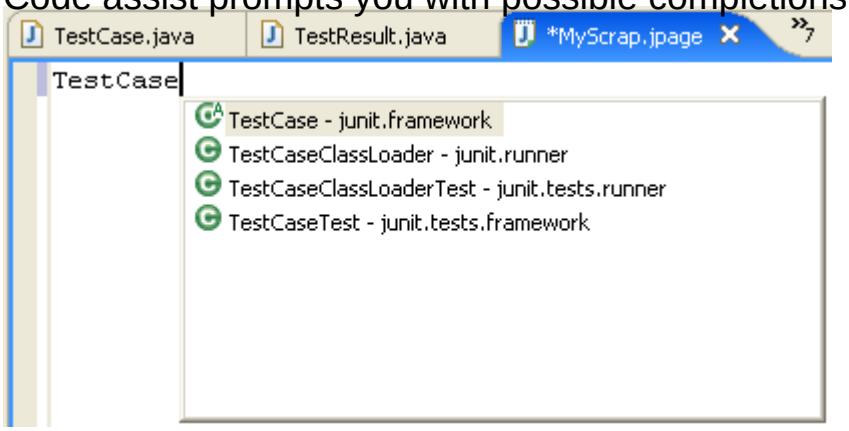
Related reference

Expressions view
Console view

Using code assist

The scrapbook editor supports code assist similarly to the regular Java editor.

For example, type *TestCase* in the scrapbook editor and press ***Ctrl+Space***. Code assist prompts you with possible completions.



Scrapbook error reporting

Java scrapbook errors are reported in the scrapbook page editor.

Related tasks

Viewing compilation errors
Viewing runtime exceptions

Viewing compilation errors

If you try to evaluate an expression containing a compilation error, it will be reported in the scrapbook editor.

For example, type and select the (invalid) expression `System.println("hi")` in the editor and click **Execute** in the toolbar.

The error message *The method println(java.lang.String) is undefined for the type java.lang.System* appears in the editor at the point of the error.

Viewing runtime exceptions

If an expression you evaluate causes a runtime exception, the exception will be reported in the editor. For example:

Type and select the expression `Object x = null; x.toString()` in the editor and click **Display** in the toolbar.

The error message:

An exception occurred during evaluation: java.lang.NullPointerException

will be displayed in the editor.

Related concepts

Java editor

Related tasks

Displaying the result of evaluating an expression
Inspecting the result of evaluating an expression
Executing an expression

Using the batch compiler

Finding the batch compiler

The batch compiler class is located in the JDT Core plug-in. The name of the class is *org.eclipse.jdt.compiler.batch.BatchCompiler*. It is packaged into `plugins/org.eclipse.jdt.core_3.4.0.<qualifier>.jar`. Since 3.2, it is also available as a separate download. The name of the file is `ecj.jar`. Its corresponding source is also available. To get them, go to the download page and search for the section **JDT Core Batch Compiler**. This jar contains the batch compiler and the javac ant adapter.
Since 3.3, this jar also contains the support for jsr199 (Compiler API) and the support for jsr269 (Annotation processing). **In order to use the annotations processing support, a 1.6 VM is required.**
So it can be used as a standalone application and inside an Ant build outside of Eclipse.

Running the batch compiler

- From the command line. `java -jar org.eclipse.jdt.core_3.4.0<qualifier>.jar -classpath rt.jar A.java`
  or:
  `java -jar ecj.jar -classpath rt.jar A.java`
- Using the static `compile(String commandLine, PrintWriter outWriter, PrintWriter errWriter, CompilationProgress progress)` method of the class BatchCompiler.

```
org.eclipse.jdt.compiler.CompilationProgress progress = null; // instantiate your subclass
org.eclipse.jdt.internal.compiler.batch.BatchCompiler.compile(
  "-classpath rt.jar A.java",
  new PrintWriter(System.out),
  new PrintWriter(System.err),
  progress);
```

You can control how progress is reported, or how the batch compiler is canceled, by subclassing the class *org.eclipse.jdt.compiler.CompilationProgress*.

Which options are available?

The recommended options have an orange background.
When some options are being set multiple times, the batch compiler consumes them from left to right. When the warning option (-warn:....) is used without '+' or '-', this overrides the set of warnings previously specified. So the user should make sure that such an option is given before any other usage of the -warn option.
Same applies for the -err: option.

Name	Usage
Classpath options	

-bootclasspath <dir 1>;<dir 2>;...;<dir P>	This is a list of directories or jar files used to bootstrap the class files used by the compiler. By default the libraries of the running VM are used. Entries are separated by the platform path separator. Each directory or file can specify access rules for types between '[' and ']'. If no bootclasspath is specified, the compiler will infer it using the following system properties `sun.boot.class.path`, `vm.boot.class.path` or `org.apache.harmony.boot.class.path` in this order respectively.
-cp -classpath <dir 1>;<dir 2>;...;<dir P>	This is a list of directories or jar files used to compile the source files. The default value is the value of the property "java.class.path". Entries are separated by the platform path separator. Each directory or file can specify access rules for types between '[' and ']' (e.g. [-X] to forbid access to type X, [~X] to discourage access to type X, [+p/X:-p/*] to forbid access to all types in package p but allow access to p/X). The compiler follows the `Class-Path` clauses of jar files' manifests recursively and appends each referenced jar file to the end of the classpath, provided it is not on the classpath yet.
-extdirs <dir 1>;<dir 2>;...;<dir P>	This is a list of directories used to specify the location of extension zip/jar files. Entries are separated by the platform path separator.
-endorseddirs <dir 1>;<dir 2>;...;<dir P>	This is a list of directories used to specify the location of endorsed zip/jar files. Entries are separated by the platform path separator.
-sourcepath <dir 1>;<dir 2>;...;<dir P>	This is a list of directories used to specify the source files. Entries are separated by the platform path separator. Each directory can specify access rules for types between '[' and ']'.
-d <dir 1>|none	This is used to specify in which directory the generated .class files should be dumped. If it is omitted, no package directory structure is created. If you want to generate no .class file at all, use -d none.

-encoding <encoding name>	Specify default encoding for all source files. Custom encoding can also be specified on a per file basis by suffixing each input source file/folder name with [<encoding name>]. For example X.java[utf8] would specify the UTF-8 encoding for the compilation unit X.java located in the current user directory. If multiple default source file encodings are specified, the last one will be used. For example: `... -encoding UTF-8 X.java[Cp1252] Y.java[UTF-16] Z.java ....` All source files will be read using UTF-8 encoding (this includes Z.java). X.java will be read using Cp1252 encoding and Y.java will be read using UTF-16 encoding. `... -encoding UTF-8 -encoding UTF-16 ....` All source files will be read using UTF-16 encoding. The -encoding option for UTF-8 is ignored. `... -encoding Cp1252 /foo/bar/X.java[UTF-16] /foo/bar[UTF-8] ....` All source files will be read using Cp1252 encoding. X.java is the only file inside the /foo/bar directory to be read using the encoding UTF-16. All other files in that directory will use UTF-8 encoding.
Compliance options	
-target 1.1 to 1.7 or (5, 5.0, etc)	This specifies the .class file target setting. The possible value are: 1.1 (major version: 45 minor: 3)1.2 (major version: 46 minor: 0)1.3 (major version: 47 minor: 0)1.4 (major version: 48 minor: 0)1.5, 5 or 5.0 (major version: 49 minor: 0)1.6, 6 or 6.0 (major version: 50 minor: 0)1.7, 7 or 7.0 (major version: 51 minor: 0) Defaults are: 1.1 in -1.3 mode1.2 in -1.4 mode1.5 in -1.5 mode1.6 in -1.6 mode1.7 in -1.7 modeclcd1.1 can be used to generate the StackMap attribute.
-1.3	Set compliance level to 1.3. Implicit -source 1.3 -target 1.1.
-1.4	Set compliance level to 1.4 (default). Implicit -source 1.3 -target 1.2.
-1.5	Set compliance level to 1.5. Implicit -source 1.5 -target 1.5.
-1.6	Set compliance level to 1.6. Implicit -source 1.6 -target 1.6.
-1.7	Set compliance level to 1.7. Implicit -source 1.7 -target 1.7.

<table>
<tr><td>-source 1.1 to 1.7 or (5, 5.0, etc)</td><td>This is used to specify the source level expected by the compiler.
The possible value are: 1.31.41.5, 5 or 5.01.6, 6 or 6.01.7, 7 or 7.0 Defaults are: 1.3 in -1.3 mode1.3 in -1.4 mode1.5 in -1.5 mode1.6 in -1.6 mode1.7 in -1.7 mode In 1.4, assert is treated as a keyword. In 1.5 and 1.6, enum and assert are treated as a keywords.</td></tr>
<tr><td colspan="2">Warning options</td></tr>
<tr><td>-?:warn -help:warn</td><td>Display advanced warning options</td></tr>
<tr><td>-warn:...</td><td>Specify the set of enabled warnings.
e.g. -warn:unusedLocal,deprecation
<table>
<tr><th>Default</th><th>Token name</th><th>Description</th></tr>
<tr><td>-</td><td>allDeadCode</td><td>dead code including trivial if(DEBUG) check</td></tr>
<tr><td>-</td><td>allDeprecation</td><td>deprecation even inside deprecated code</td></tr>
<tr><td>-</td><td>allJavadoc</td><td>invalid or missing javadoc</td></tr>
<tr><td>-</td><td>allOver-ann</td><td>all missing @Override annotations (superclass and superinterfaces)</td></tr>
<tr><td>+</td><td>assertIdentifier</td><td>occurrence of assert used as identifier</td></tr>
<tr><td>-</td><td>boxing</td><td>autoboxing conversion</td></tr>
<tr><td>+</td><td>charConcat</td><td>when a char array is used in a string concatenation without being converted explicitly to a string</td></tr>
<tr><td>+</td><td>compareIdentical</td><td>comparing identical expressions</td></tr>
<tr><td>-</td><td>conditionAssign</td><td>possible accidental boolean assignment</td></tr>
</table></td></tr>
</table>

+	constructorName	method with constructor name
+	deadCode	dead code excluding trivial if (DEBUG) check
-	dep-ann	missing @Deprecated annotation
+	deprecation	usage of deprecated type or member outside deprecated code
+	discouraged	use of types matching a discouraged access rule
-	emptyBlock	undocumented empty block
+	enumIdentifier	occurrence of *enum* used as identifier
-	enumSwitch	incomplete enum switch
-	fallthrough	possible fall-through case
-	fieldHiding	field hiding another variable
+	finalBound	type parameter with final bound
+	finally	finally block not completing normally
+	forbidden	use of types matching a forbidden access rule
-	hashCode	missing hashCode() method when overriding equals()

-	hiding	macro for fieldHiding, localHiding, typeHiding and maskedCatchBlock
-	includeAssertNull	raise null warnings for variables that got tainted in an assert expression
-	indirectStatic	indirect reference to static member
+	intfAnnotation	annotation type used as super interface
+	intfNonInherited	interface non-inherited method compatibility
-	intfRedundant	find redundant superinterfaces
-	javadoc	invalid javadoc
-	localHiding	local variable hiding another variable
+	maskedCatchBlock	hidden catch block
-	nls	non-nls string literals (lacking of tags //$NON-NLS-<n>)
+	noEffectAssign	assignment with no effect
-	null	potential missing or redundant null check
-	nullDereference	missing null check
-	over-ann	missing @Override annotation (superclass only)
-	paramAssign	assignment to a parameter

+	pkgDefaultMethod	attempt to override package-default method
+	raw	usage a of raw type (instead of a parameterized type)
-	semicolon	unnecessary semicolon or empty statement
+	serial	missing serialVersionUID
-	specialParamHiding	constructor or setter parameter hiding another field
-	static-access	macro for indirectStatic and staticReceiver
-	static-method	an instance method that could be as a static method
+	staticReceiver	if a non static receiver is used to get a static field or call a static method
-	super	overriding a method without making a super invocation
+	suppress	enable @SuppressWarnings
-	syncOverride	missing synchronized in synchronized method override
-	syntheticAccess	when performing synthetic access for innerclass

-	tasks	enable support for tasks tags in source code
+	typeHiding	type parameter hiding another type
+	unavoidableGenericProblems	ignore unavoidable type safety problems due to raw APIs
+	unchecked	unchecked type operation
-	unnecessaryElse	unnecessary else clause
-	unqualifiedField	unqualified reference to field
-	unused	macro for unusedAllocation, unusedArgument, unusedImport, unusedLabel, unusedLocal, unusedPrivate, unusedThrown and unusedTypeArgs
-	unusedAllocation	allocating an object that is not used
-	unusedArgument	unused method argument
+	unusedImport	unused import reference
+	unusedLabel	unused label
+	unusedLocal	unused local variable
+	unusedPrivate	unused private member declaration
-	unusedThrown	unused declared thrown exception

	+	unusedTypeArgs	unused type arguments for method and constructor
	-	uselessTypeCheck	unnecessary cast/instanceof operation
	+	varargsCast	varargs argument need explicit cast
	+	warningToken	unhandled warning token in @SuppressWarnings
-nowarn	No warning (equivalent to -warn:none)		
-err:...	Specify the set of enabled warnings that are converted to errors. e.g. -err:unusedLocal,deprecation unusedLocal and deprecation warnings will be converted to errors. All other warnings are still reported as warnings.		
-deprecation	Equivalent to -warn:+deprecation.		

-properties <file>	Set warnings/errors option based on the properties file contents. This option can be used with -nowarn, -err:.. or -warn:.. options, but the last one on the command line sets the options to be used.
	The properties file contents can be generated by setting project specific settings on an existing java project and using the file in `.settings/org.eclipse.jdt.core.prefs` file as a properties file, or a simple text file that is defined entry/value pairs using the constants defined in the `org.eclipse.jdt.core.JavaCore` class.
	`...`
	`org.eclipse.jdt.core.compiler.problem.annotationSuperInterface=warning`
	`org.eclipse.jdt.core.compiler.problem.assertIdentifier=warning`
	`org.eclipse.jdt.core.compiler.problem.autoboxing=ignore`
	`...`		

Debug options			
-g[:none|:lines,vars,source]	Set the debug attributes level		
-preserveAllLocals	Explicitly request the compiler to preserve all local variables (for debug purpose). If omitted, the compiler will remove unused locals.		
Annotation processing options (require a 1.6 VM or above and are used only if the compliance is 1.6)			

-Akey[=value]	Annotation processors options that are passed to annotation processors. `key` is made of identifiers separated by dots
-proc:[only|none]	If `-proc:only` is specified, the annotation processors will run but no compilation will be performed. If `-proc:none` is specified, annotation processors will not be discovered or run; compilation will proceed as if no annotation processors were found. By default the compiler must search the classpath for annotation processors, so specifying `-proc:none` may speed compilation if annotation processing is not required.
-processor <class1[,class2,...]>	Qualified class names of annotation processors to run. If specified, the normal processor discovery process will be skipped.
-processorpath <dir 1>;<dir 2>;...;<dir P>	A list of directories or jar files which will be searched for annotation processors. Entries are separated by the platform path separator. If not specified, the classpath will be searched instead.
-s <dir>	The directory where generated source files will be created.
-XprintProcessorInfo	Print information about which annotations and which elements a processor is asked to process
-XprintRounds	Print information about annotation processing rounds
-classNames <class1[,class2,...]>	Qualified names of binary types that need to be processed
Ignored options (for compatibility with javac options)	
-J<option>	Pass option to the virtual machine
-X<option>	Specify non-standard option. -Xemacs is not ignored.
-X	Print non-standard options and exit
-O	Optimize for execution time
Advanced options	
@<file>	Read command-line arguments from file
-maxProblems <n>	Max number of problems per compilation unit (100 by default)
-log <filename>	Specify a log file in which all output from the compiler will be dumped. This is really useful if you want to debug the batch compiler or get a file which contains all errors and warnings from a batch build. If the extension is **.xml**, the generated log will be an xml file.

-Xemacs	Use emacs style to present errors and warnings locations into the console and regular text logs. XML logs are unaffected by this option. With this option active, the message:
	`2. WARNING in /workspace/X.java (at line 8)...`
	is presented as:
	`/workspace/X.java:8: warning: The method...`

-proceedOnError [:Fatal]	Keep compiling in spite of errors, dumping class files with problem methods or problem types. This is recommended only if you want to be able to run your application even if you have remaining errors. With ":Fatal", all optional errors are treated as fatal and this leads to code that will abort if an error is reached at runtime. Without ":Fatal", optional errors don't prevent the proper code generation and the produced .class files can be run without a problem.
-verbose	Print accessed/processed compilation units in the console or the log file if specified.
-referenceInfo	Compute reference info. This is useful only if connected to the builder. The reference infos are useless otherwise.
-progress	Show progress (only in -log mode).
-time	Display speed information.
-noExit	Do not call System.exit(n) at end of compilation (n=0 if no error).
-repeat <n>	Repeat compilation process <n> times (perf analysis).
-inlineJSR	Inline JSR bytecode (implicit if target >= 1.5).
-enableJavadoc	Consider references inside javadoc.
Helping options	
-? -help	Display the help message.
-v -version	Display the build number of the compiler. This is very useful to report a bug.
-showversion	Display the build number of the compiler and continue. This is very useful to report a bug.

Examples

`d:\temp -classpath rt.jar -time -g -d d:/tmp`	It compiles all source files in d:\temp and its subfolders. The classpath is simply rt.jar. It generates all debug attributes and all generated .class files are dumped in d:\tmp. The speed of the compiler will be displayed once the batch process is completed.

`d:\temp\Test.java -classpath d:\temp;rt.jar -g:none`	It compiles only Test.java and its dependant files if any, retrieving dependant files from d:\temp. The classpath is d:\temp followed by rt.jar, which means that all necessary classes are searched first in d:\temp and then in rt.jar. It generates no debug attributes and all generated .class files are dumped in d:\temp.

Using the ant javac adapter

The Eclipse compiler can be used inside an Ant buildfile using the javac adapter. In order to use the Eclipse compiler, you simply need to define the **build.compiler** property in your buildfile.

In order to get the batch compiler working in an ant buildfile, the ant runtime classpath needs to contain the Eclipse batch compiler. When you run your ant buildfile:

1. outside of Eclipse: the easiest way to set up the ant runtime classpath is to add the `ecj.jar` file using the `-lib` argument or dumping it inside the `ANT_HOME` location.
2. inside Eclipse using the same JRE than Eclipse: the Eclipse batch compiler is implicitly added to the ant runtime classpath.
3. inside Eclipse using the different JRE: the Eclipse batch compiler must be explicitly added to the ant runtime classpath. This can be done using the `ecj.jar` file or using the org.eclipse.jdt.core jar file and the `jdtCompilerAdapter.jar` file located inside the org.eclipse.jdt.core jar file (this jar file needs to be extracted first).

Here is a small example:

```
<?xml version="1.0" encoding="UTF-8"?> <project name="compile" default="main" basedir="../.">   <property name="build.compiler" value="org.eclipse.jdt.core.JDTCompilerAdapter"/>   <property name="root" value="${basedir}/src"/>   <property name="destdir" value="d:/temp/bin" />   <target name="main">   <javac srcdir="${root}" destdir="${destdir}" debug="on" nowarn="on" extdirs="d:/extdirs" source="1.4">
<classpath>         <pathelement location="${basedir}/../org.eclipse.jdt.core/bin"/>        </classpath>
</javac>  </target> </project>
```

The syntax used for the javac Ant task can be found in the Ant javac task documentation. The current adapter supports the Javac Ant task 1.4.1 up to 1.6.5 versions.

If you are using a version above 1.5.0, you can use the nested compiler argument element (`<compilerarg>`) to specify compiler specific options.

```
... <javac srcdir="${root}" destdir="${destdir}" debug="on" nowarn="on" extdirs="d:/extdirs" source="1.4">
  <classpath>       <pathelement location="${basedir}/../org.eclipse.jdt.core/bin"/>     </classpath>
<compilerargcompiler="org.eclipse.jdt.core.JDTCompilerAdapter"line="-1.5 -warn:+boxing"/> </javac> ...
```

Note:

1. To prevent compiler dependant buildfiles, we *strongly* advise you to use a `<compilerarg>` whose "compiler" attribute value is `org.eclipse.jdt.core.JDTCompilerAdapter`. If this is not set, the buildfile can only be used with the Eclipse compiler. If set, the nested compiler argument is ignored if the name is different from the compiler name specified by the `build.compiler` property.
2. `<compilerarg>` should not be used to set values like target value, source value, debug options, or any options that could be set using the defined attributes of the `javac` ant task. Its usage must be reserved to pass compiler specific options like warning options. When a command-line argument is specified more than once, the Eclipse batch compiler can report errors like:

   ```
   duplicate target compliance setting specification: 1.5
   ```

Excluding warnings using @SuppressWarnings

Since Java 5.0, you can disable compilation warnings relative to a subset of a compilation unit using the `java.lang.SuppressWarning` annotation.

```
@SuppressWarning("unused") public void foo() {
 String s;
}
```

Without the annotation, the compiler would complain that the local variable `s` is never used. With the annotation, the compiler silently ignores this warning locally to the `foo` method. This enables to keep the warnings in other locations of the same compilation unit or the same project.
The list of tokens that can be used inside a `SuppressWarnings` annotation is:
- all to suppress all warnings
- boxing to suppress warnings relative to boxing/unboxing operations
- cast to suppress warnings relative to cast operations
- dep-ann to suppress warnings relative to deprecated annotation
- deprecation to suppress warnings relative to deprecation
- fallthrough to suppress warnings relative to missing breaks in switch statements
- finally to suppress warnings relative to finally block that don't return
- hiding to suppress warnings relative to locals that hide variable
- incomplete-switch to suppress warnings relative to missing entries in a switch statement (enum case)
- javadoc to suppress warnings relative to javadoc warnings
- nls to suppress warnings relative to non-nls string literals
- null to suppress warnings relative to null analysis
- rawtypes to suppress warnings relative to usage of raw types
- restriction to suppress warnings relative to usage of discouraged or forbidden references
- serial to suppress warnings relative to missing serialVersionUID field for a serializable class
- static-access to suppress warnings relative to incorrect static access
- static-method to suppress warnings relative to methods that could be declared as static
- super to suppress warnings relative to overriding a method without super invocations
- synthetic-access to suppress warnings relative to unoptimized access from inner classes
- unchecked to suppress warnings relative to unchecked operations
- unqualified-field-access to suppress warnings relative to field access unqualified
- unused to suppress warnings relative to unused code and dead code

Using the Formatter Application

The JDT has a command line Eclipse application that allows users to format Java source code either with the Eclipse default code formatter options or with custom options.

- Related tasks

Running the formatter application

- Related reference

Code Formatter

Running the Formatter Application

Running the formatter application is as simple as running the org.eclipse.jdt.core.JavaCodeFormatter application from the commandline:

```
eclipse -vm <path to virtual machine> -application org.eclipse.jdt.core.JavaCodeFormatter [ OPTIONS ]
<files>
```

When invoked on MacOS, the paths to point to the configuration file or the source files can be relative, but they will be computed from the location of the eclipse.ini file. This is a limitation of the Eclipse launcher on MacOS. On all other platforms, the relative paths are computed relative to the current user directory.

<files>	Java source files and/or directories to format. Only files ending with .java will be formatted in the given directory.

OPTIONS	Description
-config <file>	Use the formatting style from the specified properties file. Refer to Generating a config file for the formatter application for details.
-help	Display the help message.
-quiet	Only print error messages.
-verbose	Be verbose about the formatting job.

Related reference

Code formatter

Generating a Config File for the Formatter Application

Generating a config file for the formatter application involves modifying the code formatter settings for a Java project and copying org.eclipse.jdt.core.prefs out of the .settings directory for that project.

1. Select a Java project, open the pop-up menu and choose **Properties**.
2. Select the **Java Code Style > Formatter** page and check **Enable project specific settings**.
3. Select or edit a profile as explained above.
4. Click **OK** when you are done.
5. Use either a file manager or the command line to copy **`workspace/YourJavaProject/.settings/org.eclipse.jdt.core.prefs`** to a new location.
6. If the files you want to format are using 1.5 constructs, you must add the following lines inside the preference file you just created:

```
org.eclipse.jdt.core.compiler.compliance=1.5

org.eclipse.jdt.core.compiler.codegen.targetPlatform=1.5

          org.eclipse.jdt.core.compiler.source=1.5
```

Related reference

Code formatter

Breakpoint Enable Condition

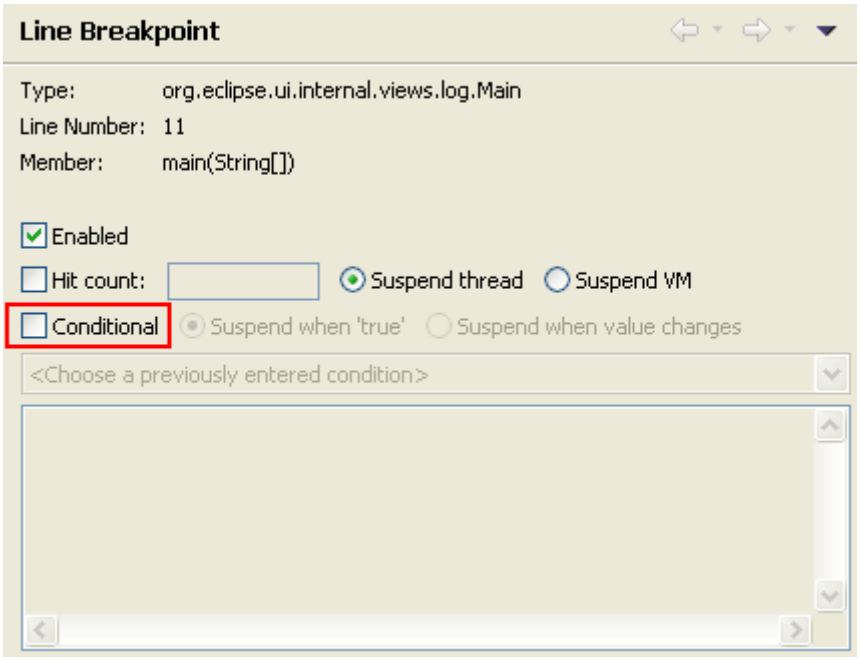
Select the **Conditional** option to enable the ability to provide a custom condition for the breakpoint.
Each breakpoint can have a unique condition that determines when the breakpoint will be hit.
A condition for a breakpoint can be any logical expression that evaluates to either true or false. The expression is evaluated in the scope of the breakpoint location, meaning you cannot make reference to a class, etc., outside the scope of the breakpoint location when composing your expression.
Consider the following example:

```
public class Person {


 String name = "";
 int age = 0;


 public Person(String name, int age) {
  this.name = name;
  this.age = age;  //breakpoint here
 }
};
```

If we take the above example, place a breakpoint where indicated and go to the breakpoint properties we can add our condition. In this case we are limited only to the members of the class the breakpoint is contained in and those provided by Object.
For example a valid condition could be:

```
 age == 56
```

meaning the breakpoint would only suspend when age was equal to 56.
Conditions can be added to breakpoints in the **Breakpoints View** detail pane or with the **Breakpoints Properties...** shown below.



Related concepts

Breakpoints

Related tasks

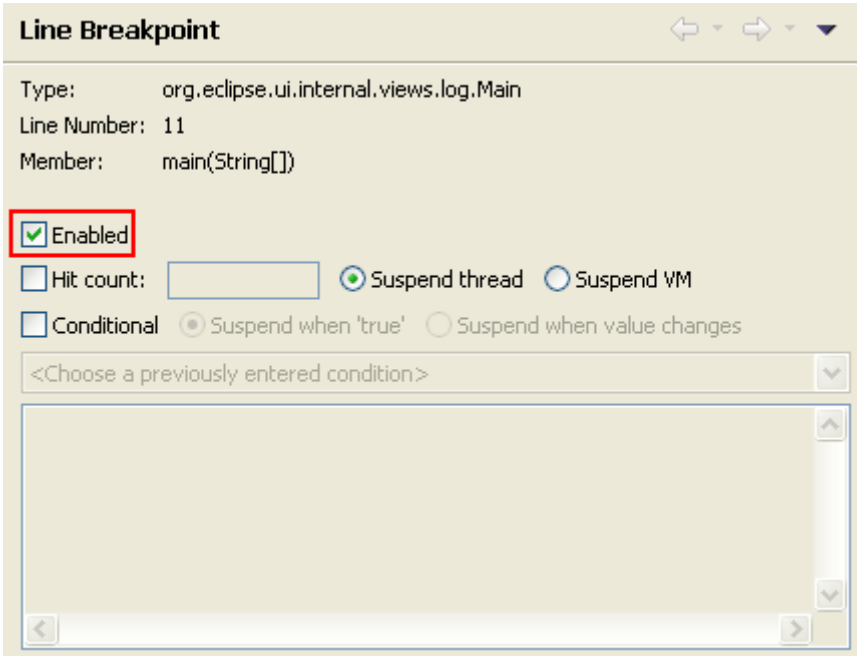
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Enabled Option
Hit Count
Suspend Policy
Method Entry
Method Exit

Breakpoint Enabled

Select the **Enabled** option to enable or disable a breakpoint.
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, by right-clicking a breakpoint, or by selecting the breakpoint or its grouping in the **Breakpoints View**.



Related concepts

Breakpoints

Related tasks

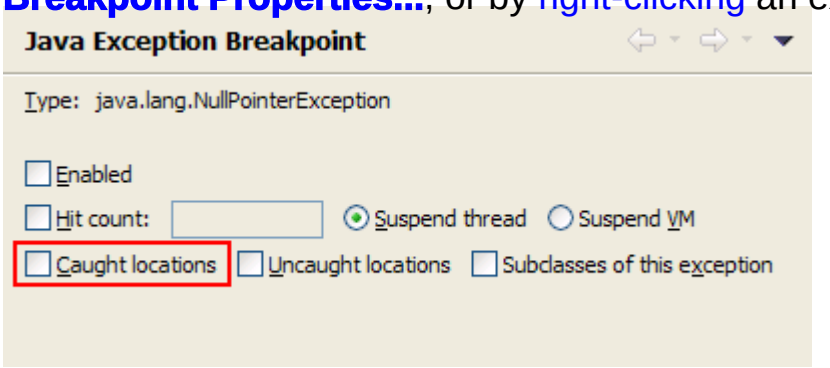
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Hit Count
Suspend Policy

Exception Breakpoint Caught Exception

Select the **Caught locations** option to suspend execution when an exception of the same type as the breakpoint is thrown in a caught location.
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking an exception breakpoint.



Related concepts

Breakpoints

Related tasks

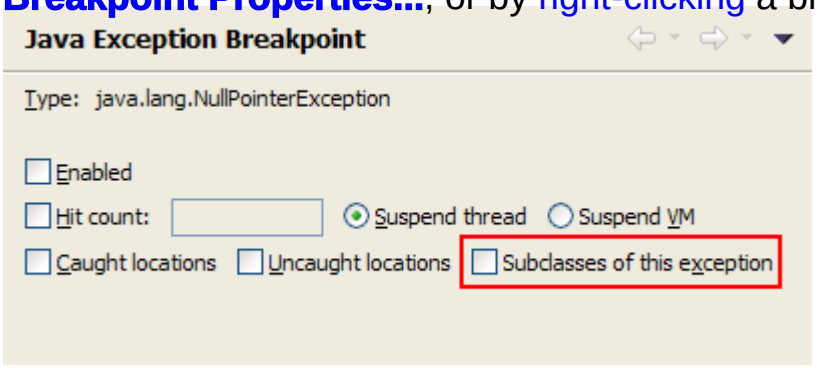
Catching exceptions
Creating Exception Breakpoint Filters
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Add Java exception breakpoint
Enabled Option
Hit Count
Suspend Policy

Exception Breakpoint Suspend on Subclass of this Exception

Select the **Subclasses of this exception** option to suspend execution when subclasses of the exception type are encountered.
For example, if an exception breakpoint for `RuntimeException` is configured to suspend on subclasses, it will also be triggered by a `NullPointerException`.
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a breakpoint.



Related concepts

Breakpoints

Related tasks

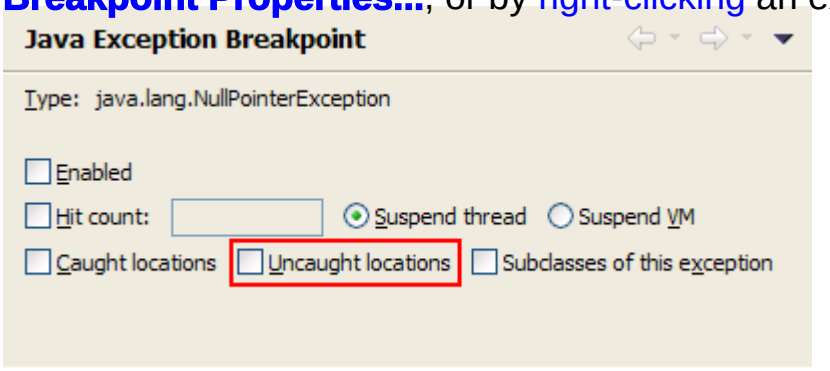
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Add Java exception breakpoint
Enabled Option
Hit Count
Suspend Policy
Caught Option
Uncaught Option

Exception Breakpoint Uncaught Exception

Select the **Uncaught locations** option to suspend execution when an exception of the same type as the breakpoint is thrown in an uncaught location.
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking an exception breakpoint.



Related concepts

Breakpoints

Related tasks

Catching exceptions
Creating Exception Breakpoint Filters
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

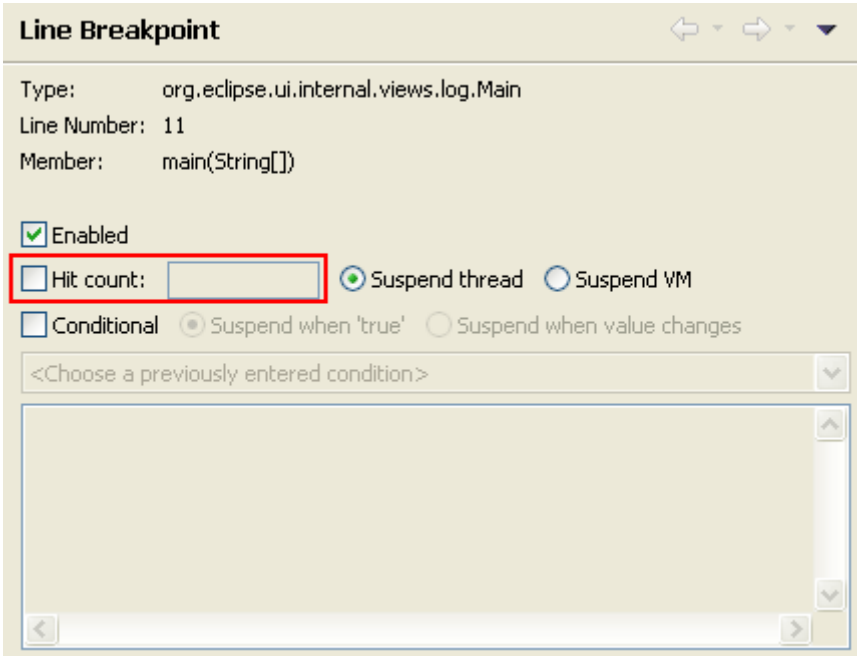
Related reference

Add Java exception breakpoint

Breakpoint Hit Count

Select the **Hit Count** option to define a hit count for the selected breakpoint. The **Hit Count** option is used to determine when your program should suspend on that breakpoint.
If a breakpoint has a hit count of *N*, execution will suspend when the breakpoint is encountered for the *N*th time. After being hit, the breakpoint is disabled until either it is re-enabled or its hit count is changed.

This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a breakpoint.



Related concepts

Breakpoints

Related tasks

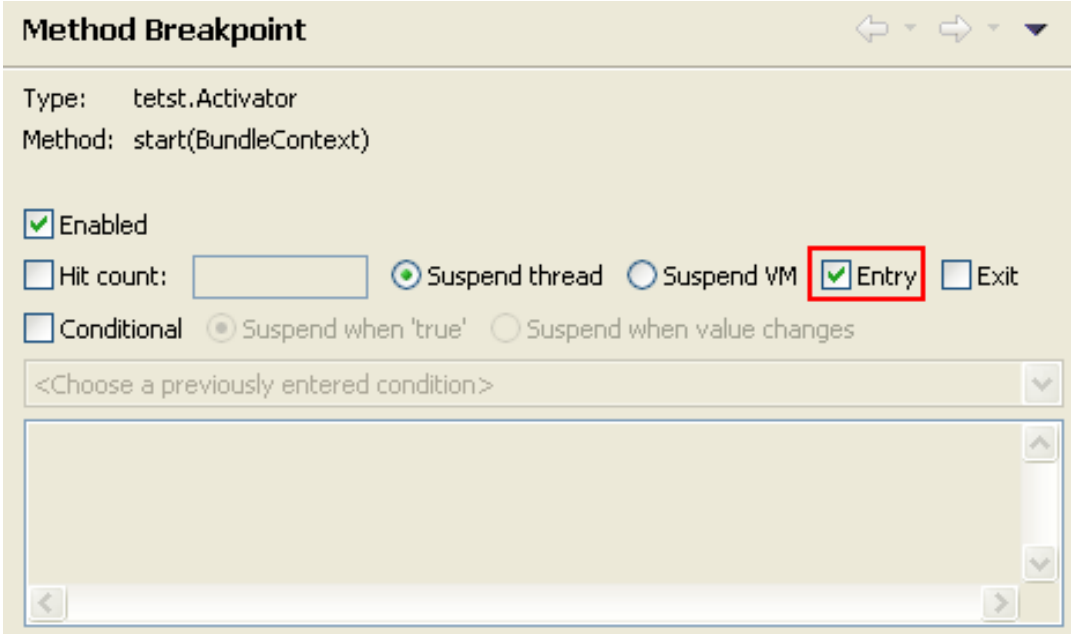
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Enabled Option
Suspend Policy

Breakpoint Method Entry

Select the **Entry** option to suspend execution when the method associated with the breakpoint is entered.
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a method breakpoint.



Related concepts

Breakpoints

Related tasks

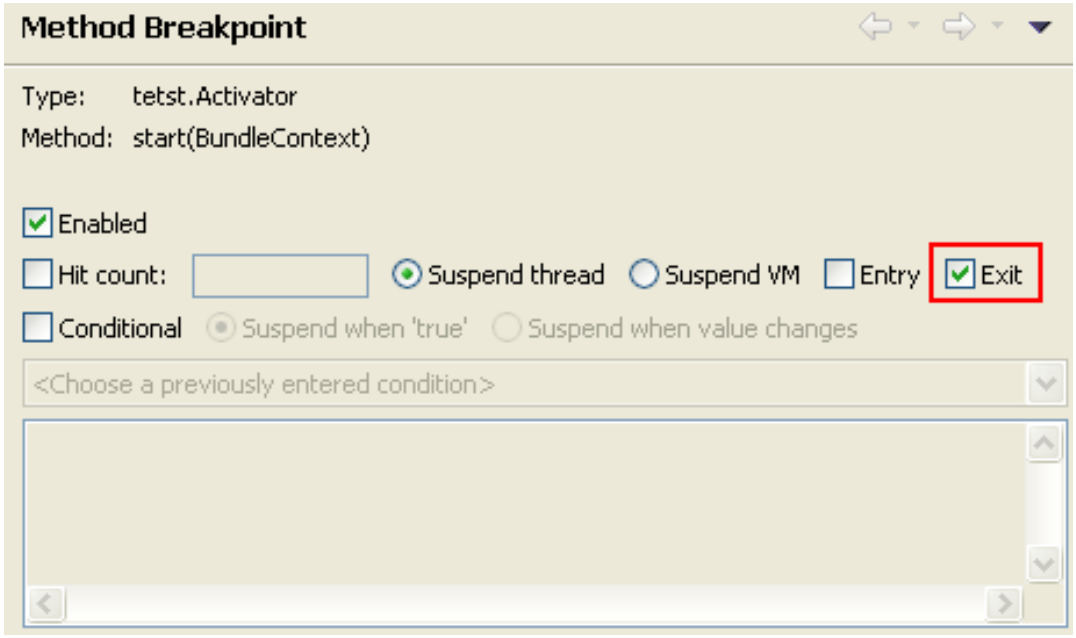
Setting method breakpoints
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Enabled Option
Hit Count
Suspend Policy
Method Exit
Breakpoint Conditions

Breakpoint Method Exit

Select the **Exit** option to suspend execution when the method associated with the breakpoint is exited.
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a method breakpoint.



Related concepts

Breakpoints

Related tasks

Setting method breakpoints
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

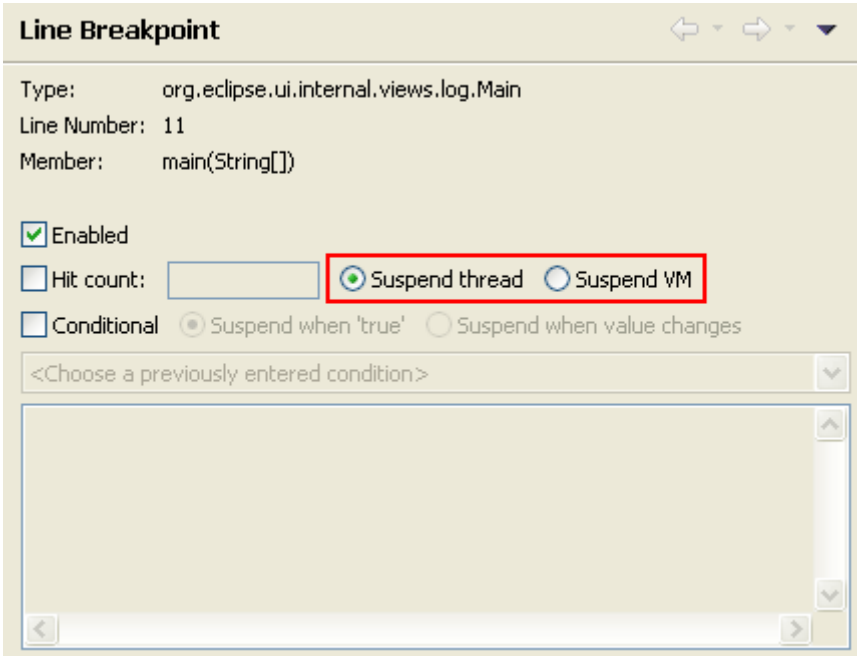
Related reference

Enabled Option
Hit Count
Suspend Policy
Method Exit
Breakpoint Conditions

Breakpoint Suspend Policy

A breapoint's suspend policy determines what should be suspended when a breakpoint is hit.
There are two options for Java breakpoints:
- **Suspend thread** - suspends only the thread that encountered the breakpoint
- **Suspend VM** - suspends the entire VM when the breakpoint is encountered

This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a breakpoint.



You can also change the default setting for the suspend policy for all newly created breakpoints.
To do this go to the [Image: /topic/org.eclipse.help/command_link.png]**Java >Debug** preference page and change it there.

Related concepts

Breakpoints

Related tasks

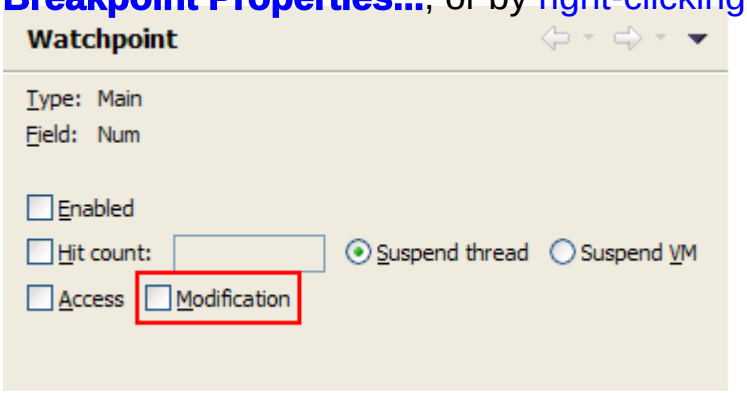
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Enabled Option
Hit Count

Watchpoint Field Modification

Select the **Modification** option to suspend execution when the associated field is modified (written).
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a watchpoint.



Related concepts

Breakpoints

Related tasks

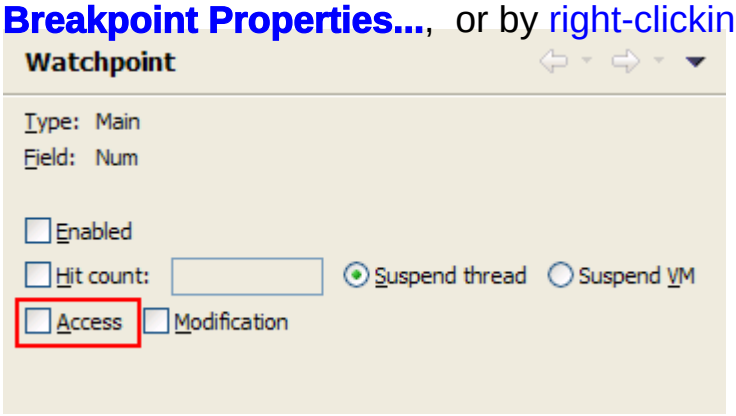
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Opening the Breakpoint Properties Dialog
Watchpoint Modification Context Action
Enabled Option
Hit Count
Suspend Policy

Watchpoint Field Access

Select the **Access** option to suspend execution when the associated field is accessed (read).
This option can be changed in the **Breakpoints View** detail pane, with the **Breakpoint Properties...**, or by right-clicking a watchpoint.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Opening the Breakpoint Properties Dialog
Watchpoint Access Context Action
Enabled Option
Hit Count
Suspend Policy

JDT Actions

JDT actions are available from

- Menu bar
- Toolbar
- Context menus in views

Related concepts

Java development tools (JDT)

Related reference

Frequently asked questions on JDT
JDT glossary
File menu actions
Edit menu actions
Source menu actions
Refactor menu actions
Navigate menu actions
Search menu actions
Project menu actions
Run menu actions
Java toolbar actions
Run and debug toolbar actions
Java editor actions

File actions

File menu commands:

Name	Function	**Keyboard Shortcut**
New	Create a Java element or a new resource. Configure which elements are shown in the submenu in **Window > Customize Perspective**. In a Java perspective, by default action for creating a project, package, class, interface, enum, annotation, source folder, Java working set, JUnit test, file and folder are available.	Alt + Shift + N
Open File	Opens a file from the file system in an editor.	
Close	Close the current editor. If the editor contains unsaved data, a save request dialog will be shown.	Ctrl + W
Close All	Close all editors. If editors contains unsaved data, a save request dialog will be shown.	Ctrl + Shift + W
Save	Save the content of the current editor. Disabled if the editor does not contain unsaved changes.	Ctrl + S
Save As	Save the content of the current editor under a new name.	

Save All	Save the content of all editors with unsaved changes. Disabled if no editor contains unsaved changes.	Ctrl + Shift + S
Revert	Revert the content of the current editor back to the content of the saved file. Disabled if the editor does not contain unsaved changes.	
Move	Move a resource. Disabled on Java Elements. To move Java elements use **Refactor > Move** (with updating all references to the file) or **Edit > Cut / Paste** (no updating of references).	
Rename	Renames a resource. Disabled on Java Elements. To rename Java elements use **Refactor > Rename** (with updating all references to the file).	
Refresh	Refreshes the content of the selected element with the local file system. When launched from no specific selection, this command refreshes all projects.	F5
Convert Line Delimiters To	Convert the line delimiters of all the text files in the selection to the selected delimiter kind.	
Print	Prints the content of the current editor. Enabled when an editor has the focus.	Ctrl + P

Switch Workspace	Switch to another workspace.	
Restart	Restarts the application.	
Import	Opens the import wizard dialog. JDT does not contribute any import wizards.	
Export	Opens the export wizard dialog. JDT contributes the JAR file export wizard and the Javadoc generation wizard.	
Properties	Opens the property pages of the select elements. Opened on Java projects the Java Build Path page and the Javadoc Location page are available. For JAR archives, configure the JAR's Source Attachment and Javadoc Location here.	Alt + Enter
Exit	Exit Eclipse	

Related concepts

Java development tools (JDT)

Related tasks

Creating JAR files

Related reference

New Java project wizard
New Java package wizard
New Java class wizard
New Java enum wizard
New Java interface wizard
New Java annotation wizard
JAR file exporter
Javadoc generation
Javadoc location properties
Java build path properties
Source attachment properties

Edit actions

Edit menu commands shown when a Java editor is visible:

Name	Function	**Keyboard Shortcut**
Undo	Revert the last change in the editor	Ctrl + Z
Redo	Revert an undone change	Ctrl + Y
Cut	Copies the currently selected text or element to the clipboard and removes the element. On elements, the remove is not performed before the clipboard is pasted.	Ctrl + X
Copy	Copies the currently selected text or elements to the clipboard	Ctrl + C
Copy Qualified Name	Copies the fully qualified name of the currently selected element to the clipboard	
Paste	Paste the current content as text to the editor, or as a sibling or child element to the a currently selected element.	Ctrl + V
Delete	Delete the current text or element selection.	Delete
Select All	Select all the editor content.	Ctrl + A

Expand Selection To	Enclosing Element: Selects the enclosing expression, block, or method in the code. This action is aware of the Java syntax. It may not function properly when the code has syntax errors. (Arrow Up) Next Element: Selects the current and next elements. (Arrow Right) Previous Element: Selects the current and previous elements. (Arrow Left) Restore Last Selection: Restores the previous selection after an invocation of *Expand Selection To*. (Arrow Down)	Alt + Shift + Arrow Keys
Find / Replace	Open the Find / Replace dialog. Editor only.	Ctrl + F
Find Next	Finds the next occurrence of the currently selected text. Editor only.	Ctrl + K
Find Previous	Finds the previous occurrence of the currently selected text. Editor only.	Ctrl + Shift + K
Incremental Find Next	Starts the incremental find mode. After invocation, enter the search text as instructed in the status bar. Editor only.	Ctrl + J

Incremental Find Previous	Starts the incremental find mode. After invocation, enter the search text as instructed in the status bar. Editor only.	Ctrl + Shift + J
Add Bookmark	Add a bookmark to the current text selection or selected element.	
Add Task	Add a user defined task to the current text selection or selected element.	
Smart Insert Mode	Toggles the Insert Mode. When smart insert mode is disabled, typing aids like automatic indentation, closing of brackets etc. are disabled.	Ctrl + Shift + Insert
Show Tooltip Description	Shows the value of a hover that would appear at the current cursor location. The dialog shown is scrollable and does not shorten descriptions.	F2

Content Assist	Opens a content assist dialog at the current cursor position. By default content assist supports five different categories of proposals. Default - union all of proposalsType Proposals *(e.g. java.lang.String, List)*Other Java Proposals *(e.g. method or field names)*Template Proposals *(e.g. 'iterate over array' template)*Word Proposals *(Proposes words which have been typed already)* See the Java Editor preference page for configuring the behavior of code assist and the contents of the *Default* proposal category. See the Templates preference page for available templates.	Ctrl + Space
Word Completion	Proposes word completions for the current string based on all words found in any open editor. Works in all text based editors.	Alt + /
Quick Fix	Opens a dialog with possible solutions if the cursor is located near a problem indication.	Ctrl + 1
Set Encoding	Toggles the encoding of the currently shown text content.	

Related concepts

Java editor

Java development tools (JDT)

Related reference

Java editor
Java editor preferences
Java outline
Views and editors

Source Actions

Source menu commands:

Name	Function	**Keyboard Shortcut**
Toggle Comment	Comment or uncomment all lines containing the current selection.	Ctrl + /
Add Block Comment	Adds a block comment around all lines containing the current selection.	Ctrl + Shift + /
Remove Block Comment	Removes a block comment from all lines containing the current selection.	Ctrl + Shift + \
Generate Element Comment	Adds a comment to the selected element. See the Code templates preference page to specify the format of the generated comments. Available on types, fields, constructors, and methods.	Alt + Shift + J
Shift Right	Increments the level of indentation of the currently select lines. Only activated when the selection covers multiple lines or a single whole line.	
Shift Left	Decrements the level of indentation of the currently select lines. Only activated when the selection covers multiple lines or a single whole line.	
Correct Indentation	Corrects the indentation of the lines denoted by the current text selection.	Ctrl + I

Format	Uses the code formatter to format the current text selection. The formatting options are configured on the Code Formatter preference page.	Ctrl + Shift + F
Format Element	Uses the code formatter to format the Java element comprising the current text selection. The Format Element action works on method and type level. The formatting options are configured on the Code Formatter preference page.	
Add Import	Creates an import declaration for a type reference currently selected. If the type reference if qualified, the qualification will be removed if possible. If the referenced type name can not be mapped uniquely to a type of the current project you will be prompted to specify the correct type. Add Import tries to follow the import order as specified in the Organize Import preference page	Ctrl + Shift + M

Organize Imports	Organizes the import declarations in the compilation unit currently open or selected. Unnecessary import declarations are removed, and required import declarations are ordered as specified in the Organize Imports preference page. Organize imports can be executed on incomplete source and will prompt you when a referenced type name can not be mapped uniquely to a type in the current project. You can also organize multiple compilation units by invoking the action on a package or selecting a set of compilation units.	Ctrl + Shift + O
Sort Members	Sorts the members of a type according to the sorting order specified in the Member Sort Order preference page	
Clean Up	Performs various changes in order to clean up your code according to the settings specified in the Clean Up preference page	

Override/Implement Methods	Opens the Override Method dialog that allows you to override or implement a method in the current type. Available on types or on a text selection inside a type.	
Generate Getter and Setter	Opens the Generate Getters and Setters dialog that allows you to create Getters and Setters for fields in the current type. Available on fields and types or on a text selection inside a type.	
Generate Delegate Methods	Opens the Generate Delegate Methods dialog that allows you to create method delegates for fields in the current type. Available on fields and types with fields.	
Generate hashCode() and equals()	Opens the Generate HashCode and Equals dialog that allows you to start and control the generation of hashCode and equals methods in the current type.	
Generate toString()	Opens the Generate toString() dialog that allows you to start and control the generation of a toString() method in the current type.	
Generate Constructor using Fields	Adds constructors which initialize fields for the currently selected types. Available on types, fields or on a text selection inside a type.	

Add Constructor from Superclass	Adds constructors as defined in the super class for the currently selected types. Available on types or on a text selection inside a type.	
Surround With	Surround the selected statements with a code template. Create your own templates on the Template preference page. Further, you can use *Expand Selection to* from the Edit menu to get a valid selection range.	Alt + Shift + Z
Externalize Strings	Opens the Externalize strings wizard. This wizards allows you to replace all strings in the code by statements accessing a property file.	
Find Broken Externalized Strings	Searches for broken externalized strings in a selected property file, package, project or set of projects.	

Related concepts

Java editor
String externalization
Java development tools (JDT)

Related tasks

Externalizing strings

Related reference

Java editor
Java editor preferences
Java outline
Views and editors

Refactor Actions

Refactor menu commands:

Name	Description
Rename	Renames the selected element and (if enabled) corrects all references to the elements (also in other files).
Move	Moves the selected elements and (if enabled) corrects all references to the elements (also in other files).
Change Method Signature	Changes parameter names, parameter types, parameter order and updates all references to the corresponding method. Additionally, parameters and thrown exceptions can be removed or added and method return type and method visibility can be changed.
Extract Method	Creates a new method containing the statements or expression currently selected and replaces the selection with a reference to the new method. This feature is useful for cleaning up lengthy, cluttered, or overly-complicated methods.
Extract Local Variable	Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.
Extract Constant	Creates a static final field from the selected expression and substitutes a field reference, and optionally rewrites other places where the same expression occurs.
Inline	Inline local variables, methods or constants.
Convert Anonymous Class to Nested	Converts an anonymous inner class to a member class.

Move Type to New File	Creates a new Java compilation unit for the selected member type or the selected secondary type, updating all references as needed. For non-static member types, a field is added to allow access to the former enclosing instance, if necessary.
Convert Local Variable to Field	Turn a local variable into a field. If the variable is initialized on creation, then the operation moves the initialization to the new field's declaration or to the class's constructors.
Extract Superclass	Extracts a common superclass from a set of sibling types. The selected sibling types become direct subclasses of the extracted superclass after applying the refactoring.
Extract Interface	Creates a new interface with a set of methods and makes the selected class implement the interface.
Use Supertype Where Possible	Replaces occurrences of a type with one of its supertypes after identifying all places where this replacement is possible.
Push Down	Moves a set of methods and fields from a class to its subclasses.
Pull Up	Moves a field or method to a superclass of its declaring class or (in the case of methods) declares the method as abstract in the superclass.
Extract Class	Replaces a set of fields with new container object. All references to the fields are updated to access the new container object.
Introduce Parameter Object	Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduce parameter.
Introduce Indirection	Creates a static indirection method delegating to the selected method.

Introduce Factory	Creates a new factory method, which will call a selected constructor and return the created object. All references to the constructor will be replaced by calls to the new factory method.
Introduce Parameter	Replaces an expression with a reference to a new method parameter, and updates all callers of the method to pass the expression as the value of that parameter.
Encapsulate Field	Replaces all references to a field with getter and setter methods.
Generalize Declared Type	Allows the user to choose a supertype of the reference's current type. If the reference can be safely changed to the new type, it is.
Infer Generic Type Arguments	Replaces raw type occurrences of generic types by parameterized types after identifying all places where this replacement is possible.
Migrate JAR File	Migrates a JAR File on the build path of a project in your workspace to a newer version, possibly using refactoring information stored in the new JAR File to avoid breaking changes.
Create Script	Creates a script of the refactorings that have been applied in the workspace. Refactoring scripts can either be saved to a file or copied to the clipboard. See *Apply Script*.
Apply Script	Applies a refactoring script to projects in your workspace. Refactoring scripts can either be loaded from a file or from the clipboard. See *Create Script*.
History	Browses the workspace refactoring history and offers the option to delete refactorings from the refactoring history.

Refactoring commands are also available from the context menus in many views and the Java editor.

Related concepts

Refactoring support

Related reference

Refactoring dialogs
Java preferences

Navigate Actions

Navigate menu commands:

Name	Function	**Keyboard Shortcut**
Go Into	Sets the view input to the currently selected element. Supported by the Package Explorer view.	
Go To	Back: Sets the view input to the input back in history: Only enabled when a history exists (**Go Into** was used) Forward: Sets the view input to the input forward in history: Only enabled when a history exists (**Go Into**, **Go To > Back** were used) Up One Level: Sets the input of the current view to its input's parent element Type: Browse for a type and reveal it in the current view. Supported by the Package Explorer view Package: Browse for a package and reveal it in the current view. Supported by the Package Explorer view Resource: Browse for a resource and reveal it in the current view.	
Open Declaration	Tries to resolve the element referenced at the current code selection and opens the file declaring the reference.	F3

Open Type Hierarchy	Tries to resolve the element referenced at the current code selection and opens the element in the Type Hierarchy view. Invoked on elements, opens the type hierarchy of the element. Supported in the Java editor and views showing Java elements.	F4
Open Call Hierarchy	Tries to resolve the method referenced at the current code selection and opens the element in the Call Hierarchy view.	Ctrl+Alt+H
Open Implementation	Opens the implementation of the currently selected method if there's only one implementor, else opens the Quick Type Hierarchy on that method.	
Open Super Implementation	Open an editor for the super implementation of the currently selected method or method surrounding the current cursor position. No editor is opened if no method is selected or the method has no super implementation.	

Open Attached Javadoc	Opens the Javadoc documentation of the currently selected element or text selection. The location of the Javadoc of a JAR or a project is specified in the Javadoc location property page on projects or JARs. Note that this attached Javadoc documentation may not be up to date with the Javadoc specified in the current code. You can create Javadoc documentation for source files in a Java project using the Javadoc export wizard.	Shift+F2
Open from Clipboard	Tries to open the matching Java element in the editor if the clipboard contains a single line. Otherwise it opens the contents in the Java Stack Trace Console. Examples: java.lang.StringString String#getBytesString .getBytesjava.lang.St ring.getBytes(String) String.java:123at java.lang.String.matc hes(String.java:1550) java.lang.String.value Of(char) line: 1456currentTimeMilli s()	
Open Type	Brings up the Open Type dialog to open a type in the editor. The Open Type selection dialog shows all types existing in the workspace.	Ctrl+Shift+T

Open Type In Hierarchy	Brings up the Open Type dialog to open a type in the editor and the Type Hierarchy view. The Open Type selection dialog shows all types that exist in the workspace.	Ctrl+Shift+H
Open Resource	Opens the Open Resource dialog to open any resource in your workspace.	Ctrl+Shift+R
Show In Breadcrumb	Shows the currently selected element in the breadcrub bar of the Java editor.	Ctrl+Shift+B
Show In	Choose to show the currently selected element in Package ExplorerHistoryNavigatorOutline	Alt+Shift+W
Quick Outline	Opens the lightweight outliner for the currently selected type. Pressing Ctrl+O a second time will show all inherited fields, types and methods as well (for the types marked with ▸ on the left). Inherited members are shown in blue.	Ctrl+O
Quick Type Hierarchy	Opens the lightweight hierarchy viewer for the currently selected type. Pressing Ctrl+T while the type hierarchy view is shown will toggle between supertype hierarchy and subtype hierarchy.	Ctrl+T
Last Edit Location	Reveals the location where the last edit occurred.	Ctrl+Q

Go to Line	Opens a dialog which allows entering the line number to which the editor should jump to. Editor only.	Ctrl+L
Back	Reveals the previous editor location in the location history.	Alt+Left
Forward	Reveals the next editor location in the location history.	Alt+Right

Related concepts

Java views
Java development tools (JDT)

Related tasks

Opening an editor for a selected element
Showing an element in the package explorer
Opening a type in the package explorer
Opening an editor on a type
Opening a package

Related reference

Package Explorer view
Type Hierarchy view
Javadoc location properties
Javadoc export wizard

Search Actions

Search menu commands:

Name	Function	**Keyboard Shortcut**
Search...	Opens the search dialog	Ctrl + H
File...	Opens the search dialog on the File search page	
Java...	Opens the search dialog on the Java search page	
Text	Searches for the selected text in the chosen scope: Workspace (**Ctrl + Alt + G**)ProjectHierarchyWorking Set	
References	Finds all references to the selected Java element in the chosen scope: Workspace (**Ctrl + Shift + G**)ProjectHierarchyWorking Set	
Declarations	Finds all declarations of the selected Java element in the chosen scope: Workspace (**Ctrl + G**)ProjectHierarchyWorking Set	
Implementors	Finds all implementors of the selected interface in the chosen scope: (Workspace, Project or Working Set) WorkspaceProjectWorking Set	
Read Access	Finds all read accesses to the selected field in the chosen scope: WorkspaceProjectHierarchyWorking Set	

Write Access	Finds all write accesses to the selected field in the chosen scope: WorkspaceProjectHierarchyWorking Set	
Occurrences in File	Finds all occurrences of the selected Java element in its file Identifier: Occurrences of all identifiers resolving to the selected elementImplementing Methods: All methods implemented by the selected super-interface declarationThrowing Exceptions: All statements that possible throw the selected declared exceptionMethod Exits: All statements that can exit the method of the selected return typeBreak/Continue Target: The targets of the selected break and continue statement	Ctrl + Shift + U
Referring Tests	Finds all JUnit tests that refer to the currently selected type	

Search scopes submenu:

Scope	**Availability**	Description
Workspace	all elements	Searches in the full workspace
Project	all elements	Searches in the project enclosing the selected element
Hierarchy	types and members	Searches in the type's hierarchy
Workings Set	all elements	Searches in a working set

Scopes can be saved and names in the working set dialog. Existing instances of working sets are also available in the Search Scope submenu

A Java search can also be conducted via the context menu of all Java views. The search context menu is also available in the Java editor. The search is only performed if the currently selected text can be resolved to a Java element.

The type of the selected Java element defines which search context menus are available. The Java editor does not constrain the list of available Java searches based on the selection.

- Related concepts

Java search

- Related reference

Java search tab

Project Actions

Project menu commands:

Name	Function	**Keyboard Shortcut**
Open Project	Shows a dialog that can be used to select a closed project and open it	
Close Project	Closes the currently selected projects	
Build All	Builds all projects in the workspace. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	Ctrl + B
Build Project	Builds the currently selected project. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	
Build Working Set	Builds the projects contained in the currently selected working set. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	
Clean...	Shows a dialog where the projects to be cleaned can be selected.	

Build Automatically	If selected, all modified files are automatically rebuilt if saved. This is an incremental build, meaning that the builder analyzes the changes since the last time of build and minimizes the number of changed files.	
Generate Javadoc...	Opens the Generate Javadoc wizard on the currently selected project.	
Properties	Opens the property pages on the currently selected project.	

Related concepts

Java projects
Java builder

Run Menu Actions

The **Run Menu**, found on the main menu of the **Eclipse** workbench contains all of the actions required run, debug, step through code and work with breakpoints. Different parts of the
menu are visible at different times, as each perspective can be customized to show only specific capabilities.
The **Run Menu** commands.

Run Menu Commands

Command	Name	Description	Shortcut
	**Add Java Exception Breakpoint**	Opens the Add Java Exception dialog to select the exception breakpoint type to create	
	**Add Class Load Breakpoint**	Opens the Select Type dialog to select the Java type to add a class load breakpoint to	
	**All Instances...**	Displays all of the instances of the selected type in the current VM in a popup (only available with a Java 1.6 VM)	Ctrl + Shift + N
	**All References...**	Displays all of the references of the selected type in the current VM in a popup (only available with a Java 1.6 VM)	
	Debug As	Presents a sub menu of registered debug launch shortcuts. Launch shortcuts provide support for workbench or active editor selection sensitive launching.	
	Debug History	Presents a sub menu of the recent history of launch configurations launched in debug mode	

	**Debug Last Launched / Debug**	Allows you to quickly repeat the most recent launch in debug mode or to debug the current selection, if that mode is supported (based on your current launch settings).	F11
	**Display**	Uses the Display view to show the result of evaluating the selected expression in the context of a stack frame or variable in that thread. If the current active part is a Java Snippet Editor, the result is displayed there.	Ctrl + Shift + D
	**Execute**	Within the context of the Java snippet editor, this command allows you to evaluate an expression but does not display a result.	Ctrl + E
	Force Return	Forces the current method to return with the specified value	Alt + Shift + F
	**Inspect**	Uses the Expressions view to show the result of inspecting the selected expression or variable in the context of a stack frame or variable in that thread.	Ctrl + Shift + I
	**Open Debug Dialog...**	This command opens the launch configuration dialog to manage debug mode launch configurations.	
	**Open Run Dialog...**	Opens the launch configuration dialog to manage run mode launch configurations.	

	**Remove All Breakpoints**	Removes all of the breakpoints from the current workspace	
	**Resume**	Resumes a suspended thread.	F8
	Run As	Presents a sub menu of registered run launch shortcuts. Launch shortcuts provide support for workbench or active editor selection sensitive launching.	
	Run History	Presents a sub menu of the recent history of launch configurations launched in run mode	
	**Run Last Launched / Run**	Allows you to quickly repeat the most recent launch in run mode or quickly run the selected resource, if that mode is supported (based on your current launch settings).	Ctrl + F11
	**Run To Line**	Executes the program to the currently selected line in the editor	Ctrl + R
	**Step Into**	Steps into the current statement.	F5
	Step Into Selection	Steps into the selected statement in the editor	Ctrl + F5

	**Step Over**	Steps over the highlighted statement. Execution will continue at the next line either in the same method or (if you are at the end of a method) it will continue in the method from which the current method was called. The cursor jumps to the declaration of the method and selects this line.	F6
	**Step Return**	Steps out of the current method. This option stops execution after exiting the current method.	F7
	**Suspend**	Suspends the selected thread of a target so that you can browse or modify code, inspect data, step, and so on.	
	**Terminate**	Terminates the selected debug target.	Ctrl + F2
	**Toggle Breakpoint**	Toggles the appropriate type of breakpoint based on what is currently selected	Ctrl + Shift + B
	**Toggle Line Breakpoint**	Toggles a line breakpoint on the current executable line of code	
	**Toggle Method Breakpoint**	Toggles a method breakpoint on the currently selected method declaration	
	**Toggle Watchpoint**	Toggles a watchpoint on the currently selected field	

	**Use Step Filters**	Toggles step filters on/off. When on, all step functions apply step filters.	Shift + F5
	**Watch**	Used to create a watch item. A watch item is an expression in the Expressions view whose value is updated as you debug.	

Related concepts

Debugger
Java development tools (JDT)
Local Debugging
Remote Debugging

Related tasks

Running and Debugging
Connecting to a remote VM with the Remote Java application launch configuration
Adding Line breakpoints
Setting method breakpoints
Catching exceptions

Related reference

Debug Preferences
Run and Debug toolbar actions

Java Toolbar Actions

Java Actions

Toolbar Button	Command	Description
	Create a Java Project	This command helps you create a new Java project. See new Java project wizard
	Create a Java Package	This command helps you create a new Java package. See new Java package wizard
	Create a Java Class	This command helps you create a new Java class. See new Java class wizard
	Create a Java enum	This command helps you create a new Java enum. See new Java enum wizard
	Create a Java Interface	This command helps you create a new Java interface. See new Java interface wizard
	Create a Java annotation	This command helps you create a new Java annotation. See new Java annotation wizard

	Create a JUnit test case	This command helps you create a new JUnit test case.
	Open Type	This command allows you to browse the workspace for a type to open in the defined default Java editor. You can optionally choose to display the type simultaneously in the Type Hierarchy view. See Open Type dialog.

Related concepts

Java development tools (JDT)

Related tasks

Opening an editor on a type

Related reference

New Java project wizard
New Java package wizard
New Java class wizard
New Java enum wizard
New Java interface wizard
New Java annotation wizard
New Java scrapbook page wizard
Views and editors

Run and Debug toolbar actions

Run and Debug toolbar actions

Toolbar Button	Command	Description
	**Run**	This command re-launches the most recently launched application, or launches the selected resource or active editor depending on the launch operation preference settings found on the **Run/Debug > Launching** preference page
	**Debug**	This command re-launches the most recently launched application under debugger control, or launches the selected resource or active editor depending on the launch operation preference settings found on the **Run/Debug > Launching** preference page

Related concepts

Debugger
Local Debugging
Remote Debugging

Related tasks

Running and Debugging
Connecting to a remote VM with the Remote Java application launch configuration
Line breakpoints
Setting method breakpoints
Catching exceptions

Related reference

Debug View
Debug Preferences
Launching Preferences
Run menu actions

Java editor
Toolbar actions

Toolbar Button	Command	Description
	Toggle Java Editor Breadcrumb	This button enables the Java editor breadcrumb. The enablement is remembered for each perspective separately.
	Toggle Mark Occurrences	Turns mark occurrences on and off in the Java editor.
	Toggle Block Selection Mode	This button enables block (aka column) selection mode in the editor.
	Show Whitespace Characters	This button enables the display of whitespace characters in an editor.

	Show Source of Selected Element Only	This button enabled display of a segmented view of the source of a compilation unit. This button is only shown if you customize your perspective to show the **Editor Presentation** actions. For example, if a method is selected in the Outline view, the **Show Source Of Selected Element Only** option causes only that method to be displayed in the editor, as opposed to the entire class. **Off**: The entire compilation unit is displayed in the editor, with the selected Java element highlighted in the marker bar with a range indicator. **On**: Only the selected Java element is displayed in the editor, which is linked to the selection in the Outline or Type Hierarchy view.
	Go to Next Problem	This command navigates to the next problem marker in the active editor.
	Go to Previous Problem	This command navigates to the previous problem marker in the active editor.

Key binding actions

The following actions can only be reached through key bindings. The **Key bindings** field in **Window > Preferences > General > Keys** must be set to 'Emacs'.

Key binding	Description
`Alt+0Ctrl+K`, `Esc0Ctrl+K`	Deletes from the cursor position to the beginning of the line.
`Ctrl+K`	Deletes from the cursor position to the end of the line.
`Ctrl+Space`, `Ctrl+2`	Sets a mark at the current cursor position.
`Ctrl+XCtrl+X`	Swaps the cursor and mark position if any.

Related concepts

Java editor

Related tasks

Opening an editor for a selected element

Related reference

Java outline
Java editor preferences
JDT actions
Views and editors

Java Preferences

The following Java preferences can be set on the [Image: /topic/org.eclipse.help/command_link.png]**Java** preference page and its sub-pages.

- Appearance
- Build Path
- Code Style
- Compiler
- Debug
- Editor
- Installed JREs
- JUnit
- Properties File Editor

Option	Description	Default
Action on double click in the Package Explorer	**Go into the selected element** When you double click a container, a **Go Into** command is executed. See **Go Into** from the Navigate menu. **Expand the selected element** When you double click a container, it is expanded and its children are revealed.	Expand the selected element

When opening a Type Hierarchy	**Open a new Type Hierarchy Perspective** Opens a new Type Hierarchy perspective whenever a Type Hierarchy view is opened. **Show the Type Hierarchy View in the current perspective** The Type Hierarchy view is displayed in the current perspective. *Note: On the Workbench preferences page, you can choose whether new perspectives open in a new window, in the current window, or as a replacement for the current perspective.*	Show the Type Hierarchy View in the current perspective
Refactoring Java code	**Save all modified resources automatically prior to refactoring** If this option is turned off, refactorings may prompt to save modified files if required. If it is turned on, all modified files are saved without prompting prior to opening a refactoring wizard.	Off

	Rename in editor without dialog If this option is turned on, then the rename refactoring will not show a dialog if invoked inside a Java editor. Instead the new name for the element to rename can be typed into the editor right away.	On
Search	**Use reduced search menu** If this option is turned on, the search context menus show only the most frequently used search actions.	On
Java dialogs	**Clear all 'do not show again' settings and show all hidden dialogs again** If pressed, all 'do not show' settings are cleared.	Button

Related concepts

Java views

Related reference

Package Explorer view

Type Hierarchy view

Java Appearance Preferences

The appearance of Java elements in views can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Appearance** preference page and on its sub pages:

- Member sort order
- Type filters

The options are:

Option	Description	Default
Show method return types	If enabled, methods displayed in views show the return type.	Off
Show method type parameters	If enabled, methods displayed in views show their type parameters.	On
Show categories	If enabled, method, field and type labels contain the categories specified in their Javadoc comment.	On
Show members in Package Explorer	If enabled, Java elements below the level of Java files and Class files are displayed as well.	On
Fold empty packages in hierarchical layout	If enabled, empty packages which do not contain resources or other child elements are folded.	On
Compress package name segments	If enabled, package names, except for the final segment, are compressed according to the compression pattern.	Off
Abbreviate package names	If enabled, package names are abbreviated according to the specified abbreviation rules.	Off

Stack views vertically in the Java Browsing perspective	If enabled, views in Java Browsing perspective will be stacked vertically, rather than horizontally.	Off

Note: The **Colored Labels** settings is now available on the [Image: /topic/org.eclipse.help/command_link.png] General > Appearance preference page.

Related concepts

Java views

Related reference

Package Explorer view

Member Sort Order Preferences

Indicate your preferences for the Member Sort Order settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Appearance > Member Sort Order** preference page.
This settings are used by Java views as well as by the Sort Members Action.

Option	Description	Default
Choose the order in which members will be displayed.	**Up**Move members of selected category closer to the beginning of a type.**Down**Move members of selected category closer to the end of a type.	Types, Static Fields, Static Initializers, Static Methods, Fields, Initializers, Constructors, Methods
Sort members in same category by visibility	If enabled, members of same category are sorted by visibility within the category. **Up**Move members of selected visibility closer to the beginning of a category.**Down**Move members of selected visibility closer to the end of a category.	Off Public, Private, Protected, Default

Related concepts

Java views

Related reference

Package Explorer view
Source Actions

Type Filters Preferences

Indicate your preferences for the Type Filters settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Appearance > Type Filters** preference page.

Specify types and packages to hide in the 'Open Type' dialog and content assist or quick fix proposals on this preference page. If a qualified type name or a package match one of the checked entries in the list, they will be hidden.

The default is to hide nothing but forbidden references.

Option	Description
Add	Adds a new entry to the list. Entries will be compared against qualified type name (using dots as package name separators) and can use wild cards ('*' and '?')
Add Packages	Adds a new entry to the list from the list of existing packages currently in the workspace.
Edit	Edits the currently selected entry
Remove	Removes the currently selected entries
Check All	Sets the checkmark on all entries
Uncheck All	Removes the checkmark on all entries

Access Restrictions

Option	Description	Default
Hide forbidden references	If enabled, references to Java elements forbidden by access rules are not displayed.	On
Hide discouraged references	If enabled, references to Java elements discouraged by access rules are not displayed.	Off

Related concepts

Quick fix
Access Rules

Related reference

Open type
Content/Code assist

Java Build Path Preferences

Indicate your preferences for the Build Path settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Build Path** preference page and its sub pages:
- Classpath Variables
- User Libraries

The options are:

Option	Description	Default
Source and output folder	**Project**Use the project root folder as the folder where Java source files and the generated class files are stored by default.**Folders**Specify separate source and output folders to store the Java source files separated from the generated class files.	Folders Source Folder: src Output Folder: bin
As JRE library use	Select what kind of JRE library should be used for newly created Java projects.	JRE container

Related concepts

Classpath variables

Related reference

New Java project wizard

Classpath Variables Preferences

Configurable variables

Classpath variables can be used in a Java Build Path to avoid a reference to the local file system. Using a variable entry, the classpath only contains a variable and the build path can be shared in a team. The value of such variables is configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Build Path > Classpath Variables** preference page.

Command	Description
New...	Adds a new variable entry. In the resulting dialog, specify a name and path for the new variable. You can click the **File** or **Folder** buttons to browse for a path.
Edit...	Allows you to edit the selected variable entry. In the resulting dialog, edit the name and/or path for the variable. You can click the **File** or **Folder** buttons to browse for a path.
Remove	Removes the selected variable entry.

Reserved class path variables

Certain class path variables are set internally and can not be changed in the Classpath variables preferences:

- JRE_LIB: The archive with the runtime JAR file for the currently used JRE.
- JRE_SRC: The source archive for the currently used JRE.
- JRE_SRCROOT: The root path in the source archive for the currently used JRE.

These variables are deprecated and should not be used anymore. Instead use a JRE System Library.

Related concepts

Classpath variables

Related reference

Installed JREs

User Libraries Preferences

Indicate your preferences for the User Libraries settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Build Path > User Libraries** preference page.

A user library is a set of JAR files. A user library can be added to a projects build path through the build path properties page. The User Libraries preference page allows to define, edit, import, export, or remove user libraries.

Option	Description
New...	Creates a new user library
Edit...	Edits the currently selected library, JAR or JAR attribute
Add JARs...	Adds one or more JARs to the currently selected user library
Remove..	Removes the currently selected user library or JAR
Up	Moves the currently selected JAR up in the list of JARs of a user library. The order can be important if more than one JAR contain a type with the same qualified name.
Down	Moves the currently selected JAR down in the list of JARs of a user library. The order can be important if more than one JAR contain a type with the same qualified name.
Import...	Adds new libraries from an import file
Export...	Exports libraries to an import file

Related reference

Java Build Path page

Java Code Style Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style** preference page allows to configure naming conventions, style rules and comment settings. These preferences are used when new code has to be generated.
Sub pages offer more code style settings:
- Clean Up
- Code Templates
- Formatter
- Organize Imports

The options are:

Naming Conventions

The list defines the naming conventions for fields (static and non-static), parameters and local variables. For each variable type it is possible to configure a list of prefix or suffix or both.
Naming conventions are used by all actions and 'Quick Fix' proposals that create fields, parameters and local variables, in particular the Source actions.

Action	Description
Edit...	Opens a dialog to edit the list of prefix and suffixes for the currently selected variable type

Code Conventions

The following settings specify how newly generated code should look like. The names of getter methods can be specified as well as the format of field accesses, method comments, annotations and exception variables.

Action	Description	Default
Qualify all generated field accesses with 'this.'	If selected, field accesses are always prefixed with 'this', regardless whether the name of the field is unique in the scope of the field access or not.	Off
Use 'is' prefix for getters that return boolean	If selected, the names of getter methods of boolean type are prefixed with 'is' rather than 'get'.	On

Add '@Override' annotation for overriding methods	If selected, methods which override an already implemented method are annotated with an '@Override' annotation. See the Compiler preference page for settings related to annotations.	On
Exception variable name in catch blocks	Specify the name of the exception variable declared in catch blocks.	e

Related reference

Source actions
Java editor
Java editor preferences
Java compiler preferences
Code templates preferences

Clean Up Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Clean Up** preference page lets you manage your code clean up profiles for the Java code clean up.

Action	Description
Active profile	Selects the profile active for the workspace or current project.
Edit...	Shows a dialog which displays the settings stored in the active profile. Only user-defined profiles can be modified.
Remove	Removes the selected profile. This action is only available on user-defined profiles.
New...	Shows the dialog to create a new profile. The dialog requires you to enter a name for the new profile. Additionally, you may select a built-in or user-defined existing profile to base your new profile on.
Import...	Imports profiles from the file system.
Export All...	Exports all the profiles to the file system.
Details	Displays a summary of the enabled options of the active profile

Disable **Show profile selection dialog for the 'Source > Clean Up' action** to apply the active profile when you invoke **Source > Clean Up** without being disturbed by a dialog.

Related reference

Java editor
Java editor preferences

Code Templates Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Code Templates** preference page lets you configure the format of newly generated code and comments.

Code and Comments

The code and comment page contains code templates that are used by actions that generate code. Templates contain variables that are substituted when the template is applied. Some variables are available in all templates, some are specific to templates.

Action	Description	Default
Edit...	Opens the Code Template dialog to edit the currently selected code template.	n/a
Import...	Imports code templates from the file system.	n/a
Export...	Exports all selected code templates to the file system.	n/a
Export All...	Exports all code templates to the file system.	n/a
Automatically add comments for new methods and types	If selected, newly generated methods and types are automatically generated with comments where appropriate. The formats of the generated comments are defined by the Comment Templates.	Off

Comment Templates

Comment templates can contain the variable **${tags}** that will be substituted by the standard Javadoc tags (@param, @return..) for the commented element. The 'Overriding method' comment can additionally contain the template **${see_to_overridden}**

Template Name	Specifies
Files	Header comment for new files. Note that this template can be referenced in the 'New Java File' template with **${filecomment}**.

Types	The comment for new types. Note that this template can be referenced in the 'New Java File' template with **${typecomment}**.
Fields	The comment for new fields.
Constructors	The comment for new constructors.
Methods	The comment for new methods that do not override an method in a base class and that do not delegate to any other method.
Overriding methods	The comment for new methods that override an method in a base class. By default the comment is defined as a non-Javadoc comment (Javadoc will replace this comment with the comment of the overridden method). You can change this to a real Javadoc comment if you want
Delegating methods	The comment for new methods which delegate to existing methods. Such methods can be created with the **Source > Generate Delegate Methods...** action.
Getters	The comment for getter methods.
Setters	The comment for setter methods.

Code Templates

Template Name	Description
New Java files	Used by the New Type wizards when a new Java file is created. The template can specify where comments are added. Note that the template can contain the variable **${typecomment}** and **${filecomment}** that will be substituted by the evaluation of the **Types** respectively **Files** comment template.
Class body	Used by the New Class wizards when a new Java class is created.
Interface body	Used by the New Interface wizards when a new Java interface is created.

Enum body	Used by the New Enum wizards when a new Java enum is created.
Annotation body	Used by the New Annotation wizards when a new Java annotation is created.
Method body	The 'Method body' templates are used when new method with a body is created that still needs some code to complete its functionality. It contains the variable **${body_statement}** that resolves to a return statement or/and a super-call.
Constructor body	The 'Constructor body' templates are used when new method or constructor with body is created. It contains the variable **${body_statement}** that resolves a super call.
Getter body	The 'Getter body' templates are used when new getter method is created. It contains the variable **${body_statement}** that resolves to the appropriate return statement.
Setter body	The 'Setter body' templates are used when new setter method is created. It contains the variable **${body_statement}** that resolves to the appropriate assignment statement.
Catch block body	The 'Catch block body' template is used when a catch block body is created. It can use the variables **${exception_type}** and **${exception_var}**.

Code Template dialog

The following fields and buttons appear in the dialog:

Action	Description
Description	A description of the template
Pattern	The template pattern.
Insert Variables...	Displays a list of pre-defined template specific variables.

Related reference

Code Formatter Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Formatter** preference page lets you manage your code formatter profiles for the Java code formatter.

Action	Description
Active profile	Selects the profile active for the workspace or current project.
Edit...	Shows a dialog which displays the settings stored in the active profile. Only user-defined profiles can be modified.
Remove	Removes the selected profile. This action is only available on user-defined profiles.
New...	Shows the dialog to create a new profile. The dialog requires you to enter a name for the new profile. Additionally, you may select a built-in or user-defined existing profile to base your new profile on.
Import...	Imports profiles from the file system.
Export All...	Exports all the profiles to the file system.
Details	Displays a sample how the active profile formats a Java code snippet

To change your formatter options you can either select one of the built-in profiles or create a new profile. To create a new profile select **New...**. You can then edit the new profile by selecting **Edit...**.

If your project is shared, e.g. through CVS, and uses a formatter profile which is not managed by yourself (you have not created the profile) then the profile will be marked as **Unmanaged Profile**. You are not allowed to change such a profile, only the creator (manager) of the profile can change it.

Related reference

Java editor
Java editor preferences

Organize Imports Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Organize Imports** preference page defines how the Organize Imports command generates the import statements in a compilation unit.

Option	Description	Default
Import order list	This list of prefixes shows the sequential order for packages imported into a Java compilation unit. Each entry defines a block. Different blocks are separated by a spacer line.	java javax org com
New...	Adds a package name prefix to the import order list. In the resulting dialog, type a package name or package name prefix.	n/a
New Static...	Adds a package name prefix to the import order list. In the resulting dialog, type a package name or package name prefix.	n/a
Edit...	Edits the currently selected entry in the import order list. In the resulting dialog, type a package name or package name prefix.	n/a
Remove	Removes the currently selected entry from the import order list.	n/a
Up	Moves the selected package name prefix up in the import order list.	n/a
Down	Moves the selected package name prefix down in the import order list.	n/a

Import...	Loads a list of package name prefixes from a file.	n/a
Export...	Saves the list of package name prefixes to a file.	n/a
Number of imports needed for .*	The number of import statements that are allowed for types in the same package before *<package>.** is used.	99
Number of static imports needed for .*	The number of static import statements that are allowed for static members of the same type before *type* is used.	99
Do not create imports for types starting with a lower case letter	If enabled, types starting with a lowercase letter are not imported.	On

Related reference

Source actions

Java Compiler Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler** preference page lets you configure the various settings related to compiling of Java source code and class file generation.

An Eclipse-based product may change the compiler defaults, so they may be different than the ones indicated here.

Sub pages offer more compiler settings:

- Building
- Error/Warnings
- Javadoc
- Task Tags

The options are:

JDK Compliance

Option	Description	Default
Compiler compliance level	Specifies the compiler compliance level.	1.4
Use default compliance settings	If enabled, the default compliance settings for the compiler compliance level are applied.	On
Generated class files compatibility	Specifies the generated class file compatibility.	1.2
Source compatibility	Specifies the compatibility of the accepted source code.	1.3
Disallow identifiers called 'assert'	When enabled, the compiler will issue an error or a warning whenever 'assert' is used as an identifier (reserved keyword in J2SE 1.4).	Warning
Disallow identifiers called 'enum'	When enabled, the compiler will issue an error or a warning whenever 'enum' is used as an identifier (reserved keyword in J2SE 1.5).	Warning

Classfile generation

Add variable attributes to generated class files	If enabled, variable attributes are added to the class file. This will enable local variable names to be displayed in the debugger (in places where variables are definitely assigned) The resulting .class file is then bigger.	On
Add line number attributes to generated class files	If enabled, line number information is added to the class file. This will enable source code highlighting in the debugger.	On
Add source file name to generated class file	If enabled, the source file name is added to the class file. This will enable the debugger to present the corresponding source code.	On
Preserve unused local variables	If enabled, unused local variables (i.e. never read) are not stripped from the class file. If stripped this potentially alters debugging.	On
Inline finally blocks	If enabled, finally blocks are inlined in the generated class files. This positively affects performance, but may result in larger class files.	Off

Related concepts

Java builder

Related tasks

Working with JREs

Related reference

Classpath variables preferences
Java build path properties
Building preference page
Errors/Warnings preference page

Javadoc preference page
Task tags preference page

Java Building Preferences

Indicate your preferences for the Building settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Compiler > Building** preference page.

General

Option	Description	Default
Maximum number of reported problems per compilation unit	Specifies how many problems should be reported for a compilation unit.	100
Enable use of exclusion patterns in source folders	When disabled, no entry on a project classpath can be associated with an exclusion pattern.	On
Enable use of multiple output locations for source folders	When disabled, no entry on a project classpath can be associated with a specific output location, preventing thus usage of multiple output locations.	On

Build path problems

Abort build when build path errors occur	Allow to toggle the builder to abort if the classpath is invalid.	On
Incomplete build path	Indicate the severity of the problem reported when an entry on the classpath does not exist, is not legitimate or is not visible (e.g. a reference project is closed).	Error
Circular dependencies	Indicate the severity of the problem reported when a project is involved in a cycle.	Error
Incompatible required binaries	Indicated the severity of the problem reported when a project requires incompatible binaries.	Ignore

Output folder

Duplicated resources	Indicate the severity of the problem reported when more than one occurrence of a resource is to be copied into the output location.	Warning
Scrub output folders when cleaning projects	Indicate whether the Java Builder is allowed to clean the output folders when performing full build operations.	On
Rebuild class files modified by others	Indicate whether class files which have been modified by others should be rebuilt to undo the modification.	Off
Filtered resources	A comma separated list of file patterns which are not copied to the output folder.	Empty

Related concepts

Java builder
Inclusion and exclusion patterns
Build classpath

Java Compiler Errors/Warnings Preferences

Indicate your preferences for the Errors/Warnings settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Compiler > Errors/Warnings** preference page.

Code style

Option	Description	Default
Non-static access to a static member	When enabled, the compiler will issue an error or a warning whenever a static field or method is accessed with an expression receiver. A reference to a static member should be qualified with a type name.	Warning
Indirect access to a static member	When enabled, the compiler will issue an error or a warning whenever a static field or method is indirectly accessed. A static field of an interface should be qualified with the declaring type name.	Ignore
Unqualified access to instance field	When enabled, the compiler will issue an error or a warning whenever it encounters a field access which is not qualified (e.g. misses a 'this').	Ignore
Undocumented empty block	When enabled, the compiler will issue an error or a warning whenever it encounters an empty block statement with no explaining comment.	Ignore

Access to a non-accessible member of an enclosing type	When enabled, the compiler will issue an error or a warning whenever it emulates access to a non-accessible member of an enclosing type. Such accesses can have performance implications.	Ignore
Method with a constructor name	Naming a method with a constructor name is generally considered poor style programming. When this option is enabled, the compiler will signal such scenario either as an error or a warning.	Warning
Parameter assignment	Assigning a value to a parameter is generally considered poor style programming. When this option is enabled, the compiler will signal such scenario either as an error or a warning.	Ignore
Non-externalized strings	When enabled, the compiler will issue an error or a warning for non externalized String literal (i.e. non tagged with //$NON-NLS-<n>$) or for non externalized String tags which do not belong to a String.	Ignore
Method can be static	When enabled, the compiler will issue an error or a warning for methods which are private or final and which refer only to static members.	Ignore

Method can potentially be static	When enabled, the compiler will issue an error or a warning for methods which are not private or final and which refer only to static members. Note that methods can be overridden in a subclass, so if you make a "potentially static" method static, this may break existing clients.	Ignore

Potential programming problems

Serializable class without serialVersionUID	When enabled, the compiler will issue an error or a warning whenever a type implementing 'java.io.Serializable' does not contain a serialVersionUID field.	Warning
Assignment has no effect (e.g. 'x = x')	When enabled, the compiler will issue an error or a warning whenever an assignment has no effect (e.g. 'x = x').	Warning
Possible accidental boolean assignment (e.g. 'if (a = b)')	When enabled, the compiler will issue an error or a warning whenever it encounters a possible accidental boolean assignment (e.g. 'if (a = b)').	Ignore
'finally' does not complete normally	When enabled, the compiler will issue an error or a warning whenever a 'finally' statement does not complete normally (e.g. contains a return statement).	Warning

Empty statement	When enabled, the compiler will issue an error or a warning whenever it encounters an empty statement (e.g. a superfluous semicolon).	Ignore
Using a char array in string concatenation	When enabled, the compiler will issue an error or a warning whenever a char[] expression is used in String concatenations, `"hello" + new char[]{'w','o','r','l','d'}`	Warning
Hidden catch block	Locally to a try statement, some catch blocks may hide others , e.g. `try { throw new java.io.CharConversionException(); } catch (java.io.CharConversionException e) { } catch (java.io.IOException e) {}` When enabled, the compiler will issue an error or a warning for hidden catch blocks corresponding to checked exceptions.	Warning
Inexact type match for vararg arguments	When enabled, the compiler will issue an error or a warning whenever it encounters an inexact type match for vararg arguments.	Warning

Boxing and unboxing conversions	When enabled, the compiler will issue an error or a warning whenever it encounters a boxing or unboxing conversion. Autoboxing may affects performance negatively.	Ignore
Enum type constant not covered on 'switch'	When enabled, the compiler will issue an error or a warning whenever it encounters a switch statement which does not contain case statements for every enum constant of the referenced enum.	Ignore
'switch' case fall-through	When enabled, the compiler will issue an error or a warning when a case may be entered by falling through a preceding, non empty case.	Ignore
Null pointer access	When enabled, the compiler will issue an error or a warning when it encounters that a local variable which is certainly null is dereferenced. Note that the analysis can not find all null pointer accesses, see **Potential null pointer access**.	Warning
Potential null pointer access	When enabled, the compiler will issue an error or a warning when it encounters that a local variable which may be null is dereferenced. Note that the analysis is fairly conservative, it only considers cases where there is something suspicious.	Ignore

Comparing identical	When enabled, the compiler will issue an error or a warning if a comparison is involving identical operands (e.g 'x == x').	Warning
Missing synchronized modifier on inherited method	When enabled, the compiler will issue an error or a warning when it encounters an inherited method which is missing the synchronized modifier.	Ignore
Class overrides 'equals()' but not 'hashCode()'	When enabled, the compiler will issue an error or a warning when it encounters a class which overrides 'equals()' but not 'hashCode()'.	Ignore
Dead code	When enabled, the compiler will issue an error or a warning when it encounters dead code (e.g 'if (false)').	Warning

Unused object allocation	When enabled, the compiler will issue an error or a warning when it encounters an allocated object which is not used, e.g.
	`if (name == null)`
	`  new IllegalArgumentException();`
		Ignore

Name shadowing and conflicts

Field declaration hides another field or variable	When enabled, the compiler will issue an error or a warning if a field declaration hides another inherited field.	Ignore

Local variable declaration hides another field or variable	When enabled, the compiler will issue an error or a warning if a local variable declaration hides another field or variable.	Ignore
Include constructor or setter method parameters	When enabled, the compiler additionally will issue an error or a warning if a constructor or setter method parameter hides another field or variable.	Off
Type parameter hides another type	When enabled, the compiler will issue an error or a warning if i.e. a type parameter of an inner class hides an outer type.	Warning
Method does not override package visible method	A package default method is not visible in a different package, and thus cannot be overridden. When this option is enabled, the compiler will signal such scenario either as an error or a warning.	Warning
Interface method conflicts with protected 'Object' method	When enabled, the compiler will issue an error or a warning whenever an interface defines a method incompatible with a non-inherited Object method. Until this conflict is resolved, such an interface cannot be implemented, e.g. `interface I { int clone(); }`	Warning

Deprecated and restricted API

Deprecated API	When enabled, the compiler will signal use of deprecated API either as an error or a warning.	Warning
Signal use of deprecated API inside deprecated code	When enabled, the compiler will signal use of deprecated API inside deprecated code. The severity of the problem is controlled with option "Deprecated API".	Off
Signal overriding or implementing deprecated method	When enabled, the compiler will signal overriding or implementing a deprecated method The severity of the problem is controlled with option "Deprecated API".	Off
Forbidden reference (access rules)	When enabled, the compiler will signal a forbidden reference specified in the access rules.	Error
Discouraged reference (access rules)	When enabled, the compiler will signal a discouraged reference specified in the access rules.	Warning

Unnecessary code

Value of local variable is not used	When enabled, the compiler will issue an error or a warning whenever a local variable is declared but its value never used within its scope.	Warning
Value of parameter is not used	When enabled, the compiler will issue an error or a warning whenever a parameter is declared but its value never used within its scope.	Ignore

Ignore in overriding and implementing methods	When enabled, the compiler will not issue an error or a warning whenever a parameter is declared but never used within its scope in a method that overrides or implements another method.	On
Ignore parameters documented with '@param' tag	When enabled, the compiler will not issue an error or a warning whenever an unread parameter is documented with an '@param' tag.	On
Unused import	When enabled, the compiler will issue an error or a warning for unused import reference.	Warning
Unused private members	When enabled, the compiler will issue an error or a warning whenever a private member is declared but never used within the same unit.	Warning
Redundant null check	When enabled, the compiler will issue an error or a warning whenever a local variable which can not be null is tested for null.	Ignore
Unnecessary 'else' statement	When enabled, the compiler will issue an error or a warning whenever it encounters an unnecessary else statement (e.g. if (condition) return; else doSomething();).	Ignore

Unnecessary cast or 'instanceof' operation	When enabled, the compiler will issue an error or a warning whenever it encounters an unnecessary cast or 'instanceof' operation (e.g. if (object instanceof Object) return;).	Ignore
Unnecessary declaration of thrown exception	When enabled, the compiler will issue an error or a warning whenever it encounters an unnecessary declaration of a thrown exception.	Ignore
Ignore in overriding and implementing methods	When enabled, the compiler will not issue an error or a warning whenever it encounters an unnecessary declaration of a thrown exception in a method that overrides or implements another method.	On
Ignore exceptions documented with '@throws' or '@exception' tags	When enabled, the compiler will not issue an error or a warning whenever an unnecessary declaration of a thrown exception is documented with an '@throws' or '@exception' tag.	On
Ignore 'Exception' and 'Throwable'	When enabled, the compiler will not issue an error or a warning whenever it encounters an unnecessary declaration of 'Exception' and 'Throwable' exception	On

Unused 'break' or 'continue' label	When enabled, the compiler will issue an error or a warning whenever it encounters an unused 'break' or 'continue' label.	Warning
Redundant super interface	When enabled, the compiler will issue an error or a warning whenever it encounters a type which explicitly implements an interface that is already implemented by any of its supertypes.	Ignore

Generic types

Unchecked generic type operation	When enabled, the compiler will issue an error or a warning whenever it encounters an unchecked generic type operation.	Warning
Usage of a raw type	When enabled, the compiler will issue an error or a warning whenever it encounters a usage of a raw type (i.e. List instead of List<String>).	Warning
Generic type parameter declared with a final type bound	When enabled, the compiler will issue an error or a warning whenever it encounters a type bound involving a final type.	Warning

Ignore unavoidable generic type problems	When enabled, the compiler will issue an error or a warning even when it detects a generic type problem that could not have been avoided by the programmer. As an example, a type may be forced to use raw types in its method signatures and return types because the methods it overrides from a super type are declared to use raw types in the first place.	Off

Annotations

Missing '@Override' annotation	When enabled, the compiler will issue an error or a warning whenever it encounters a method overriding another implemented method, and the '@Override' annotation is missing.	Ignore
Include implementations of interface methods (1.6 or higher)	When enabled, the compiler will also issue an error or a warning whenever it encounters a method overriding or implementing a method declared in an interface, and the '@Override' annotation is missing. Note that '@Override' is only allowed on such methods if the compiler compliance level is 1.6 or higher, so this error or warning will never appear in 1.5 code.	On

Missing '@Deprecated' annotation	When enabled, the compiler will issue an error or a warning whenever it encounters a deprecated type without additional '@Deprecated' annotation.	Ignore
Annotation is used as super interface	When enabled, the compiler will issue an error or a warning whenever it encounters a type implementing an annotation. Although possible, this is considered bad practice.	Warning
Unhandled token in '@SuppressWarnings'	When enabled, the compiler will issue an error or a warning whenever it encounters an unhandled token in a '@SuppressWarnings' annotation.	Warning
Enable '@SuppressWarnings' annotations	When enabled, the compiler will process '@SuppressWarnings' annotations.	On
Unused '@SuppressWarnings' token	When enabled, the compiler will issue an error or a warning whenever it encounters an unused token in a '@SuppressWarnings' annotation.	Warning
Suppress optional errors with '@SuppressWarnings'	When enabled, the '@SuppressWarnings' annotation will also suppress the optional compiler errors.	Off

Enable **Include 'assert' in null analysis** to honor 'assert' statements when doing the null access analyis.
When **Treat errors like fatal compile errors** is enabled, all generated errors, fatal or configurable, lead to non-executable code. If disabled, then your code can be executed as long as it has no fatal error (i.e. syntax error).

Related concepts

Problems View

Quick Fix
Java builder

Java Compiler Javadoc Preferences

Indicate your preferences for the Javadoc settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Compiler > Javadoc** preference page.

Option	Description	Default
Process Javadoc comments (includes search and refactoring)	The builder detects problems in Javadoc comments if this option is enabled. For search and refactoring to work on Javadoc comments it is also required to enable this option.	On
Malformed Javadoc comments	When enabled, the compiler will issue an error or warning whenever it encounters a malformed Javadoc comment.	Ignore
Only consider members as visible as	The compiler only checks members with the specified or a broader visibility.	Public
Validate tag arguments (@param, @throws, @exception, @see, @link)	When enabled, the compiler will issue an error or warning whenever it encounters an error in a Javadoc comment tag argument for @param, @throws, @exception, @see or @link.	Off
Report non visible references	When enabled, the compiler will issue an error or warning whenever it encounters a reference to a non visible element.	Off
Report deprecated references	When enabled, the compiler will issue an error or warning whenever it encounters a reference to a deprecated element.	Off

Missing tag descriptions	Choose between 'Validate all standard tags', 'Validate @return tags' and 'Ignore'. When enabled, the compiler will issue an error or warning whenever it encounters a missing tag description	'Validate @return tags'
Missing Javadoc tags	When enabled, the compiler will issue an error or warning whenever it encounters a missing Javadoc tag (i.e. missing @param tag for a parameter).	Ignore
Only consider members as visible as	The compiler only checks members with the specified or a broader visibility.	Public
Ignore in overriding and implementing methods	When enabled, methods that override or implement another method are not checked for missing Javadoc tags.	On
Ignore method type parameters	When enabled, missing Javadoc tags are not reported for method type parameters.	On
Missing Javadoc comments	When enabled, the compiler will issue an error or warning whenever it encounters a missing Javadoc comment for a Java member.	Ignore
Only consider members as visible as	The compiler only checks members with the specified or a broader visibility.	Public
Ignore in overriding and implementing methods	When enabled, methods that override or implement another method are not checked for missing Javadoc comments.	On

Java Task Tags Preferences

Task tags can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Task Tags** preference page. When the tag list is not empty, the compiler will issue a task marker whenever it encounters one of the corresponding tag inside any comment in Java source code. Generated task messages will include the tag, range until the next line separator, comment ending or non-empty tag, and will be trimmed. If the same line of code carries multiple tags, they will be reported separately.

See the Compiler preference page for information on how to enable task tags in your source code.

Action	Description
New...	Adds a new task tag. In the resulting dialog, specify a name and priority for the new task tag.
Remove	Removes the selected task tag.
Edit...	Allows you to edit the selected task tag. In the resulting dialog, edit the name and/or priority for the task tag.
Default	Sets the currently selected task tag as the default task tag. The default task tag is the one that is used in the code templates as specified on the Code Templates preference page. The default task tag is displayed in bold font.

Case sensitivity of the task tags can be specified at the bottom of the preference page using the option **Case sensitive task tag names**.

Related reference

Java compiler preferences
Code template preferences

Java Debug Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug** preference page.

These options apply to java specific debuggers and have to do with suspending options and hot code replace.

Option	Description	Default
Suspend execution on uncaught exceptions	This option controls if a program will suspend if an uncaught exception is thrown. For example if you try to call a method on a null object and a NullPointerException is thrown, with this option on your program will suspend at the location the exception was thrown	On
Suspend execution on compilation errors	This option controls if a program will suspend when a compilation error is encountered	On
Suspend on breakpoints during evaluations	This options controls if breakpoints will suspend during an evaluation of code containing a breakpoint. For more information about evaluations go here	On
Open popup when suspended on exception	This option controls if a popup window will be displayed when execution of a program stops on an exception. The exception the program suspended on is made available for inspection in the popup	Off

Default suspend policy for new breakpoints	This option allows the default suspend policy to be set for new breakpoints. The suspend policy is used by breakpoints to tell the VM what to suspend, in this case either the thread the breakpoint is active in or the entire running VM	Suspend Thread
Show error when hot code replace fails	This option controls if the user will be presented with an error dialog when a hot code replace fails	On
Show error when hot code replace is not supported	This option controls if the user will be presented with an error dialog when hot code replace is not supported, and the user has made and saved changes to currently running code	On
Show error when obsolete methods remain after hot code replace	This option controls if the user will be presented with an error dialog when a hot code replace completed, but there were obsolete methods left over	On
Replace class files containing compilation errors	This option controls is class files containing compilation errors will be replaced	On

Debugger timeout	This option describes the length of time (in milliseconds) the debugger will wait trying to communicate with a running VM before giving up and disconnecting	3000
Launch timeout	This option describes the length of time (in milliseconds) that a launch will wait to complete the launching process before giving up and terminating. This option has no bearing on the length of time a program will run, only how long the process of *trying* to run a program will wait	20000
Warn when unable to install breakpoint due to missing line number attributes	This option controls if the user will be notified if they try to set a line breakpoint on a line that has invalid line information	On

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Detail Formatters Preferences
Heap Walking Preferences
Installed JREs Preferences
Logical Structures Preferences
Primitive Display Preferences
Run/Debug Preferences
Step Filtering Preferences

Detail Formatters Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Detail Formatters** preference page.

These preferences are used to create and manage detail formatters, which are used in the details panes of variables views, or for variable labels themselves.

Option	Description	Default
Types with detail formatters	The listing of types that currently have detail formatters (the list itself), and the enabled state of the detail formatter, i.e. if it is on use (the checked state of a list item)	
Add button	Used to add a new detail formatter to the current listing	
Edit button	Used to edit the selected detail formatter	
Remove button	Used to remove the selected detail formatter	
Detail formatter code snippet for selected type	Displays the current code snippet in use for the detail formatter of the selected type. This field is not editable, the only way to edit a detail formatter is to use the Edit... button	

Show variables details	This option controls where you want to see the detail formatter appear, you have one of three choices: As a label for variables with detail formatters - will appear as the primary display label only for associated variables the have detail formatters As a label for all variables - will appear for all associated variables, regardless of them having detail formatters already In detail pane only - will only appear in variables view details panes and nowhere else	

- Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

- Related tasks

Launching a Java program
Running and debugging

- Related reference

Heap Walking Preferences
Installed JREs Preferences
Java Debug Preferences
Logical Structures Preferences
Primitive Display Preferences
Run/Debug Preferences
Step Filtering Preferences

Heap Walking Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Heap Walking** preference page.

These preferences allow count settings for Java 1.6 heap walking features to be changed.

Option	Description	Default
Maximum instances to display	The maximum number of type instances to display using the all instances command. Entering zero will always show all of the instances of the selected type in the current VM	100
Maximum references to display	The maximum number of type references to display using the all references command. Entering zero will always show all of the references to the selected type in the current VM	100

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Detail Formatters Preferences
Installed JREs Preferences
Java Debug Preferences
Logical Structures Preferences
Primitive Display Preferences
Run/Debug Preferences
Step Filtering Preferences

Logical Structures Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Logical Structures** preference page.

These preferences are used to add, edit and remove logical structures, which are used to display collection-type objects

Option	Description	Default
Logical Structures	Shows a listing of all of the current logical structures available. Note that logical structures with a yellow background cannot be edited or removed	
Add button	Used to add a new logical structure to the listing	
Edit button	Used to edit the selected logical structure	
Remove button	Used to remove the selected logical structure	
Preview	Shows the code snippet to the selected logical structure. This preview is not editable, you must use the **Edit...** button to edit a logical structure	

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Detail Formatters Preferences
Heap Walking Preferences
Installed JREs Preferences

Primitive Display Options Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Primitive Display Options** preference page.

These preferences affect how primitive values are displayed (int, double, etc).

Option	Description	Default
Display hexadecimal values (byte, short, char, int, long)	This option controls whether hexadecimal values will be displayed for the primitive types byte, short, char, int and long	Off
Display ASCII char value (byte, short, int, long)	This option controls whether ASCII char values will be displayed for the primitive types byte, short, int and long	Off
Display unsigned values (byte)	This option controls whether unsigned values will be displayed for bytes	Off

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Detail Formatters Preferences
Heap Walking Preferences
Installed JREs Preferences
Java Debug Preferences
Logical Structures Preferences
Run/Debug Preferences
Step Filtering Preferences

Step Filtering Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Step Filtering** preference page.

These preferences are used to configure step filters and step filtering options.

Option	Description	Default
Use step filters	This option controls is step filtering should be enabled or not. See also the step filtering command in the **Debug View**	On
Defined step filters	This is a listing of all of the defined step filters (the listing itself) and whether the step filters are in use or not (the checked state of step filter)	
Add Filter button	Used to add a new step filter that is based on a regular expression	
Add Class button	Used to add a new Java class to the listing of step filters	
Add Packages button	Used to add new Java packages to the listing of step filters	
Remove button	Used to remove the selected step filter(s)	
Select All button	Used to set all defined step filters as enabled (or checked)	
Deselect All button	Used to set all defined step filters as disabled (or unchecked)	

Filter synthetic methods	This option controls if synthetic method should always be filtered or not while stepping. This option requires that the VM used support synthetic methods	Off
Filter static initializers	This option controls if static initializers should always be filtered or not while stepping	Off
Filter simple getters	This option controls if simple Java bean-style getters should always be filtered or not while stepping	Off
Filter simple setters	This option controls if simple Java bean-style setters should always be filtered or not while stepping	Off
Filter constructors	This option controls if constructors should always be filtered or not while stepping	Off

Step through filters	This option controls step filters to always return from a filtered location or step through to a non-filtered location. For example, if `java.util` is a filtered location, stepping into code in `HashMap` could result in a call-back to your application code to check the equality of an object. If you choose to **Step through filters** a step into would end up in your application code. However, when the **Step through filters** option is disabled, a step into `HashMap` would behave like a step over.	On

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Detail Formatters Preferences
Heap Walking Preferences
Installed JREs Preferences
Java Debug Preferences
Logical Structures Preferences
Primitive Display Preferences
Run/Debug Preferences

Java Editor Preferences

The following Java editor preferences can be set on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor** preference page and its sub-pages.

- Content Assist
  - Advanced
  - Favorites
- Folding
- Hovers
- Mark Occurrences
- Save Actions
- Syntax Coloring
- Templates
- Typing

Note that some options that are generally applicable to text editors can be configured on the text editor preference page.

Option	Description	Default
Smart caret positioning in Java names (overrides platform behavior)	If enabled, there are additional word boundaries inside |Camel|Case| Java names.	On
Report problems as you type	If enabled, the editor marks errors and warnings as you type, even if you do not save the editor contents. The problems are updated after a short delay.	On
Highlight matching brackets	If enabled, whenever the cursor is next to a parenthesis, bracket or curly braces, its opening or closing counter part is highlighted. The color of the bracket highlight is specified with **Appearance color options**.	On

Light bulb for quick assists	If enabled, a  shows up in the vertical ruler whenever a **quick assist** is available. See the quick assist section for a list of the available assists.	Off
Only show the selected Java element	If enabled, the Java editor will only show the selected Java element which is currently selected (i.e. in the Outline View or the Package Explorer).	Off

Appearance color options	The colors of various Java editor appearance features are specified here. **Matching brackets highlight**The color of brackets highlight.**Parameter hints background**The background color of the parameter hint window**Parameter hints foreground**The foreground color of the parameter hint window**Completion overwrite background**The background color of the completion overwrite window**Completion overwrite foreground**The foreground color of the completion overwrite window**Source hover background**The background color for the source hover. The source hover shows the source code of the element under the mouse pointer.	default colors

Related concepts

Java editor

Related reference

Java editor
Code formatter preferences
Java outline
Java content assist
Quick Fix
Quick Assist

Java Content Assist Preferences

Indicate your preferences for the Content Assist settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Editor > Content Assist** preference page.

Option	Description	Default
Insertion		
Completion inserts/Completion overwrites	If **Completion inserts** is on, the completion text is inserted at the caret position, so it never overwrites any existing text. If **Completion overwrites** is on, the completion text replaces the characters following the caret position until the end of the word. Note that pressing **Ctrl** when applying a completion proposal toggles between the two insertion modes.	Completion inserts
Insert single proposals automatically	If enabled, code assist will choose and insert automatically single proposals.	On
Insert common prefixes automatically	If enabled, code assist will automatically insert the common prefix of all possible completions similar to Unix shell expansion. This can be used repeatedly, even while the code assist window is being displayed.	Off

Add import instead of qualified name	If enabled, type proposals which are in other packages will invoke the addition of the corresponding import declaration. Otherwise, the type will be inserted fully qualified.	On
Use static imports	If enabled, method proposals for static methods will invoke the addition of the corresponding static import declaration. Otherwise, a non static import for the declaring type will be added. Add the method or type containing the method to the Favorites preference page to get a proposal for such methods.	On
Fill method arguments and show guessed arguments	If enabled, code assist will add argument when completing a method. It will also offer matching variables, fields or method invocations from the context where a method proposal is inserted.	On
Insert parameter names	If selected, code assist will fill the arguments with the parameter names used in the method declaration.	Off
Insert best guessed arguments	If selected, code assist will fill the arguments with the best matching variable, field or method invocation from the context where a method proposal is inserted.	Off

Sorting and Filtering

Configure Type Filters.		
Sort proposals	Select how the proposals should be sorted in the content assist pop up.	by relevance
Show camel case matches	If enabled, camel case matches are displayed (i.e. NPE is expanded to NullPointerException).	On
Hide proposals not visible in the invocation context	If enabled, the Java element proposals are limited by the rules of visibility. For example, private field proposals of other classes would not be displayed.	On
Hide deprecated references	If enabled, references to deprecated Java elements are not displayed.	Off
Auto Activation		
Enable auto activation	If enabled, code assist can be invoked automatically. The condition for automatic invocation is specified with the preferences **Auto activation delay**, **Auto activation triggers for Java** and **Auto activation triggers for Javadoc**.	On
Auto activation delay	The time in milliseconds after which code assist is triggered after an auto activation trigger character is encountered and no other character has been typed.	200

Auto activation triggers for Java	If one of the trigger characters is typed inside Java source code (but not inside a Javadoc comment) and no other character is typed before the auto activation delay times out, the code assist is invoked.	'.'
Auto activation triggers for Javadoc	If one of the trigger characters is typed inside a Java doc and no other character is typed before the auto activation delay times out, the code assist is invoked.	'@#'

Related reference

Content/Code Assist
Type Filters

Advanced Java Content Assist Preferences

Indicate your preferences for the Advanced settings on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist > Advanced** preference page.

Option	Description	Default
Select the proposal kinds contained in the 'default' content assist list	Select all the proposal kinds which are contained in the content assist list when invoking content assist the first time with Ctrl+Space.	Other Java Proposals, SWT Template Proposals, Template Proposals, Type Proposals
Content assist cycling	Select the proposal kinds to cycle through and the cycle order when repeatedly invoking content assist (Ctrl+Space). Select a proposal kind and invoke Up or Down to change the order.	Template Proposals, SWT Template Proposals
Timeout for fetching a parameter name from attached Javadoc	If a parameter name could not be fetched from attached Javadoc within the specified amount of time then the attempt to fetch it is abandoned. Increase the value if you have problems fetching the name, for instance due to a slow network connection.	50 ms

Related reference

Content/Code Assist
Content Assist preference page

Java Content Assist Favorites Preferences

Indicate your preferences settings for the content assist for static members on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist > Favorites** preference page.

You can define a list of static members and types containing static members on this page. Content assist will propose those members even if the import is missing.

To see the static import proposals, make sure **Use static imports** is enabled on the [Image: /topic/org.eclipse.help/command_link.png] Content Assist preference page

Action	Description
New Type	Add a new type to the list. All static members declared by the added type will be proposed by content assist.
New Member	Add a static member to the list. The member will be proposed by content assist.
Edit	Edit the selected entry.
Remove	Removes the selected entry from the list.

Related reference

Content/Code assist
Content assist preference page

Java Editor Folding Preferences

Indicate your preferences for the Folding settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Editor > Folding** preference page.

Option	Description	Default
Enable folding	If enabled, different sections in the Java editor can be folded and expanded.	On
Comments	If enabled, comments are folded when opening a new editor.	Off
Header Comments	If enabled, header Comments are folded when opening a new editor.	On
Inner types	If enabled, inner types are folded when opening a new editor.	Off
Members	If enabled, Java members are folded when opening a new editor.	Off
Imports	If enabled, imports are folded when opening a new editor.	On

Related reference

Java Editor preference page

Java Hovers Preferences

Indicate your preferences for the Hovers settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Editor > Hovers** preference page.

Option	Description	Default
Expand vertical ruler icons upon hovering	If enabled, when there are multiple icons per line shown on vertical ruler, then the icons are expanded when hovering over it. This allows to see all icons. Changing this preference does not affect already opened editors.	Off
Text Hover key modifier preferences	The selected hovers in this list are used to generate the hover content. A key modifier can be given to each hover type to force the editor to only evaluate this hover when pressing the assigned key modifier.	Combined Hover, Source (with Shift)

Related reference

Java editor preference page

Mark Occurrences Preferences

Indicate your preferences for the Mark Occurrences settings on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Mark Occurrences** preference page.

Option	Description	Default
Mark occurrences of the selected element in the current file	If enabled, then you'll see within a file, where a variable, method, type or other element is referenced.	On

A list of elements can be enabled and disabled on this preference page. All are enabled by default.

Related reference

Java Editor preference page

Java Save Actions Preferences

Indicate your preferences for the Save Actions settings on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Save Actions** preference page.

A set of actions which are executed on each save of the Java editor can be specified on this page.

Option	Description	Default
Perform the selected actions on save	If enabled, then the selected actions below will be executed on save.	Off
Format source code	If enabled, then the source code will be formatted on each save. The code will be formatted according to the active profile selected on the **Java > Code Style > Formatter** preference page.	Off
Format all lines	If selected, all lines in the file will be formatted on save.	On
Format edited lines	If selected, only the lines that have been modified since the last save are formatted on save.	Off
Organize imports	If enabled, then the imports will be organized on each save. Change the organize import settings on the **Java > Code Style > Organize Imports** preference page.	On
Additional actions	If enabled, then additional actions will be executed on save. To add and remove actions select **Configure...**.	Off

Related reference

Java editor preference page
Clean Up preference page

Java Syntax Coloring Preferences

Indicate your preferences for the Syntax Coloring settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Editor > Syntax Coloring** preference page.

Syntax Coloring specifies how Java source code is rendered. Note that general text editor settings such as the background color can be configured on the general 'Text Editors' preference pages. Fonts may be configured on the general 'Colors and Fonts' preference page.

Option	Description	Default
Element	Each category (**Java**, **Javadoc** and **Comments**) contains a list of language elements that may be rendered with its own color and style. Note that some semantic highlighting options can be disabled by the user in order to ensure editor performance on low-end systems.	default colors and styles
Preview	Displays the preview of a Java source code respecting the current colors and styles.	n/a

Related concepts

Java editor

Related reference

Java editor preference page

Java Editor Templates Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Templates** preference page allows to create new and edit existing templates. A template is a convenience for the programmer to quickly insert often reoccurring source code patterns.

The following buttons allow manipulation and configuration of templates:

Action	Description
New...	Opens the Template dialog to create a new template.
Edit...	Opens the Template dialog to edit the currently selected template.
Remove	Removes all selected templates.
Restore Removed	Restores any preconfigured templates that have been removed.
Revert to Default	Restores any preconfigured templates to their default. This does not modify user-created templates.
Import...	Imports templates from the file system.
Export...	Exports all selected templates to the file system.
Use code formatter	If enabled, the template is formatted according to the code formatting rules specified in the Code Formatter preferences, prior to insertion. Otherwise, the template is inserted as is, but correctly indented.

Related concepts

Templates
Template variables

Related reference

Template editing
Java content assist
Task tag preferences
Code templates preferences
Code style preferences

Java Editor Typing Preferences

Indicate your preferences for the smart insert settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Editor > Typing** preference page. Those preferences are only considered if **Smart Insert Mode** appears in the status line. This can be toggled in the **Edit** menu.

Option	Description	Default
Automatically close	Select for which characters a closing character should be inserted when typing the opening character.	All on
Automatically insert at correct position	Select which characters should be automatically inserted when they are required.	All off
Enter key adjusts the indentation on the new line	If enabled, smart indentation will be used to indent the new line after pressing the 'Enter' key.	On
Tab key adjusts the indentation of the current line	If enabled then the 'Tab' key can be used to indent the current line.	On
Adjust indentation	When pasting Java code from the clipboard adjust its indentation to the current indentation level.	On
Update imports	When pasting Java code from the clipboard add the required import statements to the import section.	On
Wrap automatically	If enabled, string literals are wrapped when they exceed the max line length.	On
Escape text when pasting into a string literal	If enabled, special characters in pasted strings are escaped when they are pasted into an existing string literal.	Off

Related reference

Java editor preference page

Installed JREs Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs** preference page.

This page allows a default JRE to be selected as well as adding new JRE installations.

Option	Description
Installed JREs	The current listing of installed JREs, allowing you to select the one to act as the workspace default
Add...	Adds a new JRE definition to the workbench.
Edit...	Allows you to edit the selected JRE.
Duplicate...	Creates a new JRE with the same attributes as the selected JRE.
Remove	Removes the selected JRE from the workbench.
Search...	Automatically searches for JREs installed in the local file system and creates corresponding JRE definitions in the workspace.

Related concepts

Classpath Variables
Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Working with JREs
Adding a new JRE definition
Deleting a JRE definition
Assigning the default JRE for the workbench
Choosing a JRE for launching a project
Launching a Java program
Running and debugging

Related reference

Execution Environment Preferences
Java Debug Preferences
Run/Debug Preferences
Source Attachment

Execution Environments Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Java > Execution Environments** preference page.

These preferences describe all of the available execution environments, allowing default JRE installations to be specified for any given environment.

Option	Description	Default
Execution environments	The listing of all available execution environments	
Compatible JREs	The listing of compatible JRE installations that are compatible with the selected execution environment. You can select one (or none) of the installations to use as the default for the selected execution environment Any JRE installations that perfectly match the execution environment are marked with a label '(perfect match)'	
Environment description	Provides a description of the selected execution environment	

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Installed JREs Preferences
Java Debug Preferences

JUnit Preferences

The [Image: /topic/org.eclipse.help/command_link.png]**Java > JUnit** preference page lets you configure the stack trace filter patterns for JUnit. The patterns are used in the *JUnit* view to control which entries are visible in stack traces.

Action	Description
Enable assertions for new JUnit launch configurations	If enabled, then JUnit test will be launched with assertions enabled.
Show newly launched test in all JUnit views	By default, a new JUnit test run only shows up in the window where the test was started. Enable this option, if you prefer to see the latest test in all open JUnit views.
Add Filter	Allows to add a custom filter pattern. This action inserts an empty pattern which can be edited.
Add Class...	Allows to add classes to be filtered. This action opens the **Open Type** dialog to choose a type to be filtered in the stack traces.
Add Packages...	Allows to add packages to be filtered. This action opens a package selection dialog to choose the packages to be filtered in the stack traces.
Remove	Removes the currently selected stack trace filter pattern.
Enable All	Enables all stack trace filter patterns.
Disable All	Disables all stack trace filter patterns.

Related tasks

Using JUnit

Related reference

Open type

Properties Files Editor Preferences

Indicate your preferences for the Properties Files Editor settings on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Properties Files Editor** preference page.

Syntax coloring specifies how the content of '.properties' files rendered. Note that general text editor settings such as the background color can be configured on the general 'Text Editors' preference pages.

Option	Description	Default
Element	A list of elements that may be rendered with its own foreground color and style.	default colors and styles
Preview	Displays the preview of a properties file respecting the current colors and styles.	n/a

Debug Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug** preference page. The preferences here are generic to all debuggers, and govern stylistic and prompting options

Option	Description	Default
Reuse editor when displaying source code	The debugger displays source code in an editor when stepping through an application. When this option is on, the debugger will reuse the editor that it opened to display source from different source files. This prevents the debugger from opening an excessive number of editors. When this option is off, the debugger will open a new editor for each source file that needs to be displayed.	On
Activate the workbench when a breakpoint is hit	This option brings attention to the debugger when a breakpoint is encountered, by activating the associated window. The visual result varies from platform to platform. For example, on Windows, the associated window's title bar will flash.	On

Activate the debug view when a breakpoint is hit	This option brings attention to the debug view when a breakpoint is encountered. If the view is already open it will be activated. If the view is not already open it will be opened automatically.	On
Skip breakpoints during a 'Run to Line' operation	This option controls whether breakpoints are ignored when performing a 'Run to Line' operation. When the option is on, the debugger does not suspend at breakpoints encountered when a 'Run to Line' operation is invoked. When the option is off, breakpoints behave normally.	Off
Prompt for conformation when deleting all breakpoints	This option controls whether you will be prompted for confirmation when you try to delete all of your breakpoints	On
Prompt for confirmation when deleting breakpoint containers	This option controls if you will be prompted for confirmation when you try to delete a breakpoint container, e.g. a breakpoint working set	On
Changed value color	This option allows you to change the color of a changed value in the variables view, expressions view, memory view, anywhere running program variables are rendered	Red

Changed value background color	This option allows you to change the selection color of a changed variable, e.g. in the variables view showing columns	Yellow
Memory unbuffered color	This option allows you to change the rendering color of unbuffered memory blocks in the memory view	Grey
Memory buffered color	This option allows you to change the rendering color of buffered memory blocks in the memory view	Black

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Console Preferences
Installed JREs Preferences
Java Debug Preferences
Launching Preferences
Perspectives Preferences
String Substitution Preferences
View Management Preferences

Console Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Console** preference page.

The console displays output from running applications, and allows keyboard input to be read by running applications.

Option	Description	Default
Fixed width console	This preference controls whether the console has a fixed character width. When on, a maximum character width must also be specified. Some applications write long lines to the console which require horizontal scrolling to read. This can be avoided by setting the console to use a fixed width, automatically wrapping console output.	Off
Limit console output	This preference limits the number of characters buffered in the console. When on, a maximum buffer size must also be specified. When console output surpasses the specified maximum, output is truncated from the beginning of the buffer.	On
Displayed Tab Width	Allows the default width, in characters, of a tab to be specified	8

Show When Program Writes to Standard Out	This preference will force the console to show when something is written to the system out stream. A forced-show can mean that a console will be opened, or that the console will be brought to the top if it is already open.	On
Show When Program Writes to Standard Error	This preference will force the console to show when something is written to the system err stream. A forced-show can mean that a console will be opened, or that the console will be brought to the top if it is already open.	On
Standard Out Text Color	This preference controls the color of text written to the standard output stream by an application.	Black
Standard Error Text Color	This preference controls the color of text written to the standard error stream by an application.	Red
Standard In Text Color	This preference controls the color of text typed into the console to be read by an application.	Green
Background Color	This preference controls the color of the background of the console	White

You can also click the **Change** button to set the font for the Console.

- Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

- Related tasks

Launching a Java program
Running and debugging

- Related reference

Console View
Installed JREs Preferences
Java Debug Preferences
Launching Preferences
Perspectives Preferences
Run/Debug Preferences
String Substitution Preferences
View Management Preferences

Launching Preferences

These preferences allow you to configure options related to launching programs in general, and govern such things as waiting for builds, saving files before launching, etc.

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Launching** preference page.

Option	Description	Default
Save required dirty editors before launching	This option controls whether the user will be prompted to save any dirty editors before an application is launched. The allowable settings are: Always - when this option is selected, the user is never prompted to save dirty editors, and editors are automatically saved. Never - when this option is selected, the user is never prompted to save dirty editors, and editors are not automatically saved. Prompt - when this option is selected, the user is prompted to save dirty editors before launching an application.	Prompt

Wait for ongoing build to complete before launching	This option controls whether a launch will wait for an already executing build to complete before launching an application. The allowable settings are: Always - when this option is selected, a launch will always wait for the pending build to completeNever - when this option is selected, a launch will never wait for the pending build to completePrompt - when this option is selected, the user is prompted to wait for the pending build to complete when they launch	Always

Launch in debug mode whenever the workspace contains breakpoints	This option controls whether a launch will be performed in debug mode, even when run is pressed, when the workspace contains breakpoints. The allowable options are: Always - when this option is selected, a launch will always be performed in debug mode when breakpoints are present, no matter which launch button is pressed (run or debug)Never - when this option is selected, a launch will always be performed in the user specified modePrompt - when this option is selected, the user is prompted to launch in debug mode when the workspace contains breakpoints	Never
Continue launch if project contains errors	This option controls whether a launch should be performed when a related project contains an error. The allowable options are: Always - when this option is selected, a launch will continue in the face of compilation errorsPrompt - when this option is selected, the user is prompted to proceed with a launch when relevant compilation errors are in an associated project	Prompt

Build (if required) before launching	If the workspace requires building, an incremental build will be performed prior to launching an application.	On
Remove terminated launches when a new launch is created	When an application is launched, all terminated applications in the Debug view are automatically cleared.	On
Size of recently launched applications list	This option controls how many launches will appear in the Run/Debug pull-down launch history menus.	10
Launch Operation	New in 3.3 is the ability to launch the selected resource or editor. The launch operation preference allows users to switch between using the old 'always launch last' behavior and the new 'launch selected resource or editor' behavior. If the preference to use launch selected resource or editor is selected, you can then customize what happens if the selected resource or editor is not launchable. There are two choices for this try to launch the parent project of the resource or editor launch the previous thing you launched.	Launch Selected with Launch Project

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

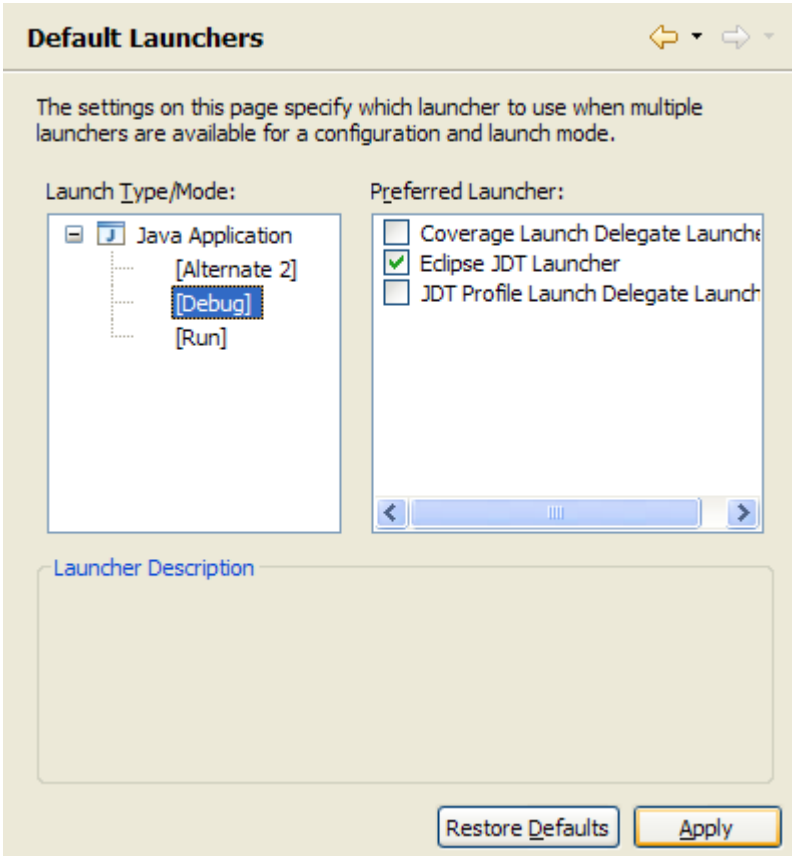
Run and Debug toolbar actions
Console Preferences
Default Launchers Preferences
Installed JREs Preferences
Java Debug Preferences
Launch Configurations Preferences
Perspectives Preferences
Run/Debug Preferences
String Substitution Preferences
View Management Preferences

Default Launchers Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Launching > Default Launchers** preference page.

The **Default Launchers** preference allows you to select what launch tooling is to be used if more than one kind of tooling exists for the same thing, for example, consider using two profilers at the same time for Java programs. For the most part, this page remains disabled, as concurrent overlapping tooling is very rare, but in the event there is overlap, the page will enable and present the user with the launch types that conflict and what modes they conflict on.

The following image shows the page with conflicting tooling for Java types in the *run*, *debug* and *Alternate2* modes. In this example a preferred launcher can be selected independently for each of the conflicting modes.



Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Installed JREs Preferences

Java Debug Preferences
Launch Configurations Preferences
Launching Preferences
Run/Debug Preferences

Launch Configurations Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Launching > Launch Configurations** preference page.

This page allows you set filtering options that are used throughout the workbench to limit the exposure of certain kinds of launch configurations. These filtering setting affect the launch dialog, launch histories and the workbench.

Option	Description	Default
Filter configurations in closed projects	Filter out configurations that are associated with a project that is currently closed	On
Filters configuration in deleted or missing projects	Filter out configurations that are associated with a project that has been deleted or are simply no longer available (not in the workspace, etc)	On
Apply windows working set	Applies the filtering from any working sets currently active to the visibility of configurations associated to resources in the active working sets. i.e. if project P has two configurations associated with it, but is not in the currently active working set, the configurations do not appear in the UI, much like P does not.	On

Filter check configuration types	Filter all configurations of the selected type regardless of the other filtering options. For example if you specify to filter Java type configurations, all Java type configurations will be filtered from the UI, not just ones that fall under the other filtering categories.	Off
Delete configurations when associated project is deleted	Any launch configurations associated with a project being deleted will also be deleted if this option is enabled. Once deleted the configurations are not recoverable.	On

Migration	As Eclipse matures and new features are added to the launching framework, there sometimes exists the need to make changes to launch configurations. Some of these changes are made automatically, but those that are not (nonreversible ones) are left up to the end-user. The migration section allows users to self-migrate any launch configurations that require it. Upon pressing the **Migrate...** button, if there are any configurations requiring migration, they are presented to the user, and the user can select the ones they want to migrate. Configuration migration is not un-doable.	

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Default Launchers Preferences
Installed JREs Preferences
Java Debug Preferences
Launching Preferences
Run/Debug Preferences

Perspectives Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Perspectives** preference page.

The preferences on this page allow you to set which, if any perspectives are automatically opened when launchers are used in certain modes

Option	Description	Default
Open the associated perspective when launching	This option controls if the specified perspective will be switched to automatically or not when launching Always - the specified perspective will always be opened Never - the specified perspective will never be opened Prompt - the user will be prompted to switch to the specified perspective	Never
Open the associated perspective when an application suspends	This option controls if the specified perspective will be switched to when the application suspends Always - the specified perspective will always be opened Never - the specified perspective will never be opened Prompt - the user will be prompted to switch to the specified perspective	Prompt

Launcher / Perspective choices	This option area allows you to customize what perspective will be opened for a given launch and mode that it supports. For example, you can say that you want to open the Java perspective when using the Java JDT launcher in the run mode only.	

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

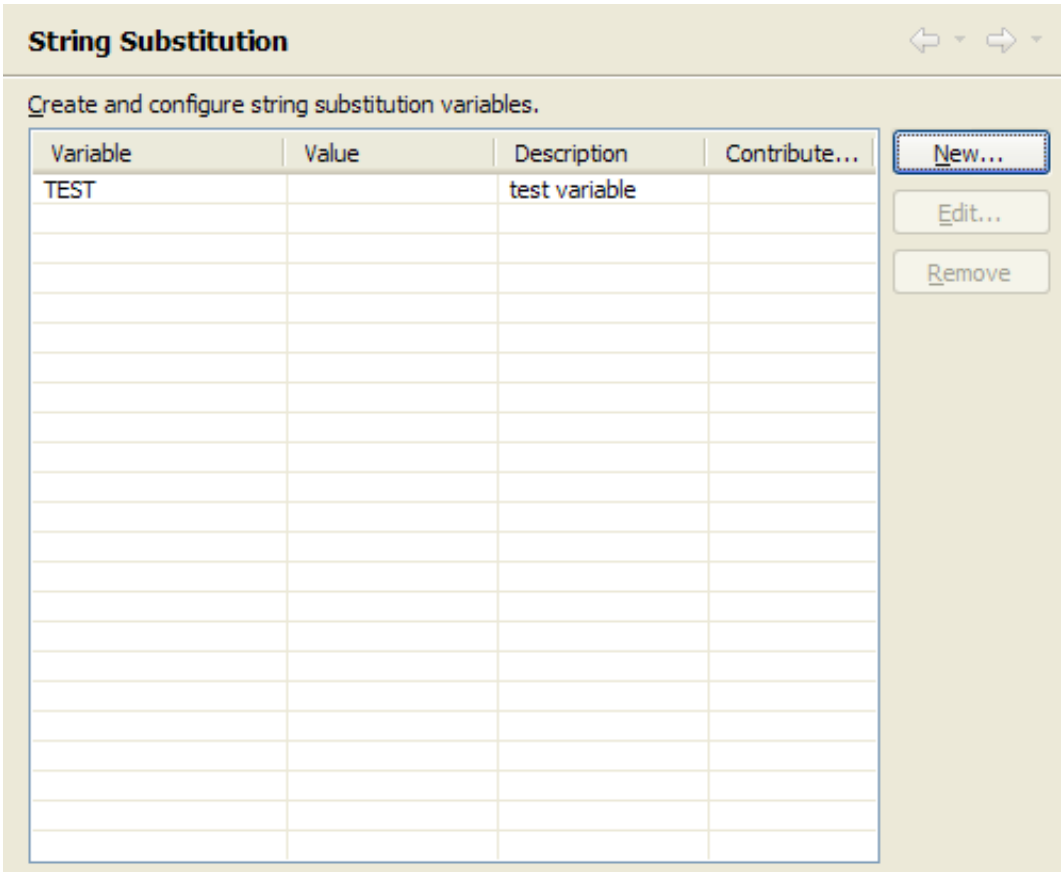
Console Preferences
Installed JREs Preferences
Java Debug Preferences
Launching Preferences
Run/Debug Preferences
String Substitution Preferences
View Management Preferences

String Substitution Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > String Substitution** preference page.

This preference page is used solely for the purpose of creating string variables to be reused in other locations that accept string variable substitution.

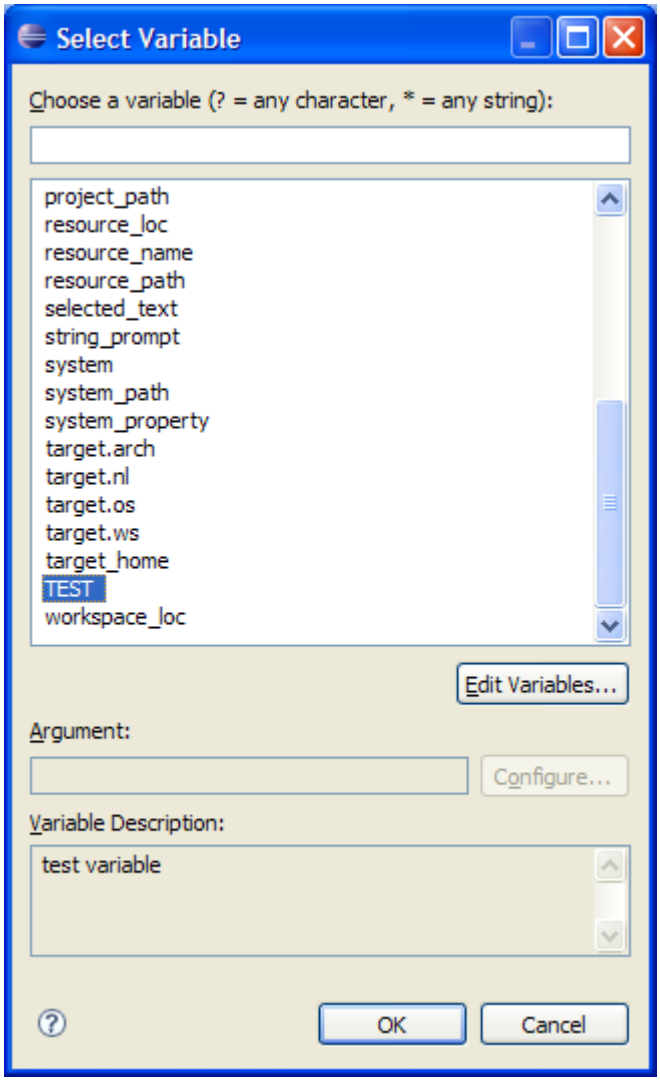
The platform SDK for Eclipse uses string variables in the launch dialog for setting program and VM arguments for certain types of launch configurations, as well as allowing substitutions for Ant targets in the External Tools launch dialog. Consider the following screen shot which shows the **String Substitution** preference page with a new *TEST* string variable created.



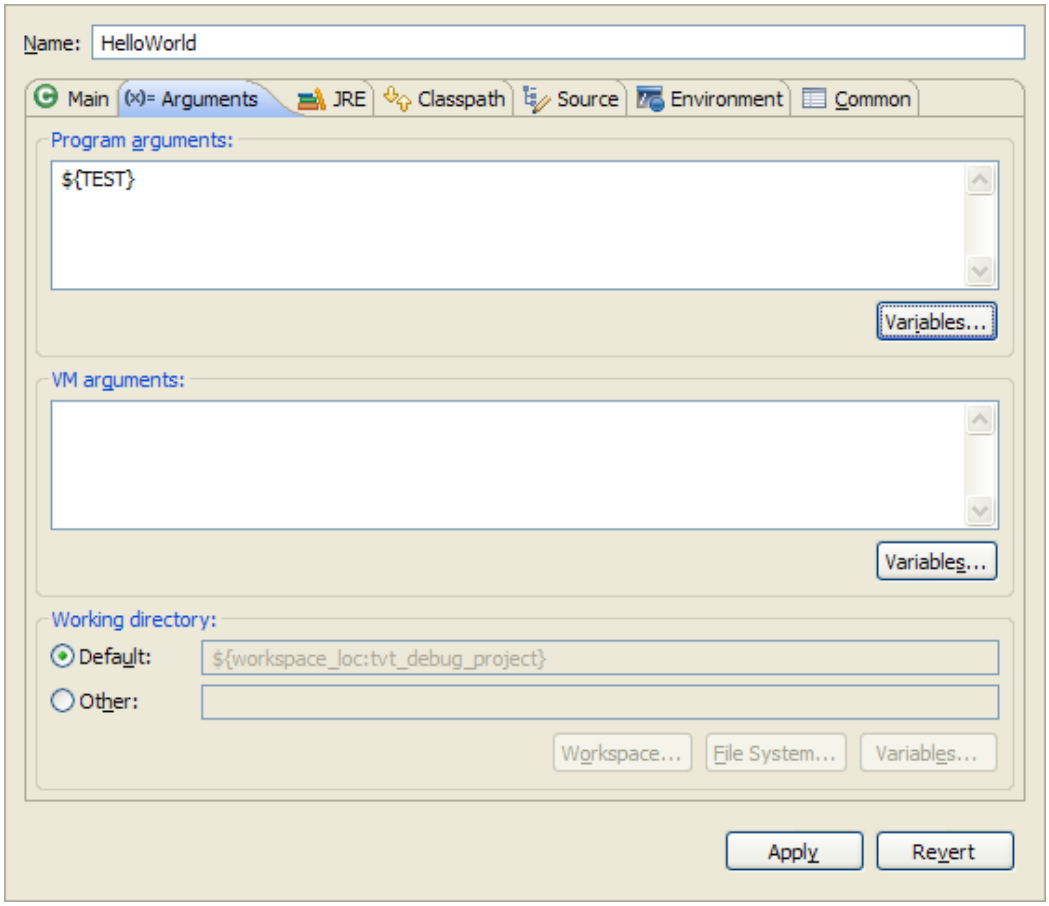
In this case the *TEST* variable does nothing, but in the 'real' world it could point to a directory or program that you would like to reuse the path for in configurations, etc.

If we now want to leverage this new variable, we can do so from the launch dialog on the arguments tab (for those configurations that have a standard arguments tab), by selecting the **Variables...** button for either the program or VM arguments text area. Selecting the Variables button opens a dialog with a listing of all of the string variables available.

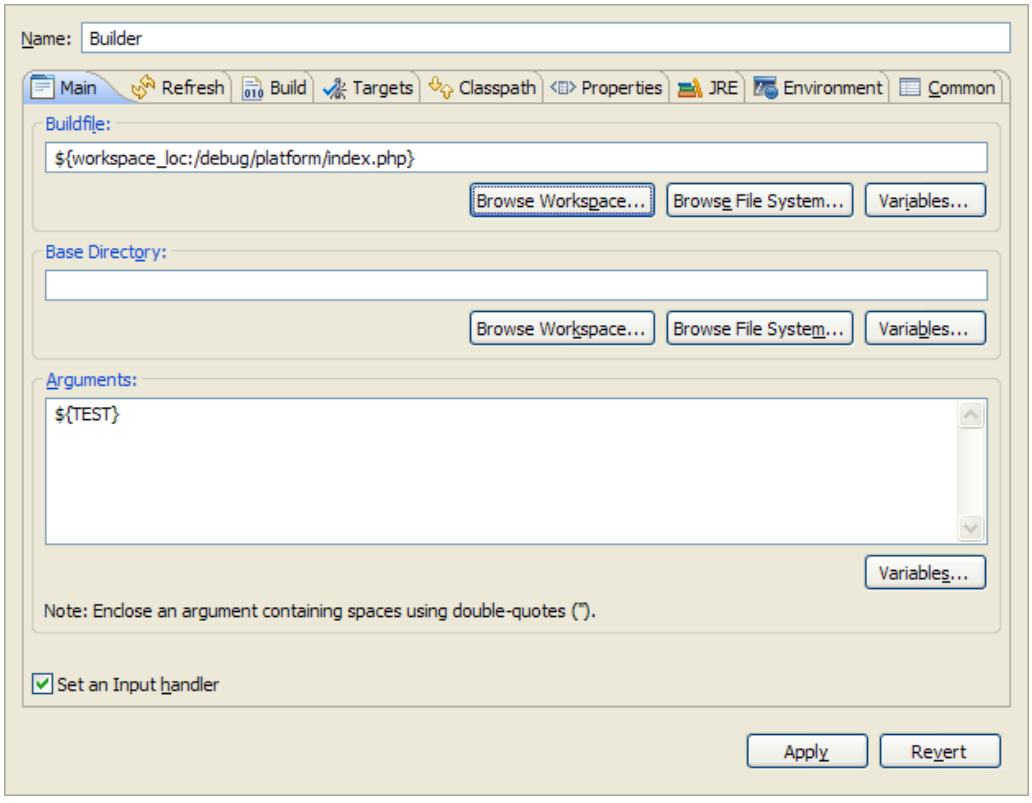
In the following screen shot we can see the *TEST* variable available in the list.



We can now select the *TEST* variable and insert in as an argument if we wish, as shown in the following screen shot.



The other mentioned platform use of string substitution in the External Tools launch dialog, used with Ant configurations, as shown in the following screen shot.



Related concepts

Debugger

Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Console Preferences
Installed JREs Preferences
Java Debug Preferences
Launching Preferences
Perspectives Preferences
Run/Debug Preferences
View Management Preferences

View Management Preferences

The following preferences can be set using the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > View Management** preference page.

These preferences allow you to configure which views will actively support automatic view opening.

Option	Description	Default
Perspectives	The listing of perspectives to select which ones will support automatic view opening	
Do not automatically open/close views which have been manually opened/closed	This option controls if views manually opened/closed will be affected by view management	On
Reset button	Resets the listing of supporting perspectives.	

Related concepts

Debugger
Java perspectives
Java views
Local debugging
Remote debugging

Related tasks

Launching a Java program
Running and debugging

Related reference

Console Preferences
Installed JREs Preferences
Java Debug Preferences
Launching Preferences
Perspectives Preferences
Run/Debug Preferences
String Substitution Preferences

Javadoc Location Dialog

This dialog lets you define the location of the Javadoc documentation for a JAR or a Java project.

You can reach this dialog the following ways:

- Select a JAR or Java project, open the context menu and select Properties > Javadoc Location or use **Properties** from the File menu
- In the Javadoc generation wizard, on the Standard doclet settings page, choose **Configure**

Javadoc can be attached to JARs, class folders or Java projects. For projects it documents the elements of all source folders, for JARs and class folders, elements contained in the JAR are documented. The location is used by

- **Open Attached Javadoc** in the Navigate menu to open the attached Javadoc of an element
- Context Help (F1) to point to a Javadoc location
- Javadoc Export Wizard to link to other documentation or as default destination for a project's documentation

Valid locations are URLs that point to a folder containing the API documentation's *index.html* and *package-list* file. Examples are:

file:///M:/JAVA/JDK1.2/DOCS/API/
http://java.sun.com/j2se/1.4/docs/api/

Option	Description	Default
Javadoc URL	Specify the location of the generated Javadoc documentation. You can **Browse** in the local file system for a Javadoc location (will result in a file:// URL)	<empty>
Validate	Validate the current location by trying to access the *index.html* and *package-list* file with the given URL. If the validation was successful, you can directly open the documentation.	

Javadoc for JARs and class folders can also be located inside an archive available on the local file system. In that case the location of the archive as well as the location of the doc inside the archive has to be specified.

Option	Description	Default

Archive location	Specify the location of the archive that contains the generated Javadoc documentation. You can **Browse** in the local file system for a Javadoc location	<empty>
Path within archive	Specify the path inside the archive that contains the generated Javadoc documentation. You can **Browse** to see the content of the archive.	<empty>
Validate	Validate the current location by trying to access the *index.html* and *package-list* file with the given URL. If the validation was successful, you can directly open the documentation.	

Java Build Path

The options in this page indicate the build path settings for a Java project. You can reach this page through the project properties (Project > Properties > Java Build Path) from the context menu on a created project or the File menu of the workbench.
The build class path is a list of paths visible to the compiler when building the project.

Source tab

Source folders are top-level folders in the project hierarchy. They are the root of packages containing .java files. The compiler will translate the contained files to .class files that will be written to the output folder.
Source folders allow to structure the project, for example to separate test from the application in two source folders. Within a source folder, a more detailed structuring can be achieved by using packages.
Each source folder can define an exclusion filter to specify which resources inside the folder should not be visible to the compiler.
Resources existing in source folders are copied to the output folder unless the setting in the Java > Compiler > Building preference page specifies that the resource is filtered. The output folder is defined per project except if a source folder specifies its own output folder.
Source folder options:

Option	Description
Add Folder	Creates a new folder to contain source
Link Source	Creates a new folder that links to an location outside of the workspace
Edit	Allows to modify the currently selected source folder or source folder attribute.
Remove	Removes the selected folders from the class path. This does not delete the folders nor their contents.
Allow output folder per source folder	Shows/Hides the 'output folder' attribute of the source folders

Source folder attributes:

Attribute	Description
Exclusion filter	Selects which resources are not visible to the compiler. For details see Inclusion and exclusion patterns.

Output folder	Only available when **Allow output folder per source folder** is checked. Defines a source folder specific output location. If not set the project's default output folder is used.
Native library location	Defines the folder that contains the native libraries (for example 'dll' or 'o' files) required at runtime by the sources in the source folder.

At the bottom of this page, the **Default output folder** field allows you to enter a path to a folder path where the compilation output for this project will reside. The default output is used for source folders that do not specify an own output folder. Use **Browse** to select an existing location from the current project.

Projects tab

In the **Required projects on the build path** list, you can add project dependencies by selecting other workbench projects to add to the build path for this new project. Adding a required project indirectly adds all its classpath entries marked as 'exported'. Setting a classpath entry as exported is done in the Order and Export tab.

The projects selected here are automatically added to the referenced projects list. The referenced project list is used to determine the build order. A project is always build after all its referenced projects are built.

Action	Description
Add	Add another project in the workspace to the build path of this project.
Edit	Edit the classpath attribute of a required project.
Remove	Removes the selected required projects from the list.

Project entry attributes:

Attribute	Description
Native library location	Specifies where native library required for the project to operate can be found.
Access rules	Specifies access rules for project contained in the library. This allows to hide content of a project.

Libraries tab

On this page, you can add libraries to the build path.

By default, the library list contains an entry representing the Java runtime library. This entry points to the JRE selected as the default JRE. The default JRE is configured in the Java > Debug > Installed JREs preferences page.

Libraries tab options:

Option	Description
Add JARs	Allows you to navigate the workbench hierarchy and select JAR files to add to the build path.
Add External JARs	Allows you to navigate the file system (outside the workbench) and select JAR files to add to the build path.
Add Variable	Allows you to add classpath variables to the build path. Classpath variables are an indirection to JARs with the benefit of avoiding local file system paths in a classpath. This is needed when projects are shared in a team. Variables can be created and edited in the Java > Build Path > Classpath Variables preference page.
Add Library	Allows to add a predefined libraries like the JRE System Library. Such libraries can stand for an arbitrary number of entries (visible as children node of the library node)
Add Class Folder	Allows to navigate the workbench hierarchy and select a class folder for the build path. The selection dialog also allows you to create a new folder.
Add External Class Folder	Allows to navigate the file system (outside the workbench) and select a class folder for the build path. The selection dialog also allows you to create a new folder.
Edit	Allows you to modify the currently selected library entry or entry attribute
Remove	Removes the selected element from the build path. This does not delete the resource.
Migrate JAR File	Migrate a JAR on the build path to a newer version. If the newer version contains refactoring scripts the refactoring stored in the script will be executed.

Libraries have the following attributes (presented as library entry children nodes):
Library entry attributes:

Attribute	Description
Javadoc location	Specifies where the library's Javadoc documentation can be found. If specified you can use **Shift+F2** on an element of this library to open its documentation.

Source attachment	Specifies where the library's source can be found.
Native library location	Specifies where native library required for the library to operate can be found.
Access rules	Specifies access rules for resources contained in the library. This allows to hide content of a library.

Order and Export tab

In the **Build class path order** list, you can click the **Up** and **Down** buttons to move the selected path entry up or down in the build path order for this new project. Checked list entries are marked as exported. Exported entries are visible to projects that require the project. Use the **Select All** and **Deselect All** to change the checked state of all entries. Source folders are always exported, and can not be deselected.

Related concepts

Build classpath
Classpath variables
Inclusion and exclusion patterns

Related reference

Frequently asked questions on JDT
Classpath variables preferences
Build path preferences
Compiler preferences

Java Compiler Page

The options in this page indicate the compiler settings for a Java project.

You can reach this page through the

- Java Compiler property page (File > Properties > Java Compiler) from the context menu on a created project or the File menu

A project can either reuse workspace default settings or use its own custom settings.

Option	Description
Enable project specific settings	Once selected, compiler settings can be configured for this project. All Java compiler preferences can be customized. At any time, it is possible to revert to workspace defaults, by using the button **Restore Defaults**.

Related concepts

Build classpath

Related tasks

Java compiler preferences
Java Build Path properties
Frequently asked questions on JDT

Java Task Tags Page

The options in this page indicate the task tags for a Java project.

You can reach this page through the

- Java task tags property page (File > Properties > Java Compiler > Task Tags) from the context menu on a created project or the File menu

A project can either reuse workspace default settings or use its own custom settings.

Option	Description
Enable project specific settings	Once selected, task tags can be configured for this project as in the Task tags preference page. At any time, it is possible to revert to workspace defaults, by using the button **Restore Defaults**.

Related reference

Task tag preferences

Source Attachment Property Page

To browse the source of a type contained in library you can attach a source archive or source folder to this library. The editor will then show the source instead of a the decompiled code. Setting the source attachment also allows source level stepping with the debugger.

The Source Attachment dialog can be reached in several ways:

- Select a JAR in the Package Explorer and choose **Properties > Java Source Attachment** from the context menu or the Project menu
- Open the Java Build Path page of a project (**Projects > Properties > Java Build Path**). On the **Libraries** page expand the library's node and select the **Source attachment** attribute and press **Edit**
- Open an editor on a class file. If the source attachment has not already been configured for this JAR, the editor contains a button Attach Source

Depending of how a JAR was contributed to the classpath, you can see different types of Source attachment dialogs:

JAR

In the Location path field, enter the path of an archive or a folder containing the source. Use either the **Workspace**, **External File** or the **External Folder** button to browse for a location.

Variable

In the Location Variable Path field enter a *variable path* that points to the source attachment's location. A variable path has as first segment a variable (which will resolve to a folder or file), the rest is an optional path extension (e.g. *MYVARIABLE/src.jar*). Use either the Variable button to select an existing variable and the Extension button to select the extension path. The Extension button is only enabled when the variable can be extended (resolves to a folder)

JRE_SRC is a reserved variable that points to a JRE selected in the Installed JREs preference page ([Image: /topic/org.eclipse.help/command_link.png]Java > Installed JREs). Go to this preference page to configure the source attachment for the JRE's library..

Related concepts

Build classpath

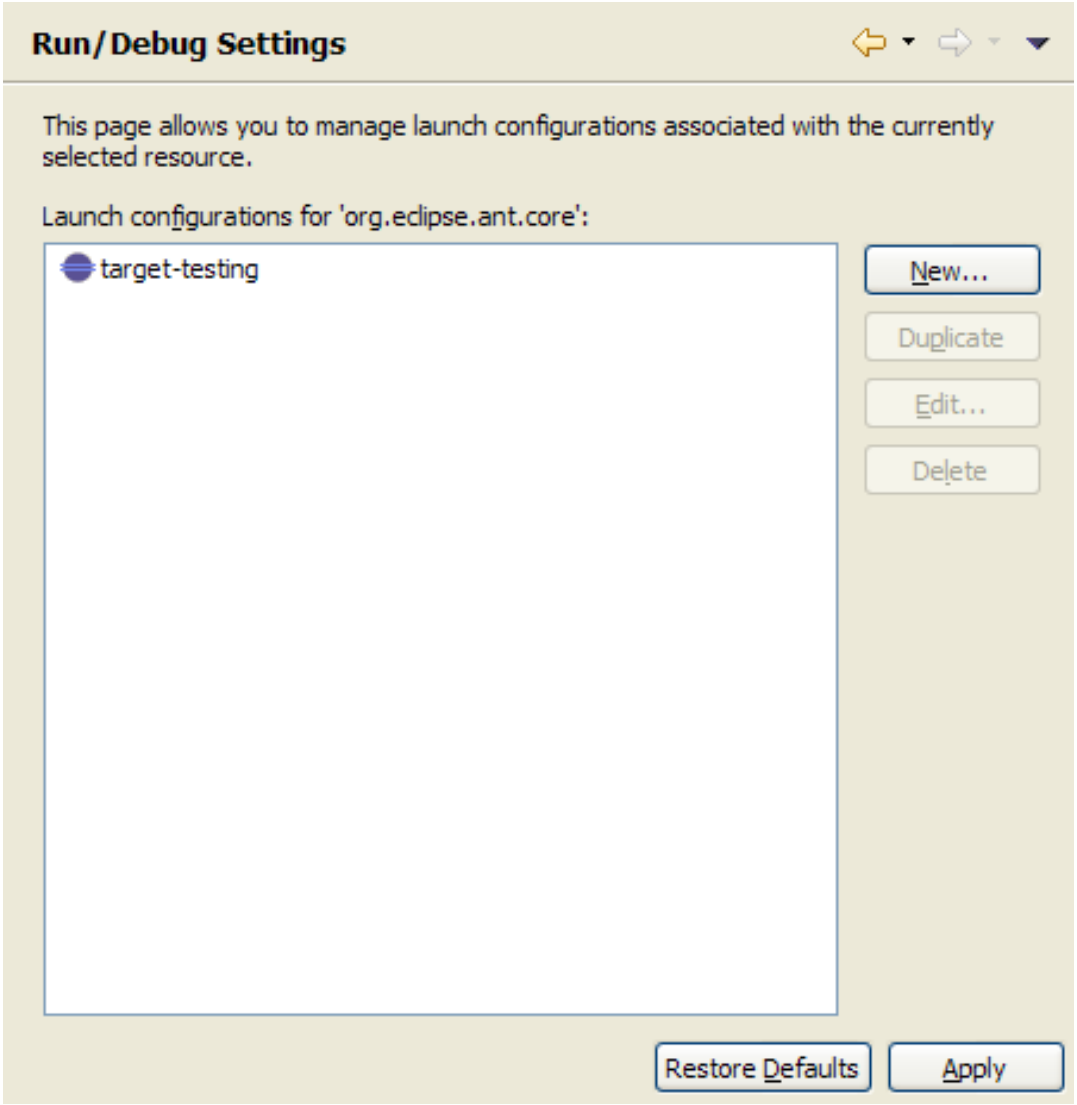
Related reference

Installed JREs preferences
Java build path properties

Run / Debug Properties Page

This property page allows you to view and manage all launch configurations that apply to the pages' corresponding resource.

Option	Description
Launch Configuration List	The list shows all of the launch configurations in the current workspace that apply to the property pages' corresponding resource.
New...	The **New...** button can be used to create a new launch configuration that applies to the corresponding resource. When selected, the New... button will present you with a dialog of all of the launch configuration types that can be used to make a configuration that applies to the corresponding resource.
Duplicate	The **Duplicate** button is used to make an identical copy of a selected launch configuration.
Edit...	The **Edit...** button will open the launch configuration edit dialog, allowing you to change the selected launch configuration.
Delete	The **Delete** button is used to delete the selected launch configuration.



Related concepts

Local Debugging
Remote Debugging

Related tasks

Launching a Java Program
Launching a Java program in debug mode
Preparing to Debug
Re-launching a program
Launching a Java applet
Disconnecting from a VM
Using the remote Java application launch configuration

Related reference

Run and Debug toolbar actions

Refactoring Wizard

A dialog based user interface guides you through the steps necessary to execute a selected refactoring. Depending on the complexity of the refactoring, either a wizard or a simple dialog is used to gather information that is required for the refactoring.

Input pages

The input pages gather information that is required for the refactoring. After you have provided the required input you can click **Ok** or **Finish** to carry out the refactoring without previewing the results. If you want to preview the changes press **Preview** or **Next**

Preview page

The JDT allows you to preview the results of a refactoring action before you execute it.

The preview page consists of two parts:

- A tree at the top containing all Java elements affected by the refactoring. Each top-level node in the tree represents one compilation unit.
  Some refactorings allow to filter the tree by different kind of changes made by the refactoring. Use the **Filter Changes** drop down to change the filtering.
- A compare viewer at the bottom. The left side of the compare viewer shows the original, the right side displays the refactored source.

Problem page

The Refactoring Problem page indicates if there are suspected, potential, or definite problems with the refactoring action you are attempting.

Four types of problems are possible:

- **Information**
  - A problem described as Information will not affect the refactoring in any way, nor will it negatively affect the code in the workbench. You can most likely ignore this type of problem.
- **Warnings**
  - Warnings attempt to predict compiler warnings. This type of problem most likely will not negatively affect the code in your workbench.
- **Errors**
  - A problem described as an Error is very likely to cause compiler errors or change your workbench code semantically. You can choose to continue with the refactoring in spite of these errors, although it is not recommended.
- **Stop problems**
  - This type of problem prevents the refactoring from taking place. For example, if you select a comment and choose the Extract Method command from it, the workbench will issue a stop problem on the refactoring attempt because you cannot extract a comment.

If there aren't any stop problems then the refactoring can be carried out by pressing the **Finish** button. To preview the results of the refactoring action, press the **Next >** button.

Refactoring without Dialog

It is also possible to rename a Java element without showing a dialog. This can be enabled and disabled on the Java preference page. If enabled, then the new name for an Java element can be typed into the editor when the rename refactoring is invoked.

Related concepts

Refactoring support

Related reference

Refactoring actions

Icons

Extract Method Errors

When you attempt to extract a method, you may get one or more of the following common errors:

- `Selected block references a local type declared outside the selection` A local type declaration is not part of the selection but is referenced by one of the statements selected for extraction. Either extend the selection that it includes the local type declaration or reduce the selection that no reference to the local type declaration is selected.
- `A local type declared in the selected block is referenced outside the selection` The selection covers a local type declaration but the type is also referenced outside the selected statements. Either extend the selection that is includes all references to the local type or reduce the selection that the local type declaration isn't selected.
- `Ambiguous return value: selected block contains more than one assignment to local variable` More than one assignment to a local variable was found inside the selected block. Either reduce the selection that only one assignment is selected or extend the selection that at least all reference except of one to the local variables are covered by the selection too.
- `Ambiguous return value: expression access to local and return statement selected` The selected statement generates more than one return value. This is for example the case if an expression is selected and an expression's argument is modified as well. To remedy this problem extend the selection to cover the read access of the modified argument as well.
- `Selection contains a break statement but the corresponding break target isn't selected` To remedy the problem either extend the selection to include the break / continue target or reduce the selection that the break / continue statement isn't covered by the selection.
- `Selection contains a continue statement but the corresponding continue target isn't selected` To remedy the problem either extend the selection to include the break / continue target or reduce the selection that the break / continue statement isn't covered by the selection.
- `Selection starts inside a comment` Parts of a comment cannot be extracted. Either extend the selection that it covers the whole comment or reduce the selection that the comment isn't covered at all.
- `Selection ends inside a comment` Parts of a comment can't be extracted. Either extend the selection that it covers the whole comment or reduce the selection that the comment isn't covered at all.
- `Cannot extract selection that ends in the middle of a statement` Adjust selection so that it fully covers a set of statements or expressions. The users can extend the selection to a valid range using the **Expand Selection to** in the Edit menu.

Related concepts

Java development tools (JDT)

Refactoring support

Related reference

Source menu

Refactor menu

Java Search Tab

This tab in the Search dialog allows you to search for Java elements. To show the tab invoke **Search > Java**.

Search string

In this field, type the expression for which you wish to search, using the wildcard characters mentioned in the dialog as needed. This field is initialized based on the current selection.

- Type patterns have the following syntax: **`[qualification '.']typeName ['<' typeArguments '>']`**

  Examples:
  - `java.lang.Object`
  - `Runnable`
  - `List<String>`

  Type arguments can be specified to search for references to parameterized types using following syntax:

  `'<' { [ '?' {'extends'|'super'} ] type ( ',' [ '?' {'extends'|'super'} ] type )* | '?' } '>'`

  Note that:
  - *'*' is not valid inside type arguments definition <>*
  - *'?' is treated as a wildcard when it is inside <> (i.e. it must be put on first position of the type argument)*

- Method patterns have the following syntax: **`[declaringType '.'] ['<' typeArguments '>'] methodName ['(' parameterTypes ')'] [returnType]`**

  Type arguments have the same syntax as explained in the type patterns section.

  Examples:
  - `java.lang.Runnable.run() void`
  - `main(*)`
  - `<String>toArray(String[])`

- Constructor patterns have the following syntax: **`['<' typeArguments '>'] [declaringQualification '.'] typeName ['(' parameterTypes ')']`**

  Type arguments have the same syntax as explained in the type patterns section.

  Note that the constructor name should not be entered as it is always the same as the type name.

  Examples:
  - `java.lang.Object()`
  - `Test(*)`
  - `<Exception>Sample(Exception)`

- Field patterns have the following syntax: **`[declaringType '.'] fieldName [fieldType]`**

Examples:
- `java.lang.String.serialVersionUID long`
- `field*`

- Package patterns have the following syntax: **`packageNameSegment {'.' packageNameSegment}`**
Examples:
- `java.lang`
- `org.e*.jdt.c*e`

Search For

Select to search for one of the following kinds of elements:
- Type
- Method
- Package
- Constructor
- Field

Limit To

Select to limit your search results to one of the following kinds of matches:
- Declarations
- Implementors (available only when searching for types)
- References
- All occurrences
- Read access (available only when searching for fields)
- Write access (available only when searching for fields)
- Match locations (available only when searching for types or methods references).
Match locations allow to further narrow the location of matches.
Select the location where to search for:
  - type reference:
    - Super class declarations
    - Annotations
    - Field types
    - Local variable types
    - Method return types
    - Method parameter types
    - Thrown exception types
    - Type parameter bounds
    - Wildcard bounds
    - Type argument
    - Cast expressions
    - Catch clauses
    - Class instance creations
    - 'instanceof' checks
  - method reference:
    - 'this' references
    - Implicit 'this' references
    - 'super' references
    - Qualified references

Search In

Select where in the scope to search for results
- Sources: Search in all source files in the scope
- Required projects: Search in all required projects
- JRE libraries: Search in JRE libraries on the build path
- Application libraries: Search in libraries on the build path

Scope

Select to limit your search results to one of the following scope
- Workspace
- Selected resources
- Enclosing Projects
- Working Set

Press Choose to select or create a working set.

Related concepts

Java search

Related reference

Search

Java Views and Editors

- Related concepts

Java editor
Java views
Java development tools (JDT)

- Related tasks

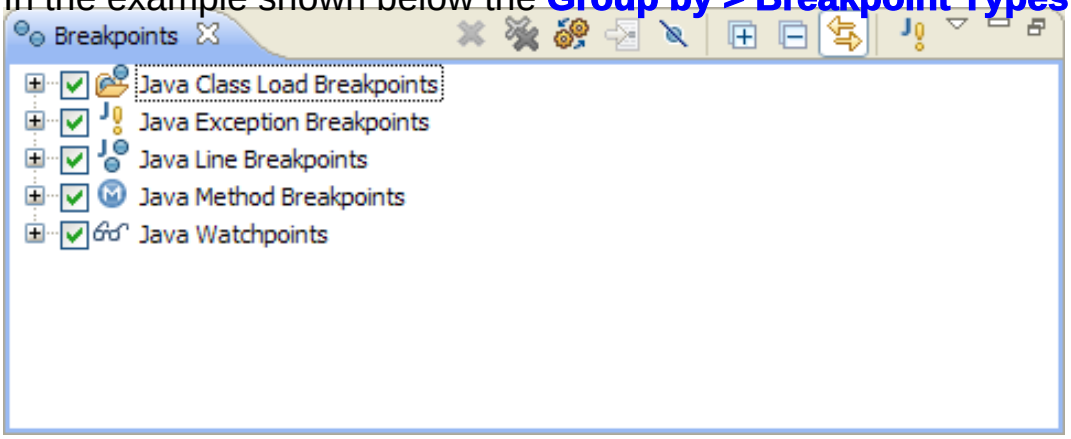
Changing the appearance of the console view
Opening an editor for a selected element
Opening an editor on a type

- Related reference

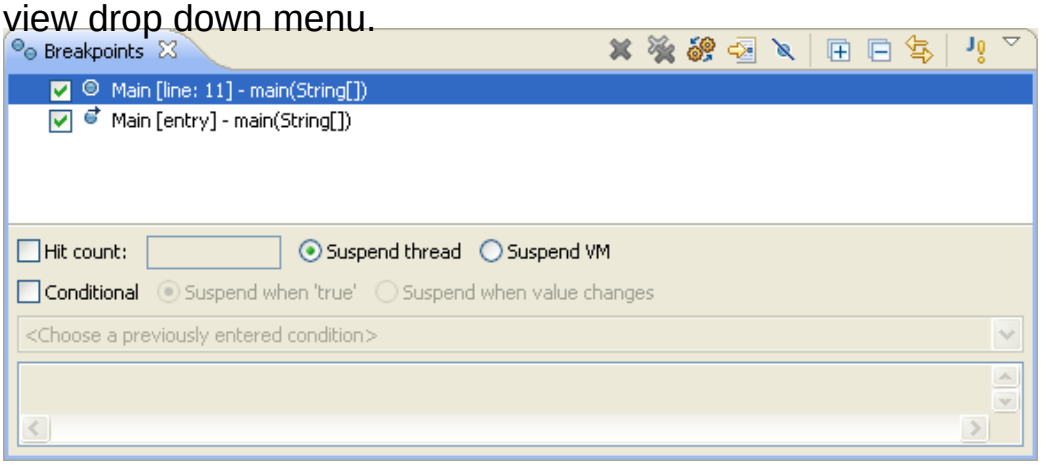
Java editor actions
Breakpoints view
Console view
Debug view
Display view
Expressions view
Java editor
Package Explorer view
Variables view
Java outline
Java scrapbook Page
Type Hierarchy view
Call Hierarchy view

Breakpoints View

The **Breakpoints View** lists all the breakpoints you currently have set in your workspace.
You can double-click a breakpoint to display its location in the editor (if applicable).
You can also enable or disable breakpoints, delete them, add new ones, group them by working set, or set hit counts.
In the example shown below the **Group by > Breakpoint Types** option is turned on.



The breakpoints view displays details of the selected breakpoint and can be used to configure attributes of the selected breakpoint similar to the **Breakpoint Properties...** action. The orientation of the detail pane can be configured from the view drop down menu.



The commands available in the **Breakpoints View** are listed below.

Breakpoints View Commands

Command	Name	Description	Availability
	Access	Changes if the selected watchpoint should suspend on access to its associated field.	Context menu
	Add Exception Breakpoint	Opens the create exception breakpoint dialog.	View action
	Breakpoint Properties...	Opens the breakpoints properties dialog.	Context menu

	Caught	Changes if the selected exception breakpoint should suspend when the specified type of exception is caught.	Context menu
	**Collapse All**	Collapses all of the items in the view.	View action
	**Copy**	Copies the selected breakpoints to the system clipboard.	Context menu
	**Disable**	Changes the selected breakpoint(s) to be disabled.	Context menu
	**Enable**	Changes the selected breakpoint(s) to be enabled.	Context menu
	Entry	Changes if the selected method breakpoint should suspend on entry to the associated method.	Context menu
	Exit	Changes if the selected method breakpoint should suspend on exit from the associated method.	Context menu
	**Expand All**	Expands all of the items in the view.	View action
	**Export Breakpoints...**	Opens the export breakpoints wizard.	Context menu
	**Go to File**	Opens the corresponding location of the breakpoint in the java editor.	Context menu and view action
	Group By...	Allows you to select an alternate grouping for your breakpoints or create your own.	View action
	**Hit Count**	Allows you to set or change the hit count for the selected breakpoint.	Context menu
	**Import Breakpoints...**	Opens the import breakpoints wizard.	Context menu

	**Link with View**	Changes if the breakpoints should be linked to the Debug View.	View action
	Modification	Changes if the selected watchpoint should suspend when its associated field is modified.	Context menu
	**Paste**	Pastes copied breakpoints into the view.	Context menu
	**Remove All**	Removes all breakpoints from the view.	Context menu and view action
	**Remove Selected Breakpoints**	Removes only the selected breakpoint(s) from the view.	Context menu and view action
	Select All	Selects all of the breakpoints in the view.	Context menu
	**Select Default Working Set**	Allows you to choose which working set will be the default one.	View action
	**Show Qualified Names**	Changes if qualified names are shown or not.	View action
	**Show Supported Breakpoints**	Changes if only supported breakpoints should be shown or not.	View action
	**Skip All**	Sets all breakpoints to be skipped.	View action
	**Suspend**	Allows you to choose what to suspend when the selected breakpoint is hit.	Context menu
	Uncaught	Changes if the selected exception breakpoint should suspend when the specified type of exception is not caught.	Context menu
	Working Sets...	Opens the working sets dialog.	View action

Related concepts

Breakpoints
Java views
Java perspectives

Related tasks

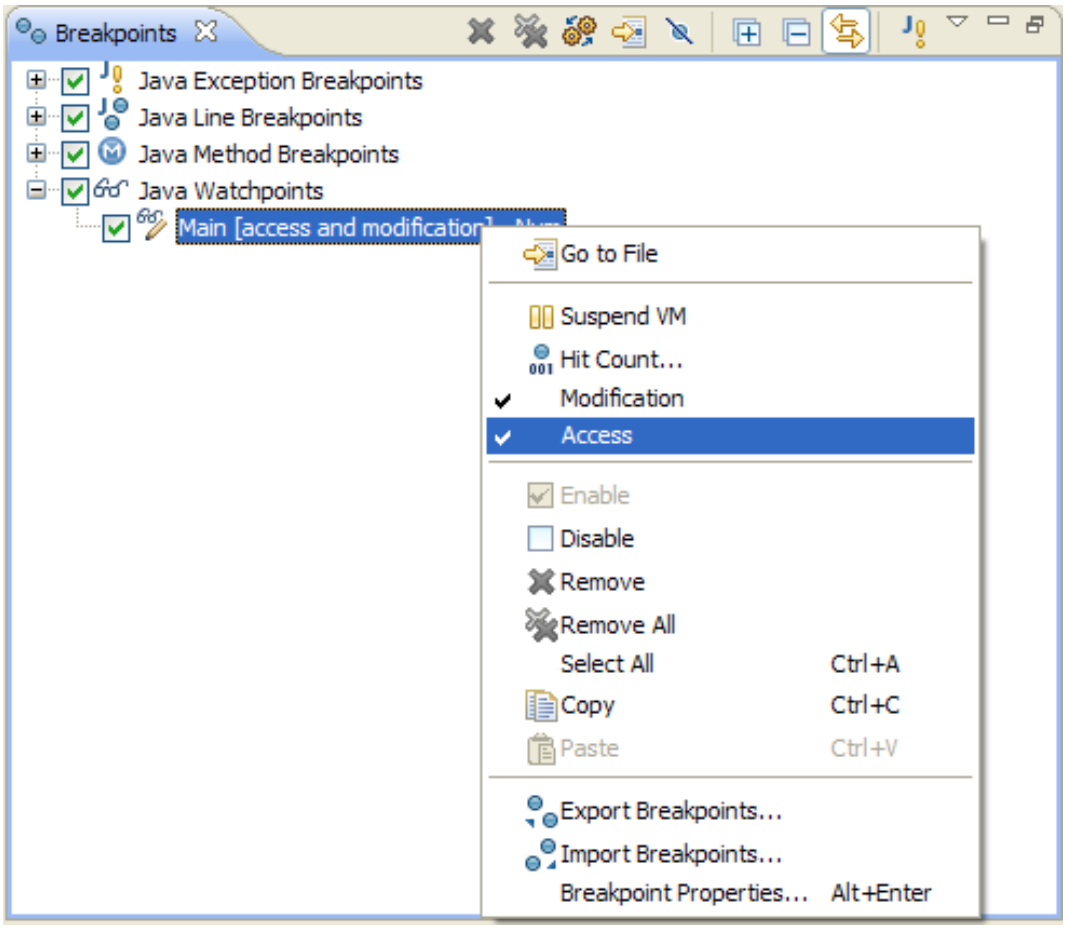
Adding breakpoints
Applying hit counts
Catching Java exceptions
Removing breakpoints
Enabling and disabling breakpoints
Managing conditional breakpoints
Setting method breakpoints

Related reference

Views and editors

Watchpoint Access

Select the **Access** command to change if the currently selected watchpoint will suspend when its associated field is accessed or read.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Watchpoint Access Option

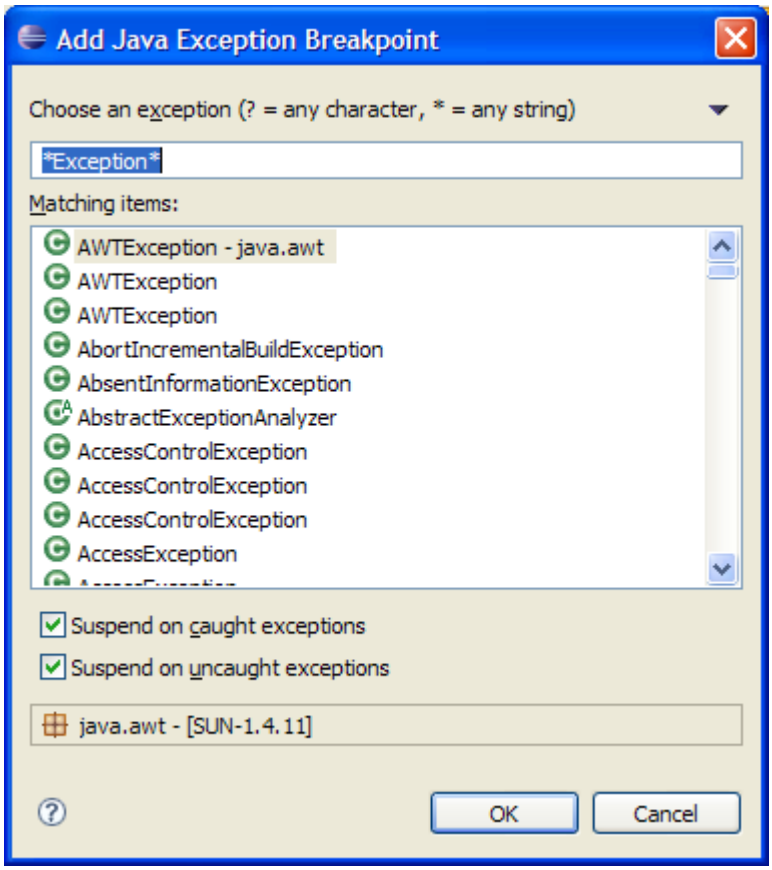
Add Java Exception Breakpoint

Select the **Add Exception Breakpoint** command [ ] to add a Java exception breakpoint.

In the resulting dialog:

- In the **Choose an Exception** field, type a string that is contained in the name of the exception you want to add. You can use wildcards as needed ("*" for any string and "?" for any character).
- In the **Matching types** list , select the exception you want to add.
- Select **Caught** and **Uncaught** as needed to indicate on which exception type you want to suspend the program.

The **Add Java Exception Breakpoint Dialog**.



Related concepts

Breakpoints

Related tasks

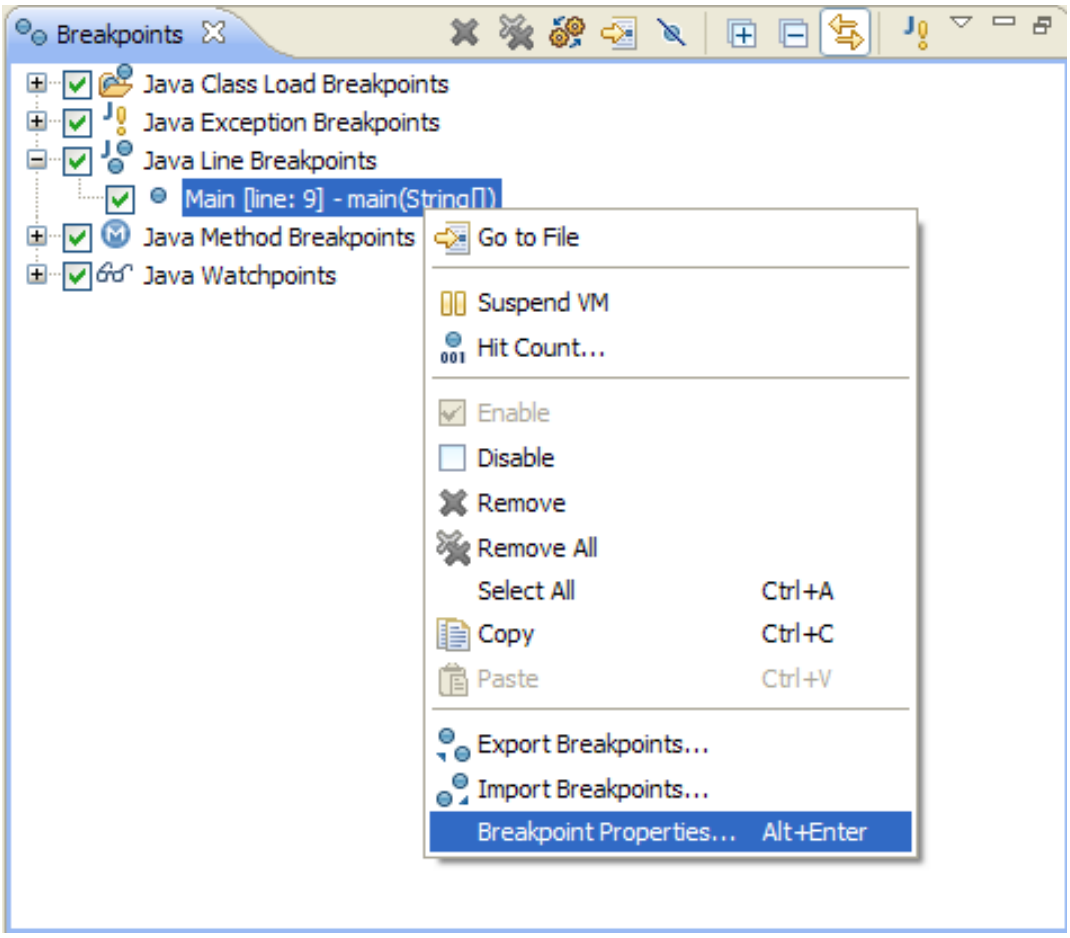
Catching Java exceptions
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

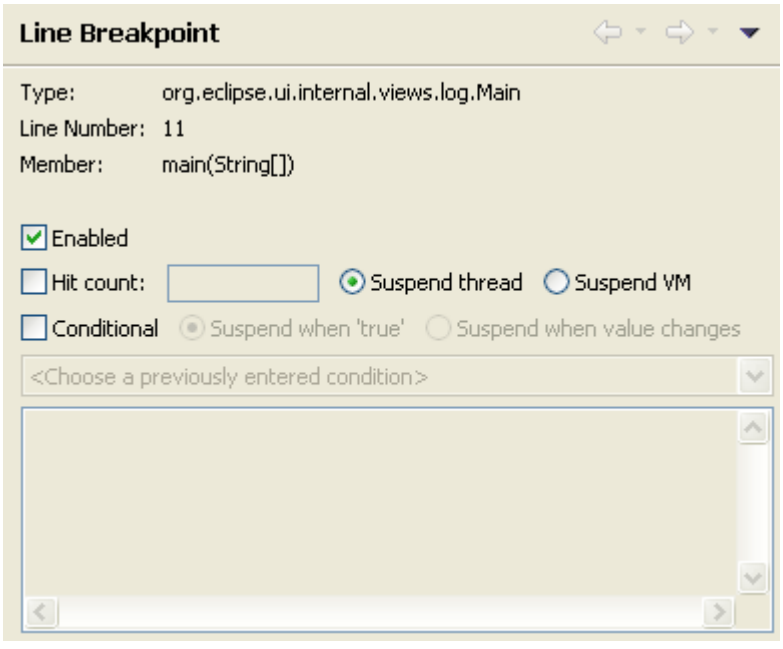
Breakpoints View
Run Menu
Caught Exception Option
Uncaught Exception Option

Breakpoint Properties...

Select the **Breakpoint Properties...** command to open the breakpoint properties dialog for the currently selected breakpoint.



There are separate properties for each type of breakpoint, thus resulting in varying properties dialogs. The example given below is for a line breakpoint.



Related tasks

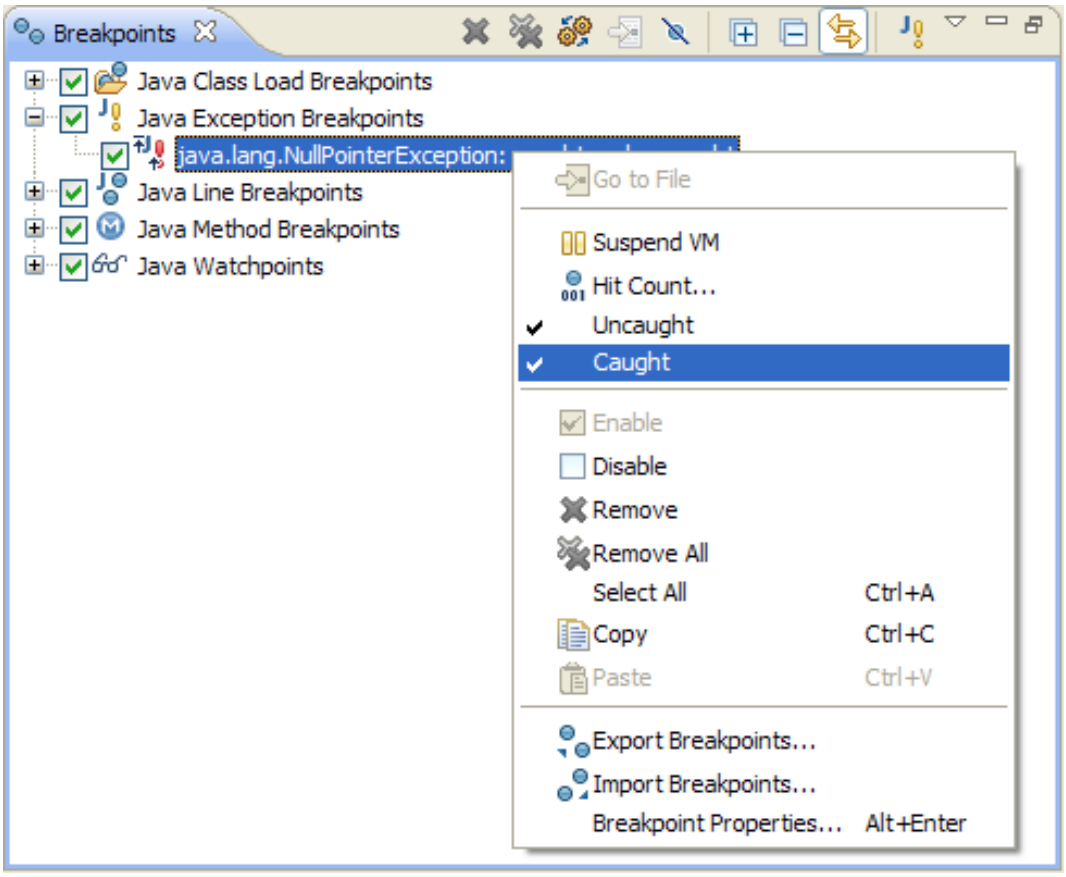
Adding breakpoints
Applying hit counts
Catching Java exceptions
Removing breakpoints
Enabling and disabling breakpoints
Managing conditional breakpoints
Setting method breakpoints

- Related reference

Breakpoints View
Condition option
Enabled option
Hit count option
Exception breakpoint caught option
Exception breakpoint suspend on subclass option
Exception breakpoint uncaught option
Method breakpoint entry option
Method breakpoint exit option
Suspend Policy
Watchpoint access option
Watchpoint modification option

Exception Breakpoint Caught

Select the **Caught** command to change if the currently selected Java exception breakpoint will suspend on caught exceptions of the same type.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Catching Java exceptions
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Caught Exception Option

Collapse All

Select the **Collapse All** command [] to collapse all of the currently elements in the view.

This command applies to:

- **Breakpoints View**
- **Expressions View**
- Registers View
- **Variables View**

Copy

Select the **Copy** command [] to copy the selected contents from the view onto the system clipboard. You can also use the standard keyboard shortcut ***Ctrl+C***.
This command applies to:
- **Breakpoints View**
- **Console View**
- **Debug View**
- **Display View**
- **Expressions View**
- Registers View
- **Variables View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)

Disable Breakpoints

Select the **Disable** command [☐] to disable the currently selected breakpoint(s).

- Related concepts

Breakpoints

- Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

- Related reference

Breakpoints View
Breakpoint Enabled Option

Enable Breakpoints

Select the **Enable** command [☑] to enable the currently selected breakpoint(s).

- Related concepts

Breakpoints

- Related tasks

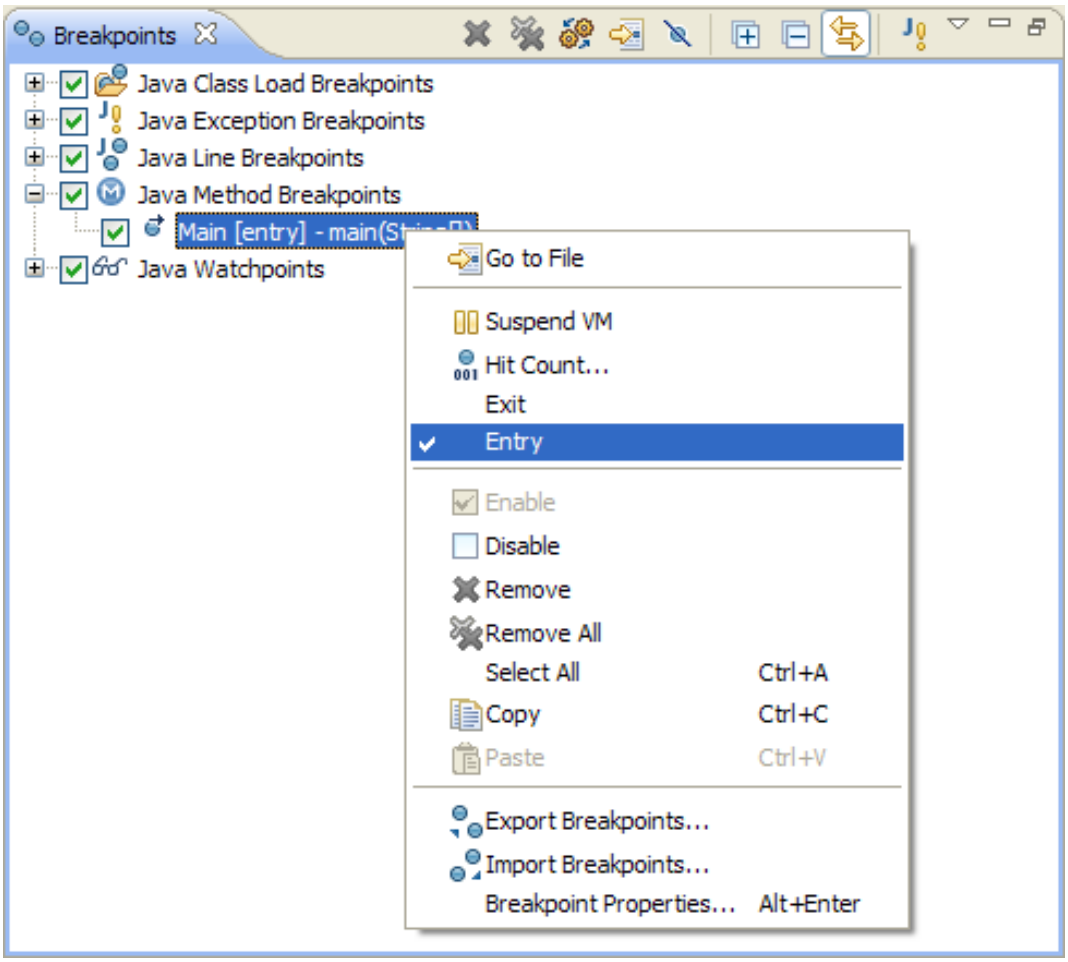
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

- Related reference

Breakpoints View
Breakpoint Enabled Option

Method Breakpoint Entry

Select the **Entry** command to change if the selected method breakpoint will suspend on entry to the associated method.



Related concepts

Breakpoints

Related tasks

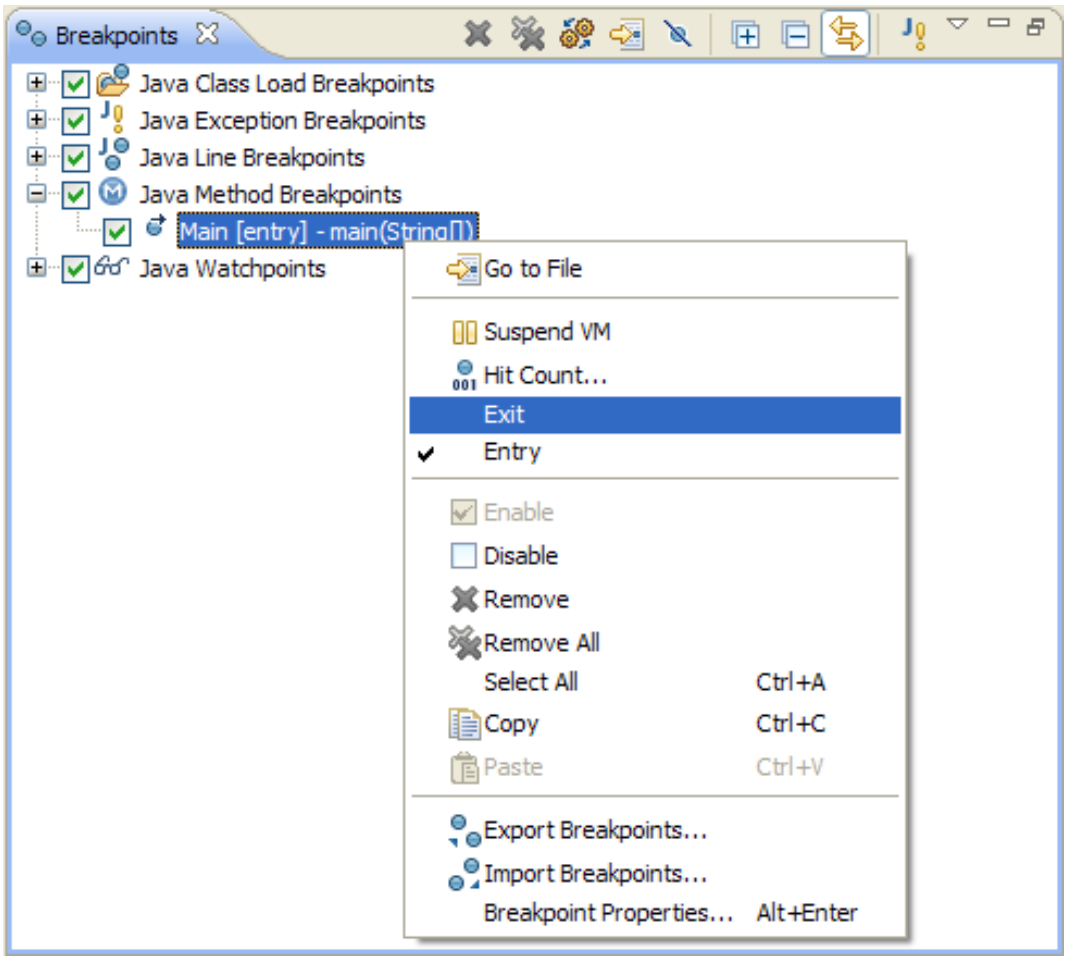
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Method Breakpoint Entry Option

Method Breakpoint Exit

Select the **Exit** command to change if the selected method breakpoint will suspend on exit from the associated method.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Method Breakpoint Exit Option

Expand All

Select the **Expand All** command [] to expand all of the items in the **Breakpoints View**.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View

Export Breakpoints

Select the **Export Breakpoints...** command [ ] to start the [Image: /topic/org.eclipse.help/command_link.png]**Export Breakpoints** wizard which will help you export breakpoints to a file.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Export Breakpoints wizard
Import Breakpoints wizard
Import Breakpoints command

Go to File for Breakpoint

Select the **Go to File** command [] to open the associated resource for the breakpoint, make it active and highlight the location of the breakpoint. If the resource is already open it is made active and the breakpoint location is highlighted.

- Related concepts

Breakpoints

- Related tasks

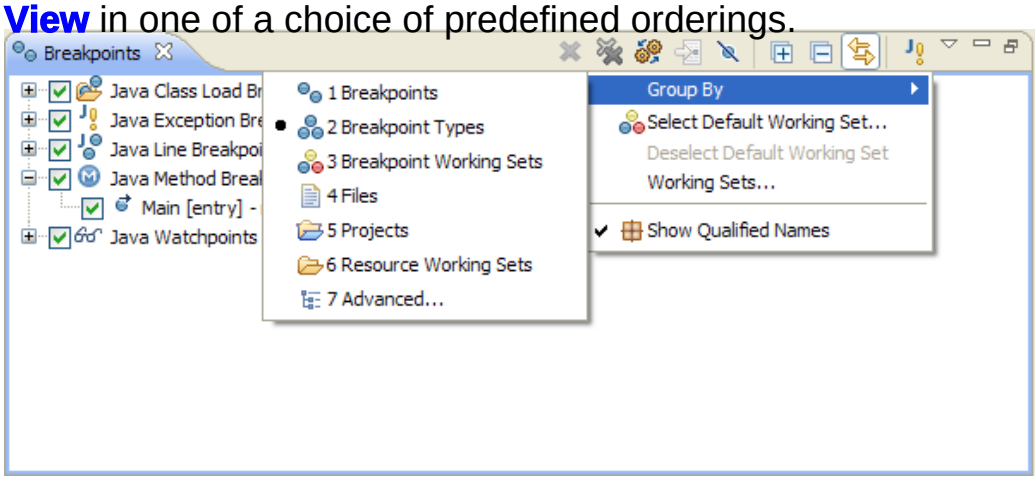
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

- Related reference

Breakpoints View

Group Breakpoints By

Select the **Group By** view menu item to group the breakpoints in the **Breakpoints View** in one of a choice of predefined orderings.

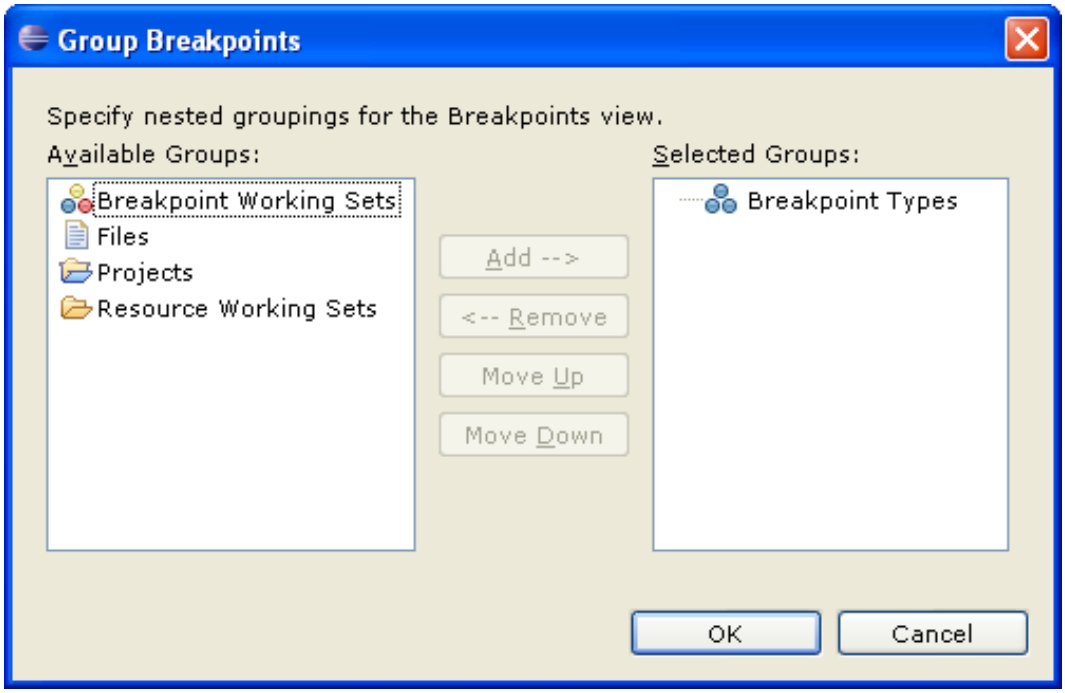


The orderings are as follows:

- **Breakpoints** - a standard list of all breakpoints
- **Breakpoint Types** - all breakpoints organized into their respective types
- **Breakpoint Working Sets** - all breakpoints are organized into the working sets they belong to
- **Files** - all breakpoints are organized by the files that they belong to
- **Projects** - all breakpoints are organized by the project that they belong to
- **Resource Working Sets** - all breakpoints are organized by the resource working sets that they belong to
- **Advanced...**- See below.

The **Advanced...** command opens a dialog that allows you to specify multiple levels of groupings for your breakpoints. For example you could have breakpoint grouped by type, which are then further grouped by resource working set, which are then further grouped by projects.

Below is the **Group Breakpoints** dialog.



- Related concepts

Breakpoints

- Related tasks

Adding breakpoints

Create a breakpoint working set

Removing breakpoints

Launching a Java program

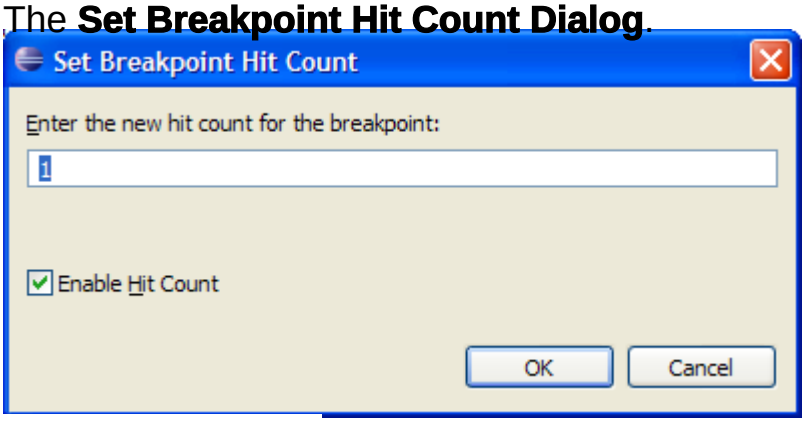
Running and debugging

- Related reference

Breakpoints View

Breakpoint Hit Count

Select the **Hit Count** command [ ] to edit the hit count for the currently selected breakpoint.
The command opens the **Set Breakpoint Hit Count** dialog which allows you to enter an integer value hit count to apply to the selected breakpoint.
The **Set Breakpoint Hit Count Dialog**.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Breakpoint Hit Count Option

Import Breakpoints

Select the **Import Breakpoints...** command [] to start the [Image: /topic/org.eclipse.help/command_link.png]**Import Breakpoints** wizard which will help you import breakpoints into your workspace.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Export Breakpoints wizard
Import Breakpoints wizard
Export Breakpoints command

Link Breakpoints View with Debug View

Select the **Link with Debug View** command [] to have the **Breakpoints View** be updated with information from the **Debug View**. When an application suspends on a breakpoint in the **Debug View**, the associated breakpoint will be highlighted in the **Breakpoints View**.

- Related concepts

Breakpoints

- Related tasks

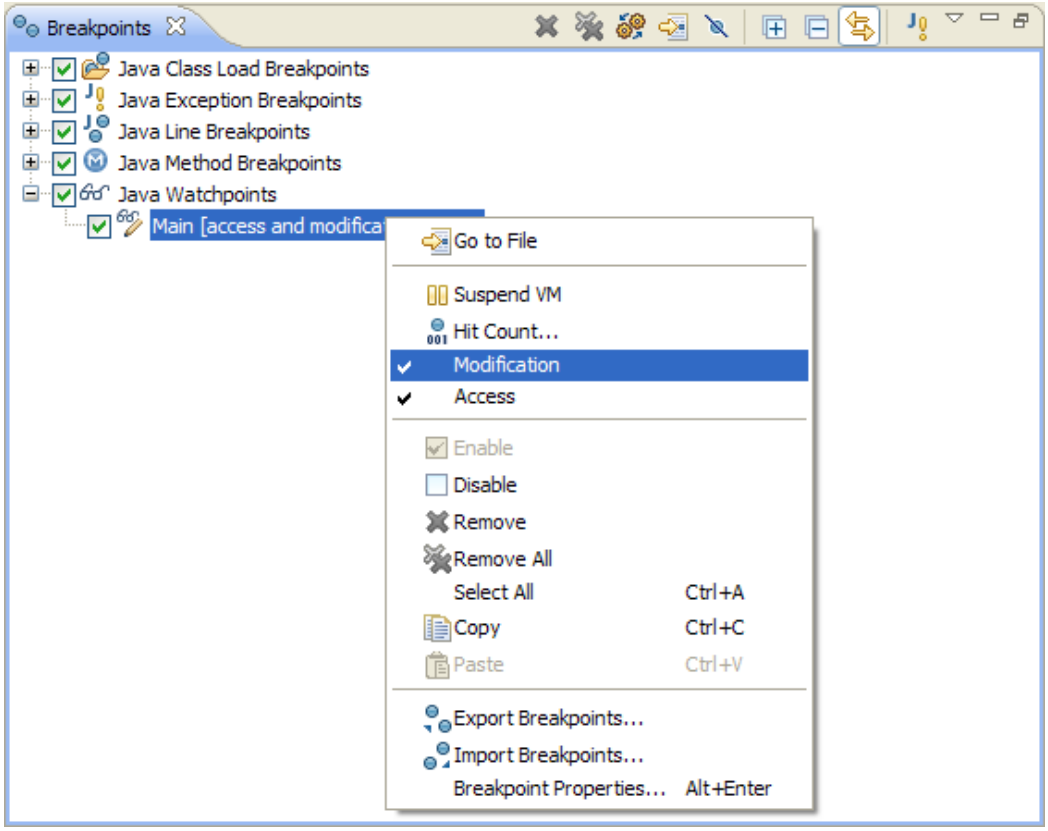
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

- Related reference

Breakpoints View

Watchpoint Modification

Select the **Modification** command to change if the currently selected watchpoint will suspend when its associated field is modified or written to.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Watchpoint Modification Option

Paste

Select the **Paste** command [ ] to copy material from the system clipboard into the current view. You can also use the standard keyboard shortcut ***Ctrl+V***.
This command applies to:
- **Breakpoints View**
- **Console View**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)

Remove Selected Breakpoint

Select the **Remove** command [✖] to remove the selected breakpoint(s) from the **Breakpoints View**.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View

Remove All Breakpoints

Select the **Remove All** command [] to remove all breakpoints from the **Breakpoints View**.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View

Select All

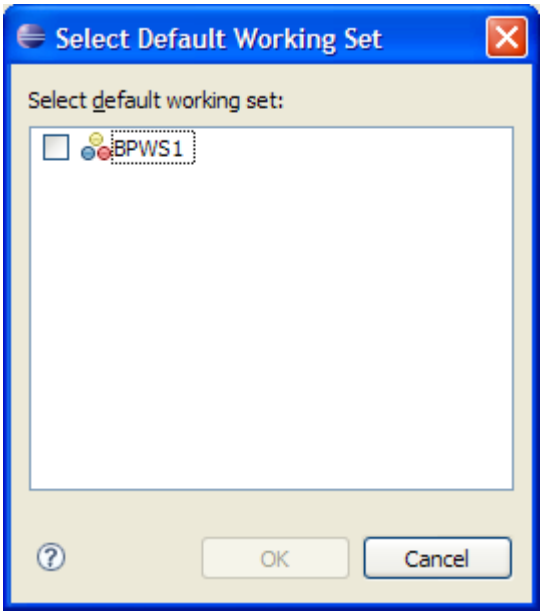
Choose this command to select all of the content in the view. You can also use the standard keyboard shortcut ***Ctrl+A***.

This command applies to:

- **Breakpoints View**
- **Console View**
- **Display View**
- **Expressions View**
- Registers View
- **Variables View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)

Select Default Breakpoint Working Set

Choose the **Select Default Working Set...** command [ ] to select a default working set.
The command opens the **Select Default Working Set** dialog, which is then used to select a working set to be the default.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Create Working set
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View

Show Qualified Names

Select the **Show Qualified Names** command [ ] the change whether qualified names should be shown in the view or not.

This command applies to:

- **Breakpoints View**
- **Expressions View**
- **Variables View**

Related reference

View Display Commands

Show Supported Breakpoints

Select the **Show Supported Breakpoints** command [] to indicate if only breakpoints applicable to the current debug target should be visible in the **Breakpoints View**.

Example: If you have C/C++ and Java breakpoints, with this option turned on you will only see those breakpoints applicable to what you are currently debugging. Meaning that when you are debugging a Java program the **Breakpoints View** will only display Java breakpoints.

- Related concepts

Breakpoints

- Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

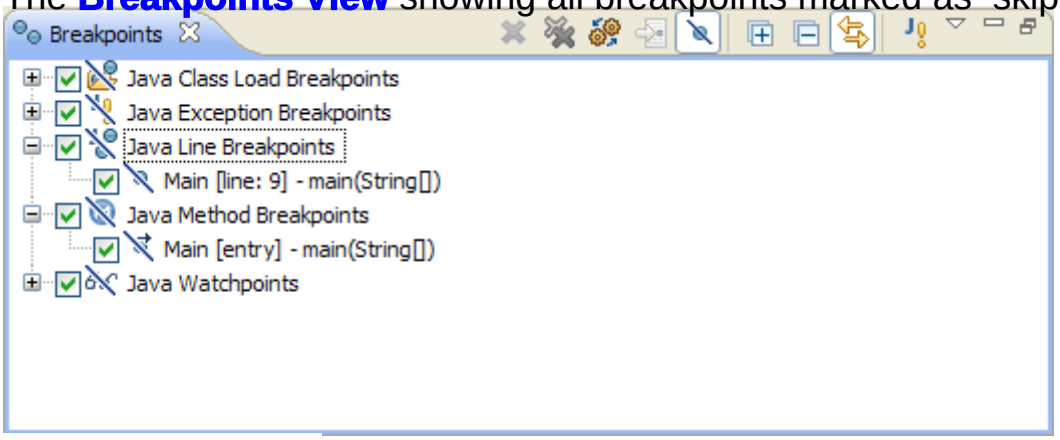
- Related reference

Breakpoints View

Skip All Breakpoints

Select the **Skip All Breakpoints** command [ ] to mark all breakpoints in the current view as skipped. Breakpoints marked as skipped will not suspend execution.

The **Breakpoints View** showing all breakpoints marked as skipped.



Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View

Breakpoint Suspend

Select the **Suspend** command [ ] to change the suspend policy of a breakpoint between suspending the entire VM and the thread in which the breakpoint suspended.

There is only one menu item visible at any one given time, showing you which suspend policy you *will be changing to* should you select it.

You can change the default suspend policy for all newly created breakpoints on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug** preference page.

Related concepts

Breakpoints

Related tasks

Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

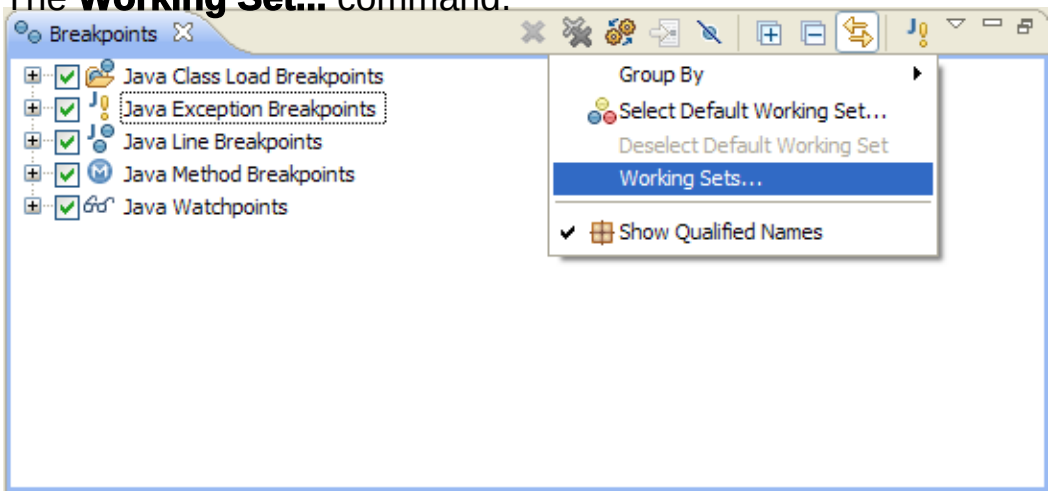
Related reference

Breakpoints View
Breakpoint Suspend Policy Option

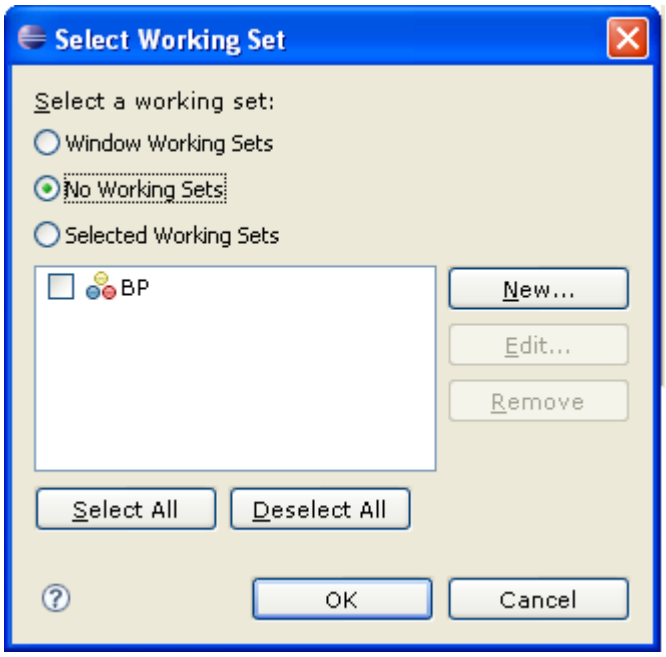
Breakpoint View Working Sets

Select the **Working Sets...** command to open the working sets dialog, which allows you to add/remove/edit working sets.

The **Working Set...** command.



The **Select Working Set** dialog.



Related concepts

Breakpoints

Related tasks

Adding breakpoints

Create Working set

Removing breakpoints

Launching a Java program

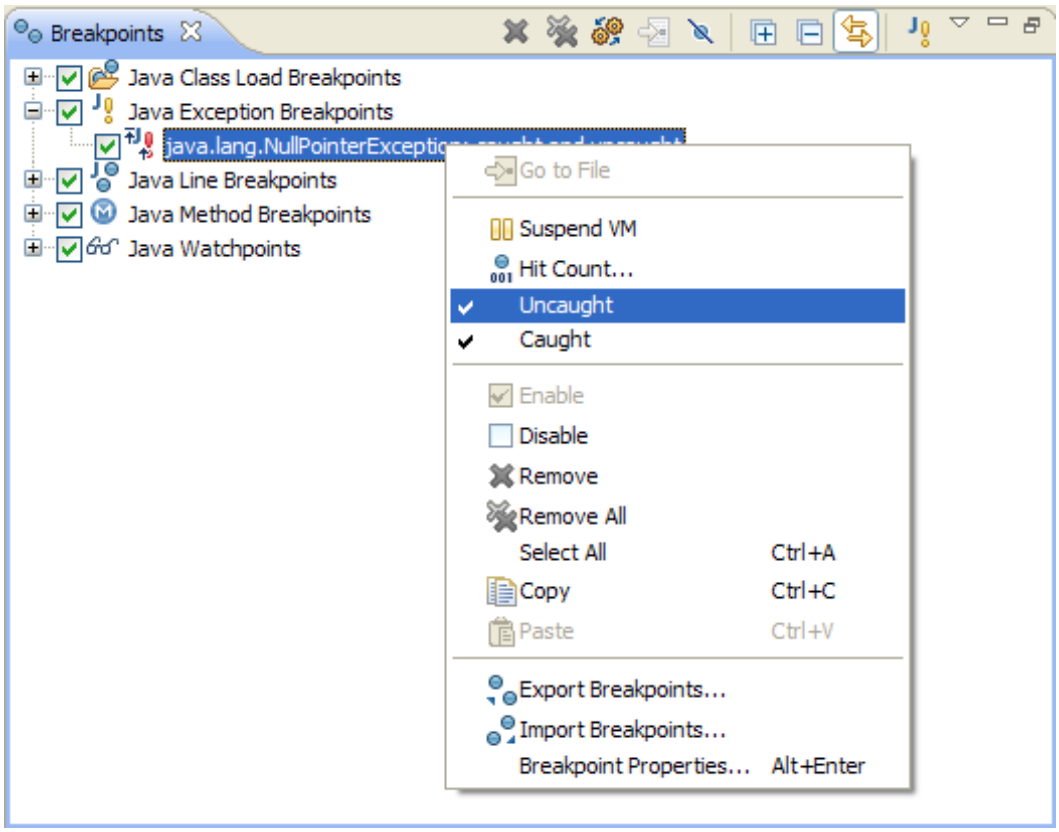
Running and debugging

Related reference

Breakpoints View

Exception Breakpoint Uncaught

Select the **Uncaught** command to change if the selected Java exception breakpoint will suspend for uncaught exceptions of the same type.



Related concepts

Breakpoints

Related tasks

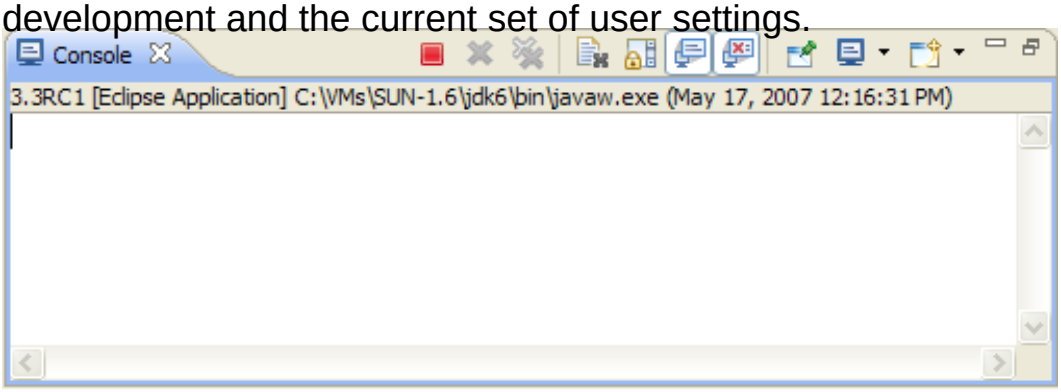
Adding breakpoints
Removing breakpoints
Launching a Java program
Running and debugging

Related reference

Breakpoints View
Exception Breakpoint Uncaught Option

Console View

The **Console View** displays a variety of console types depending on the type of development and the current set of user settings.



The three consoles that are provided by default with the Eclipse Platform are:
- The **Process Console**
- The **Stacktrace Console**
- The **CVS Console**

You can change settings for consoles on the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Console** preference page.
The commands available in the **Console View** are listed below.

Console View Commands

Command	Name	Description	Availability
	Clear Console	Clears the currently active console, and is available as both a view command and a contextual menu item.	Context menu and view action
	Display Selected Console	Opens a listing of current consoles and allows you to select which one you would like to see.	View action
	Open Console	Opens a new console of the selected type.	View action
	Pin	Pins the current console to remain on top of all other consoles.	View action
	Scroll Lock	Changes if scroll lock should be enabled or not in the current console.	Context menu and view action

Related concepts

Java views
Java perspectives

Related tasks

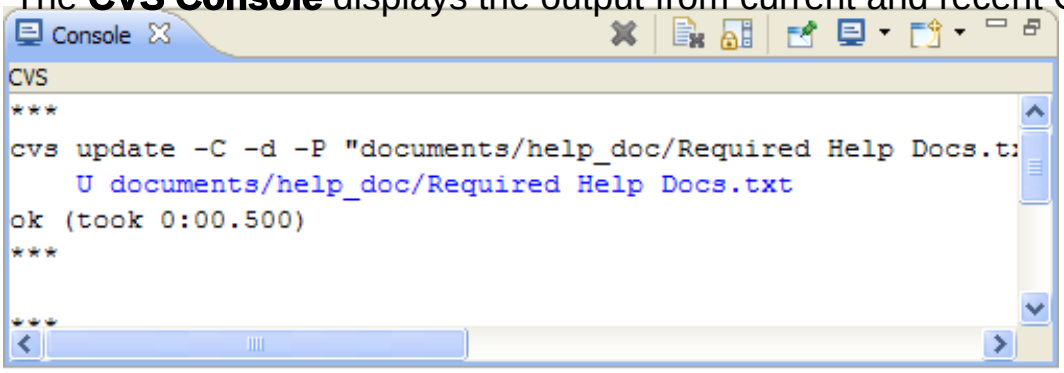
Changing the appearance of the console view
Views and editors

Related reference

Console Preferences
CVS Console
Process Console
Stacktrace Console
Views and editors

CVS Console

The **CVS Console** displays the output from current and recent CVS operations.



The extra commands available in the **CVS Console** are listed below.

CVS Console Commands

Command	Name	Description	Availability
	Copy	Copies the selected material from the console onto the system clipboard.	Context menu
	Cut	Copies the selected material to the system clipboard and removes it from the console.	Context menu
	Find/Replace	Allows you to search for and replace a specified expression.	Context menu
	Open Link	Allows you to follow the hyperlink which was right-clicked on in the current stacktrace.	Context menu
	Paste	Pastes material saved on the system clipboard into the current console.	Context menu
	Select All	Selects all of the contents of the current console.	Context menu

Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
Process Console
Stacktrace Console

Cut

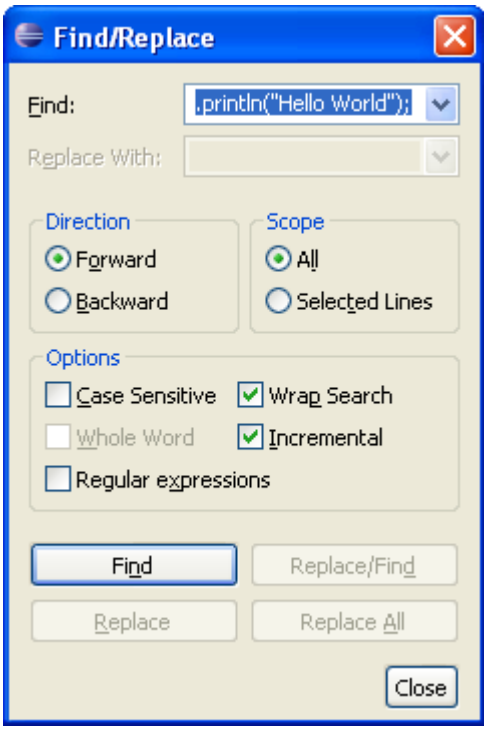
Select the **Cut** command [ ] to copy the selected contents from the view to the system clipboard and remove the selection from the view. You can also use the standard keyboard shortcut ***Ctrl+X***.

This command applies to:

- **Console View**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)

Find/Replace

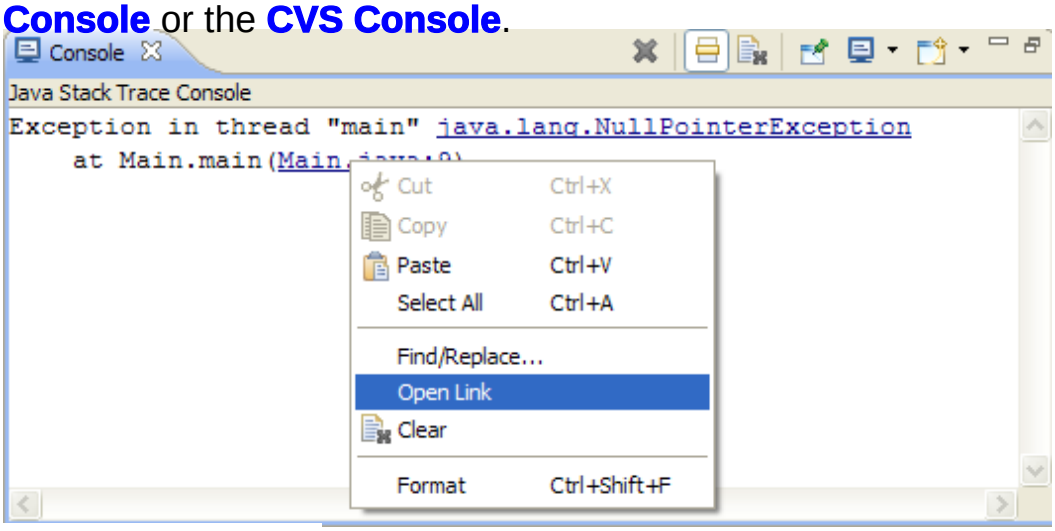
Select the **Find/Replace** command to search for and replace specific statements or portions of statements. You can also use the keyboard shortcut ***Ctrl+F***.
The resulting find/replace dialog.



This command applies to:
- **Console View**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)

Open Link

Select the **Open Link** command to follow a detected hyperlink in the **Stacktrace Console** or the **CVS Console**.



Related concepts

Java views
Java perspectives

Related tasks

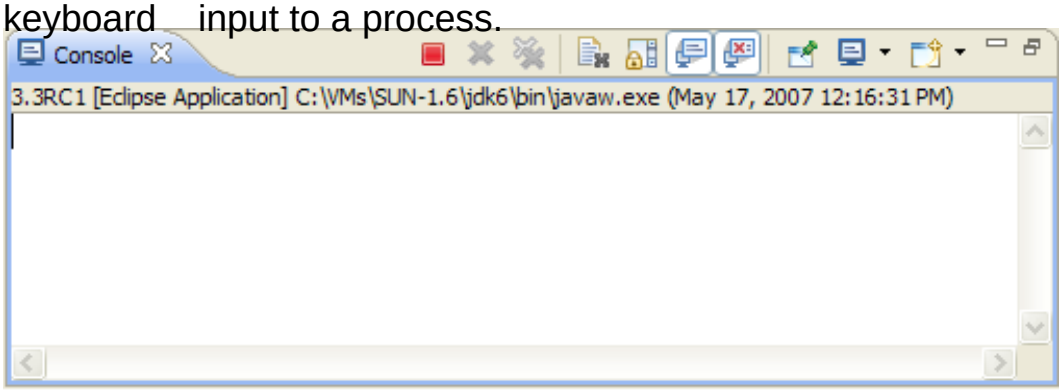
Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Process Console

The **Process Console** shows the output of a process and allows you to provide keyboard input to a process.



The **Process Console** shows three different kinds of text:

- Standard output
- Standard error
- Standard input

You can choose the different colors for these kinds of text on the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Console** preference page.

The extra commands available in the **Process Console** are listed below.

Process Console Commands

Command	Name	Description	Availability
	Copy	Copies the selected material from the console onto the system clipboard.	Context menu
	Cut	Copies the selected material to the system clipboard and removes it from the console.	Context menu
	Find/Replace	Allows you to search for and replace s specified expression.	Context menu
	Paste	Pastes material saved on the system clipboard into the current console.	Context menu
	Preferences...	Opens the **Console PreferencePage**, allowing you to customize you consoles.	Context menu

	Remove All Terminated Launches	Removes all of the terminated launches from the current console.	Context menu and view action
	**Remove Launch**	Removes the current launch from the console.	View action
	**Terminate**	Terminates the running launch in the current console.	View action
	Select All	Selects all of the contents of the current console.	Context menu
	**Show Console When Standard Out Changes**	Will open (if needed) and bring a console to the front when information is written to the System.out stream	View Action
	**Show Console When Standard Error Changes**	Will open (if needed) and bring a console to the front when information is written to the System.err stream	View Action

Related concepts

Java views
Java perspectives

Related tasks

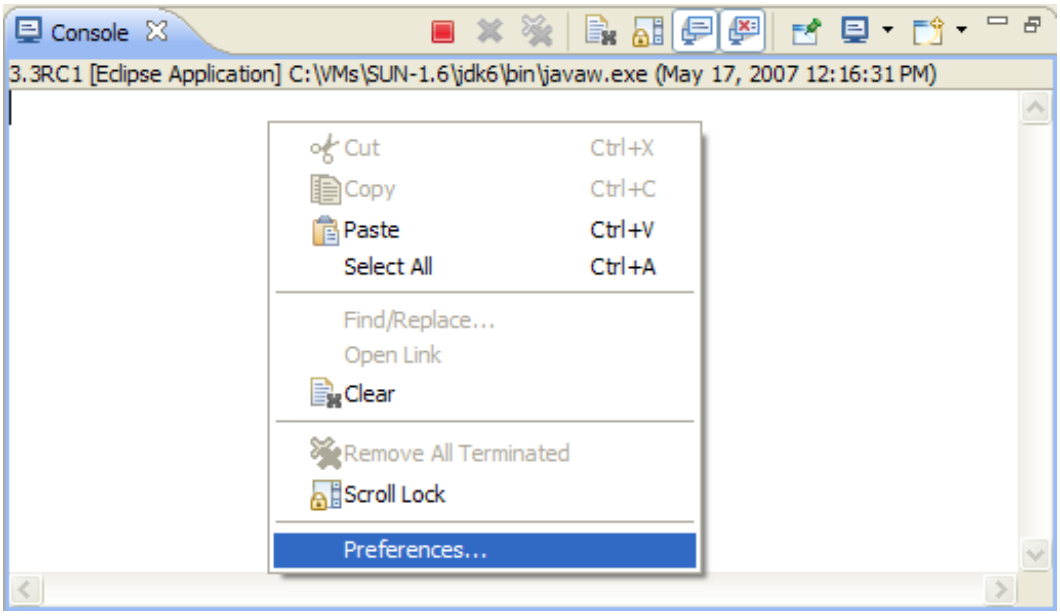
Changing the appearance of the console view
Views and editors

Related reference

Console View
Stacktrace Console
CVS Console

Console Preferences

Select the **Preferences...** command to open the [Image: /topic/org.eclipse.help/command_link.png]**Run/Debug > Console** preference page. This command only applies to the **Process Console**.



Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Remove All Terminated Launches

Select the **Remove All Terminated** command [] to remove all of the terminated launches from the **Process Console**.

- Related concepts

Java views
Java perspectives

- Related tasks

Changing the appearance of the console view
Views and editors

- Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Remove Launch

Select the **Remove Launch** command [✖] to remove the current launch from the **Process Console**.

Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Show Console When Standard Out Changes

Select the **Show Console When Standard Out Changes** command [ ] to have a console opened (if not open already) and brought to the front if the associated process of the current **Process Console** writes to Standard.out.

- Related concepts

Java views
Java perspectives

- Related tasks

Changing the appearance of the console view
Views and editors

- Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Show Console When Standard Error Changes

Select the **Show Console When Standard Error Changes** command [] to have a console opened (if not open already) and brought to the front if the associated process of the current **Process Console** writes to Standard.err.

- Related concepts

Java views
Java perspectives

- Related tasks

Changing the appearance of the console view
Views and editors

- Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Terminate

Select the **Terminate** command [] to terminate the process that is associated with the current **Process Console**.

Related concepts

Java views
Java perspectives

Related tasks

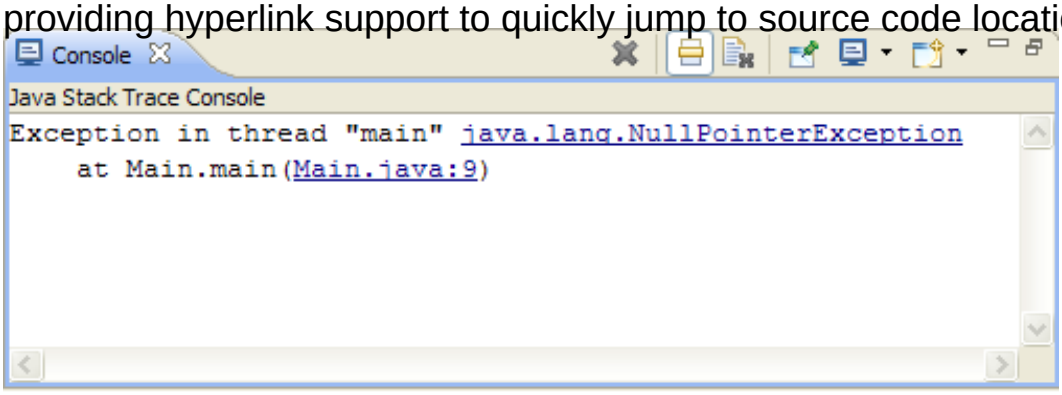
Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Stacktrace Console

The **Stacktrace Console** displays a Java stacktrace in a nicely formatted manner, providing hyperlink support to quickly jump to source code locations.



The extra commands available in the **Stacktrace Console** are listed below.

Stacktrace Console Commands

Command	Name	Description	Availability
	Auto Format	Automatically formats stacktraces when pasted into a stacktrace console	View Action
	Copy	Copies the selected material from the console onto the system clipboard.	Context menu
	Cut	Copies the selected material to the system clipboard and removes it from the console.	Context menu
	Find/Replace	Allows you to search for and replace s specified expression.	Context menu
	Format	Reformats the current stacktrace. This command in only available to stacktrace consoles.	Context menu
	Open Link	Allows you to follow the hyperlink which was right-clicked on in the current stacktrace.	Context menu

	**Paste**	Pastes material saved on the system clipboard into the current console.	Context menu
	Select All	Selects all of the contents of the current console.	Context menu

Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console

Console Autoformat

Select the **Autoformat** command [] to automatically reformat any stacktrace pasted into the **Stacktrace Console**.
This command is only available to stacktrace consoles.

Related concepts

Java views
Java perspectives

Related tasks

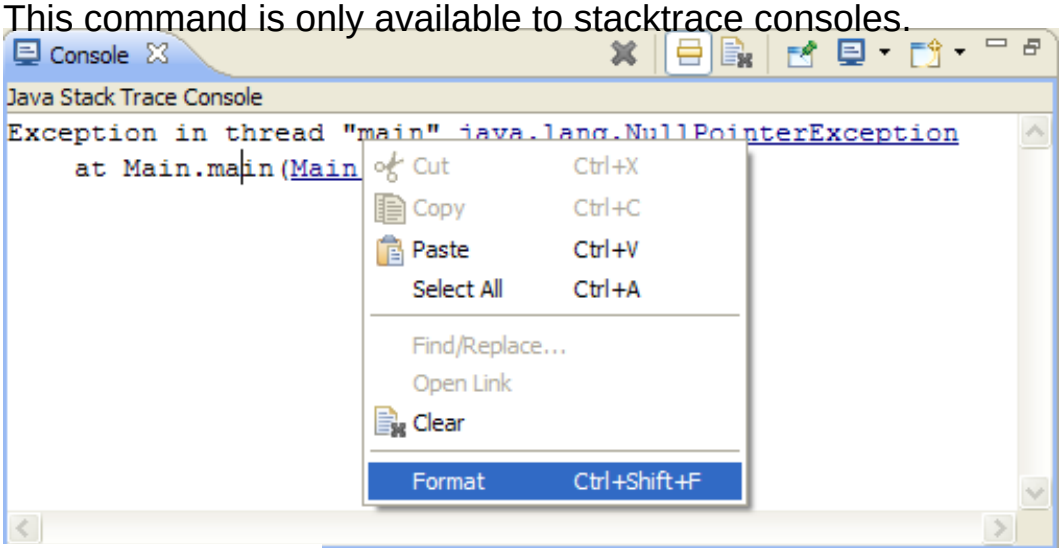
Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console

Console Format

Select the **Format** command to reformat the current stacktrace in the **Stacktrace Console**. You can also use the keyboard shortcut ***Ctrl+Shift+F***.
This command is only available to stacktrace consoles.



Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console

Clear Console

Select the **Clear Console** command [] to clear all of the contents in the **Console View**.

Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Display Selected Console

Select the **Display Selected Console** command [] to bring the console selected from the resulting list into focus.
Note: this command is only enabled if you have more than one console open.

- Related concepts

Java views
Java perspectives

- Related tasks

Changing the appearance of the console view
Views and editors

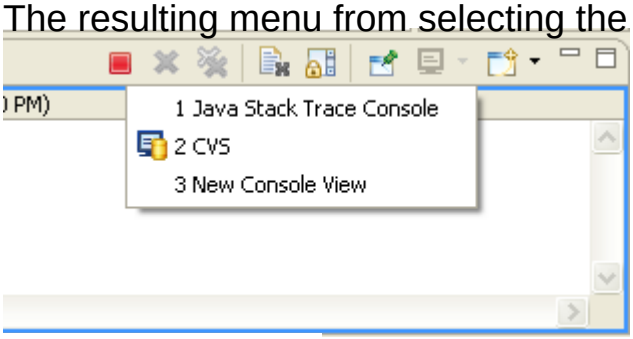
- Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Open Console

Select the **Open Console** command [ ] to open a new specific type of console in the **Console View**.

The resulting menu from selecting the drop down arrow on the command.



Related concepts

Java views
Java perspectives

Related tasks

Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Pin Console

Select the **Pin Console** command [] to ensure that the current console remains on top of all other consoles.
For example if you had two consoles, say a **Process Console** and a **Stacktrace Console**, and you had the stacktrace console pinned, even if output is written to the process console it will not come to focus in front of the stacktrace console.

- Related concepts

Java views
Java perspectives

- Related tasks

Changing the appearance of the console view
Views and editors

- Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Scroll Lock

Select the **Scroll Lock** command [] to change whether scroll lock should be enabled or disabled for all open consoles.

Related concepts

Java views
Java perspectives

Related tasks

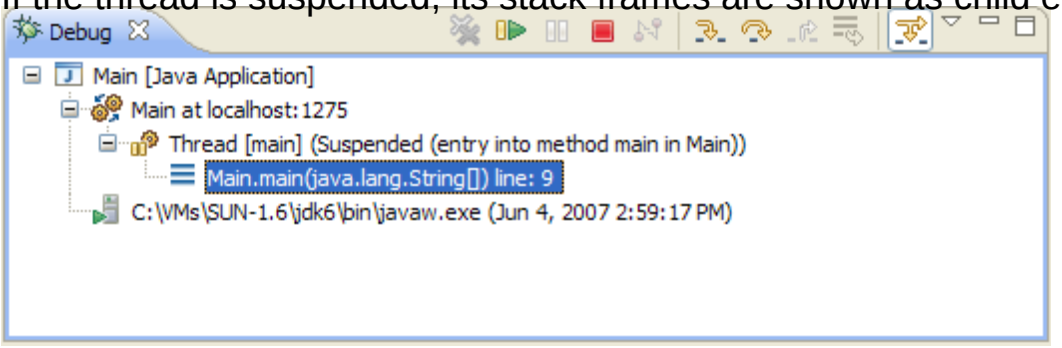
Changing the appearance of the console view
Views and editors

Related reference

Console View
CVS Console
Process Console
Stacktrace Console

Debug View

The **Debug View** allows you to manage the debugging or running of a program in the workbench. It displays the stack frame for the suspended threads for each target you are debugging. Each thread in your program appears as a node in the tree. It displays the process for each target you are running.
If the thread is suspended, its stack frames are shown as child elements.



The **Execution Control Commands** allow you to debug your program, starting, stopping and stepping through the code.

Debug View Commands

Command	Name	Description	Availability
	Automatic	Configures the debug view to switch layout between a tree and a breadcrumb viewer automatically when the view is resized.	View action
	Auto-Expand Breadcrumb	Configures the Debug Breadcrumb Drop-down to automatically expand elements when opened.	View action
	Breadcrumb	Configures the debug view layout to use a breadcrumb viewer.	View action

	**Copy Stack**	Copies the selected stack of suspended threads as well as the state of the running threads to the clipboard.	Context menu
	**Disconnect**	Disconnects the debugger from the selected debug target when debugging remotely.	Context menu and view action

	**Drop to Frame**	This command lets you drop back and reenter a specified stack frame. This feature is similar to "running backwards" and restarting your program part-way through. To drop back and reenter a specified stack frame, select the stack frame that you want to "drop" to, and select **Drop to Frame**. Some caveats apply to this feature: You cannot drop past a native method on the stack. Global data are unaffected and will retain their current values. For example, a static vector containing elements will not be cleared. *Note: This command is only enabled if the underlying VM supports this feature.*	Context menu and view action

	Edit [configuration name]...	Opens the launch configuration dialog on the associated launch configuration, allowing you to make changes.	Context menu
	Edit Step Filters	Opens the Step Filtering preference page to allowing editing of step filters	Context menu
	**Edit Source Lookup**	Opens the source lookup dialog, allowing you to make changes.	Context menu
	Filter Package	Adds the containing package for the type from the current suspended execution to the listing of step filters	Context menu
	Filter Type	Adds the type from the current suspended execution to the listing of step filters	Context menu
	Find...	Opens the debug view element searching dialog.	Context menu
	Lookup Source	Forces source lookup to take place	Context menu
	Open Declared Type	Opens the declared type for the selected stack frame in a new editor.	Context menu

	Open Declared Type Hierarchy	Opens the type hierarchy for the declared type of the selected stack frame.	Context menu
	Properties	This command displays the properties of the selected launch. It also allows you to view the full command line for a selected process.	Context menu
	**Relaunch**	This command relaunches the selected debug target.	Context menu
	**Remove All Terminated Launches**	Clears all terminated debug targets from the view display.	Context menu and view action
	**Resume**	Resumes a suspended thread.	Context menu, Run menu and view action
	**Show Monitors**	This option can be toggled to display or hide monitors. *Note: this command will only display monitor information if the underlying VM supports it.*	View action
	**Show Qualified Names**	This option can be toggled to display or hide qualified names.	View action
	**Show System Threads**	This option can be toggled to display or hide system threads	View action
	**Show Thread Groups**	This option can be toggled to display or hide thread groups	View action

	**Step Into**	Steps into the highlighted statement.	Context menu, Run menu and view action
	**Step Over**	Steps over the highlighted statement. Execution will continue at the next line either in the same method or (if you are at the end of a method) it will continue in the method from which the current method was called. The cursor jumps to the declaration of the method and selects this line.	Context menu, Run menu and view action
	**Step Return**	Steps out of the current method. This option stops execution after exiting the current method.	Context menu, Run menu and view action
	**Suspend**	Suspends the selected thread of a target so that you can browse or modify code, inspect data, step, and so on.	Context menu, Run menu and view action
	**Terminate**	Terminates the selected debug target.	Context menu, Run menu and view action
	**Terminate/Disconnect All**	Terminates all active launches in the view.	Context menu
	**Terminate and Relaunch**	Terminates the selected debug target and relaunches it.	Context menu

	Terminate and Remove	Terminates the selected debug target and removes it from the view.	Context menu
	Tree	Configures the debug view layout to use a tree viewer.	View action
	**Use Step Filters**	Toggles step filters on/off. When on, all step functions apply step filters.	Context menu, Run menu and view action

Related concepts

Debugger
Java views
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug Preferences
Views and editors
Run Menu

Execution Control Commands

Execution control commands allow you to change the execution state of code being executed.

Java Execution Control Commands

Command	Name	Description	Availability
	Resume	Resumes a suspended thread.	Context menu, Run menu and view action
	**Step Into**	Steps into the highlighted statement.	Context menu, Run menu and view action
	**Step Over**	Steps over the highlighted statement. Execution will continue at the next line either in the same method or (if you are at the end of a method) it will continue in the method from which the current method was called. The cursor jumps to the declaration of the method and selects this line.	Context menu, Run menu and view action
	**Step Return**	Steps out of the current method. This option stops execution after exiting the current method.	Context menu, Run menu and view action
	**Suspend**	Suspends the selected thread of a target so that you can browse or modify code, inspect data, step, and so on.	Context menu, Run menu and view action
	**Terminate**	Terminates the selected debug target.	Context menu, Run menu and view action

	**Terminate & Relaunch**	Terminates the selected debug target and relaunches it.	Context menu
	**Terminate & Remove**	Terminates the selected debug target and removes it from the view.	Context menu
	**Terminate/Disconnect All**	Terminates all active launches in the view.	Context menu

See the **Debug View** for more information.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Run Menu

Resume

Select the **Resume** command [] to resume the execution of the currently suspended debug target.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Run Menu

Step Into

Select the **Step Into** command [] to step into the next method call at the currently executing line of code.
To step into a method you must have execution suspended and be stepping through code.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Run Menu
Step Return
Step Over

Step Over

Select the **Step Over** command [ ] to step over the next method call (without entering it) at the currently executing line of code. Even though the method is never stepped into, the method will be executed normally.
To step over a method you must have execution suspended and be stepping through code.

- Related concepts

Debugger
Local debugging
Remote debugging

- Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

- Related reference

Debug View
Debug preferences
Execution Control Commands
Run Menu
Step Return
Step Into

Step Return

Select the **Step Return** command [] to return from a method which has been stepped into. Even though we return from the method, the remainder of the code inside the method will be executed normally.

To step return from a method you must have execution suspended and be stepping through code.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Run Menu
Step Into
Step Over

Suspend

Select the **Suspend** command [ ] to halt the execution of the currently selected thread in a debug target. Once the selected thread is suspended you can then examine its stack frames.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Resume
Run Menu

Terminate

Select the **Terminate** command [ ] to terminate the launch associated with the selected debug target.
Once a launch is terminated it can be automatically removed from the **Debug View**. To change this setting use the [Image: /topic/org.eclipse.help/command_link.png] **Run/Debug > Launching** preference page.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Relaunch
Remove All Terminated
Run Menu
Terminate/Disconnect All
Terminate and Relaunch
Terminate and Remove

Terminate/Disconnect All

Select the **Terminate/Disconnect All** command [] to terminate all the running debug targets in the **Debug View**. If the target cannot be terminated, it will be disconnected.

Once a launch is terminated it can be automatically removed from the **Debug View**. To change this setting use the [Image: /topic/org.eclipse.help/command_link.png] **Run/Debug > Launching** preference page.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Relaunch
Remove All Terminated
Terminate
Terminate and Relaunch
Terminate and Remove

Terminate and Relaunch

Select the **Terminate and Relaunch** command [] to first terminate the selected debug target and secondly, relaunch it.

Once a launch is terminated it can be automatically removed from the **Debug View**. To change this setting use the [Image: /topic/org.eclipse.help/command_link.png] **Run/Debug > Launching** preference page.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Remove All Terminated Launches
Terminate
Terminate/Disconnect All
Terminate and Remove

Terminate and Remove

Select the **Terminate and Remove** command [] to terminate the launch associated with the selected debug target and remove it from the **Debug View**.

- Related concepts

Debugger
Local debugging
Remote debugging

- Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

- Related reference

Debug View
Debug preferences
Execution Control Commands
Relaunch
Remove All Terminated
Terminate
Terminate/Disconnect All
Terminate and Relaunch

Disconnect

Select the **Disconnect** command [] to terminate the connection between the debugger and the remote debug target.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences

Drop to Frame

Select the **Drop to Frame** command [] to re-enter the selected stack frame in the **Debug View**.

Note this command is only available if the current VM supports drop to frame and the selected stackframe is not the top frame or a frame in a native method.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

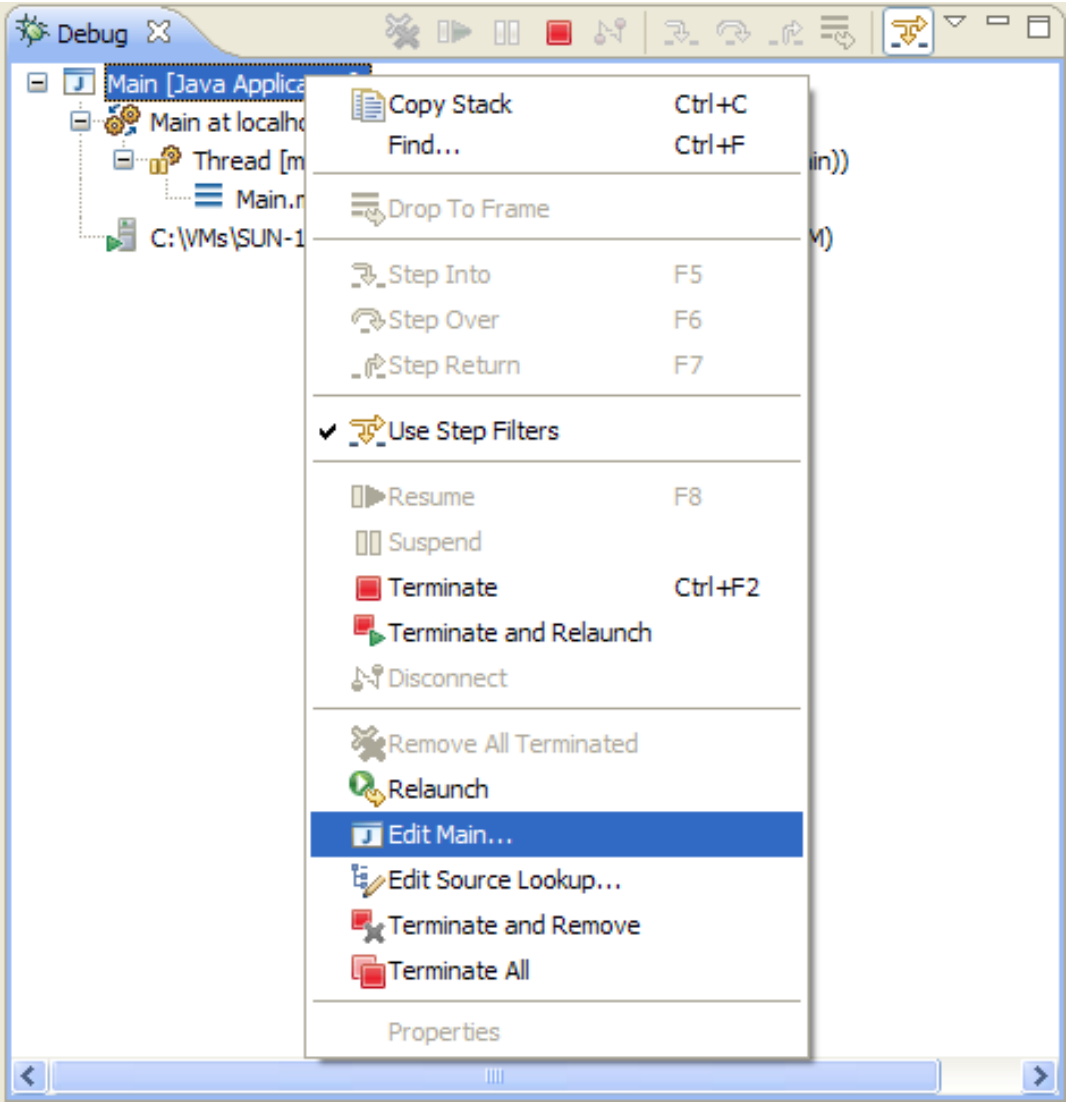
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

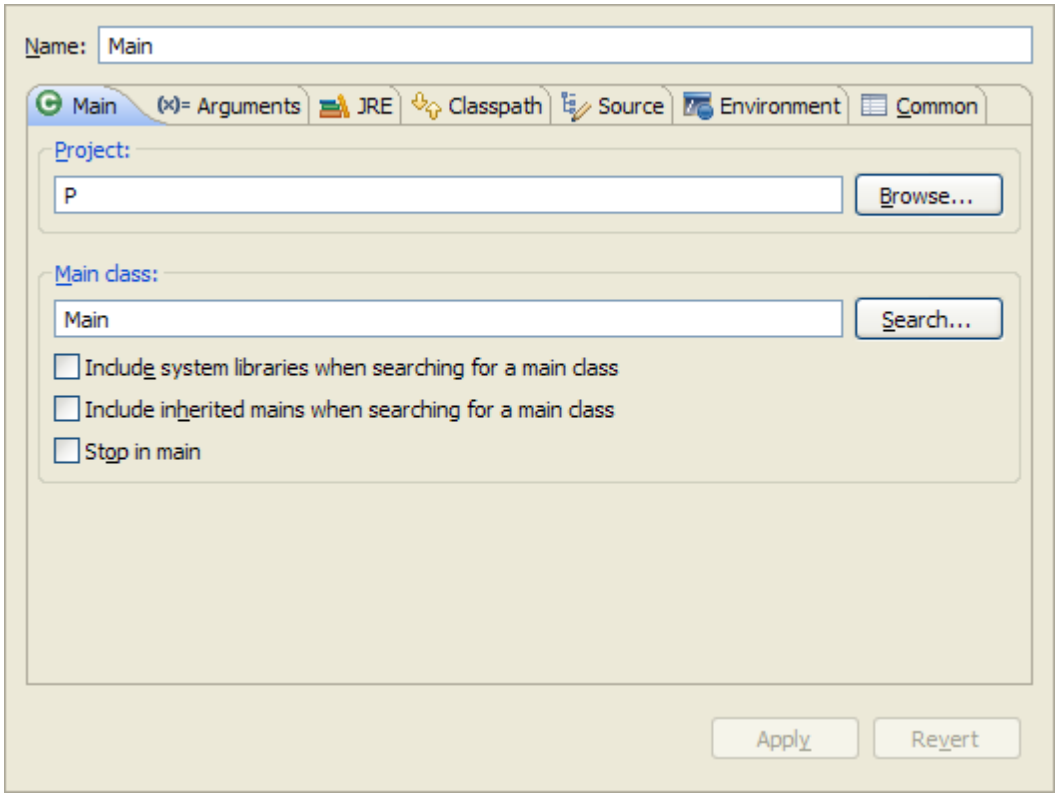
Debug View
Debug preferences

Edit...

Select the **Edit...** command to open the **Launch Configuration Dialog**, and allow you to edit the launch configuration for the selected target.



The resulting launch configuration dialog, in this example a Java Application type.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

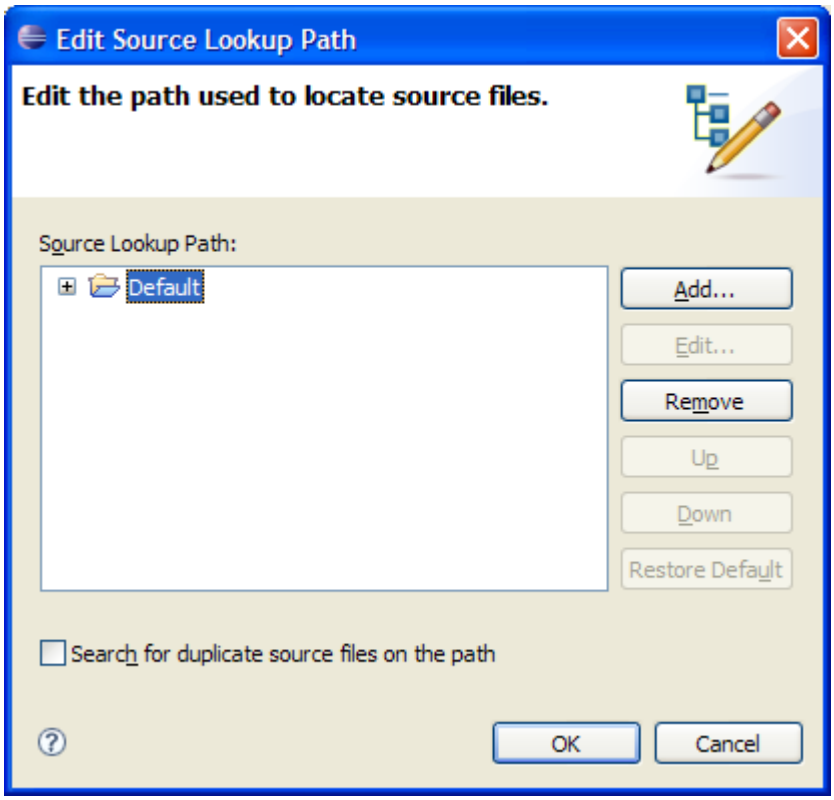
Related reference

Debug View
Debug preferences

Edit Source Lookup

Select the **Edit Source Lookup...** command [ ] to open the **Source Path Dialog**, which allows you to make changes to the source lookup path of the selected debug target.

The **Source Path Dialog**:



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

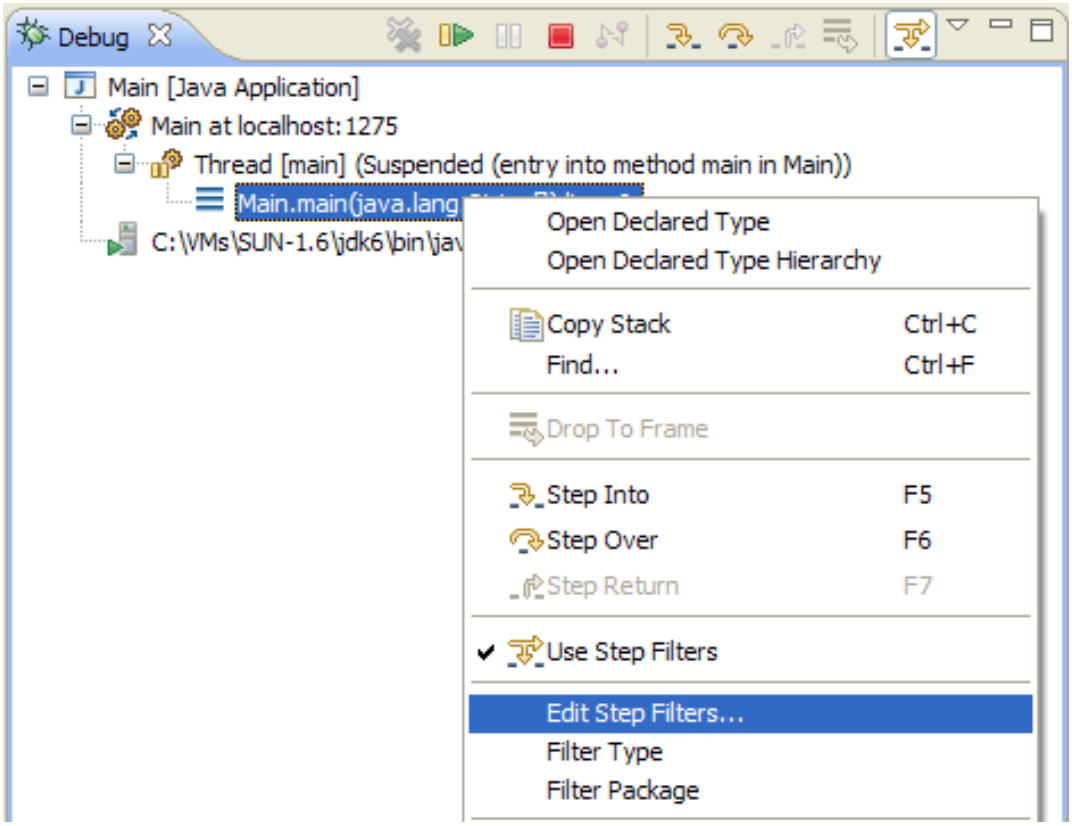
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences

Edit Step Filters

Select the **Edit Step Filters** command to open the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Step Filtering** preference page, which allows you to add and configure step filters.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

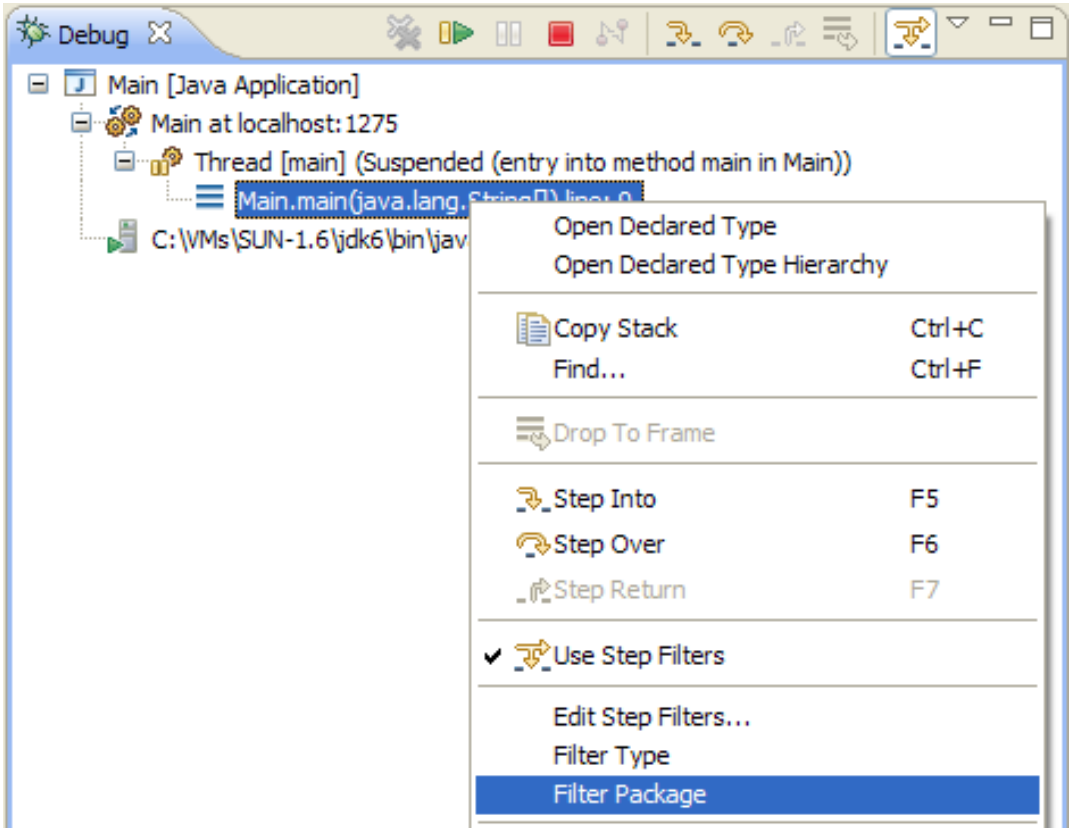
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Filter Package Command
Filter Type Command
Use Step Filters Command

Filter Package

Select the **Filter Package** command to add the package of the selected stack frame to the list of items to be filtered via step filtering.
For example, if the currently selected stack frame is of type ***java.lang.Object***, and you select the **Filter Package** command then ***java.lang.**** will be added to the list of types to be filtered via step filtering.
Warning: You can inadvertently filter out more than you expect when using this command, take care in its use.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

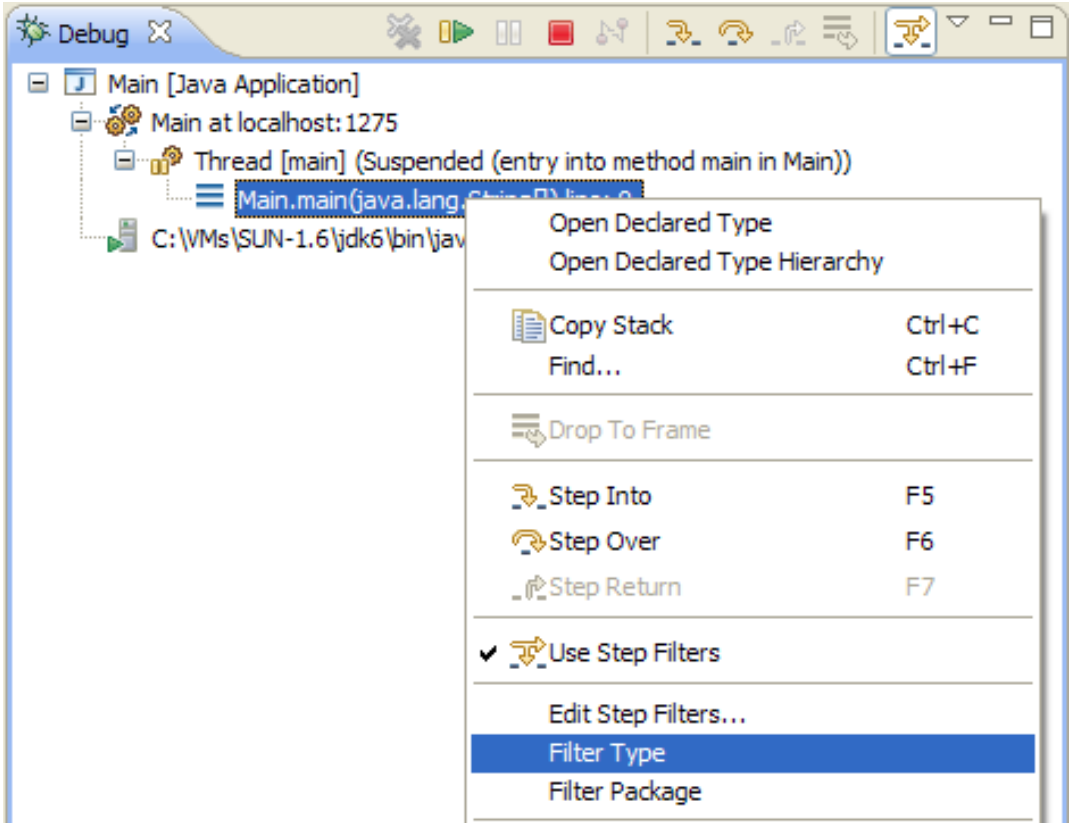
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences

Filter Type

Select the **Filter Type** command to add the type associated with the selected stack frame to the list of types to be filtered via step filtering.
For example, if the currently selected stack frame is associated with the type ***java.lang.Object***, and you select the **Filter Type** command then ***java.lang.Object*** will be added to the list of types to be filtered via step filtering.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

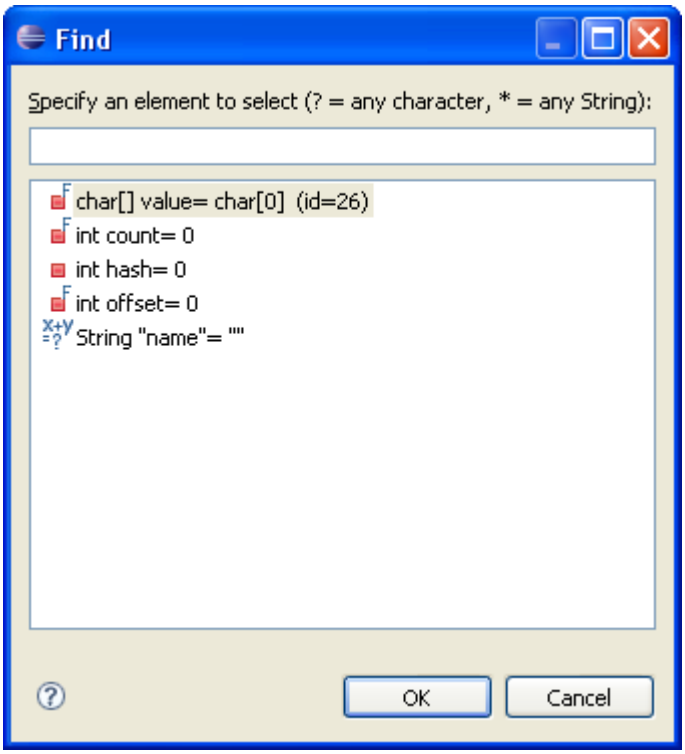
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Step Commands
Edit Step Filters Command

Find...

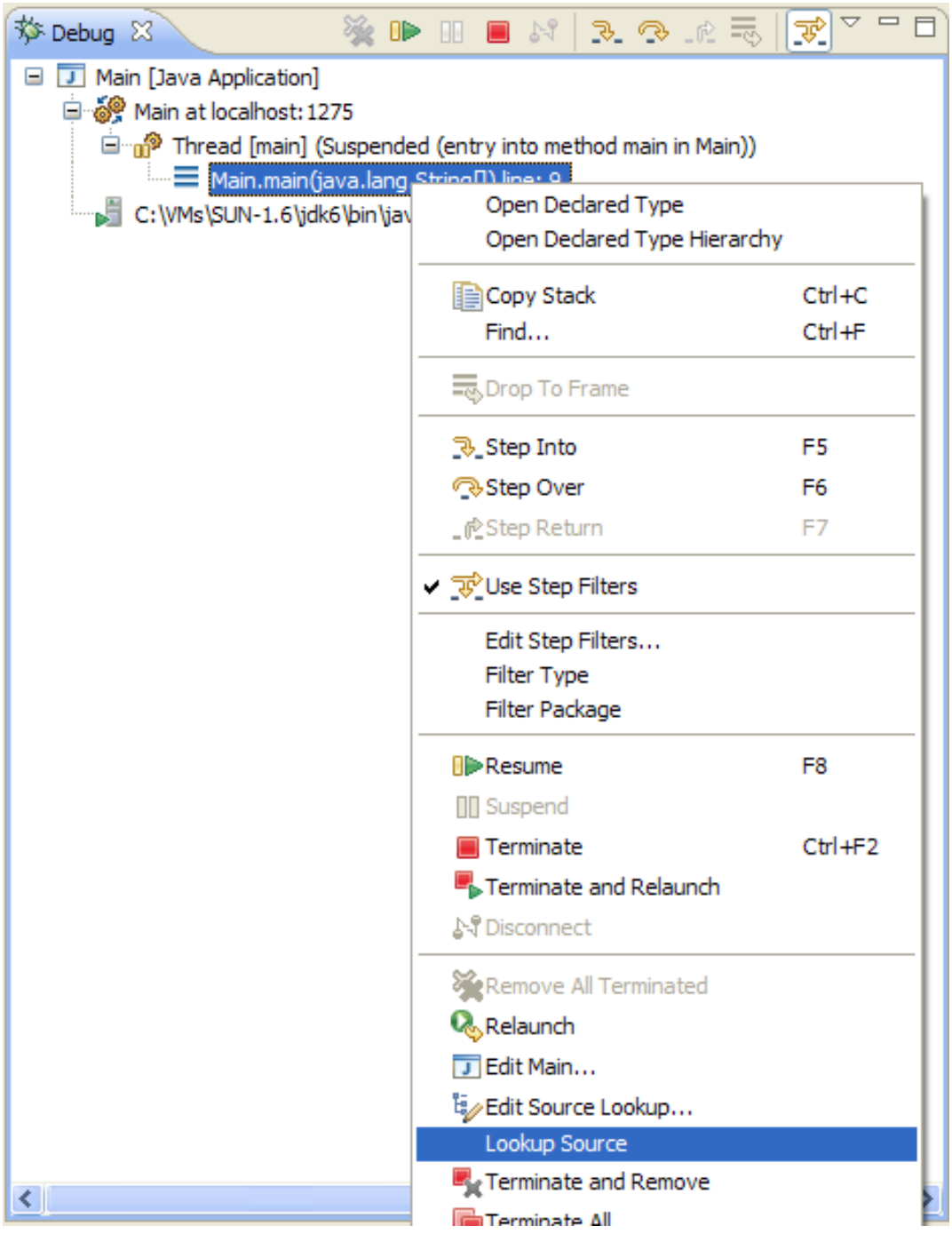
Select the **Find...** command to open the **Find Dialog** which allows you to search for specific elements within the view. You can also use the keyboard shortcut ***Ctrl+F***.
The **Find Dialog**:



This command applies to:
- **Debug View**
- **Expressions View**
- Registers View
- **Variables View**

Lookup Source

Select the **Lookup Source** command to force a source lookup to take place. If the requested source file is not open, it will be opened and the corresponding line of execution will be highlighted.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM

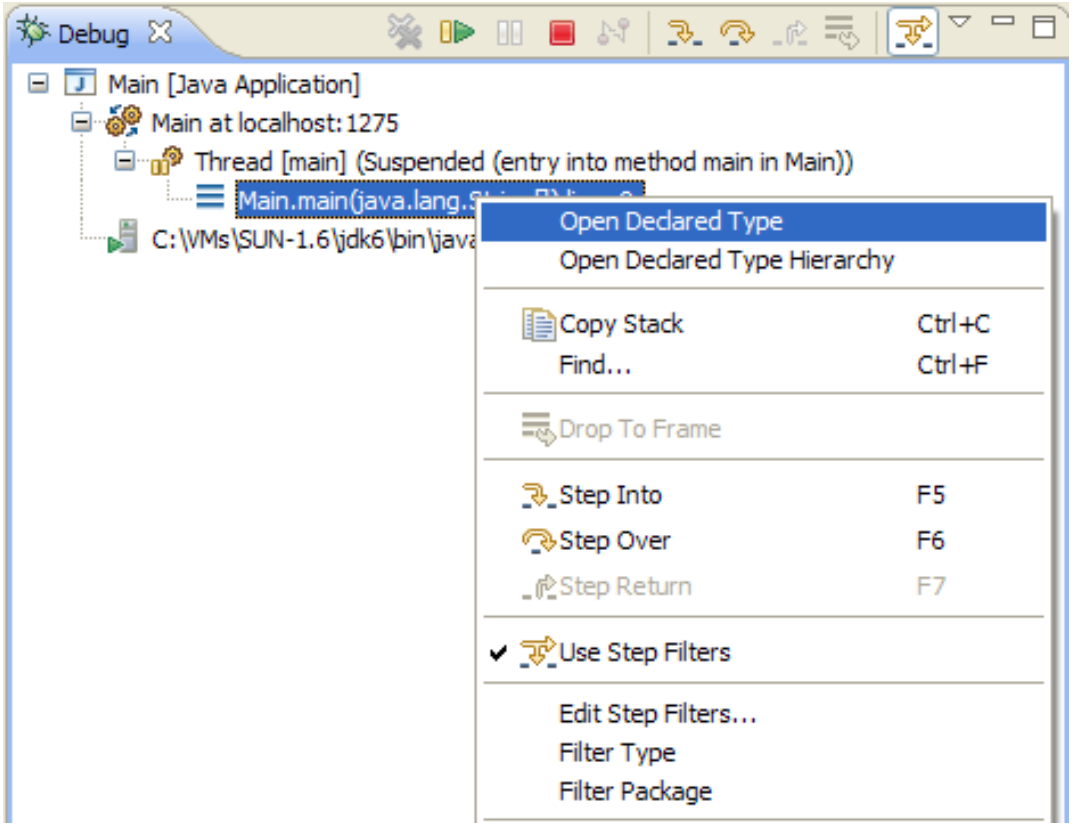
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Edit Source Lookup

Open Declared Type

Select the **Open Declared Type** command to open an editor on the declared type of the currently selected stack frame in the **Debug View**.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

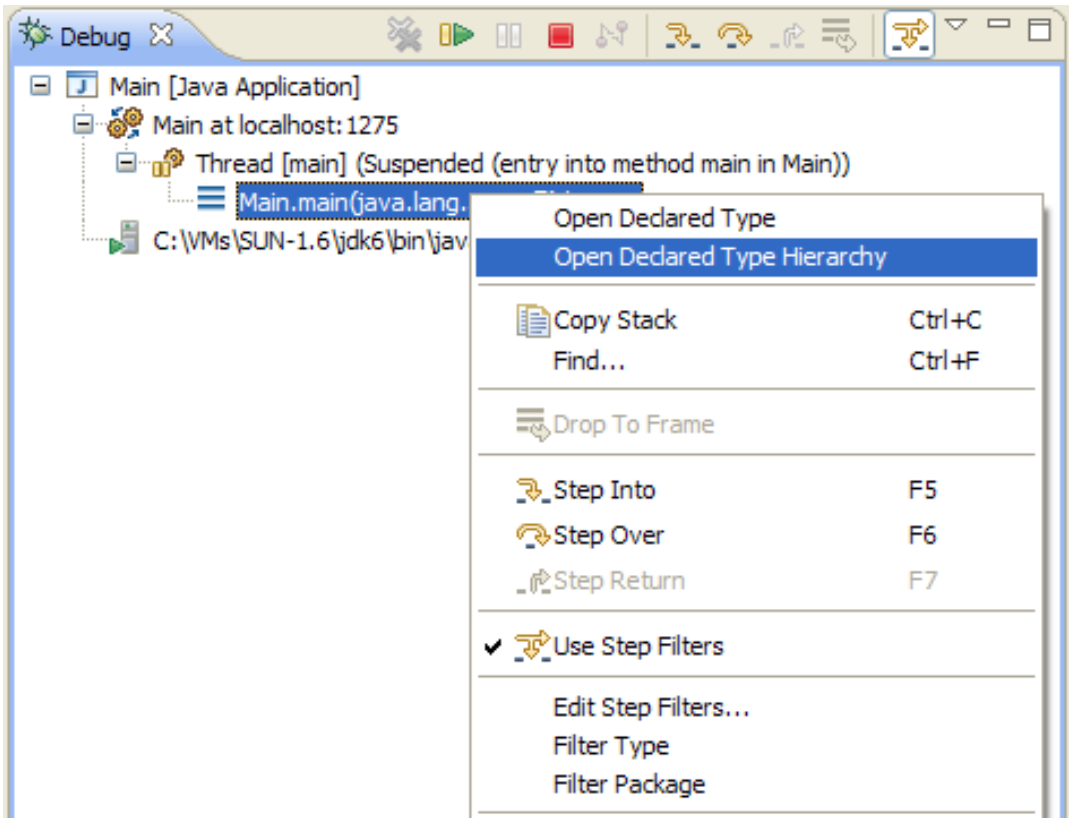
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences

Open Declared Type Hierarchy

Select the **Open Declared Type Hierarchy** command to open the declared type of the currently selected stack frame in the **Type Hierarchy View.**



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

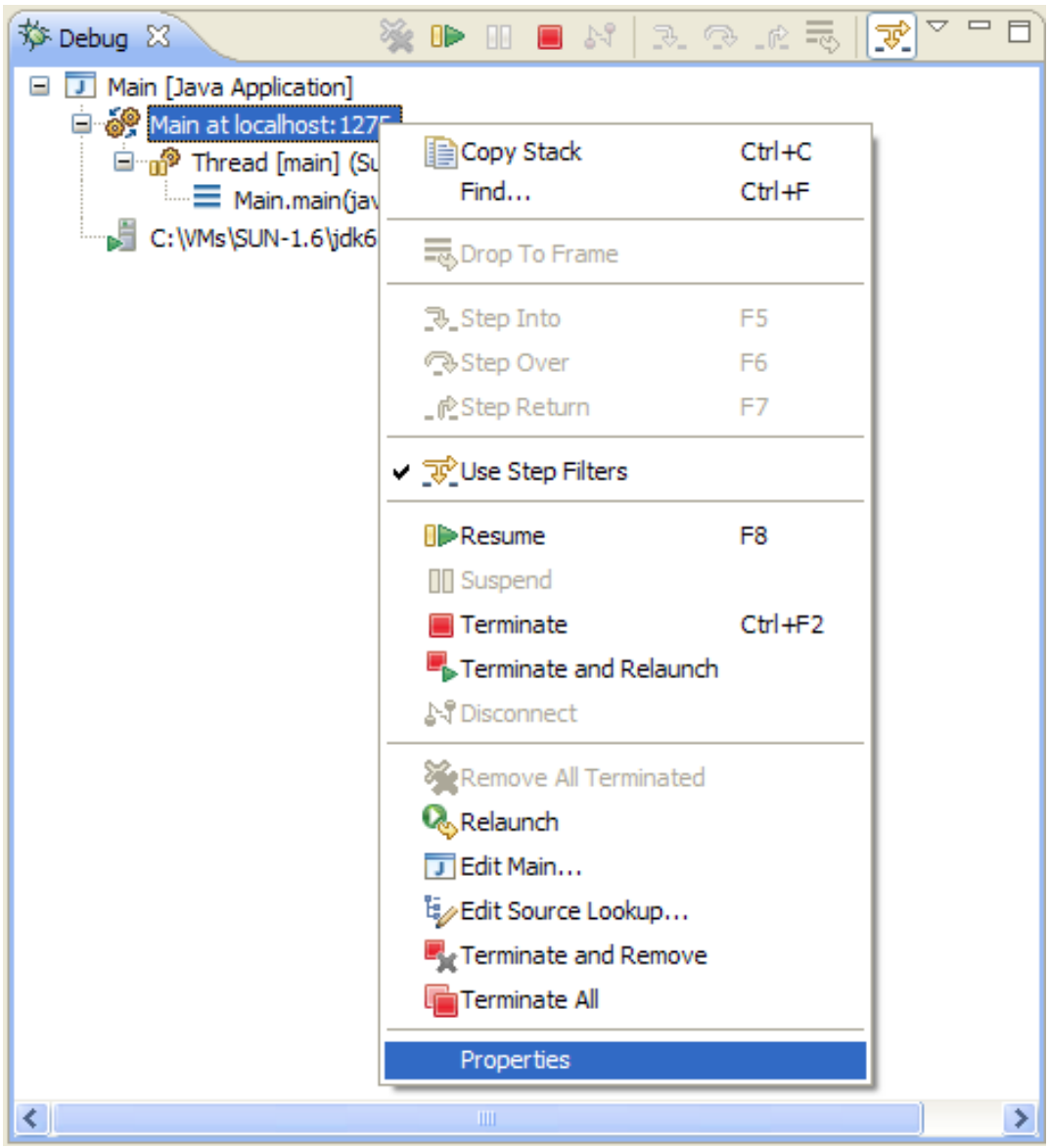
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

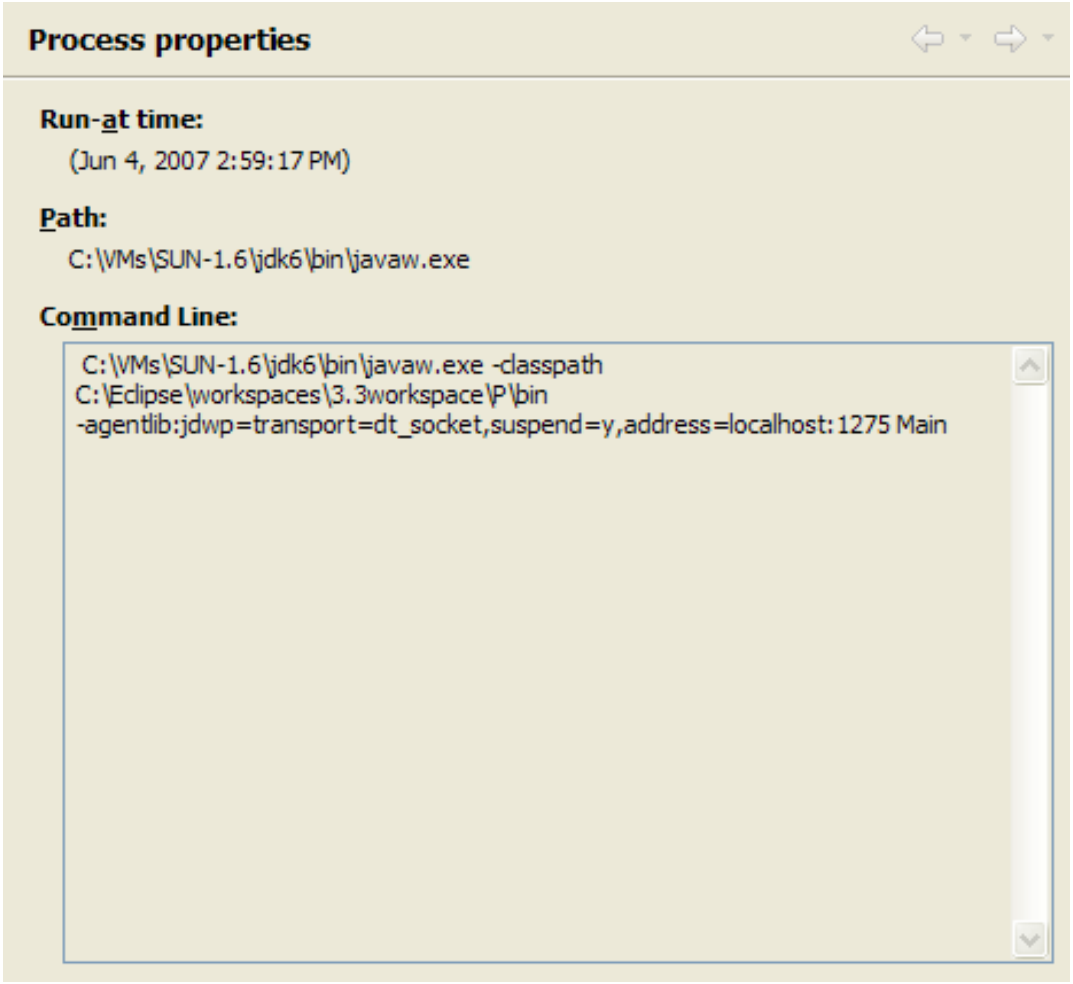
Debug View
Debug preferences

Properties

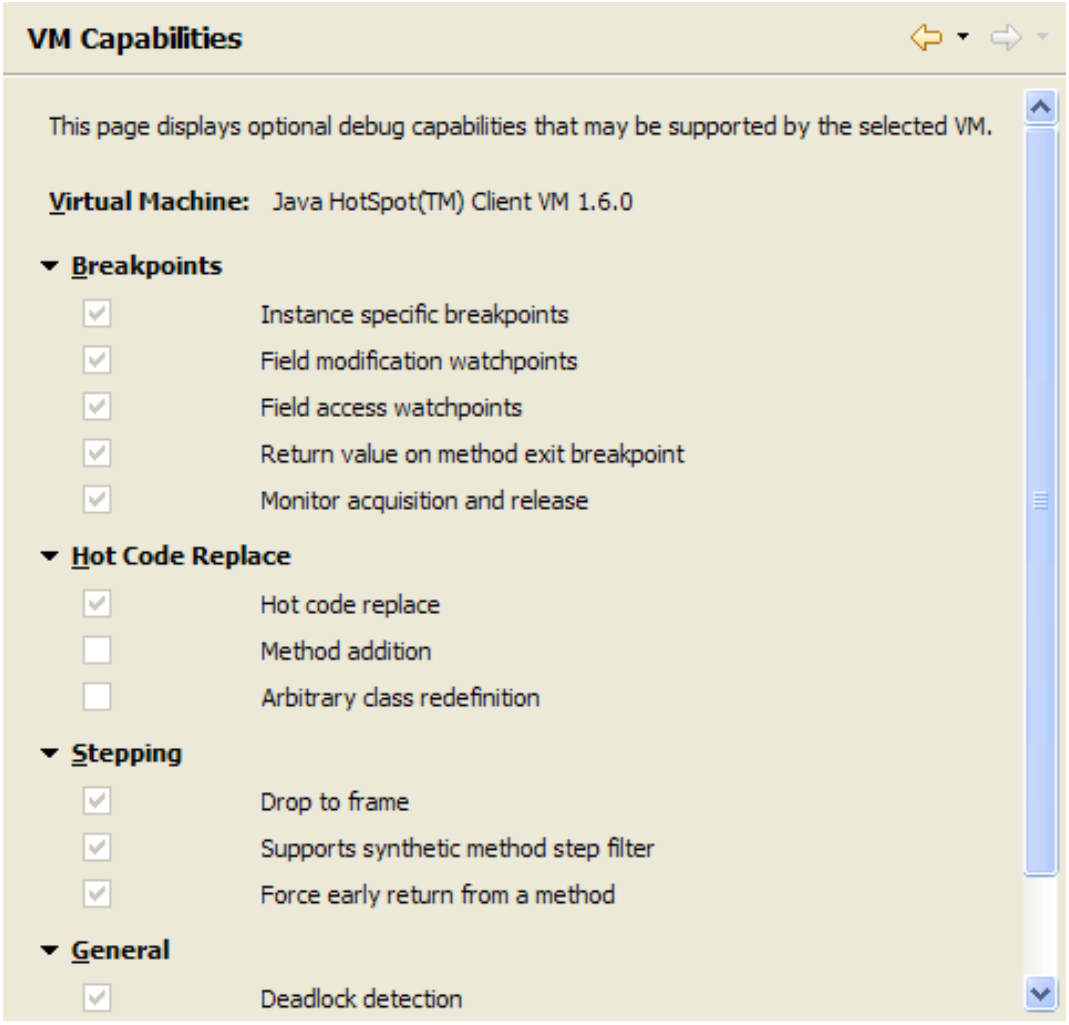
Select the **Properties** command to open the properties dialog for the selected debug target, thread, thread group, process or stackframe.



From the resulting dialog you can copy information about the selected debug target and view the actual command line used to run the target.



You can also view the capabilities of the VM used to launch the associated target



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences

Relaunch

Select the **Relaunch** command [] to relaunch the selected debug target in the **Debug View**.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Remove All Terminated Launches
Run Menu
Terminate
Terminate/Disconnect All
Terminate and Relaunch
Terminate and Remove

Remove All Terminated

Select the **Remove All Terminated** command [] to clear the **Debug View** of all terminated launches.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

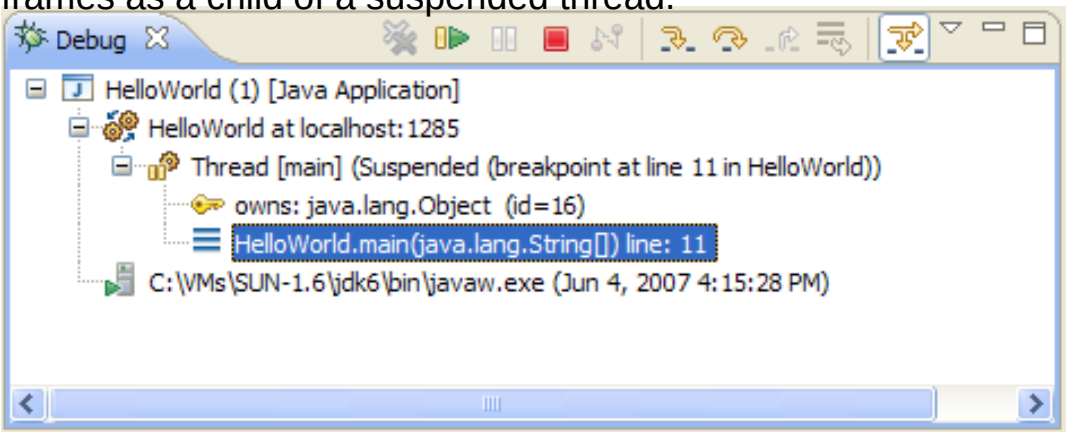
Debug View
Debug preferences
Relaunch
Terminate
Terminate/Disconnect All
Terminate and Relaunch
Terminate and Remove

Show Monitors

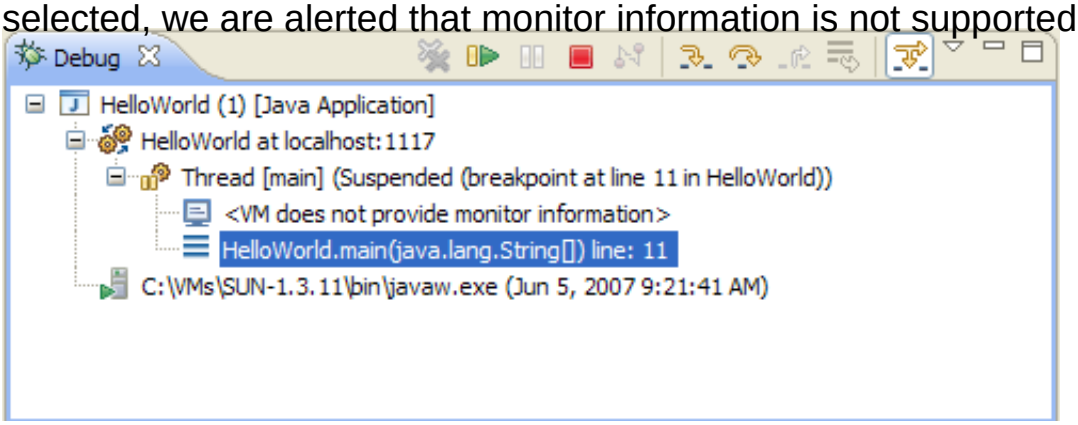
Select the **Show Monitors** command [ ] to change if monitor information will be shown for suspended threads.
Note: The display of monitor information must be supported by the underlying VM. If using an IBM or SUN VM, any version greater than 1.4 will support monitor information.
Consider the following code example:

```
/**
* Hello World
*/
public class HelloWorld {



  public static void main(String[] args) {
  Object mutex = new Object();
  synchronized (mutex) {
   System.out.println("Hello World!");
  }
  }
}
```

If the above code snippet is run on a supporting VM with **Show Monitors** selected, the monitor information is represented as a 'key' and appearing before any stack frames as a child of a suspended thread.



If however, we run the same snippet on an unsupported VM with **Show Monitors** selected, we are alerted that monitor information is not supported by the VM.



Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Show System Threads
Show Thread Groups

Show System Threads

Select the **Show System Threads** command [] to change if system threads should be shown in the **Debug View**.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Show Monitors
Show System Threads
Show Thread Groups

Show Thread Groups

Select the **Show Thread Groups** command [] to change if thread groups should be used to organize threads displayed in the **Debug View**.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Show Monitors
Show System Threads
Show Thread Groups

Use Step Filters

Select the **Use Step Filters** command [] to change whether step filters should be used in the **Debug View**. You can also use the keyboard shortcut ***Shift+F5***.
Step filters are commonly used to filter out types that you do not wish to see or step through while debugging.
For example, if you did not want to the see or step through anything from the class ***java.lang.Object***, you would add this to the list of filtered types. Adding types to the list of those to be filtered can be done in one of two ways:

1. Via the context menu - Right click on the stack frame for the type you wish to filter and use the **Filter Type** or **Filter Package** command.
2. Via the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Step Filtering** preference page.

Related concepts

Debugger
Local debugging
Remote debugging

Related tasks

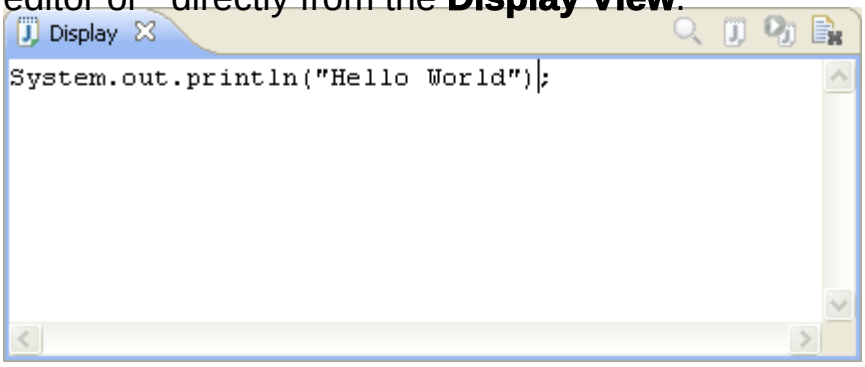
Changing debugger launch options
Connecting to a remote VM with the Remote Java application launch configuration
Disconnecting from a VM
Launching a Java program
Preparing to debug
Resuming the execution of suspended threads
Running and debugging
Stepping through the execution of a program
Suspending threads

Related reference

Debug View
Debug preferences
Execution Control Commands
Edit Step Filters
Run Menu

Display View

The **Display View** displays the result of evaluating an expression in the context of the current stack frame. You can evaluate and display a selection either from the editor or directly from the **Display View**.



The commands available in the display view are listed below.

Display View Commands

Command	Name	Description	Availability
	Clear	Clears the current contents of the view.	Context menu and view action
	Copy	Copies the selected statements to the system clipboard.	Context menu
	Content Assist	Opens the content assist popup for context sensitive coding assistance	Context menu
	Cut	Copies the selected statements to the system clipboard and removes them from the view.	Context menu
	Display	Displays the result of the selected statement inline in the view.	Context menu and view action

	**Execute**	Runs the selected statement. This action is analogous to running the statement in normal code.	Context menu and view action
	Find/Replace	Allows you to search for and replace specific statements, or portions of statements	Context menu
	**Inspect Result**	Allows you to inspect what the return value of the selected statement will be.	Context menu and view action
	**Paste**	Copies material from the system clipboard into the view	Context menu
	Select All	Selects all of the statements in the view	Context menu

Related concepts

Java views
Java perspectives

Related tasks

Evaluating expressions

Related reference

Views and editors

Clear

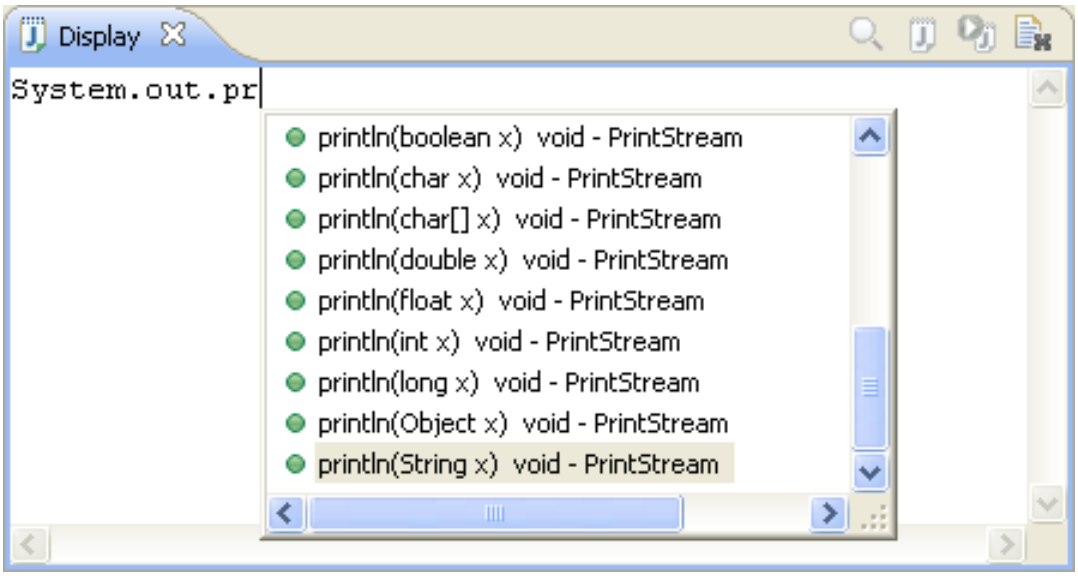
Select the **Clear** command [] to remove all of the contents from the view.

Related reference

Display View

Content Assist

Select the **Content Assist** command [ ] to open the **Content Assist** popup dialog. You can also use the keyboard shortcut ***Ctrl+Spacebar*** within the view. The resulting dialog provides you with context sensitive coding help by making available a listing of all applicable java code elements for the location content assist was activated.



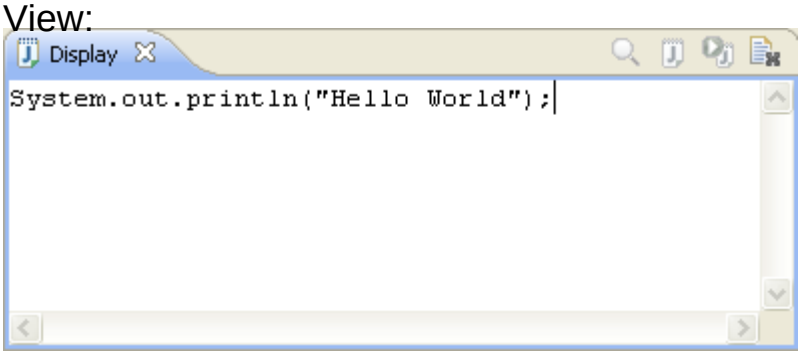
This command applies to:
- **Java Editor**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)

Related tasks

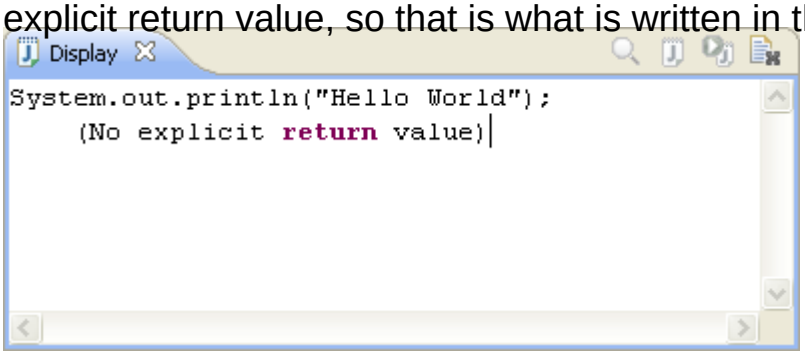
Evaluating Expressions

Display

Select the **Display** command [ ] to have the selected statement evaluated and the results displayed inline in the **Display View** or in a popup dialog (that can be moved to the Display View).
For example, consider we have the following statement selected in the Display View:



After we select the **Display** command we notice that the result is inserted inline in the view. In the following case though, a ***println()*** statement does not have an explicit return value, so that is what is written in the view.



This command applies to:
- **Java Editor**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)
- **Run Menu**

Related tasks

Evaluating Expressions

Execute

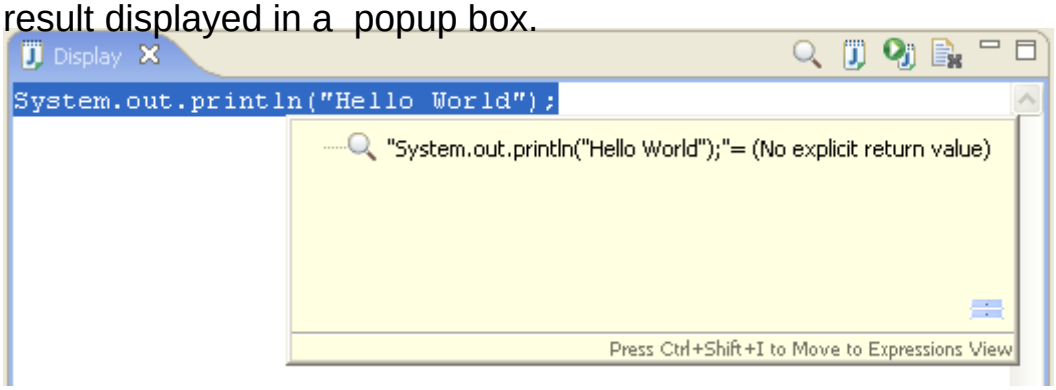
Select the **Execute** command [] to have the selected statement executed as though it was run in normal Java code.
This command applies to:
- **Java Editor**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)
- **Run Menu**

Related tasks

Evaluating Expressions

Inspect Selected Statement

Select the **Inspect** command [ ] to evaluate the selected expression and have its result displayed in a popup box.



You can also move the inspected statement to the **Expressions View** by pressing ***Ctrl+Shift+I*** once the popup has been opened.

This command applies to:

- **Java Editor**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)
- **Run Menu**

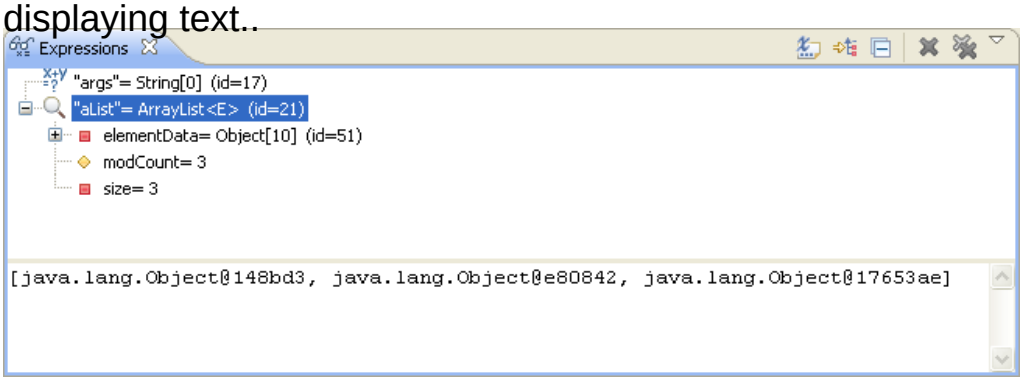
Related tasks

Evaluating Expressions

Expressions View

Data can be inspected in the **Expressions View**. You can inspect data from a scrapbook page, a stack frame of a suspended thread, and other places. The Expressions View opens automatically when an item is added to the view. Entries in the Expressions View can be selected to have more detailed information be displayed in the **Detail Pane**. When debugging a Java program, data that contains variables can be expanded to show the variables and the fields the variables contain.

The Expressions View. The Detail Pane is the area at the bottom of the view displaying text..



There are many commands available in the Expressions View:

- **View Display Commands** affect what data and variables are displayed and how they are presented.
- The **Detail Pane** has many commands available by right clicking on it.
- **View Layout Commands** affect how the detail pane is oriented.
- Other commands are listed below.

Variables View Commands

	**Add Watch Expression**	Allows you to add a watch expression.	Context menu
	**All Instances**	Opens a popup dialog displaying a list of all instances of the selected Java type. Your Java virtual machine must support instance retrieval.	Context menu

	All References	Opens a popup dialog displaying a list of all Java objects that have references to the selected variable. Your Java virtual machine must support reference retrieval.	Context menu
	Change Value...	Allows you to change the value for the underlying selected variable.	Context menu
	Collapse All	Collapses all the currently expanded variables.	View action
	Copy Expressions	Copies the selected expressions and variables to the system clipboard.	Context menu
	Convert to Watch Expression	Converts the selected inspect expression to a corresponding watch expression.	Context menu
	Disable	Disables a currently enabled expression.	Context menu
	Edit Logical Structure	Allow you to edit the logical structure of the selected variable	Context menu
	Edit Watch Expression	Allows you to edit the currently selected expression.	Context menu

	Enable	Enables a currently disabled expression.	Context menu
	Find...	Opens the search dialog to find elements in the variables view.	Context menu
	**Inspect**	Creates a new inspect expression for the selected variable and adds it to the view.	Context menu
	Instance Breakpoints...	Allows you to filter existing breakpoints to the selected variable instance.	Context menu
	Java Preferences...	Opens several preference pages containing options that affect the view.	View action
	New Detail Formatter...	Allows you to create your own detail formatter for that type of variable.	Context menu
	Open Actual Type	Opens the actual type of the selected variable.	Context menu
	Open Actual Type Hierarchy	Opens the actual type hierarchy for the actual type of the selected variable.	Context menu
	Open Declared Type	Opens the declared type for the selected variable in a new editor.	Context menu

	Open Declared Type Hierarchy	Opens the type hierarchy for the declared type of the selected variable.	Context menu
	Reevaluate Expression	Reevaluates the currently selected expression.	Context menu
	**Remove**	Removes the currently selected expression(s) from the view.	Context menu and view action
	**Remove All**	Removes all of the expressions from the view.	Context menu and view action
	Select All	Selects all of the variables in the view.	Context menu
	Show Structure As...	Allows you to select a different formatter for showing the selected logical structure type variable.	Context menu
	Show Details As...	Allows you to select a different detail pane for showing detailed information about selected variables.	Context menu
	**Toggle Watchpoint**	Creates a new watchpoint on the currently selected field or removes the watchpoint if one already exists.	Context menu

Related concepts

Java views
Java perspectives
Scrapbook

Related tasks

Evaluating expressions
Suspending threads
Related reference

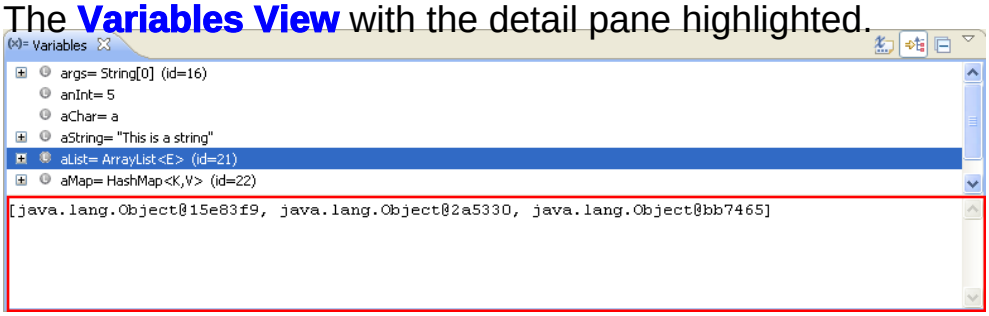
Detail Pane
View Display Commands
View Layout Commands

Detail Pane

The **Detail Pane** displays detailed information about a selection in the following views:

- **Expressions View**
- Registers View
- **Variables View**

The **Variables View** with the detail pane highlighted.



By default, the **Detail Pane** displays information as marked up text (text is highlighted, bolded, underlined, etc. according to tool specific settings). For example, when debugging a Java program, the **Detail Pane** displays `toString()` value of Java objects.

If other types of detail panes are available, they can be accessed by right clicking on a variable and going to the "Show Details As" menu. This menu will not be available unless there is more than one possible type of detail pane for the current selection. The marked up text viewer detail pane that is available by default is the only detail pane provided by the Eclipse platform.

A number of commands are available in the default **Detail Pane**. They are accessed by right-clicking the **Detail Pane** to open up a context menu.

Detail Pane Commands

Command	Name	Description	Availability
	Assign Value	Assigns a new value to the currently selected variable by evaluating the text in the detail pane.	Context menu
	Content Assist	Opens the **Content Assist** popup dialog, offering code completion help.	Context menu
	Cut	Copies the selected material to the system clipboard and removes it from the detail pane.	Context menu

	**Copy**	Copies the selected material from the detail pane onto the system clipboard.	Context menu
	**Display**	Evaluates the selected text and displays the result in a popup dialog.	Context menu
	**Execute**	Executes the selected text as normal Java code.	Context menu
	Find/Replace	Allows you to search for and replace a specified expression.	Context menu
	Force Return	Allows you to immediately return from the method in which execution is suspended.	Context menu
	**Inspect**	Allows you to inspect what the return value of the selected statement will be.	Context menu
	Max Length	Opens a dialog allowing you to change the maximum number of characters displayed in the detail pane.	Context menu
	**Paste**	Pastes material saved on the system clipboard into the detail pane.	Context menu
	Select All	Selects all text in the detail pane.	Context menu

	Wrap Text	Change if the text displayed in the detail pane should wrap at the predefined width or not.	Context menu

Related reference

Expressions View
Variables View

Assign Value

Select the **Assign Value** command to assign a new value to the currently selected variable by evaluating the text in the detail pane.

This command applies to:

- **Detail Pane** (in the **Expressions View** and **Variables View**)

Related tasks

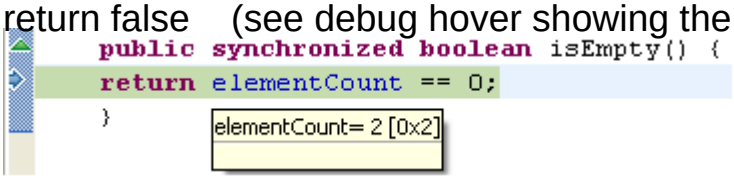
Evaluating Expressions

Force Return

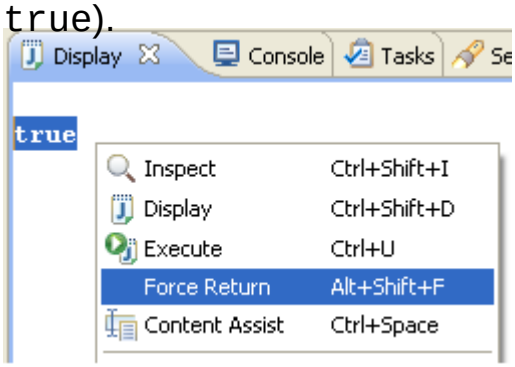
Select the **Force Return** command to return from the current method with the specified value.

You can force an early return from a method (only available when debugging on a Java SE 6 virtual machine). This returns a value from the current stack frame without executing any more instructions in the method and releases any locks obtained by synchronized blocks. A return value is created by selecting an expression and **Force Return** (**Alt+Shift+F**). This action is available from the Java editor's context menu, top level **Run** menu, in the **Display View**, and in the **detail pane** of the **Variables View**.

Forcing an early return from a non-void method requires an expression to be evaluated. For example, if a method was going to return false you could return a value of true by selecting an expression in the **Display View** and invoking **Force Return**. In the following example, `elementCount` is not equal to zero, and would return false (see debug hover showing the value of `elementCount`).



From the **Display View**, we could enter the value we want returned, select it and use the **Force Return** command to force the method `isEmpty()` to return with that value (in the following example we will force `isEmpty()` to return with the value `true`).



This command applies to:

- **Java Editor**
- **Display View**
- **Detail Pane** (in the **Expressions View** and **Variables View**)
- **Run Menu**

Related tasks

Evaluating Expressions

Max Detail Pane Length

The **Max Length...** command to opens a dialog allowing you to change the maximum number of characters displayed in the **Detail Pane**

This command applies to:

- **Detail Pane** (in the **Expressions View** and **Variables View**)

Wrap Text

Select the **Wrap Text** command to change if the text displayed in the **Detail Pane** should wrap at the predefined width or not.
This command applies to:
- **Detail Pane** (in the **Expressions View** and **Variables View**)

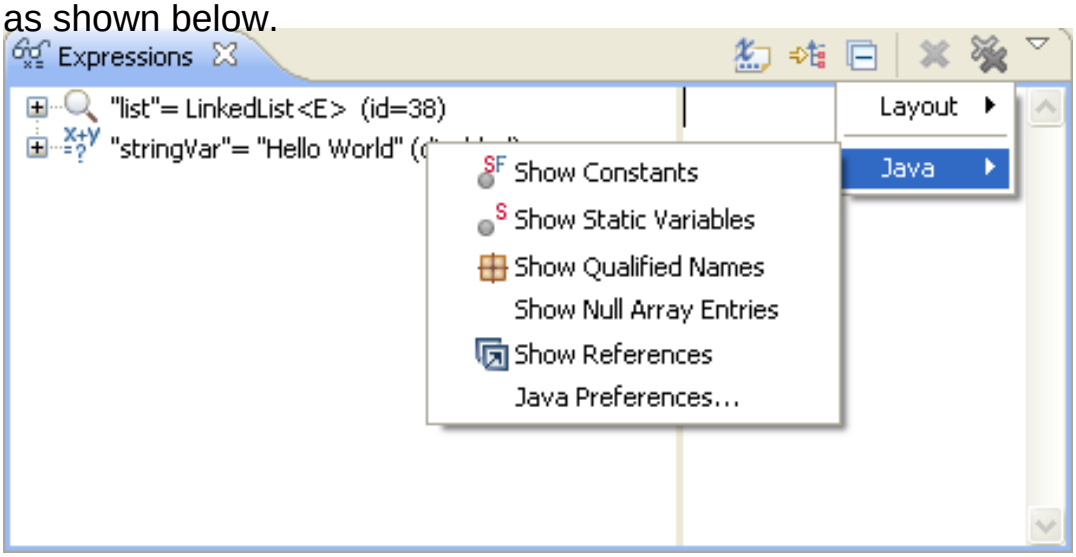
Variable Display Commands

View display commands affect what variables are displayed in the view and how they are presented.

These commands apply to (some commands apply to additional views):

- **Expressions View**
- **Variables View**

The **Show Type Names** and **Show Logical Structures** commands are available on the view toolbar. The other actions can be found in the drop down menu of the view as shown below.



View Display Commands

Command	Name	Description	Availability
	Show Constants	Changes whether constants should be shown in the view.	View action
	Show Logical Structures	Changes whether logical structures should be shown in the view.	View action
	Show Null Array Entries	Change whether null array entries should be shown in the view.	View action
	Show Qualified Names	Changes whether qualified names should be shown in the view.	View action

	**Show References**	Changes whether references to variables should be displayed in the view. Your Java virtual machine must support reference retrieval.	View action
	**Show Static Fields**	Changes whether static fields should be shown in the view.	View action
	**Show Type Names**	Changes whether type names should be shown in the view.	View action

Related reference

Expressions View
Variables View
Detail Pane

Show Constants

Select the **Show Constants** command [] to change if constants should be shown in the view or not.

This command applies to:

- **Expressions View**
- **Variables View**

Related reference

View Display Commands

Show Logical Structures

Select the **Show Logical Structures** command [] to change if logical structures should be shown in the view or not.

This command applies to:

- **Expressions View**
- Registers View
- **Variables View**

Related reference

View Display Commands

Show Null Array Entries

Select the **Show Null Array Entries** command to change if null array entries should be shown in the view or not.

This command applies to:

- **Expressions View**
- **Variables View**

Related reference

View Display Commands

Show References

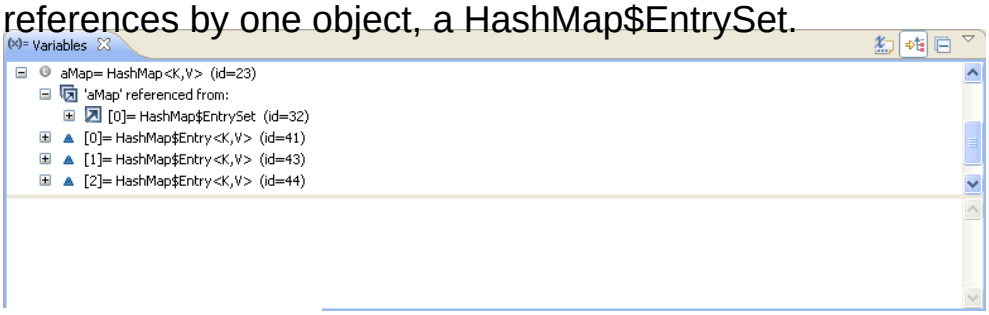
Select the **Show References** command [ ] to change if references should be displayed as variables in the view. When turned on, each Java variable that supports references will have a new child item with the name "'<variable name>' referenced from:". This new variable contains a list of all Java objects that hold a reference to the parent object. You can get a text list in the detail pane by clicking on the reference list variable or explore the details of each referencing object by expanding the variables.

References will only appear in the view if the Java Virtual Machine you are using supports reference retrieval.

This command applies to:

- **Expressions View**
- **Variables View**

The Variables View with Show References turned on. In this case, the Map is references by one object, a HashMap$EntrySet.



Related reference

View Display Commands

Show Static Fields

Select the **Show Static Fields** command [] to change if static fields should be shown in the view or not.
This command applies to:
- **Expressions View**
- **Variables View**

Related reference

View Display Commands

Show Type Names

Select the **Show Type Names** command [] to change if type names should be shown in the view or not.

This command applies to:

- **Expressions View**
- Registers View
- **Variables View**

Related reference

View Display Commands

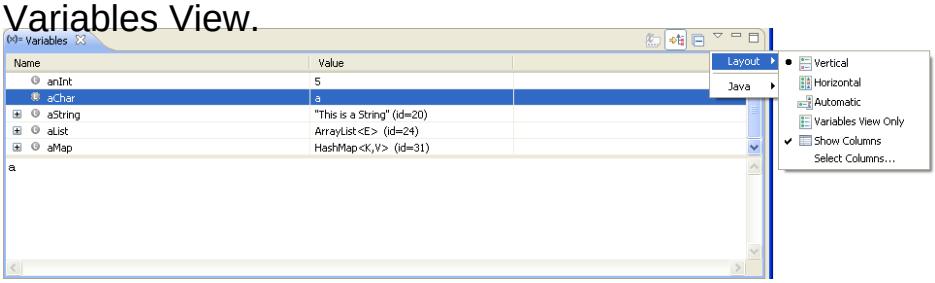
View Layout Commands

View layout commands affect the orientation of the **Detail Pane** in the view. In addition, the **Variables View** has commands that allow you to change column settings.

These commands apply to:

- **Expressions View**
- Registers View
- **Variables View**

The commands are available under the Layout menu, which is shown below in the Variables View.



Detail Pane Display Commands

Command	Name	Description	Availability
	Show Columns	Changes if columns should be shown in the view or not. Variables View only.	View action
	Select Columns...	Allows you to select which columns are shown when columns are visible. Variables View only.	View action
	Horizontal	Displays the detail pane at the right side of the view, aligning the parts of the view horizontally.	View action
	Vertical	Displays the detail pane at the bottom of the view, aligning the parts of the view vertically.	View action

	**View Only**	Hides the detail pane.	View action

Related reference

Expressions View
Variables View

Show Columns

Select the **Show Columns** command [] to change if columns should be used to display variable information in the**Variables View**. If Show Columns is turned on, the Select Columns command will be available, allowing you to choose which columns are visible.

This command applies to:

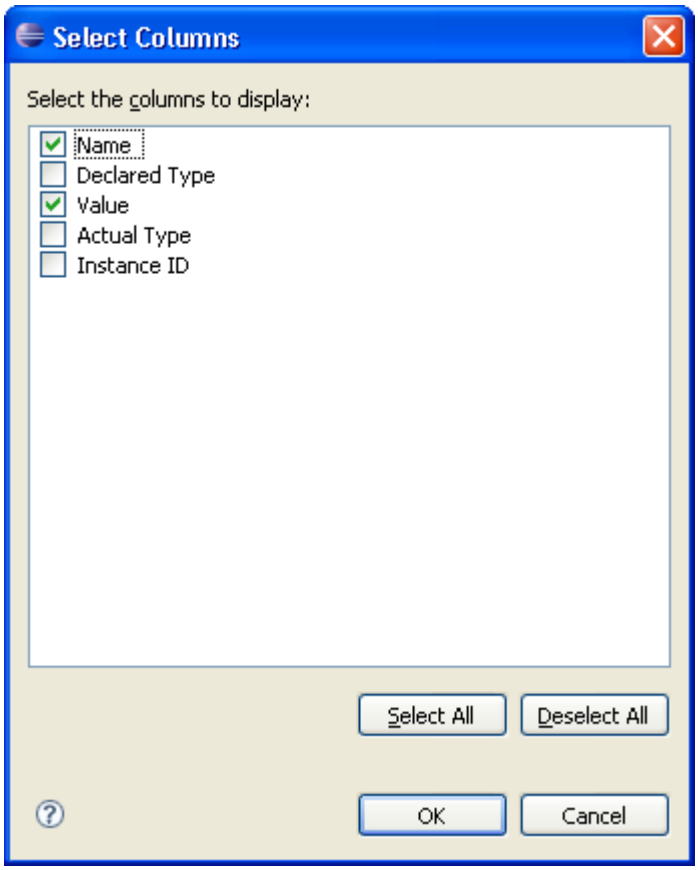
- **Variables View**

Related reference

View Layout Commands

Select Columns

Select the **Select Columns...** command to open a dialog that will allow you to choose which columns to display in the **Variables View**.
In the resulting dialog you can select which columns will be displayed.



This command applies to:
- **Variables View**

Related reference

View Layout Commands

Horizontal Layout

Select the **Horizontal** command [] to reveal the **Detail Pane** and set it in a horizontal alignment within the view.

This command applies to:

- **Expressions View**
- Memory View
- Registers View
- **Variables View**

Related reference

View Layout Commands
Detail Pane
Expressions View
Variables View

Vertical Layout

Select the **Vertical** command [] to reveal the **Detail Pane** and set it in a vertical alignment within the view.

This command applies to:

- **Expressions View**
- Memory View
- Registers View
- **Variables View**

Related reference

View Layout Commands
Detail Pane
Expressions View
Variables View

View Only

Select the **View Only** command [] to hide the **Detail Pane** within the view. The name of the command will change based on the current view. For example, in the **Variables View**, the command is named 'Variables View Only'.

This command applies to:

- **Expressions View**
- Registers View
- **Variables View**

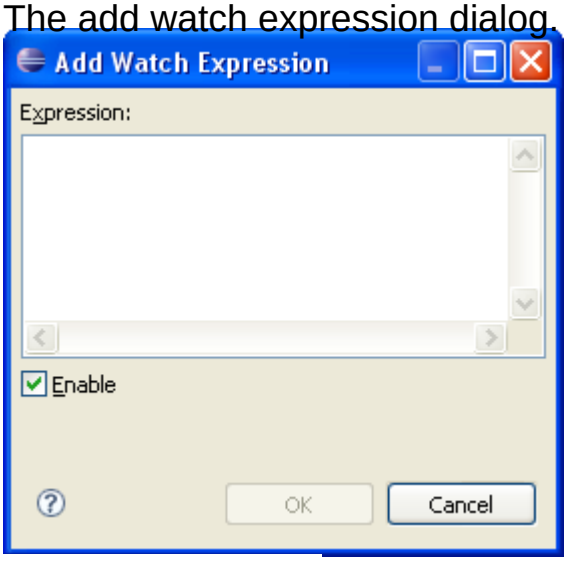
Related reference

View Layout Commands
Detail Pane
Expressions View
Variables View

Add Watch Expression

Select the **Add Watch Expression** command [ ] to open the create new expression dialog, which allows you to create a new watch expression and add it to the **Expressions View**.

The add watch expression dialog.



Related tasks

Evaluating expressions

Related reference

Expressions View

All Instances

Select the **All Instances** command [] to open a popup containing a list of all Java object instances of the selected type that exist in the current debug target. The command has a keyboard shortcut of ***Ctrl+Shift+N***. This command is only available if the Java virtual machine you are currently using supports instance retrieval.

The type to search for can be selected in several ways:

- In the **Java Editor**, highlight or place the cursor in the name of a type or a constructor for a type.
- In the **Outline View**, select a type or constructor.
- In the **Expressions View** or **Variables View**, select a Java object variable.

Once the popup has opened, you can move the list to the **Expressions View** by pressing ***Ctrl+Shift+I***.

There is a maximum number of instances the virtual machine will return. This maximum can be changed on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Heap Walking** preference page.

This command applies to:

- **Java Editor**
- **Outline View**
- **Expressions View**
- **Variables View**

Related reference

Evaluating Expressions
Run Menu

All References

Select the **All References...** command [ ] to open a popup containing a list of all Java objects in the current debug target that have references to the selected variable. The selected variable in the view must be a Java object. This command is only available if the Java virtual machine you are currently using supports reference retrieval.

Once the popup has opened, you can move the list to the **Expressions View** by pressing ***Ctrl+Shift+I***.

There is a maximum number of references the virtual machine will return. This maximum can be changed on the [Image: /topic/org.eclipse.help/command_link.png] **Java > Debug > Heap Walking** preference page.

This command applies to:

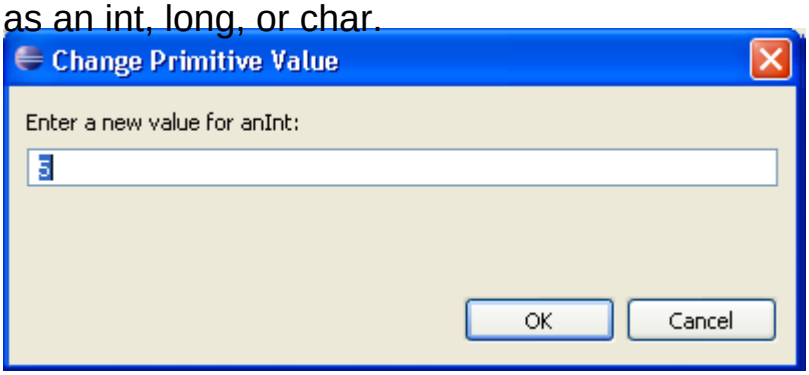
- **Expressions View**
- **Variables View**

Related reference

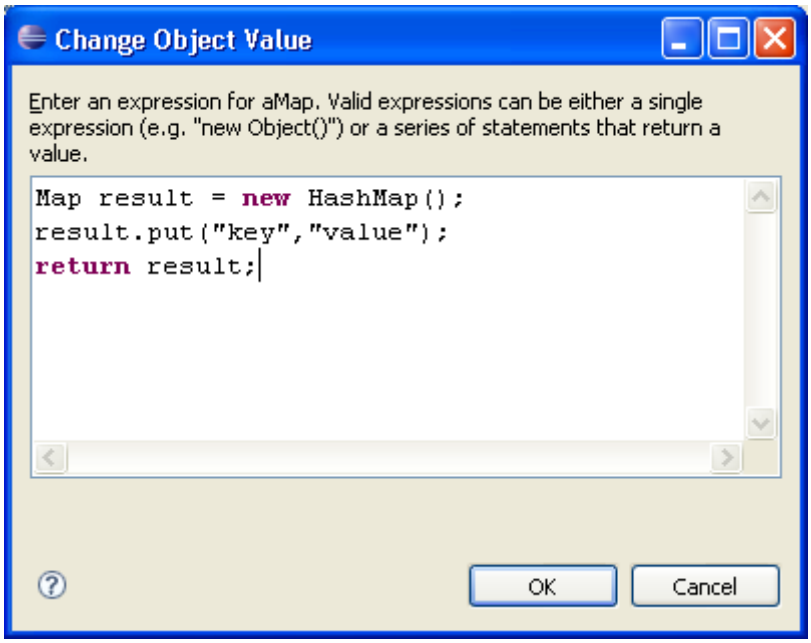
Evaluating Expressions
Run Menu

Change Variable Value

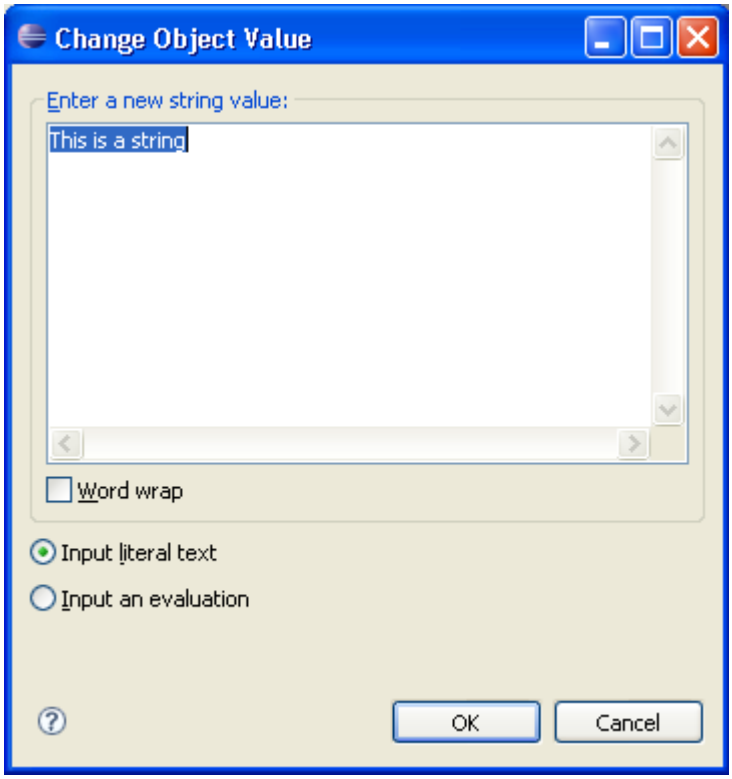
Select the **Change Value...** command [ ] to open a dialog in which you can change the value of the selected variable or register.
In the **Variables View**, if variables are displayed in the column form, variable values can also can be changed by directly editing the value in the column.
When debugging a Java program, there are three different types of dialogs that this command can open.
The **Change Primitive Value Dialog** is used to change the value of a primitive such as an int, long, or char.



The **Change Object Value Dialog** is used to change the value of an Object. There are two forms of this dialog.
For most objects you must enter an expression that will be evaluated to generate the resulting value.



However, when editing Java Strings, you may also enter literal text.



This command applies to:
- **Expressions View**
- Registers View
- **Variables View**

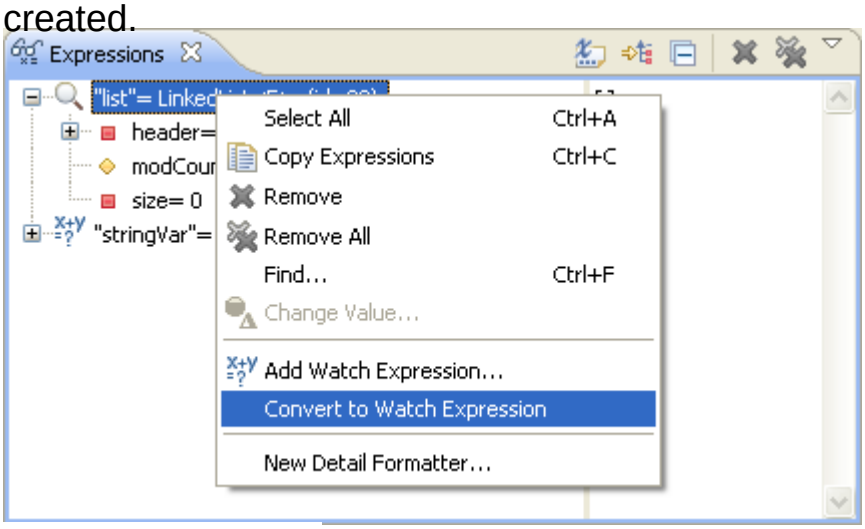
Related tasks

Evaluating Expressions

Convert to Watch Expression

Select the **Convert to Watch Expression** command to convert the currently selected inspect expression to a watch expression.

Note: the inspect expression is removed from the view once the watch expression is created.

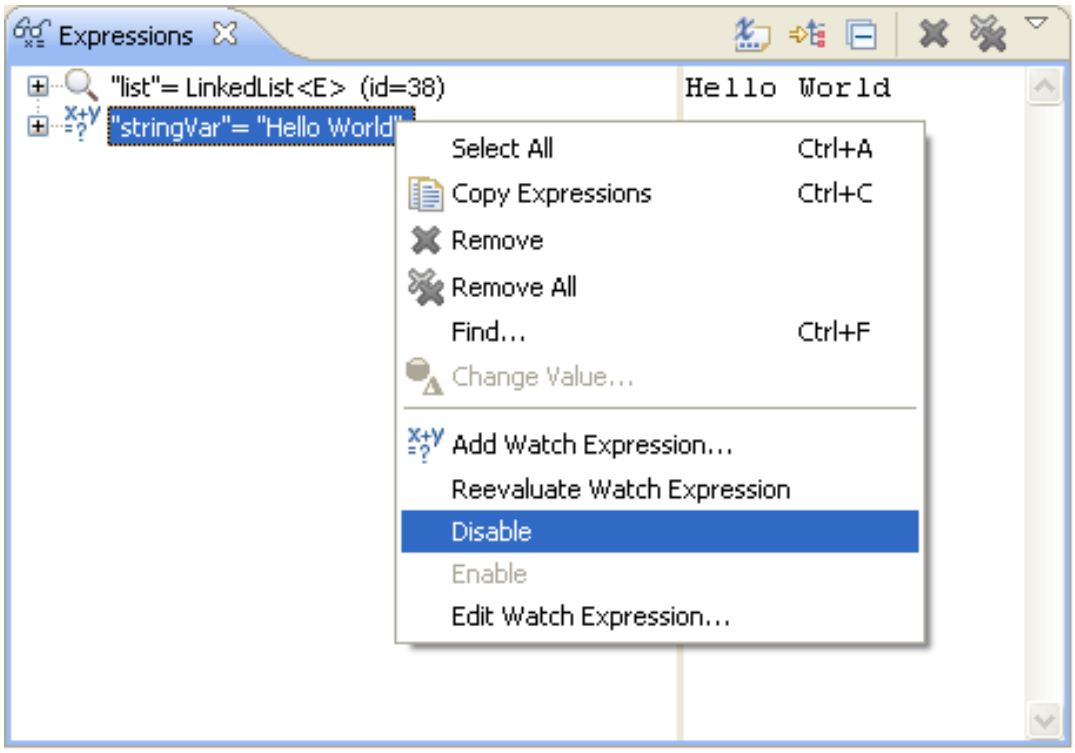


Related reference

Expressions View

Disable Watch Expression

Select the **Disable** command to disable the selected watch expression.

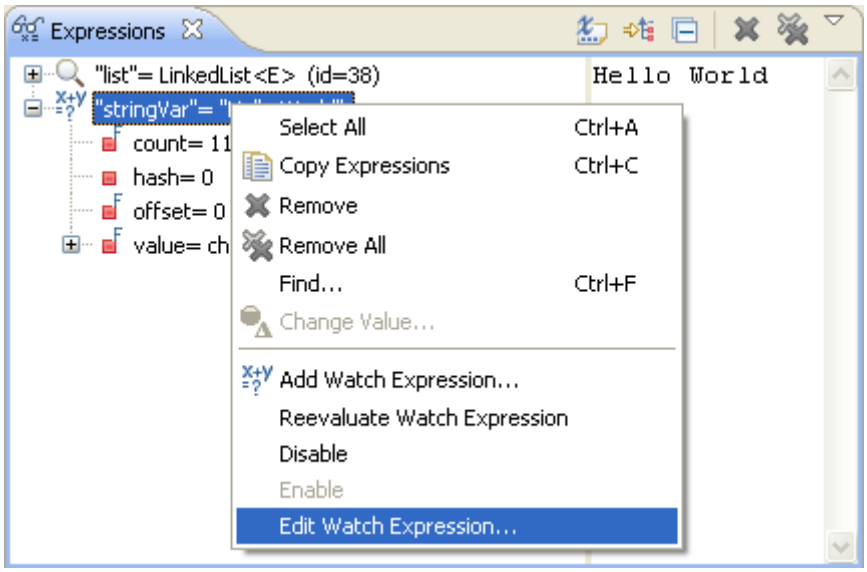


Related reference

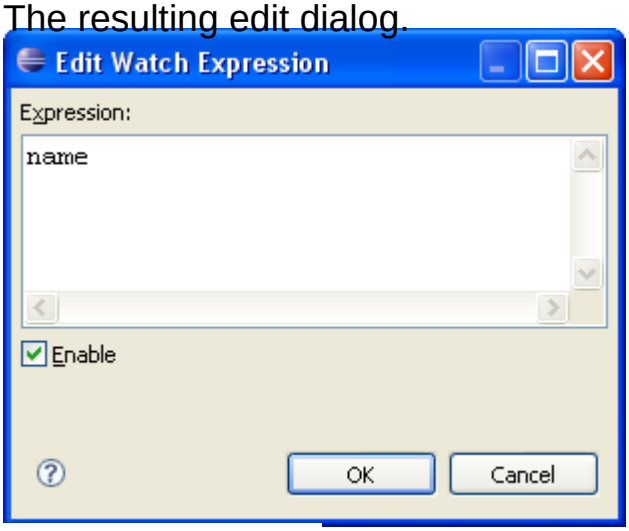
Expressions View
Enable Watch Expression

Edit Watch Expression...

Select the **Edit Watch Expression ...** command to open the editing dialog for the selected watch expression.
The edit watch expression command.



The resulting edit dialog.

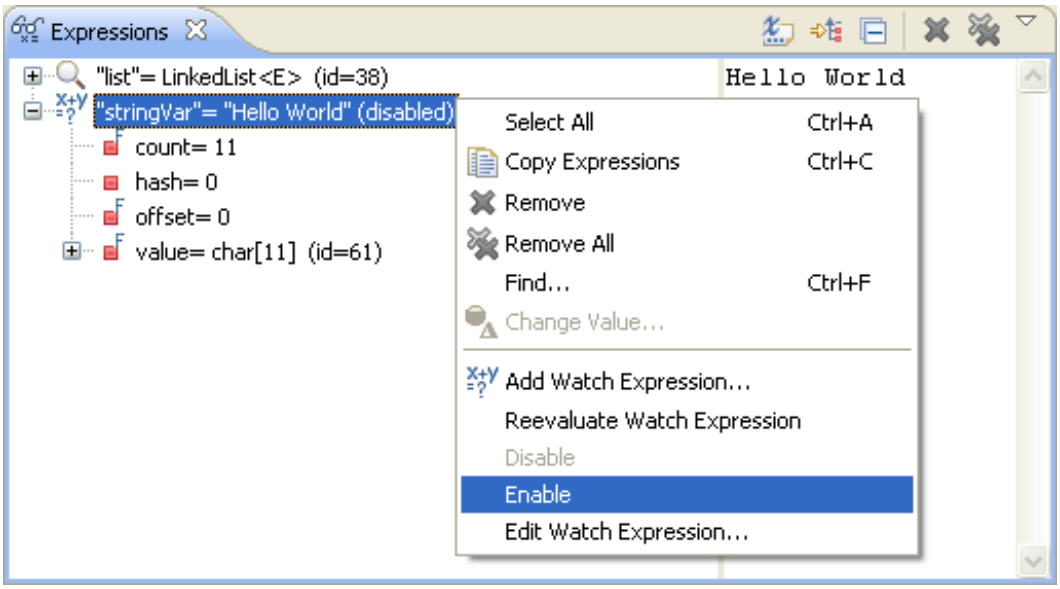


Related reference

Expressions View

Enable Watch Expression

Select the **Enable** command to enable the selected watch expression. The expression must be disabled for this command to be available.



Related reference

Expressions View
Disable Watch Expression

Inspect Selected Variable

Select the **Inspect** command [] to create an inspect expression associated with the selected variable and add it to the **Expressions View**.

This command applies to:

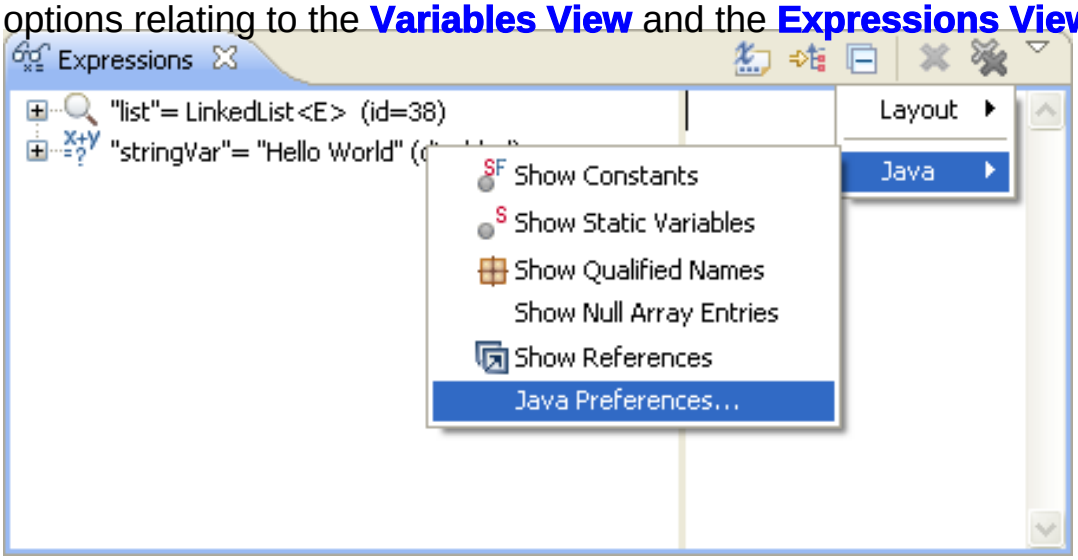
- **Expressions View**
- **Variables View**

Related tasks

Evaluating Expressions

View Java Preferences

Select the **Java Preferences...** command to open the preference pages containing options relating to the **Variables View** and the **Expressions View**.



There are four pages in total that are shown, all as entries in the tree on the left hand side of the **Preferences Dialog**.

These pages are:

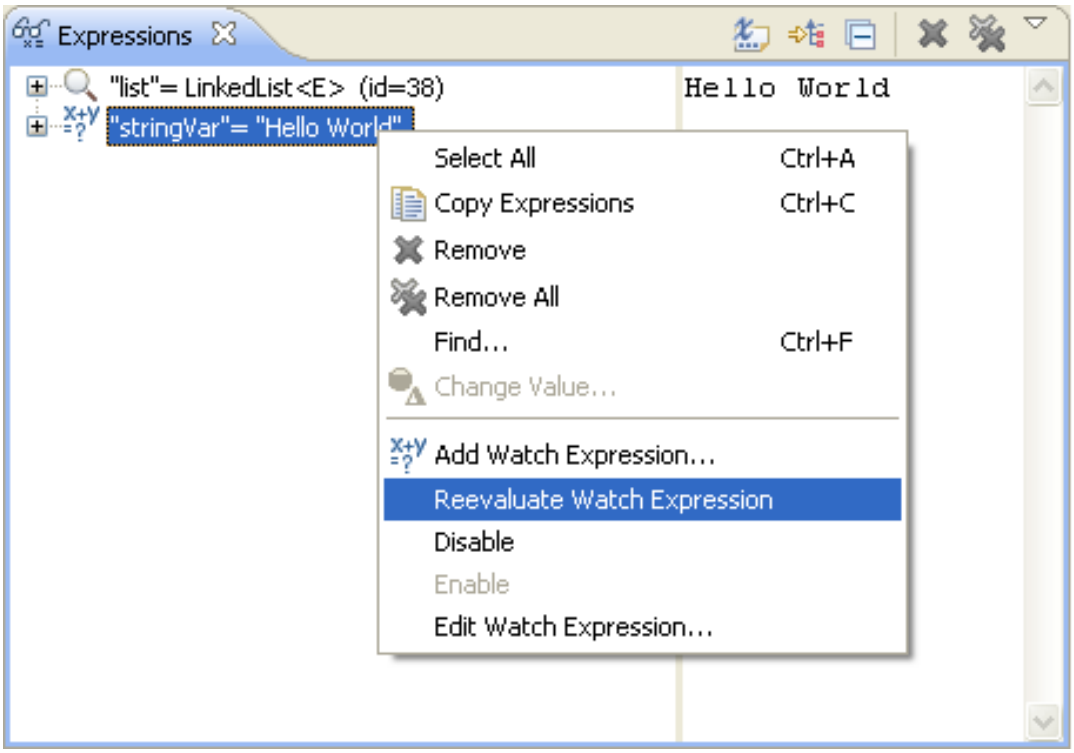
- The [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Detail Formatters** preference page.
- The [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Heap Walking** preference page.
- The [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Logical Structures** preference page.
- The [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Primitive Display Options** preference page.

This command applies to:

- **Expressions View**
- **Variables View**

Reevaluate Watch Expression

Select the **Reevaluate Watch Expression** command to force the selected watch expression to be evaluated.



Related reference

Expressions View

Remove Selected Expressions

Select the **Remove** command [] to remove the selected watch expression(s) from the **Expressions View**.

You can also perform this same action by selecting the expression(s) that you wish to remove and pressing the **Delete** key.

Related reference

Expressions View

Remove All Expressions

Select the **Remove All Expressions** command [] the remove all of the expressions from the **Expressions View**.

Related reference

Expressions View

Toggle Field Watchpoint

Select the **Toggle Watchpoint** command [] to create a new watchpoint on the currently selected field. Whenever that field is accessed or modified, execution will be suspended. If the selected field already has a watchpoint, selecting this command will remove it. You must select a Java field object to use this command. This command applies to:

- **Expressions View**
- **Variables View**

Related concepts

Breakpoints

Related tasks

Adding breakpoints

Package Explorer View

The Package Explorer view, shown by default in the Java perspective, shows the Java element hierarchy of the Java projects in your workbench. It provides you with a Java-specific view of the resources shown in the Navigator. The element hierarchy is derived from the project's build paths.

For each project, its source folders and referenced libraries are shown in the tree. You can open and browse the contents of both internal and external JAR files. Opening a Java element inside a JAR opens the CLASS file editor, and if there is a source attachment for the JAR file, its corresponding source is shown.

Toolbar buttons

Command	Name	Description
	Back	Navigates to the most recently-displayed state of the view with a different element at the top level.
	Forward	Navigates to the state of the view with a different element at the top level that was displayed immediately after the current state.
	Up	Navigates to the parent container of the package that is currently displayed at the top level in the view.
	Collapse All	Collapses all tree nodes.
	Link with Editor	Links the package explorer's selection to the active editor.

	Top Level Elements > Projects	Shows project as top level elements in the Package Explorer. Only projects which are part of the currently active working sets are shown. Following actions can be used to configure the active working sets: **Select Working Set:** Opens the **Select Working Set** dialog to allow selecting the working set from which elements should be shown.**Deselect Working Set:** Deselect the active working sets. All elements are shown after invoking this action**Edit Active Working Set:** Opens the **Edit Working Set** wizard to edit the currently active working set
	Top Level Elements > Working Sets	Shows Working Sets as top level elements in the Package Explorer. The **Configure Working Sets...** action can be used to open a dialog which allows to configure the visible working sets.
	Filters...	Opens the **Java element filters** dialog.

Related concepts

Java views
Java perspectives
Working sets

Related reference

Java element filters dialog
Views and editors

Java Element Filters Dialog

This dialog lets you define Java element filters for the Package Explorer view and other Java views.

Option	Description	Default
Name filter patterns	If enabled, a comma separated list of patterns can be specified additionally.	Off
Select the elements to exclude from the view	List of pre-defined filters which can be enabled.	.* resources, Empty library containers Empty parent packages, Import declarations, Inner class files, Package declarations, Synthetic members

The **Filter description** field displays the description for the currently selected filter.

Related concepts

Filtering elements

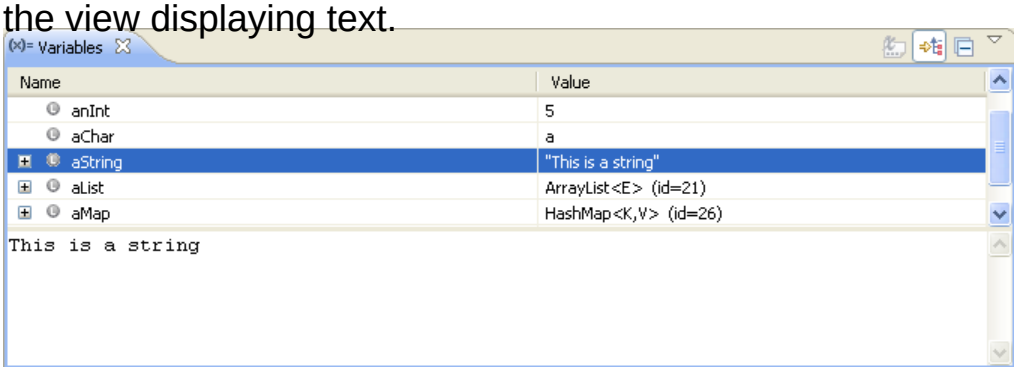
Related reference

Package Explorer view

Variables View

The **Variables View** displays information about the variables associated with the stack frame selected in the **Debug View**. When debugging a Java program, variables can be selected to have more detailed information be displayed in the **Detail Pane**. In addition, Java objects can be expanded to show the fields that variable contains.

The Variables View, shown with columns. The Detail Pane the area at the bottom of the view displaying text.



There are many commands available in the Variables View:

- **View Display Commands** affect what variables are displayed and how they are presented.
- The **Detail Pane** has many commands available by right clicking on it.
- **View Layout Commands** affect how the detail pane is oriented and whether columns are displayed.
- Other commands are listed below.

Variables View Commands

Command	Name	Description	Availability
	All Instances	Opens a popup dialog displaying a list of all instances of the selected Java type. Your Java virtual machine must support instance retrieval.	Context menu

	All References	Opens a popup dialog displaying a list of all Java objects that have references to the selected variable. Your Java virtual machine must support reference retrieval.	Context menu
	**Change Value...**	Allows you to change the value for the underlying selected variable.	Context menu
	**Collapse All**	Collapses all the currently expanded variables.	View action
	**Copy Variables**	Copies the selected variables to the system clipboard.	Context menu
	**Create Watch Expression**	Allows you to create a watch expression for the selected variable.	Context menu
	Edit Logical Structure	Allow you to edit the logical structure of the selected variable	Context menu
	Find...	Opens the search dialog to find elements in the variables view.	Context menu
	**Inspect**	Creates a new inspect statement for the selected variable and adds it to the expressions view.	Context menu

	Instance Breakpoints...	Allows you to filter existing breakpoints to the selected variable instance.	Context menu
	Java Preferences...	Opens several preference pages containing options that affect the view.	View action
	New Detail Formatter...	Allows you to create your own detail formatter for that type of variable.	Context menu
	Open Actual Type	Opens the actual type of the selected variable.	Context menu
	Open Actual Type Hierarchy	Opens the actual type hierarchy for the actual type of the selected variable.	Context menu
	Open Declared Type	Opens the declared type for the selected variable in a new editor.	Context menu
	Open Declared Type Hierarchy	Opens the type hierarchy for the declared type of the selected variable.	Context menu
	Select All	Selects all of the variables in the view.	Context menu
	Show Structure As...	Allows you to select a different formatter for showing the selected logical structure type variable.	Context menu

	Show Details As...	Allows you to select a different detail pane for showing detailed information about selected variables.	Context menu
	**Toggle Watchpoint**	Creates a new watchpoint on the currently selected field or removes the watchpoint if one already exists.	Context menu

Related concepts

Java views
Java perspectives

Related tasks

Suspending threads
Evaluating expressions

Related reference

Detail Pane
View Display Commands
View Layout Commands
Expressions View

Create Watch Expression

Select the **Create Watch Expression** command [] to create a new expression based on the selected variable and add it to the **Expressions View**.

- Related tasks

Evaluating expressions

- Related reference

Expressions View
Variables View

Java outline

This view displays an outline of the structure of the currently-active Java file in the editor area.

Toolbar buttons

Command	Name	Description
	Go into Top Level Type	Makes the top level type of the compilation unit the new input for this view. Package declaration and import statements are no longer shown.
	Collapse All	Collapse all top level types.
	Sort	This option can be toggled to either sort the outline elements in alphabetical order or sequential order, as defined on the **Java > Appearance > Member Sort Order** preference page.
	Hide Fields	Shows or hides the fields.
	Hide Static Fields and Methods	Shows or hides the static fields and methods.
	Hide Non-Public Members	Shows or hides the non-public fields and methods.
	Hide Local Types	Shows or hides the local types.

Related concepts

Java editor
Java views
Filtering in Java views
Java element decorations

Sorting elements in Java views
Presentation options for Java views

- Related tasks

Restoring a deleted workbench element
Setting method breakpoints

- Related reference

Override methods
Views and editors

Java Scrapbook Page

The scrapbook allows Java expressions to be run, inspected, and displayed, under the control of the debugger.

Note: Content assist (such as code assist) is available on scrapbook pages.

Java Scrapbook page buttons

Command	Name	Description
	Run Snippet	Running an expression evaluates an expression but does not display a result.
	Display	Displaying shows the result of evaluating an expression as a string in the scrapbook editor.
	Inspect	Inspecting shows the result of evaluating an expression in the Expressions view.
	Terminate	This command terminates the Java VM that is used to evaluate expressions.
	Set the Import Declarations	This commands sets the import declarations to be used for the context of evaluating the code

Related concepts

Scrapbook

Related tasks

Creating a Java scrapbook page
Displaying the result of evaluating an expression
Inspecting the result of evaluating an expression
Executing an expression

Related reference

New Java scrapbook page wizard

Type Hierarchy View

This view shows the hierarchy of a type. The Type Hierarchy view consists of two panes:

- Type Hierarchy tree pane
- Member list pane (optional)

Type Hierarchy tree pane toolbar buttons

The type hierarchy tree shows supertypes, suptypes or both of a given type depending on the selection made in the toolbar.

Command	Name	Description
	Previous Hierarchy Inputs	This menu displays a history of previously displayed type hierarchies.
	Show the Type Hierarchy	This command displays the type in its full context (i.e., superclasses and subclasses) in the Type Hierarchy view. To see for which type the hierarchy is shown, hover over the view title (e.g., "Types").
	Show the Supertype Hierarchy	This command displays the supertypes and the hierarchy of all implemented interfaces of the type in the Type Hierarchy view. The tree starts at the selected type and displays the result of traversing up the hierarchy. *Note: The selected type is always at the top level, in the upper-left corner.*

	Show the Subtype Hierarchy	This command displays the subtypes of the selected class and/or all implementors of the interface in the Type Hierarchy view. The tree starts at the selected type and displays the result of traversing down the hierarchy *Note: The selected type is always at the top level, in the upper-left corner.*
	Layout > Hierarchy View Only	Hides the member list pane.

Member list pane toolbar buttons

The member list pane displays the members of the currently selected type in the type hierarchy tree pane.

Command	Name	Description
	Lock View and Show Members in Hierarchy	Shows the members implementing the selected method Only types implementing the method are shown. When the view is locked, the member list pane no longer tracks the selection in the hierarchy pane above.
	Show All Inherited Members	Shows or hides all methods and fields inherited by base classes. When this option is set, the name of the type that defines the method is appended to the method name.
	Sort Members by the Defining Type	Sorts the members according to the type in which they are defined.

	Hide Fields	Shows or hides the fields.
	Hide Static Fields and Methods	Shows or hides the static fields and methods.
	Hide Non-Public Members	Shows or hides the non-public fields and methods.

Related concepts

Java views

Related reference

Views and editors

Call Hierarchy View

The Call Hierarchy view shows callers and callees for a selected Java member.
Call Hierarchy commands:

Command	Name	Description
	Refresh	Refresh the whole hierarchy
	Refresh (context menu)	Refresh the selected elements and their direct children
	Cancel Current Search	Cancel the current running search. Useful for long running searches.
	Show Caller Hierarchy	Show all callers in the search scope of the selected member.
	Show Callee Hierarchy	Show all members which are called by the currently selected member.
	Show History List	This menu displays a history of previously displayed call hierarchies.
	Pin the Call Hierarchy View	Pins the current view and enables the user to open multiple Call Hierarchy views at the same time.
	Layout	Specifies the layout of the call hierarchy views
	Field Accesses	Specifies which field accesses to show: Read or write accesses, or both
	Search In...	Specifies where in the scope the Call Hierarchy should search for results.
	Search Scope	Defines the scope for the search for callers.
	Filters	Specifies the patterns whose matching members will be hidden from the view.

	Expand with Constructors (context menu)	Expands and replaces an instance method's normal children with: the constructors of the method's declaring classa node that contains the direct callers of the methodThis is especially useful for methods in anonymous classes.
	Expand with Constructors...	Specifies the list of types whose instance methods are expanded with constructors by default. The feature can also be enabled or disabled for all methods declared in anonymous types at once using the **All methods in anonymous types** option.
	Remove from View	Remove the selected nodes from the view

Related concepts

Java views

Related reference

Views and editors

JUnit View

The JUnit view shows you the JUnit test run progress and status, and allows you to rerun tests.

JUnit View commands:

Command	Name	Description
	Next Failed Test	Navigates to the next failed test.
	Previous Failed Test	Navigates to the previous failed test.
	Show Failures Only	Shows only failed tests.
	Scroll Lock	Changes if scroll lock should be enabled or not.
	Rerun Test	Rerun the test.
	Rerun Test - Failures First	Rerun the test, and run the failures first.
	Stop JUnit Test Run	Stops the current JUnit test run.
	Test Run History	This menu displays a history of previously run tests.
	Clear Terminated	Clears all the terminated test runs from the histroy.
	Import...	Imports a test run result from a XML file.
	Import from URL...	Imports a test run result via an URL.
	Import URL from Clipboard	Imports a test run result via an URL placed on clipboard.
	Export	Exports the test run result to a XML file.
	Show Tests in Hierarchy	Shows the tests in a hierarchical layout.
	Show Execution Time	Shows execution time for the tests.

	Layout	Specifies the layout of the JUnit view.
	Activate on Error/Failure Only	Changes if the JUnit view should be activated only on error/failure or not.

Related concepts

Java views

Related reference

Views and editors

Javadoc View

The Javadoc view shows the Javadoc of the element selected in the Java editor or in a Java view.

Javadoc View commands:

Command	Name	Description
	Back	Navigates to the most recently viewed javadoc.
	Forward	Navigates to the javadoc that was displayed immediately after the current javadoc.
	Link with Selection	Links the Javadoc view's input with the selection in the Java editor or in a Java view.
	Open Input	Opens the current input element of the Javadoc view in the Java editor.
	Open Attached Javadoc in a Browser (Shift+F2)	Opens the attached javadoc of current input of the Javadoc view in a browser.

Related concepts

Java views

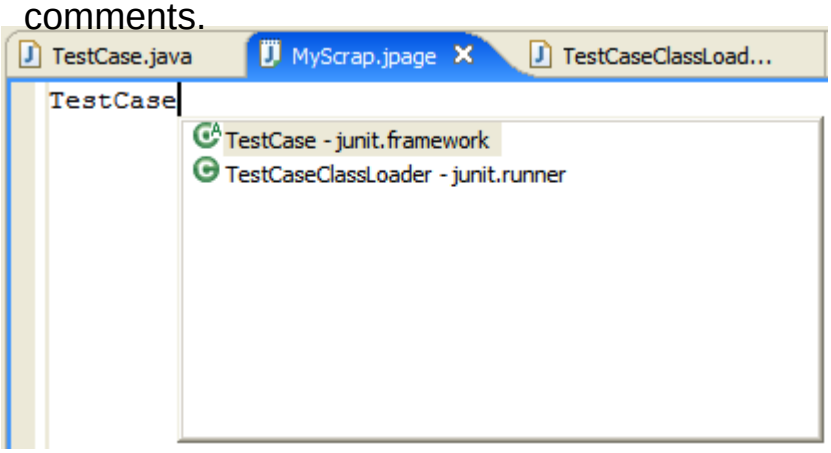
Related reference

Views and editors

Content/Code Assist

In the Java editor press **Ctrl + Space** on code to complete. This opens a list of available code completions. Some tips for using code assist are listed in the following paragraph:

- You can use the mouse or the keyboard (Up Arrow, Down Arrow, Page Up, Page Down, Home, End, Enter) to navigate and select lines in the list.
- If you select a line in the content assist list, you can view Javadoc information for that line.
- Clicking or pressing Enter on a selected line in the list inserts the selection into the editor.
- You can access specialized content assist features inside Javadoc comments.



Configure the behavior of the content assist in the Java > Editor > Code Assist preference page.

Related concepts

Java editor

Java development tools (JDT)

Related reference

Edit menu

Java editor preferences

Templates preferences

Formatter

In the Java editor press **Ctrl+Shift+F** on code to format it. If no selection is set then the entire source is formatted otherwise only the selection will be. Some tips for using the formatter are listed in the paragraphs of this chapter.

Note that the Java Formatter preferences are accessible on the [Image: /topic/org.eclipse.help/command_link.png]**Java Formatter** preference page.

Disabling formatter inside sections

You can disable/enable the formatter in one or several sections in the code as shown in the sample below:

```
public class Example {
    // area 1: formatted
    int x;

/**
 * The method foo.
 * @formatter:off
 */
public    void
foo()   {
for (int i=0;i<10;i++){
//  area    2:     NOT     formatted   !
}/* @formatter:on */
        for (int i = 0; i < 10; i++) {
            // area 3: formatted
        }/* @formatter:on */
    }

    void bar() {
        int i = 0;
        while (i++ < 10) {
        }
//@formatter:off
int j=0;while (j++<10){
//  area    4:     NOT     formatted   !
}}}
```

The snippet above use default tag names, but they can be changed on the **Off/On tags** tab of the Java Formatter preference page.

Wrap outermost method calls

Since version 3.6, the Java formatter now tries to wrap the outermost method calls first to have a better output when wrapping nested method calls.

Here is an example of a formatted code where the formatter has wrapped the line between the arguments of the outermost message call to keep each nested method call on a single line:

```
public class X {
    void test() {
        foo(bar(1, 2, 3, 4),
                bar(5, 6, 7, 8));
    }
```

A new preference allows you to disable this strategy, typically if you want to format your code as before, then uncheck the **Prefer wrapping outer expressions** preference accessible on the **Line wrapping** tab of the Java Formatter preference page.

Note: Currently the new strategy only applies to nested method calls, but that might be extended to other nested expressions in future versions.

Condense Javadoc and block comments

Users can reduce the number of lines of formatted multi-lines comments as shown in the example below:

```
/* The bar private method. */
private void bar() {
}
```

To activate this behavior uncheck the **/* and */ on separate lines** preference accessible on the **Comments** tab of the Java Formatter preference page.

The same kind of preference is also available for the Javadoc comments.

Preserve user line breaks

Users can preserve line breaks by not joining lines in code or comments.

For example, the already wrapped lines of the `return` statement in the following test case:

```
class X {
String foo() {
return "select x "
      + "from y "
      + "where z=a";
}
}
```

will be preserved by the formatter when the **Never join lines** preference is used, hence produces the following output when formatted:

```
class X {
   String foo() {
      return "select x "
            + "from y "
            + "where z=a";
   }
}
```

To activate this behavior check the **Never join lines** preference accessible on the **Line Wrapping** and the **Comments** tabs of the Java Formatter preference page.

Related concepts

Java editor

Java development tools (JDT)

Related reference

Edit menu

Java editor preferences

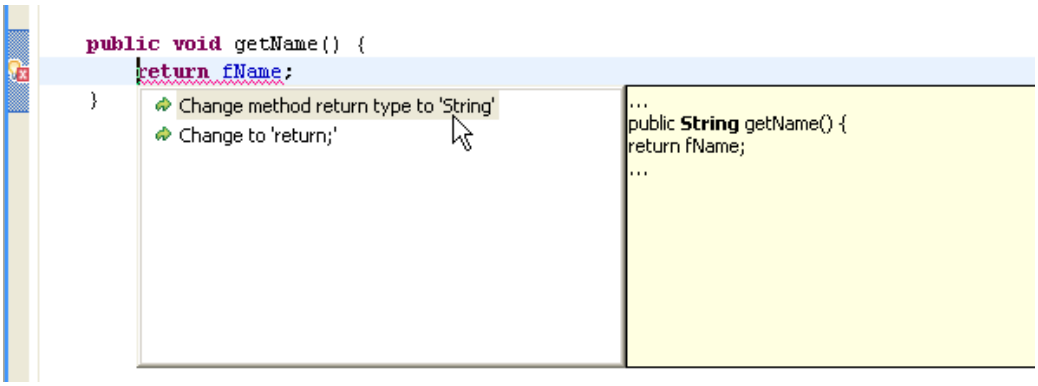
Formatter preferences

Quick Fix

The Java editor offers corrections to problems found while typing and after compiling. To show that correction proposals are available for a problem or warning, a 'light bulb' is visible on the editor's annotation bar.

Left click on the light bulb or invoking **Ctrl+1 (Edit > Quick Fix)** brings up the proposals for the problem at the cursor position.

Each quick fix shows a preview when selected in the proposal window.



Some selected quick fixes can also be assigned with direct shortcuts. You can configure these shortcuts on the [Image: /topic/org.eclipse.help/command_link.png]**General > Keys** preference page (in the 'Source' category).

Some quick fixes offer to fix all problems of the same kind in the current file at once. The information text in proposal window contains this information for all applicable proposals. To fix all problems of the same kind, press **Ctrl + Enter**.

The following quick fixes are available:

Package Declaration	Add missing package declaration or correct package declaration Move compilation unit to package that corresponds to the package declaration
Imports	Remove unused, unresolvable or non-visible import Invoke 'Organize imports' on problems in imports

Types	Create new class, interface, enum, annotation or type variable for references to types that can not be resolved Change visibility for types that are accessed but not visible Rename to a similar type for references to types that can not be resolved Add import statement for types that can not be resolved but exist in the project Add explicit import statement for ambiguous type references (two import-on-demands for the same type) If the type name is not matching with the compilation unit name either rename the type or rename the compilation unit Remove unused private types Add missing type annotation attributes
Constructors	Create new constructor for references to constructors that can not be resolved (this, super or new class creation) Reorder, add or remove arguments for constructor references that mismatch parameters Change method with constructor name to constructor (remove return type) Change visibility for constructors that are accessed but not visible Remove unused private constructor Create constructor when super call of the implicit default constructor is undefined, not visible or throws an exception If type contains unimplemented methods, change type modifier to 'abstract' or add the method to implement

Methods	Create new method for references to methods that can not be resolved Rename to a similar method for references to methods that can not be resolved Reorder or remove arguments for method references that mismatch parameters Correct access (visibility, static) of referenced methods Remove unused private methods Correct return type for methods that have a missing return type or where the return type does not match the return statement Add return statement if missing For non-abstract methods with no body change to 'abstract' or add body For an abstract method in a non-abstract type remove abstract modifier of the method or make type abstract For an abstract/native method with body remove the abstract or native modifier or remove body Change method access to 'static' if method is invoked inside a constructor invocation (super, this) Change method access to default access to avoid emulated method access Add 'synchronized' modifier Override hashCode() Open the 'Generate hashCode() and equals()' wizard
Fields and variables	Correct access (visibility, static) of referenced fields Create new fields, parameters, local variables or constants for references to variables that can not be resolved Rename to a variable with similar name for references that can not be resolved Remove unused private fields Correct non-static access of static fields Add 'final' modifier to local variables accessed in outer types Change field access to default access to avoid emulated method access Change local variable type to fix a type mismatch Initialize a variable that has not been initialized Create getter and setters for invisible or unused fields

Exception Handling	Remove unneeded catch block Handle uncaught exception by surrounding with try/catch or adding catch block to a surrounding try block Handle uncaught exception by adding a throw declaration to the parent method or by generalize an existing throw declaration
Build Path Problems	Add a missing JAR or library for an unresolvable type Open the build path dialog for access restriction problems or missing binary classes. Change project compliance and JRE to 1.5 Change workspace compliance and JRE to 1.5
Others	Add cast or change cast to fix type mismatches Let a type implement an interface to fix type mismatches Add type arguments to raw references Complete switch statements over enums Remove dead code Insert '//$FALL-THROUGH$' Insert null check For non-NLS strings open the NLS wizard or mark as non-NLS Add missing @Override, @Deprecated annotations Add missing Javadoc comments Add missing Javadoc tags Suppress a warning using @SuppressWarnings Throw the allocated object Return the allocated object

Quick Assists are proposals available even if there is no problem or warning. See the Quick Assist page for more information.

Related concepts

Java editor
Quick fix and assist

Related reference

Quick Assist
JDT actions

Quick Assist

Quick assists perform local code transformations. They are invoked on a selection or a single cursor in the Java editor and use the same shortcut as quick fixes (**Ctrl+1**), but quick assist are usually hidden when an error is around. To show them even with errors present on the same line, press **Ctrl+1** a second time.

A selection of quick assists can be assigned to a direct shortcut. By default, these are:
- Rename in file: **Ctrl+2, R**
- Assign to local: **Ctrl+2, L**
- Assign to field: **Ctrl+2, F**

Assign more shortcuts or change the default shortcuts on the [Image: /topic/org.eclipse.help/command_link.png]**General > Keys** preference page (in the 'Source' category).

A quick assist light bulb can be turned on on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor** preference page.

Name	Code example			Invocation location
Inverse if statement	`if (x) a(); else b();`	>	`if (!x) b(); else a();`	On 'if' statements with 'else' block
Inverse boolean expression	`a && !b`	>	`!a || b`	On a boolean expression

Invert local variable	`boolean a = false;`
	
	`if (a) {}`
		>	`boolean notA = true;`
			
			`if (!notA) {}`
				On a boolean variable

Invert equals	`a.equals(b)`	>	`b.equals(a)`	On a invocation of 'equals'
	Inverse conditional expression	`x ? b : c`	>	`!x ? c : b`
On a conditional expression	Pull negation up	`b && c`	>	`!(!b || !c)`
On a boolean expression	Push negation down	`!(b && c)`	>	`!b || !c`

On a negated boolean expression	Remove extra parentheses	`if ((a == b) && (c != d)  {}`	>	`if (a == b && c != d) {}`
On selected expressions	Put expression in parentheses	`return a > 10 ? 1 : 2;`	>	`return (a > 10) ? 1 : 2;`
On selected expression	Put expressions in parentheses	`if (a == b && c != d) {}`	>	`if ((a == b) && (c != d))  {}`
On selected expressions	Join nested if statements	`if (a) { if (b) {} }`	>	`if (a && b) {}`
On a nested if statement	Swap nested if statements	`if (a) { if (b) {} }`	>	`if (b) { if (a) {} }`
On a nested if statement	Split if statement with and'ed expression	`if (a && b) {}`	>	`if (a) { if (b) {} }`
On an and'ed expression in a 'if'	Join selected 'if' statements with ||	`if (a) x(); if (b) x();`	>	`if (a || b) x();`
On selected 'if' statements	Join 'if' sequence in if-else-if	`if (a) x(); if (b) y();`	>	`if (a) x(); else if (b) y();`
On selected 'if' statements	Split if statement with or'd expression	`if (a || b) x();`	>	`if (a) x(); if (b) x();`
On an or'd expression in a 'if'	If-else assignment to conditional expression	`if (a) x= 1; else x= 2;`	>	`x= a ? 1 : 2;`

On an 'if' statement	If-else return to conditional expression	`if (a) return 1;`
		
		`else return 2;`
			>	`return a ? 1 : 2;`

On an 'if' statement	Conditional expression assignment to If-else	`x= a ? 1 : 2;`	>	`if (a) x= 1; else x= 2;`
On a conditional expression	Conditional expression return to If-else	`return  a ? 1 : 2;`	>	`if (a) return 1; else return 2;`

On a conditional expression	Switch to If-else	`switch (kind) {`
		
		`case 1: return -1;`
		
		`case 2: return -2;`
		
		`}`
			>	`if (kind == 1) {`
				
				`return -1;`
				
				`} else if (kind == 2) {`
				
				`return -2;`
				
				`}`

On a switch statement	Add missing case statements on enums	`switch (e){`
		
		`}`
			>	`switch (e){`
				
				`case E1: break;`
				
				`case E2: break;`
				
				`}`

On a switch statement	Exchange operands	`a + b`	>	`b + a`

On an infix operation	Cast and assign	`if (obj instanceof Vector) {`
		
		`}`
			>	`if (obj instanceof Vector) {`
				
				`Vector vec= (Vector)obj ;`
				`}`

On an instanceof expression in an 'if' or 'while' statement	Add finally block	`try {`
		
		`} catch (Expression e) {`
		
		`}`
			>	`try {`
				
				`} catch (Expression e) {`
				
				`} finally {}`

On a try/catch statement	Add else block	`if (a) b();`	>	`if (a) b(); else { }`
On a if statement	Replace statement with block	`if (a) b();`	>	`if (a) { b(); }`
On a if statement	Unwrap blocks	`{ a() }`	>	`a()`
On blocks, if/while/for statements	Pick out string	`"abcdefgh"`	>	`"abc" + "de" + "fgh"`

select a part of a string literal	Convert string concatenation to StringBuilder (J2SE 5.0) or StringBuffer	`"Hello " + name`	>	`StringBuilder builder= new StringBuilder();`
				
				`builder.append("Hello ");`
				
				`builder.append(name);`

select a string literal	Convert string concatenation to MessageFormat	`"Hello " + name`	>	`MessageFormat.format("Hello {0}", name);`
select a string literal	Split variable	`int i= 0;`	>	`int i; i= 0;`
On a variable with initialization	Join variable	`int i; i= 0;`	>	`int i= 0`
On a variable without initialization	Assign to variable	`foo()`	>	`X x= foo();`

On an expression statement	Extract to local	`foo(getColor());`	>	`Color color= getColor();`
				
				`foo(color);`

On an expression	Assign parameter to field	`public A(int color) {}`	>	`Color fColor;`
				
				`public A(int color) {`
				
				`fColor= color;`
				
				`}`

On a parameter	Array initializer to Array creation	`int[] i= { 1, 2, 3 }`	>	`int[] i= new int[] { 1, 2, 3 }`

On an array initializer	Convert to 'enhanced for loop' (J2SE 1.5)	`for (Iterator i= c.iterator();i.hasNext();) {`
		
		`}`
			>	`for (x : c) {`
				
				`}`

On a for loop	Create method in super class			
On a method declaration	Rename in file			
On identifiers	Rename in workspace			

On identifiers	Extract to local variable	`a= b*8;`	>	`int x= b*8;`
				
				`a= x;`

On expressions	Extract to constant	`a= 8;`	>	`final static int CONST= 8;`
				
				`a= CONST;`

On expressions	Extract method	`int x= p * 5;`	>	`int x= getFoo(p);`
On expressions and statements	Inline local variable	`int a= 8, b= a;`	>	`int b= 8;`

On local variables	Convert local variable to field	`void foo() { int a= 8; }`	>	`int a= 8; void foo() {}`
On local variables	Convert anonymous to nested class	`new Runnable() { };`	>	`class RunnableImplementation implements Runnable { }`
On anonymous classes	Replace with getter and setter (Encapsulate Field)	`p.x;`	>	`p.getX();`

Related concepts

Java Editor
Quick Fix and Quick Assist

Related reference

Quick Fix
JDT actions

New Java Project Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Project** wizard helps you create a new Java project in the workbench.

Project name page

Option	Description	Default
Project name	Type a name for the new project.	<blank>
Location	When **'Use default location'** is selected, the New Project Wizard will create a new project with the specified name in the workspace. Otherwise, you can specify the location from which the New Java Project Wizard will retrieve an existing Java project. In this case the wizard will analyze the existing project and set up the build path automatically. Click on **Browse...** to browse for a location of an existing Java project.	workspace

JRE	**Use default JRE:** When selected, the New Java Project Wizard creates a new Java project which uses the workspace default JRE. The default JRE can be configured on the Java > Installed JREs preference page. The project will also use the default compiler compliance which can be configured on the Java > Compiler preference page. Click on **Configure default...** to configure the default JRE and compiler compliance. **Use project specific JRE:** When selected, you can explicitly specify the JRE to be used for the new Java project. The new project will use a compiler compliance which matches the version of the selected JRE. **Use an execution environment JRE:** When selected, you can specify an execution environment to be used for the new Java project. The new project will use a compiler compliance which fits best the selected execution environment. Execution environments can be configured on the Java > Installed JREs > Execution Environments	Use default JRE

	preference page.	
Project layout	**Use project folder as root for sources and class files:** When selected, the project folder is used both as source folder and as output folder for class files. **Create separate folders for sources and class files:** When selected, the New Java Project Wizard creates a source folder for Java source files and an output folder which holds the class files of the project.	Create separate folders for sources and class files
Working sets	**Add project to working sets:** When selected, the new project will be added to the working sets shown in **Working Sets** drop down field. The drop down field shows a list of previous selected working sets. Click on **Select...** to select working sets to which to add the new project.	Depends on the context from which the wizard has been started

Java Build Path page

You can skip this page by pressing Finish on the first page. Otherwise press Next to configure the Java build path.

The Java build path consist of source, library and project entries. The user interface is the same as for the Build Path property page, except for the source tab:

Source tab

Source folders are top-level folders in the project hierarchy. They are the root of packages containing .java files. The compiler will translate the contained files to .class files that will be written to the output folder.

Source folders allow to structure the project, for example to separate test from the application in two source folders. Within a source folder, a more detailed structuring can be achieved by using packages.

Each source folder can define an exclusion filter to specify which resources inside the folder should not be visible to the compiler.

Resources existing in source folders are copied to the output folder unless the setting in the Java > Compiler > Building preference page specifies that the resource is filtered. The output folder is defined per project except if a source folder specifies its own output folder.
The tree shows the project as it will look like when switching to the package explorer. Several operations can be executed on this tree to change the structure of the project.

Icon	Option	Description
	Add source folder	Add a folder to the Java build path as source folder.
	Link additional source to project	Add a link to a folder in the file system as source folder to the Java build path.
	Remove from buildpath	Remove a source folder from the Java build build path and change it into a normal folder.
	Exclude	Add a resource to the exclusion filter of its parent source folder. The excluded resource and all its children are no longer visible to the compiler.
	Include	Includes a previously excluded resource.

	Configure source folder properties	The edit source folder property menu has two actions **Configure Inclusion / Exclusions Filters**: Customize the inclusion and exclusion filters by defining string patterns. It is possible to use wildcard in patters (i.e. to exclude all java files which start with "Test" write "Test*.java").**Configure Output Folder**: Change the output folder for a source folder. This action is only enabled if **Allow output folders for source folders** is enabled.
	Undo all changes	All changes that have been applied to the project in this wizard will be withdrawn and the original state of the project is reconstructed.
	Allow output folders for source folders	If enabled, each source folder can have its own output folder. Otherwise all source folders will use the default output folder.

A shorter description of all operations is visible in the **Details** pane below the project tree.

Library tab, Project tab and Order Tab

See Build Path property page for more details.

Related concepts

Java projects

Related reference

File actions

New Java Package Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Package** wizard helps you create a folder corresponding to a new Java package. The corresponding folder of the default package always exists, and therefore doesn't have to be created.

Java Package Options

Option	Description	Default
Source folder	Type or browse to select a container (project or folder) for the new package.	The source folder of the element that was selected when the wizard has been started.
Name	Type a name for the new package	<blank>

Related reference

File actions

New Java Class Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Class** wizard helps you to create a new Java class in a Java project.

Java Class Options

Option	Description	Default
Source folder	Enter a source folder for the new class. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new class. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new class. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new class.	<blank>
Modifiers	Select one or more access modifiers for the new class. Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)abstractfinalstatic (only available if you specify an enclosing type)	public

Superclass	Type or click Browse to select a superclass for this class.	The type (not the compilation unit!) that was selected when the wizard has been started or <java.lang.Object>
Interfaces	Click Add to choose interfaces that the new class implements.	<blank>
Which method stubs would you like to create?	Choose the method stubs to create in this class: public static void main(String [] args): Adds a main method stub to the new class.Constructors from superclass: Copies the constructors from the new class's superclass and adds these stubs to the new class.Inherited abstract methods: Adds to the new class stubs of any abstract methods from superclasses or methods of interfaces that need to be implemented.	Inherited abstract methods enabled
Do you want to add comments?	When selected, the wizard adds comments to the new class where appropriate.	Do not add comments

Related reference

File actions

New Java Enum Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Enum Type** wizard helps you to create a new Java enum in a Java project.

Java Enum Options

Option	Description	Default
Source folder	Enter a source folder for the new enum. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new enum. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new enum. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new enum.	<blank>
Modifiers	Select one or more access modifiers for the new enum. Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)	public
Interfaces	Click Add to choose interfaces that the new enum implements.	<blank>

Do you want to add comments?	When selected, the wizard adds comments to the new enum where appropriate.	Do not add comments

- Related reference

File actions

New Java Interface Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Interface** wizard helps you to create a new Java interface in a Java project.

Java Interface Options

Option	Description	Default
Source folder	Enter a source folder for the new interface. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new interface. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new interface. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new interface.	<blank>

Modifiers	Select one or more access modifiers for the new interface. Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)static (only available if you specify an enclosing type)	public
Extended interfaces	Click Add to choose interfaces that the new interface extends.	<blank>
Do you want to add comments?	When selected, the wizard adds comments to the new class where appropriate.	Do not add comments

Related reference

File actions

New Java Annotation Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Annotation Type** wizard helps you to create a new Java Annotation in a Java project.

Java Annotation Options

Option	Description	Default
Source folder	Enter a source folder for the new annotation. Either type a valid source folder path or click Browse to select a source folder via a dialog.	The source folder of the element that was selected when the wizard has been started.
Package	Enter a package to contain the new annotation. You can select either this option or the Enclosing Type option, below. Either type a valid package name or click Browse to select a package via a dialog.	The package of the element that was selected when the wizard has been started.
Enclosing type	Select this option to choose a type in which to enclose the new annotation. You can select either this option or the Package option, above. Either type a valid name in the field or click Browse to select a type via a dialog.	The type or the primary type of the compilation unit that was selected when the wizard has been started or <blank>
Name	Type a name for the new annotation.	<blank>
Modifiers	Select one or more access modifiers for the new annotation. Either public, default, private, or protected (private and protected are only available if you specify an enclosing type)	public

Do you want to add comments?	When selected, the wizard adds comments to the new class where appropriate.	Do not add comments

- Related reference

File actions

New Source Folder Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Source Folder** wizard helps you to create a new source folder to a Java project.

Note that a new source folder can not be nested in existing source folders or in an output folder. You can choose to add exclusion filters to the other nesting source folders or the wizard will suggest to replace the nesting classpath entry with the new created entry. The wizard will also suggest to change the output location.

New Source Folder Options

Option	Description	Default
Project name	Enter a project to contain the new source folder. Either type a valid project name or click Browse to select a project via a dialog.	The project of the element that was selected when the wizard has been started.
Folder name	Type a path for the new source folder. The path is relative to the selected project.	<blank>
Update exclusion filter in other source folders to solve nesting	Select to modify existing source folder's exclusion filters to solve nesting problems. For example if there is an existing source folder *src* and a folder *src/inner* is created, the source folder *src* will be updated to have a exclusion filter *src/inner*.	Off

Related concepts

Java projects

Related reference

File actions

New Java Scrapbook Page Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**New Java Scrapbook Page** wizard helps you to create a new Java scrapbook page in a project.

Create Java Scrapbook Page Options

Option	Description	Default
Enter or select the parent folder	Type or browse the hierarchy below to select a container (project or folder) for the scrapbook page.	The container of the selected element
File name	Type a name for the new file. The "*.jpage* " extension is appended automatically when not added already.	<blank>

Related reference

Java scrapbook page
File actions

Export Breakpoints Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**Export Breakpoints** wizard helps you to export breakpoints to a file.

Export breakpoints page

Option	Description	Default
Breakpoint List	Select the breakpoints that you wish to export, and de-select those that you do not.	**There can be two possible defaults:** If the wizard is invoked from the breakpoints view and any breakpoints are selected, those same ones will be checked by default. If the wizard is invoked via the **File** menu than nothing will be selected.
To File	The file to export the breakpoints to, you can also click **Browse...** to search for a location to export to.	**There can be two possible defaults:** If you have never exported breakpoints before it will be blank If you have exported breakpoints before the last file name you exported to will be automatically inserted.
Overwrite	**Overwrite existing file without warning :** When selected, if the file you wish to export to already exists, it will be overwritten automatically by the wizard.	not selected

Related concepts

Breakpoints

Related tasks

Adding Line Breakpoints
Removing Line Breakpoints
Launching a Java Program
Running and Debugging

Related reference

File actions
Importing Breakpoints

Export Launch Configurations Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**Export Launch Configurations** wizard helps you to export breakpoints to a file.

Export launch configurations page

Option	Description	Default
Launch configuration list	Select the launch configurations that you wish to export, and de-select those that you do not.	None
Select All	Can be used to quickly select all launch configurations in the list.	None
Deselect All	Can be used to quickly deselect all launch configurations in the list.	None
Location	The folder to export the launch configurations to.	None
Browse...	The **Browse...** button allows you to search for a specific folder to export launch configurations to.	None
Overwrite existing file(s) without warning	When selected, if the location already contains a launch configuration being exported, it will be overwritten automatically by the wizard.	Not Selected

Related tasks

Launching a Java Program
Running and Debugging

Related reference

File actions
Importing Launch Configurations

Externalize Strings Wizard

The Externalize Strings wizard allows you to refactor a compilation unit such that strings used in the compilation unit can be translated to different languages. The wizard can be invoked on new, unprocessed files, but also modify already translated pages. It also allows you revert changes done by a previous invocation of the wizard.

On the first page of the wizard you can specify how the strings found in the compilation unit should be externalized. It is also possible to internalize strings which have been externalized before.

Field	Description
Use Eclipse's string externalization mechanism	If unchecked the standard externalization mechanism is used, otherwise the new Eclipse string externalization mechanism is used. *Note:* This field is only present if the project build path contains the `org.eclipse.osgi.util.NLS` class.
Enter common prefix for generated keys	Specifies an optional prefix for all newly generated key. A good practice is to use the name of the compilation unit so entries in the property files can easily be grouped.
Strings to externalize	Displays the list of all strings in the file. When non-externalized strings are found, the list ill be filtered to only show the new strings with proposed keys and values. To control this filter, use the checkbox on top of the table.
Externalize	Marks the selected strings to be externalized. Externalized strings will be placed in a property file and the code will be changed to a lookup to the property file.
Ignored	Marks the selected strings to be ignored from externalization. These strings will be marked with a '//$NON_NLS' comment and the compiler will ignore this string when warning about non-externalized strings.

Internalize	Marks the selected strings to be internalized again: This brings the code back in its original state before the externalize wizard was applied: The entry in the property file will be removed, and a '//$NON_NLS' comment will be removed.
Edit...	Opens a dialog for entering a new key.
Context	Displays the occurrence of the string in the context of the compilation unit.
Accessor class	The class used to access the property file. Press **Configure...** to open the **Configure Accessor Class** dialog.

Configure Accessor Class

This dialog specifies the name and location of the property file and its accessor class. An accessor class is used to retrieve strings stored in the property file given a key.

Accessor class settings:

Option	Description
Source folder	The source folder to store the accessor class in.
Package	The location for the accessor class.
Class name	The name of the class to access the resource bundle.
Substitution pattern	Specifies the source pattern to replace the string to externalize with.

Property file settings:

Option	Description
Source folder	The source folder to store the property file in.
Package	The location for the property file.
Property file name	The name of the property file.

Related concepts

String externalization
Eclipse string externalization mechanism

Related tasks

Externalizing strings

- Related reference

Source actions

Import Breakpoints Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**Import Breakpoints** wizard helps you to import breakpoints from a file.

Import breakpoints page

Option	Description	Default
From...	Type the name of the file to import from, or click the **Browse...** button to search for a file.	<blank>
Update...	**Automatically update existing breakpoints** When selected, any breakpoints that already exist in your workspace will be overwritten by those being imported from the file.	not selected
Create...	**Automatically create breakpoints workingsets:** When selected, if an imported breakpoint indicates it belongs to a working sets that does not exist in your workspace, the workingset will be created for you and the breakpoint added to it.	not selected

Related concepts

Breakpoints

Related tasks

Adding Line Breakpoints
Removing Line Breakpoints
Launching a Java Program
Running and Debugging

Related reference

File actions
Exporting Breakpoints

Import Launch Configurations Wizard

The [Image: /topic/org.eclipse.help/command_link.png]**Import Launch Configurations** wizard helps you to import breakpoints from a file.

Import launch configurations page

Option	Description	Default
From Directory	The folder to import launch configurations from.	None
Browse...	The **Browse...** button allows you to search for a specific folder to import launch configurations from.	None
Overwrite existing launch configurations without warning	When selected, any launch configurations that already exist in your workspace will be overwritten by those being imported.	Not Selected

Related tasks

Launching a Java Program
Running and Debugging

Related reference

File actions
Exporting Launch Configurations

JAR File Exporter

The [Image: /topic/org.eclipse.help/command_link.png]**Jar Export** wizard allows you to create a JAR file.

JAR package specification

JAR Specification Options:

Option	Description
Select the resources to export	In the list, check or clear the boxes to specify exactly the files that you want to export to the JAR file. This list is initialized by the workbench selection.
Export generated class files and resources	If enabled, then generated class files and resources are included in the JAR.
Export all output folders for checked projects	If enabled, then all output folders are included in the JAR.
Export Java source files and resources	If enabled, then Java source files and resources are included in the JAR.
Export refactorings for checked projects	If enabled, then refactoring scripts for the selected projects are included in the JAR. This allows clients to migrate to the new JAR by executing all the stored refactorings in the JAR.
Select the export destination	Enter an external file system path and name for a JAR file (either new or existing). Either type a valid file path in the field or click Browse to select a file via a dialog.

Options	You can select any of the following options: **Compress the contents of the JAR file** : to create a compressed JAR file **Add directory entries**: adds an entry for each directory to the JAR file, even if the directory does only contain subdirectories. **Overwrite existing files without warning** : if an existing file might be overwritten, you are prompted for confirmation. This option is applied to the JAR file, the JAR description, and the manifest file.

JAR packaging options

JAR Options:

Option	Description
Select options for handling problems	Choose whether to export classes with certain problems: Export class files with compile errors Export class files with compile warnings
Create source folder structure	Selected this option if you want the source folder structure to be rebuilt in the JAR. This option is only enabled when source files but no class files are exported.
Build projects if not build automatically	Select this option to force a rebuild before exporting. It is recommended to build be before exporting so the exported class files are up to date.
Save the description of this JAR in the workspace	If you select this option, you can create a file in the workbench describing the JAR file you are creating. Either type and/or browse to select a path and name for this new file.

JAR manifest specification

JAR Manifest Specification Options (Available when class file are exported):

Option	Description

Specify the manifest	Choose the source of the manifest file for this JAR file: **Generate the manifest file** (you can also choose either to save or reuse and save the new manifest file) **Use existing manifest from workspace**
Seal contents	Choose which packages in the JAR file must be sealed: **Seal the JAR** to seal the entire JAR file; click **Details** to exclude selectively **Seal some packages** click **Details** to choose selectively *Note: This option is only available if the manifest is generated.*
Select the class of the application entry point	Type or click **Browse** to select the main class for the JAR file, if desired. *Note: This option is only available if the manifest is generated.*

Related tasks

Creating JAR files

Related reference

File actions

Javadoc Generation

The [Image: /topic/org.eclipse.help/command_link.png]**Javadoc Generation** wizard allows you to generate Javadoc. It is a user interface for the javadoc.exe tool available in the Java JDK. Visit Sun's Javadoc Tool page for a complete documentation of the Javadoc tool.

First page

Type Selection:

Option	Description
Javadoc command	Specify which command to use to generate the javadoc.
Select types for which Javadoc will be generated	In the list, check or clear the boxes to specify exactly the types that you want to export. This list is initialized by the workbench selection.
Create Javadoc for members with visibility	**Private**: All members will be documented **Package**: Only members with default, protected or public visibility will be documented **Protected**: Only members with protected or public visibility will be documented **Public**: Only members with public visibility will be documented (default)
Use standard doclet	Start the Javadoc command with the standard doclet (default) **Destination**: select the destination to which the standard doclet will write the generated documentation. The destination is a doclet specific argument, and therefore not enabled when using a custom doclet.
Use custom doclet	Use a custom doclet to generate documentation **Doclet name**: Qualified type name of the doclet **Doclet class path**: Classpath needed by the doclet class

Standard doclet arguments

Standard Doclet Arguments (available when **Use standard doclet** has been selected):

Option	Description
Document title	Specify a document title.
Generate use page	Selected this option if you want the documentation to contain a Use page.
Generate hierarchy tree	Selected this option if you want the documentation to contain a Tree page.
Generate navigator bar	Selected this option if you want the documentation to contain a navigation bar (header and footer).
Generate index	Selected this option if you want the documentation to contain a Index page.
Separate index per letter	Create an index per letter. Enabled when **Generate Index** is selected.
@author	Selected this option if you want to the @author tag to be documented.
@version	Selected this option if you want to the @version tag to be documented.
@deprecated	Selected this option if you want to the @deprecated tag to be documented.
deprecated list	Selected this option if you want the documentation to contain a Deprecated page. Enabled when the **@deprecated** option is selected.
Select referenced archives and projects to which links should be generated	Specify to which other documentation Javadoc should create links when other types are referenced. Select All: Create links to all other documentation locations Clear All: Do not create links to other documentation locations Configure: Configure the Javadoc location of a referenced JAR or project.
Style sheet	Select the style sheet to use

General arguments

General Javadoc Options:

Option	Description

Overview	Specifies that Javadoc should retrieve the text for the overview documentation from the given file. It will be placed in overview-summary.html.
VM options	Add any number of extra VM options here.
Extra Javadoc options	Add any number of extra options here: Custom doclet options or JRE 1.4 compatibility options. Note that arguments containing spaces must enclosed in quotes. For arguments specifying html code (e.g. -header) use the & or &" to avoid spaces and quotes.
JRE source compatibility	The JRE source compatibility to be accepted by the javadoc tool.
Save the settings of this Javadoc export as an Ant script	Specify to store an Ant script that will perform the specified Javadoc export without the need to use the wizard. Existing Ant script can be modified with this wizard (Use Open Javadoc wizard' from the context menu on the generated Ant script)
Open generated index file in browser	Opens the generated index.html file in the browser (Only available when using the standard doclet)

Press **Finish** to start the Javadoc generation. If the destination is different to the location configured the project's Javadoc Location page , you will be asked if you want to set the project's Javadoc location to the new destination folder. By doing this, all invocations of Open External Javadoc will use the now created documentation.

A new process will be started and the generation performed in the background. Open the Console view (Window > Show View > Console) to see the progress and status of the generation.

Related reference

File actions
Javadoc location properties

Open Type

This dialog allows you to browse the workbench for a type to open in an editor or type hierarchy

- **Enter type name prefix or pattern**: In this field, type the first few characters of the type you want to select. The following pattern kinds are supported:

  Wildcards:
  - *"*"* for any string and *"?"* for any character
  - terminating *"<"* or *" "* (space) to prevent the automatic prefix matching, e.g. *"java.*Access<"* to match `java.util.RandomAccess` but not `java.security.AccessControlContext`

  Camel case:
  - *"TZ"* for types containing *"T"* and *"Z"* as upper-case letters in camel-case notation, e.g. `java.util.TimeZone`
  - *"NuPoEx"* or *"NuPo"* for types containing *"Nu"*, *"Po"*, (and *"Ex"*) as parts in camel-case notation, e.g. `java.lang.NullPointerException`
  - terminating "<" or " " (space) to fix the number of camel-case parts, e.g. "HMap<" and "HaMap<" match "HashMap" and "HatMapper", but not "HashMapEntry" nor "Hashmap".

  Both pattern kinds also support **package prefixes**, e.g. *"j.util.*Map<"*.

- **Matching items**: This list displays matches for the pattern you type in the **Enter type name prefix or pattern** field. Recently opened types show up in a history section at the top of the list.

The behavior of the **Open Type** dialog can be further customized using the dialog menu:

Open Type Options

Option	Description	Default
Show Status Line	When selected, the **Open Type** dialog shows an additional bar at the bottom of the dialog which displays the package and containing JRE of the selected type	Show status line
Working Set actions	The search scope can be restricted by selecting one or more working sets or the global Window Working Set.	Show all types in the workspace.

Type Filters...	Types are hidden if they match the global Java type filters.	Show all types in the workspace.

Related tasks

Opening an editor on a type

Related reference

Working sets
Type filters
Navigate actions

Create Getters and Setters

This dialog lets select the getter and setter methods to create.

Use **Generate Getters and Setters** from the Source menu or the context menu on a selected field or type, or a text selection in a type to open the dialog. The Generate Getters and Setters dialog shows getters and setters for all fields of the selected type. The methods are grouped by the type's fields.

The names of the getter and setter methods are derived from the field name. If you use a prefix or suffix for fields (e.g. fValue, _value, val_m), you can specify the suffixes and prefixes in the Code Style preference page (Windows > Preferences > Java > Code Style).

When pressing **OK**, all selected getters and setters are created.

Option	Description
Select getters and setters to create	A tree containing getter and setter methods that can be created. Getters and setters are grouped by field their associated field.
Select All	Select all getter and setter methods
Deselect All	Deselect all getter and setter methods
Select Getters	Select all getter methods
Insertion point	Defines where in the type body the new methods are inserted
Sort by	Defines how to sort the new methods: in getter/setter pairsfirst getters, then setters
Access modifiers	Defines the access modifiers of the new methods
Generate method comments	Controls whether Javadoc comments are added to the created methods. The comment templates are defined on the Code Templates preference page.

Related reference

Source actions

Generate toString() dialog

The Generate toString() dialog provides means to include specific member fields and methods in generated toString() method.

Use **Generate toString()...** from the Source menu or the context menu on a selected type or on a text selection in a type.

The members to be considered for inclusion in the toString() output are:

- fields (non-static)
- argument-less methods (non-static, non-void)
- inherited fields and methods (fulfilling adequate conditions)

When pressing **OK**, the toString() method printing out all selected members and eventually helper methods are created.

Option	Description	Default
Select fields and methods to include in the toString() method	For convenience, members are divided into up to four groups: Fields, Inherited fields, Methods and Inherited methods. Checking or unchecking a group affects the state of all its members.	Only fields which are non-transient and not inherited.
Select All	Select all fields and methods.	n/a
Deselect All	Deselect all fields and methods.	n/a
Up	Move the element with focus up the list. A member can be moved only within its group. Groups can also be moved.	n/a
Down	Move the element with focus down the list. A member can be moved only within its group. Groups can also be moved.	n/a
Sort	Sort elements in every group in lexicographical order.	n/a

Insertion point	Defines where in the type body the generated methods are inserted. If a method already exists in a class and is overwritten, its position will not be changed (this option will have no effect).	Last member or location of the cursor position if invoked from editor
Generate method comments	Controls whether a comment is added to the created toString() method. Generated comment just points to a javadoc comment for `java.lang.Object#toString()`.	off
String format	Defines which template is used to format the toString() output. A template determines what information about a class is added, how the fields are separated, and so on. For more information about templates, see toString() generator: format templates topic.	Default template
Edit...	Opens a dialog that lets you add, remove or change string format templates (see above).	n/a
Code style	Defines the method code style, that is what libraries/mechanisms it uses to create output. For more information, see toString() generator: code styles topic.	String concatenation

Configure...	Opens a dialog that lets you configure more options related to selected code style. Currently this button is only active for Custom toString() builder code style.	n/a
Skip null values	Controls whether the toString() method skips members that have no value (nulls). How it is accomplished depends on selected code style (see above).	off
List contents of arrays instead of using native toString()	Controls whether the toString() method lists items contained by arrays instead of using their default toString(). See toString() generator: listing content for details.	off
Limit number of items in arrays/collections/maps	Controls whether the toString() method limits the number of elements printed for arrays, `Collections` and `Maps`, and what the limit is. It doesn't work for arrays if "Ignore arrays' default toString()" option is off. See toString() generator: listing content for details.	off

Related reference

Source actions
toString() generator: format templates
toString() generator: code styles
toString() generator: content listing

toString() Generator: Format Templates

Format templates are used by a simple mechanism that allows you to change format of generated method's output string: beginning, ending, separator, and so on. They look similar to JDT code templates but they don't affect the generated code directly (Code Styles are used for that). Here's a list of available template variables:

${object.className}	inserts the class name as a simple String
${object.getClassName}	inserts a call to `this.getClass.getName()`
${object.superToString}	inserts a call to `super.toString()`
${object.hashCode}	inserts a call to `this.hashCode()`
${object.identityHashCode}	inserts a call to `System.identityHashCode(this)`
${member.name}	inserts the first member's name
${member.name()}	inserts the first member's name followed by parenthesis in case of methods
${member.value}	inserts the first member's value
${otherMembers}	inserts the remaining members. For each member, the template fragment between the first and the last **${member.*}** variable is evaluated and appended to the result. The characters between the last **${member.*}** and **${otherMembers}** define the separator that is inserted between members (**${otherMembers}** must stand after the last **${member.*}** variable).

For the template to work properly, the **${otherMembers}** variable must be used exactly once in a template and cannot be followed by any **${member.*}** variable. **${object.*}** variables can be placed anywhere in the template, although if one is placed in a member related fragment (that is between the first **${member.*}** variable and **${otherMembers}**), it will be repeated for every member, which is probably not a sensible solution.

The description above might seem complicated, but the template format itself is very easy to use. It should all become clear after seeing some working examples.

Template examples

1. The default template is a good example:

```
${class.name} [${member.name()}=${member.value}, ${otherMembers}]
```

   The output string for this template looks like this:

```
FooClass[aFloat=1.0, aString=hello, anInt=10, anObject=null, aCharMethod()=a]
```

2. Multiple line output is also available:

```
${object.getClassName} {
 ${member.name}: ${member.value}
 ${otherMembers}
}
```

   Example result:

```
FooClass {
 aFloat: 1.0
 aString: hello
 anInt: 10
 anObject: null
 aCharMethod: a
}
```

3. If you enclose a member in braces, don't forget to do the same with **${otherMembers}** variable:

```
{${member.name}=${member.value}},
{${otherMembers}}
```

   Here's the effect:

```
{aFloat=1.0},
{aString=hello},
{anInt=10},
{anObject=null},
{aCharMethod=a}
```

4. **${object.*}** variables can be used at the beginning and at the end of the template:

```
${object.getClassName} (hashcode:${object.hashCode})
 members: ${member.name} = ${member.value}; ${otherMembers}
[super: ${object.superToString}]
```

   This template would result in an output similar to this:

```
FooClass (hashCode:232198409832)
 members: aFloat = 1.0; aString = hello; anInt = 10; anObject = null; aCharMethod = a
[super: SuperFooClass[aField=null]]
```

Related reference

Generate toString() dialog
toString() Generator: Code Styles
toString() Generator: Content Listing

toString() Generator: Code Styles

Code style determines how the generated method works and what classes it uses. There are several code styles available to chose from the combo box in generator's dialog:

- **String concatenation**

  This style uses simple sum expressions so it's very efficient (compiler uses StringBuilder/StringBuffer to optimize the code) and relatively easy to read and modify. Here's an example outcome in the simplest case:

```
return "FooClass [aFloat=" + aFloat + ", aString=" + aString + ", anInt=" + anInt
 + ", anObject=" + anObject + "]";
```

  With "Skip null values" option turned on, the code becomes a little harder to read:

```
return "FooClass [aFloat=" + aFloat + ", "
 + (aString != null ? "aString=" + aString + ", " : "")
 + "anInt=" + anInt + ", "
 + (anObject != null ? "anObject=" + anObject : "") + "]";
```

- **StringBuilder/StringBuffer**

  This style uses StringBuilder if the project is compatible with JDK1.5 or later and StringBuffer otherwise. StringBuilder is faster (because of lack of synchronization), but only available since JDK1.5.

```
StringBuilder builder = new StringBuilder();
builder.append("FooClass [aFloat=");
builder.append(aFloat);
builder.append(", aString=");
builder.append(aString);
builder.append(", anInt=");
builder.append(anInt);
builder.append(", anObject=");
builder.append(anObject);
builder.append("]");
return builder.toString();
```

  The **"Skip null values"** option doesn't obfuscate the code as much as previously:

```
StringBuilder builder = new StringBuilder();
builder.append("FooClass [aFloat=");
builder.append(aFloat);
builder.append(", ");
if (aString != null) {
 builder.append("aString=");
 builder.append(aString);
 builder.append(", ");
}
builder.append("anInt=");
builder.append(anInt);
builder.append(", ");
if (anObject != null) {
 builder.append("anObject=");
 builder.append(anObject);
}
```

```
builder.append("]");
return builder.toString();
```

- **StringBuilder/StringBuffer with chained calls**

  Style very similar to the previous one only that append methods are called in chain. This makes the code shorter and probably easier to read. `StringBuilder builder = new StringBuilder();`

```
builder.append("FooClass [aFloat=").append(aFloat).append(", aString=").append(aString)
  .append(", anInt=").append(anInt).append(", anObject=").append(anObject).append("]");
return builder.toString();
```

  With **"Skip null values"** switched on, the chain must be broken: `StringBuilder builder = new StringBuilder();`

```
builder.append("FooClass [aFloat=").append(aFloat).append(", ");
if (aString != null) {
 builder.append("aString=").append(aString).append(", ");
}
builder.append("anInt=").append(anInt).append(", ");
if (anObject != null) {
 builder.append("anObject=").append(anObject);
}
builder.append("]");
return builder.toString();
```

- **String.format()/MessageFormat**

  This style is very pleasant for relatively short list of elements, but with longer ones it becomes hard to see which fields are associated with which variables. Unfortunately, the **"Skip null values"** option cannot be used with this style. `return String.format("FooClass [aFloat=%s, aString=%s, anInt=%s, anObject=%s]",`

```
 aFloat, aString, anInt, anObject);
```

  Because there's no `String.format()` in JDK 1.4 and earlier, `MessageFormat.format()` is used instead: `return MessageFormat.format("FooClass [aFloat={0}, aString={1}, anInt={2}, anObject={3}]",`

```
 new Object[] { new Float(aFloat), aString, new Integer(anInt), anObject });
```

- **Custom toString() builder**

  This style uses an external class to build a result string. It can use classes that fulfill the following conditions:

  - Provide a public constructor taking a single `Object` as parameter - it will be passed an object for which the toString() method is called
  - Provide methods for appending member information - these are methods with specified name, that take an `Object` and (optionally) a `String` (in any order)
  - Provide a method for retrieving result - that is a method taking no arguments and returning a `String`

  Custom builder requires some additional configuration to work properly. All necessary options can be entered in a dialog box showing up after clicking 'Configure...' button. These options include:

  - **Builder class** - a fully qualified name of a class to use. It can be typed in manually or selected from a class search dialog box (in this case it's automatically checked if selected class meets the requirements). It can be a class declared either directly in current project or in one of included libraries - it just has to be accessible on the build path. For example, ToStringBuilder from the

Apache Commons Lang library or ToStringCreator from the Spring Framework work very well with this mechanism.

- **Builder label** - any valid java identifier. It will be used to reference the builder object.
- **Append method** - the name of methods to use for appending items. If the class provides many methods with this name, methods taking two arguments (one of them must be `String`) are preferred over those taking a single argument (additionally, the `String` argument shall preferably be the first one). If there are versions of the method that take specific argument types, they are also used when possible.
- **Result method** - the name of a method to use for retrieving final result.
- **Chain invocations** - determines whether calls to the **append** methods should form chains. This option takes effect only for methods that have proper return type (that is, the builder class, or a subclass).

For example, suppose your builder class looks like this:

```
package org.foo.ToStringBuilder2;


public class ToStringBuilder2 {
 public ToStringBuilder2(Object o) {...}
 public ToStringBuilder2 appendItem(String s, Object o) {...}
 public ToStringBuilder2 appendItem(String s, float f) {...}
 public String getString() {...}
}
```

Of course in this case **builder class** should be set to `"org.foo.ToStringBuilder2"`, **builder label** can be for example `"builder"`, **append method** is `"appendItem"` and **result method** is `"getString"`. With **chain invocations** selected, generated method will look like this:

```
ToStringBuilder2 builder = new ToStringBuilder2(this);
builder.append("aFloat", aFloat).append("aString", aString).append("anInt", new Integer(anInt))
  .append("anObject", anObject);
return builder.getString();
```

Note that a primitive variable `anInt` was passed to the builder using wrapper type. This is done for projects using JDK 1.4 and earlier (for later JDKs the compiler does it automatically). In case of `aFloat` there was a specific method in builder class so no wrapping was required.

Related reference

Generate toString() dialog
toString() Generator: Format Templates
toString() Generator: Content Listing

toString() Generator: Content Listing

This topic discusses how toString() generator lists contents of arrays and how it limits number of items listed in `Collections` and `Maps`. Used method depends not only on the member type, but also on selected JDK compatibility of the project.

Listing contents of Arrays

- **Without limiting number of elements**

  For JDK1.5 and later, generated method uses `Arrays.toString()`. As this method is not available in JDK1.4, `Arrays.asList()` is used instead in this case, but only for non-primitive arrays. Primitive arrays are handled with a helper method, showed below.

- **Limiting number of elements**

  If the array enclose a non-primitive type, the same solution is used for all versions of JDK: `Arrays.asList()` with `List.subList()`. For example, for a member named anArray, generated code looks like this: `Arrays.asList(anArray).subList(0, Math.min(anArray.length, maxLen))`

  In case of primitive arrays, `Arrays.asList()` cannot be used so in JDK1.5 or lower a helper method is used instead. In JDK1.6 there's another solution: `Arrays.toString(Arrays.copyOf(anArray, Math.min(anArray.length, maxLen))`. Although copying an array is not an optimal solution, the copied part is usually rather small and efficiency is not affected. A good thing is that a helper method can be avoided.

- **Helper method**

  A helper method returns a string listing items of a given array, in the form of `[1, 2, 3]`. The method iterates over the array and uses `instanceof` to determine its enclosing type. To make the code shorter, it checks only for types that can actually be passed to it.

```
private String arrayToString(Object array, int len, int maxLen) {
 StringBuffer stringBuffer = new StringBuffer();
 len = Math.min(len, maxLen);
 stringBuffer.append("[");
 for (int i = 0; i < len; i++) {
  if (i > 0)
   stringBuffer.append(", ");
  if (array instanceof float[])
   stringBuffer.append(((float[]) array)[i]);
  if (array instanceof int[])
   stringBuffer.append(((int[]) array)[i]);
  if (array instanceof Object[])
   stringBuffer.append(((Object[]) array)[i]);
 }
 stringBuffer.append("]");
 return stringBuffer.toString();
}
```

NOTES:

- This method is overwritten every time the generator runs.

- If the number of items is not limited, the method doesn't take the `maxLen` argument and doesn't use `Math.min()`
- StringBuffer or StringBuilder is used depending on selected JDK compatibility

Listing limited contents of `Lists`

The same solution is used for all JDK versions: `aList.subList(0, Math.min(aList.size(), maxLen))`

Listing limited contents of `Collections` (helper method)

A `Collection` cannot be turned into a `List` without copying its contents (assuming it isn't a `List` already), so a helper method is used to iterate over first `maxLen` elements and build a string out of them:

```
private String toString(Collection collection, int maxLen) {
 StringBuffer stringBuffer = new StringBuffer();
 stringBuffer.append("[");
 int i = 0;
 for (Iterator iterator = collection.iterator(); iterator.hasNext() && i < maxLen; i++) {
  if (i > 0)
   stringBuffer.append(", ");
  stringBuffer.append(iterator.next());
 }
 stringBuffer.append("]");
 return stringBuffer.toString();
}
```

NOTES:

- This method is not overwritten every time the generator runs, so it can be easily customized by hand.
- StringBuilder or StringBuffer is used depending on selected JDK compatibility

Listing limited contents of `Maps`

In case of `Maps`, the same helper method is used as for `Collections` only that `map.entrySet()` is passed to it.

Summary

This table sums up what methods are used in different conditions:

	`java.util.List`	`java.util.Collection`	`java.util.Map`	Array of primitive type	Array of non-primitive type
jdk 1.4	-	-	-	helper method `arrayToString(array, len)`	`Arrays.asList(array)`

jdk 1.4/1.5, limit elements	`member.subList()`	helper method `toSting(collection, maxLen)`	helper method `toString(collection, maxLen)` with `map.entrySet()`	helper method `arrayToString(array, len, maxLen)`	`Arrays.asList(array).subList()`
jdk 1.5	-	-	-	`Arrays.toString()`	`Arrays.asList(array)`
jdk 1.6	-	-	-	`Arrays.toString()`	`Arrays.toString()`
jdk 1.6, limit elements	`member.subList()`	helper method `toString(Collection)`	helper method `toString(Collection)` with `map.entrySet()`	`Arrays.toString(Arrays.copyOf(member, ...))`	`Arrays.asList(array).subList()`

Additional notes

- If a helper method must be generated for at least one member, it is also used for other members when possible. For example, a `List` is usually handled with `subList()` method, but if there's another member of `Collection` (not `List`) type and toString(collection) is generated, `Lists` are also passed to it. This way the code is shorter and more consistent.
- Where necessary, the code responsible for listing contents is surrounded with null-checking code, for example: `array != null ? Arrays.asList(array) : null`
- If maximum number of listed items is set to 0, the generator uses simple string literal (`"[]"`) instead of methods described above.

Related reference

Generate toString() dialog
toString() Generator: Format Templates
toString() Generator: Code Styles

Override Methods

This dialog lets you define methods to override.

Use **Override/Implement Methods** from the Source menu or the context menu on a selected type or on a text selection in a type.

The dialog presents all methods that can be overridden from supertypes or implemented from interfaces. Abstract methods or unimplemented methods are selected by default.

The tree view groups methods by the type declaring the method. If more than one type in the hierarchy declare the same method, the method is only shown once, grouped to the first type in the list of supertypes that implements or defines this method.

The flat view shows only methods, sorted alphabetically.

When pressing **OK**, method stubs for all selected methods are created.

Option	Description	Default
Select methods to override or implement	Select methods to override or implement	Abstract methods from superclasses and unimplemented methods from interfaces are selected
Group methods by types	Shows methods grouped by a list of the super types in which they are declared.	selected
Select All	Select all methods	n/a
Deselect All	Deselect all methods	n/a
Insertion point	Defines where in the type body the new methods are inserted	Last member or location of the cursor position if invoked from editor
Generate method comments	Controls whether Javadoc comments are added to the created methods. The comment templates are defined on the Code Templates preference page.	off

Related reference

Source actions

Frequently Asked Questions on JDT

Can I use a Java compiler other than the built-in one (javac for example) with the workbench?

No. The JDT provides a number of sophisticated features including fully automatic incremental compilation, code snippet evaluation, code assist, type hierarchies, and hot code replace. These features require special support found in the workbench Java compiler (an integral part of the JDT's incremental project builder), but not available in standard Java compilers.

Where do Java packages come from?

A project contains only files and folders. The notion of a Java package is introduced by a Java project's class path (at the UI, the Package Explorer presents the packages as defined by the classpath). **Tip:** If the package structure is not what you expect, check out your class path. The Java search infrastructure only finds declarations for and references from Java elements on the class path.

When do I use an internal versus an external JAR library file?

An internal resource resides in some project in the workbench and is therefore managed by the workbench; like other resources, these resources can be version managed by the workbench. An external resource is not part of the workbench and can be used only by reference. For example, a JRE is often external and very large, and there is no need to associate it with a VCM system.

When should I use source folders within a Java project?

Each Java project locates its Java source files via one or more source type entries on the project's class path. Use source folders to organize the packages of a large project into useful grouping, or to keep source code separate from other files in the same project. Also, use source folders if you have files (documentation for example) which need not be on the build path.

What are source attachments, How do I define one?

Libraries are stored as JAR files containing binary class files (and perhaps other resources). These binary class files provide signature information for packages, classes, methods, and fields. This information is sufficient to compile or run against, but contains far less information that the original source code. In order to make it easier to browse and debug binary libraries, there is a mechanism for associating a corresponding source JAR (or ZIP) file with a binary JAR file.

Why are all my resources duplicated in the output folder (bin, for example)?

If your Java project is using source folders, then in the course of compiling the source files in the project, the Java compiler copies non-Java resources to the output folder as well so that they will be available on the class path of the running program. To avoid certain resources to be copied to the output location you can set a resource filter in the Java compiler preferences: **Window > Preferences > Java > Compiler > Building**

How do I prevent having my documentation files from being

copied to the project's output folder?

Use source folders and put any resources that you do not want to be copied to the output folder into a separate folder that is not included on the class path. You can also set a resource filter in the Java compiler preferences: **Window > Preferences > Java > Compiler > Building** to for example *.doc.

How do I create a default package?

You don't have to. Files in the root folder of a source folder or project are considered to be in the default package. In effect, every source folder has the capability of having a fragment of the default package.

What is refactoring?

Refactoring means behavior-preserving program transformations. The JDT supports a number of transformations described in Martin Fowler's book *Refactoring: Improving the Design of Existing Code*, Addison-Wisely 1999.

When do I use Open Declaration (F3)?

To find out the Java element that corresponds to the selected source range with the help of the compiler.

Is the Java program information (type hierarchy, declarations, references, for example) produced by the Java builder? Is it still updated when auto-build is off?

The Java program information is independent from the Java builder. It is automatically updated when performing resource changes or Java operations. In particular, all the functionality offered by the Java tooling (for example, type hierarchies, code assisting, search) will continue to perform accurately when auto-build is off; for example, when doing heavy refactoring which require to turn off the builders, you can still use code assist, which will reflect the latest changes (not yet build). Other than the launching (that is, running and debugging) of programs, the only functionality which requires the Java builder is the evaluation of code snippets.

After reopening a workbench, the first build that happens after editing a Java source file seems to take a long time. Why is that?

The Java incremental project builder saves its internal state to a file when the workbench is closed. On the first build after the project is reopened, the Java incremental project builder will restore its internal state. When this file is large, the user experiences an unusually long build delay.

I can't see a type hierarchy for my class. What can I do?

Check that you have your build class path set up properly. Setting up the proper build class path is an important task when doing Java development. Without the correct build path, you will not be able to compile your code. In addition, you will not be able to search or look at the type hierarchies for Java elements.

How do I turn off "auto compile" and do it manually when I want?

Clear the **Build automatically** checkbox on the [Image: /topic/org.eclipse.help/command_link.png]**General > Workspace** preference page. When you want to build, press `Ctrl+B`, or select **Project > Build All** from the menu bar.

Hint: when you turn "auto compile" off and build manually, you may also want to

select the **Save automatically before build** checkbox on the [Image: /topic/org.eclipse.help/command_link.png]**General > Workspace** preference page.

When I select a method or a field in the Outline view, only the source for that element is shown in the editor. What do I do to see the source of the whole file?

There is a toolbar button **Show Source of Selected Element Only** - all you have to do is un-press it.

Can I nest source folders?

Yes, you can use exclusion filters to create nested source folders.

Can I have separate output folders for each source folder?

Yes, select the **Allow output folders for source folders** checkbox in the **Java Build Path > Source** property page of your Java project.

Can I have an output or source folder that is located outside of the workspace?

Yes, you can create a linked folder that points to the desired location and use that folder as the source or output folder in your Java project.

Related concepts

Java development tools (JDT)

Related reference

Java build path page
JDT glossary

JDT Glossary

CLASS file

A compiled Java source file.

compilation unit

A Java source file.

field

A field inside a type.

import container

Represents a collection of import declarations. These can be seen in the Outline view.

import declaration

A single package import declaration.

initializer

A static or instance initializer inside a type.

JAR file

JAR (Java archive) files are containers for compiled Java source files. They can be associated with an archive (such as, ZIP, JAR) as a source attachment. The children of JAR files are packages. JAR files can be either compressed or uncompressed.

JAVA elements

Java elements are Java projects, packages, compilation units, class files, types, methods and fields.

JAVA file

An editable file that is compiled into a byte code (CLASS) file.

Java projects

Projects which contain compilable Java source code and are the containers for source folders or packages.

JDT

Java development tools. Workbench components that allow you to write, edit, execute, and debug Java code.

JRE

Java runtime environment (for example, J9, JDK, and so on).

method

A method or constructor inside a type.

package declaration

The declaration of a package inside a compilation unit.

packages

A group of types that contain Java compilation units and CLASS files.

refactoring

A comprehensive code editing feature that helps you improve, stabilize, and maintain your Java code. It allows you to make a system-wide coding change without affecting the semantic behavior of the system.

type

A type inside a compilation unit or CLASS file.

source folder

A folder that contains Java packages.

VCM

Version control management. This term refers to the various repository and versioning features in the workbench.

VM

Virtual machine.

Related concepts

Java development tools (JDT)

Related reference

Frequently asked questions on JDT

JDT Icons

Objects

	compilation unit (*.java file)
	Java file which is not on a build path
	class file
	generic file (unknown content type)
	unknown object
	Java scrapbook page (*.jpage file)
	Java scrapbook page (evaluation in progress)
	JAR description file
	JUnit test result file
	Java working set
	Java model
	library container
	JAR file with attached source
	JAR file without attached source
	class folder with attached source
	class folder without attached source
	source folder
	package
	empty package
	logical package
	empty logical package
	package only containing non Java resources
	package declaration
	import container
	import
	class (public)
	interface (public)
	enum type (public)

	annotation type (public)
	package visible class
	private class
	protected class
	default field (package visible)
	private field
	protected field
	public field
	default method (package visible)
	private method
	protected method
	public method

Object adornments

	marks project as Java project
	decorates files and folders if they are on the build path of their enclosing Java project
	decorates Java projects and working sets that contain build path errors
	this Java element causes an error
	this Java element causes a warning
	this Java element is deprecated
	constructor
	abstract member
	final member
	static member
	synchronized member
	native method
	transient field
	volatile field

	type with public static void main(String[] args)
	implements method from interface
	overrides method from super class
	type with focus in Type Hierarchy or Quick Outline/Hierarchy
	maximal expansion level in Call Hierarchy
	recursive call in Call Hierarchy
	compilation unit containing an abstract class as primary type
	compilation unit containing an interface as primary type
	compilation unit containing an enum as primary type
	compilation unit containing an annotation as primary type

Build path

	class path variable
	JAR with attached source
	JAR without attached source
	system library
	build path ordering
	inclusion filter
	exclusion filter
	access rules
	Javadoc location
	source attachment
	native library location
	output folder

Code assist

	HTML tag
	Javadoc tag
	local variable
	template
	SWT template

Compare

	field
	method

Debugger

	debug launch
	run launch
	terminated run launch
	process
	terminated process
	debug target
	suspended debug target
	terminated debug target
	thread
	suspended thread
	stack frame
	running stack frame
	adornment that marks a stack frame that may be out of synch with the target VM as a result of an unsuccessful hot code replace
	adornment that marks a stack frame that is out of synch with the target VM as a result of an unsuccessful hot code replace
	inspected object or primitive value
	watch expression
	local variable
	monitor
	a monitor in contention

	a thread in contention for a monitor
	a monitor that is owned by a thread
	a thread that owns a monitor
	current instruction pointer (top of stack)
	current instruction pointer
	enabled line breakpoint
	disabled line breakpoint
	adornment that marks a breakpoint as skipped
	adornment that marks a line breakpoint as installed
	adornment that marks a breakpoint as conditional
	adornment that marks an entry method breakpoint
	adornment that marks an exit method breakpoint
	field access watchpoint
	field modification watchpoint
	field access and modification watchpoint
	adornment that marks a watchpoint as installed
	exception breakpoint
	runtime exception breakpoint
	disabled exception breakpoint
	adornment that marks an exception breakpoint as caught
	adornment that marks an exception breakpoint as uncaught
	adornment that marks an exception breakpoint as scoped
	adornment that marks an exception breakpoint as installed

Editor

	implements
	overrides
	quick assist available
	search match

JUnit

	test
	currently running test
	successful test
	failing test
	test throwing an exception
	ignored test
	test suite
	currently running test suite
	successfully completed test suite
	test suite with failing test
	test suite with exception throwing test
	caught exception
	stack frame element

NLS tools

	skipped NLS key
	translated NLS key
	untranslated NLS key

Quick fix

	quick fixable error
	quick fixable warning
	add
	change
	change cast
	fix multiple problems
	move to another package
	remove

	remove import
	rename
	rename in file
	surround with try/catch

Refactoring

	general change
	composite change
	text change
	file change
	Stop error
	Error
	Warning
	Information
	Change filter

Search

	Java Search
	search for declarations
	search for references

Search - Occurrences in File

	a general match
	read access to local or field
	write access to local or field

Type Hierarchy view

	type outside of selected package
	interface outside of selected package

List of JDT key bindings

The list of available key bindings in Eclipse depends on many factors, including what view or editor is selected, whether a dialog is open, what plug-ins are installed, and what operating and windowing system is being used. At any time, you can obtain a list of available key bindings using Key Assist ([Image: /topic/org.eclipse.help/command_link.png]**Help > Key Assist...** or Ctrl+Shift+L). The following tables list some popular key bindings available in the Java development tools.

Java editor actions

Add Block Comment	Ctrl+Shift+/
Add Import	Ctrl+Shift+M
Content Assist	Ctrl+Space
Content Assist (Simplified Chinese)	Alt+/
Expand Selection to Enclosing Element	Alt+Shift+Up Arrow
Expand Selection: Restore Last Selection	Alt+Shift+Down Arrow
Expand Selection to Next Element	Alt+Shift+Right Arrow
Expand Selection to Previous Element	Alt+Shift+Left Arrow
Format	Ctrl+Shift+F
Next	Ctrl+. (Period)
Open External Javadoc	Shift+F2
Open on Selection	F3
Open Type	Ctrl+Shift+T
Open Type Hierarchy	F4
Open Type in Hierarchy	Ctrl+Shift+H
Organize Imports	Ctrl+Shift+O
Parameter Hints	Ctrl+Shift+Space
Parameter Hints (Simplified Chinese)	Alt+?
Previous	Ctrl+, (Comma)
Quick Assist	Ctrl+1
Quick Fix	Ctrl+1
Redo	Ctrl+Y
Remove Block Comment	Ctrl+Shift+\
Search for Declarations in Workspace	Ctrl+G

Search for References in Workspace	Ctrl+Shift+G
Shift Right	Tab
Shift Left	Shift+Tab
Show Tooltip Description	F2
Toggle Comment	Ctrl+/
Undo	Ctrl+Z

Debug actions

Debug Last Launched	F11
Display	Ctrl+Shift+D
Inspect	Ctrl+Shift+I
Resume	F8
Run Last Launched	Ctrl+F11
Run Snippet	Ctrl+U
Run to Line	Ctrl+R
Run to Return	F7
Step Into	F5
Step into Selection	Ctrl+F5
Step Over	F6
Toggle Breakpoint	Ctrl+Shift+B

Related concepts

Navigating the user interface using the keyboard
Keys and accessibility for the workbench

Related reference

List of key bindings

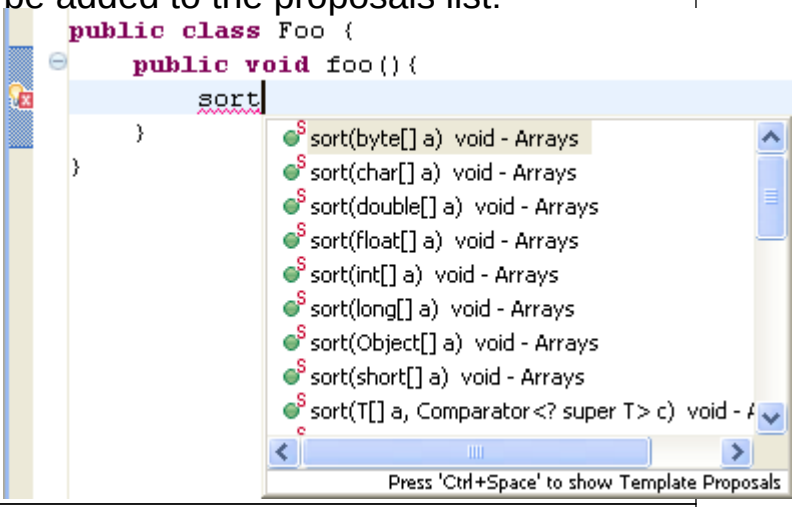
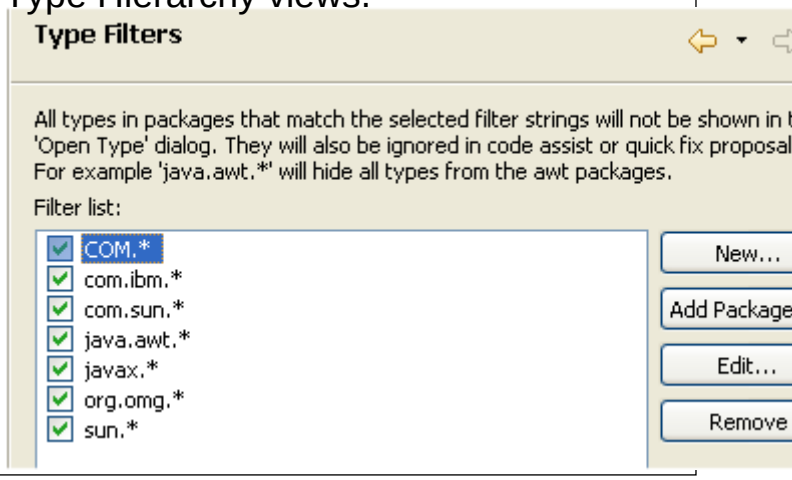
Tips and Tricks (JDT)

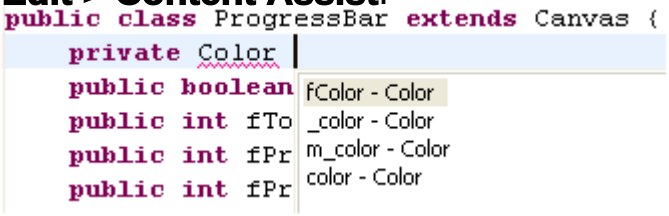
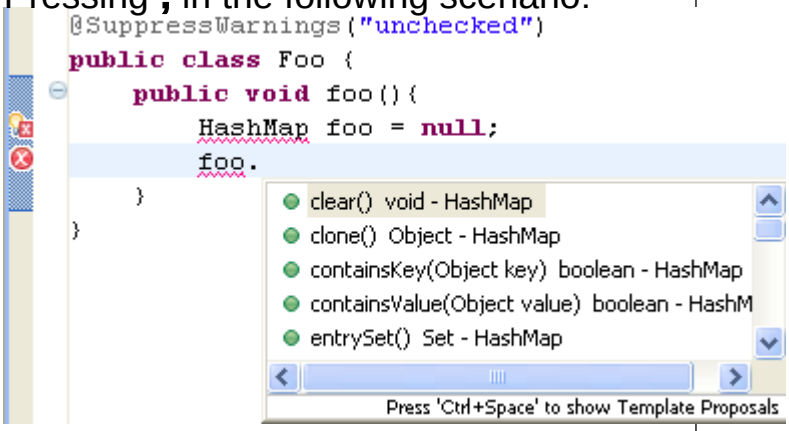
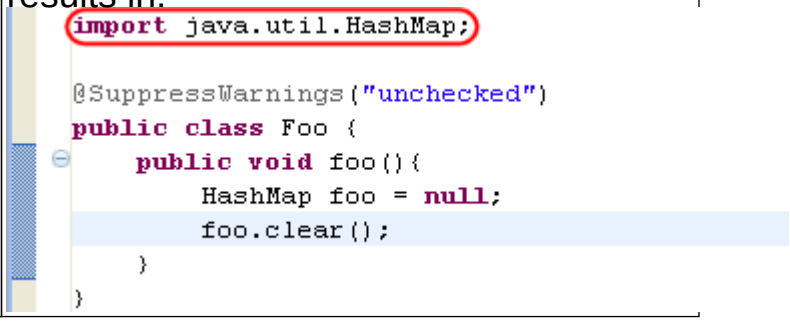
The following tips and tricks give some helpful ideas for increasing your productivity. See also Platform Tips and Tricks for general Eclipse tips and What's New in 3.6 (JDT) for features in this release.

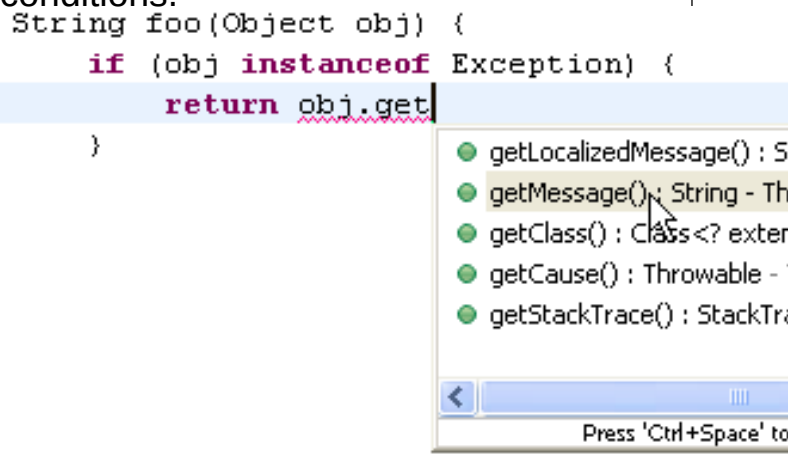
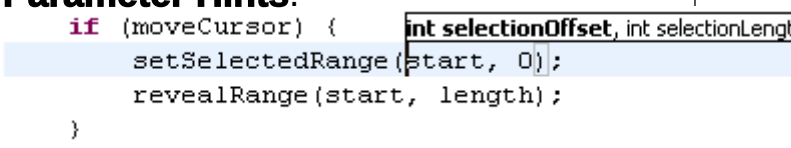
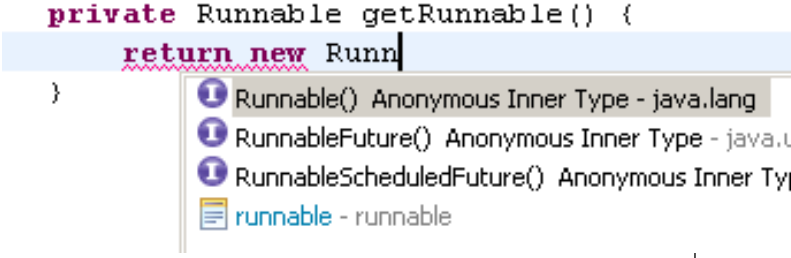
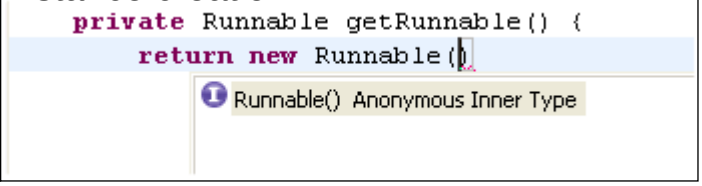
Editing | Refactoring | Searching | Navigation | Views | Miscellaneous | Debugging

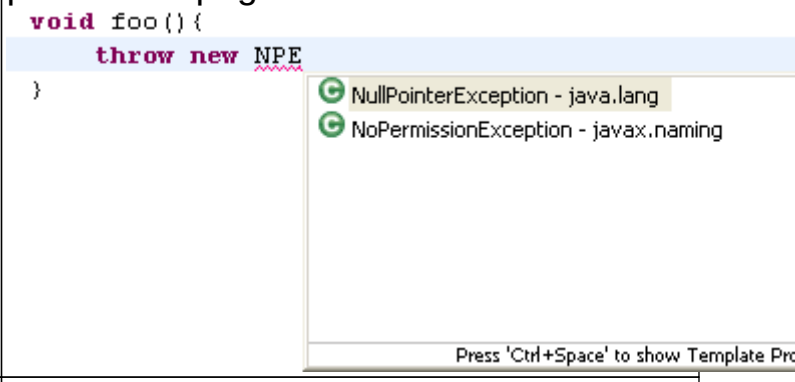
Editing source

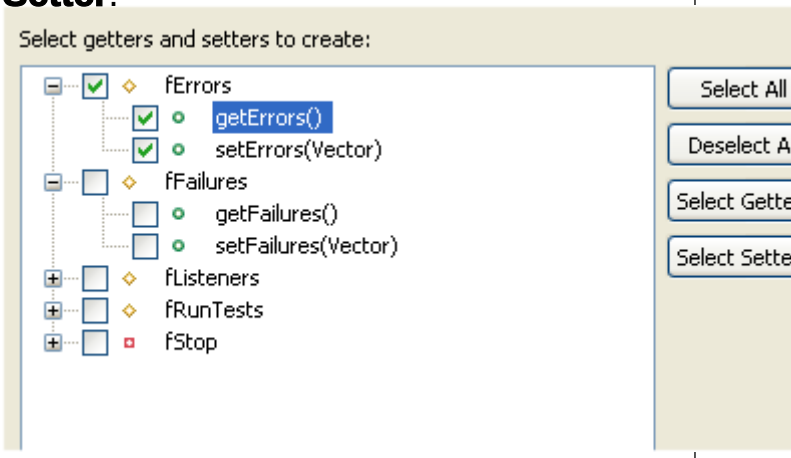
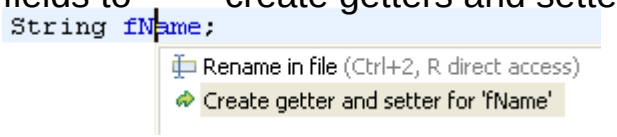
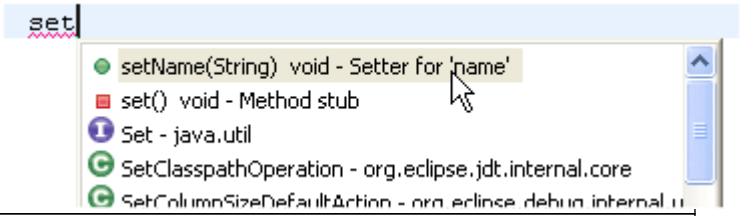
Content assist	Content assist provides you with a list of suggested completions for partially entered strings. In the Java editor press **Ctrl+Space** or use **Edit > Content Assist**.
Content assist in Javadoc comments	Content assist is also available in Javadoc comments. 

Content assist for static imports	To get content assist proposals for static members configure your list of favorite static members on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist > Favorites** preference page. For example, if you have added `java.util.Arrays.*` to this list, then all static methods of this type matching the completion prefix will be added to the proposals list: 
Suppress types in content assist	To exclude certain types from appearing in content assist, use the type filter feature configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Appearance > Type Filters** preference page. Types matching one of these filter patterns will not appear in the Open Type dialog and will not be available to content assist, quick fix and organize imports. These filter patterns do not affect the Package Explorer and Type Hierarchy views. 

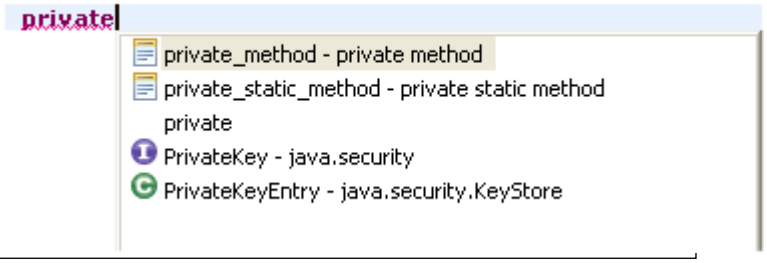
Content assist for variable, method parameter and field name completions	You can use content assist to speed up the creation of fields, method parameters and local variables. With the cursor positioned after the type name of the declaration, press **Ctrl+Space** or use **Edit > Content Assist**.  If you use a name prefix or suffix for fields, local variables or method parameters, be sure to specify this in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style** preference page.
Content assist for variable with unresolved type	Code assist also works on accesses to types that are not imported yet. Depending on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist > Add import instead of qualified name** preference the editor will either automatically add imports or fully qualify the types for such proposals. Pressing **;** in the following scenario:  results in: 

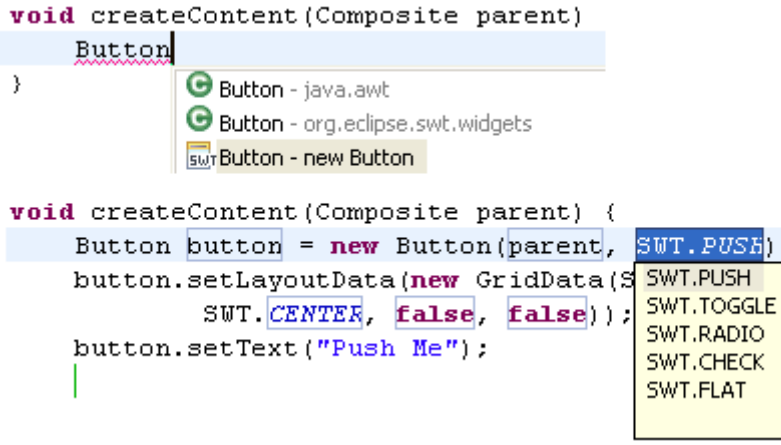
Content assist after instanceof condition	Content assist can propose members available on types used in instanceof conditions.  Eclipse will add the required cast for you when you select such a proposal.
Parameter hints	With the cursor in a method argument, you can see a list of parameter hints. In the Java Editor press **Ctrl+Shift+Space** or invoke **Edit > Content Assist > Parameter Hints**. 
Content assist on anonymous classes	Content assist also provides help when creating an anonymous class. With the cursor positioned after "new " and the beginning of an abstract class or interface name press **Ctrl+Space** or invoke **Edit > Content Assist > Default**.  This will create the body of the anonymous inner class including all methods that need to be implemented. This also works if you place the caret after the opening parentheses of a class instance creation: 

Toggle between inserting and replacing content assist	When content assist is invoked on an existing identifier, it can either replace the identifier with the chosen completion or do an insert. The default behavior (overwrite or insert) is defined in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist** preference page. You can temporarily toggle the behavior while inside the content assist selection dialog by pressing and holding the **Ctrl** key while selecting the completion.
Incremental content assist	Content assist can also **Insert common prefixes automatically**, similar to Unix shell expansion. To enable that behavior, select the check box on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist** preference page.
Camel case support in code completion	Code completion supports camel case patterns. For example, completing on NPE or NuPoiE will propose NullPointerException. This support can be disabled using the **Show camel case matches** preference on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist** preference page. 
Customize content assist categories	Repeatedly invoking content assist **(Ctrl+Space)** cycles through different proposal categories. To configure which categories to show use the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist > Advanced** preference page. You can also assign separate key shortcuts to your favorite proposal categories.

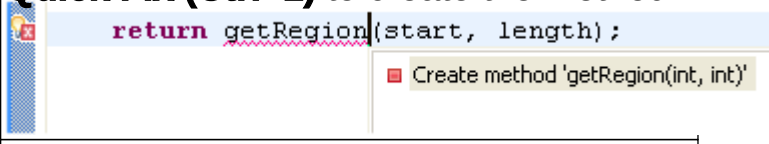
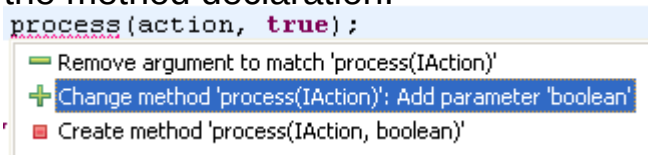
Create getters and setters	To create getter and setter methods for a field, select the field's declaration and invoke **Source > Generate Getter and Setter**.  If you use a name prefix or suffix be sure to specify this in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style** preference page.
Create getters and setters quick assist	A quick assist **(Ctrl+1)** is available on fields to create getters and setters. 
Use content assist to create getter and setters	Another way to create getters and setters is using content assist. Set the cursor in the type body between members and press **Ctrl+Space** to get the proposals that create a getter or setter method stub. 
Delete getters and setters together with a field	When you delete a field from within a view, Eclipse can propose deleting its Getter and Setter methods. If you use a name prefix or suffix for fields, be sure to specify this in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style** preference page.

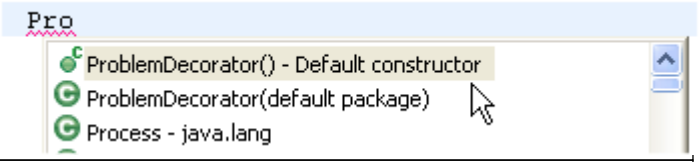
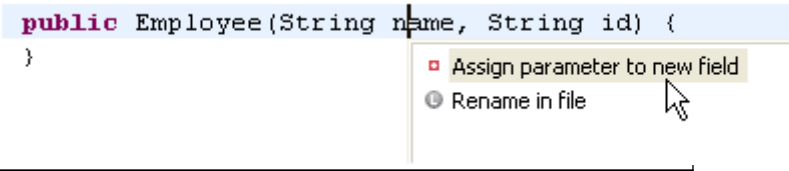
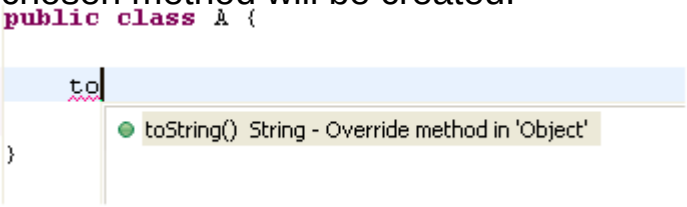
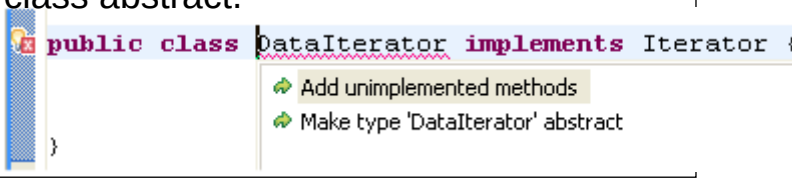
Create delegate methods	To create a delegate method for a field select the field's declaration and invoke **Source > Generate Delegate Methods**. This adds the selected methods to the type that contains a forward call to delegated methods. This is an example of a delegate method:
	`public void addModifyListener(ModifyLister`
	`fTextControl.addModifyListener(listene`
	`}`

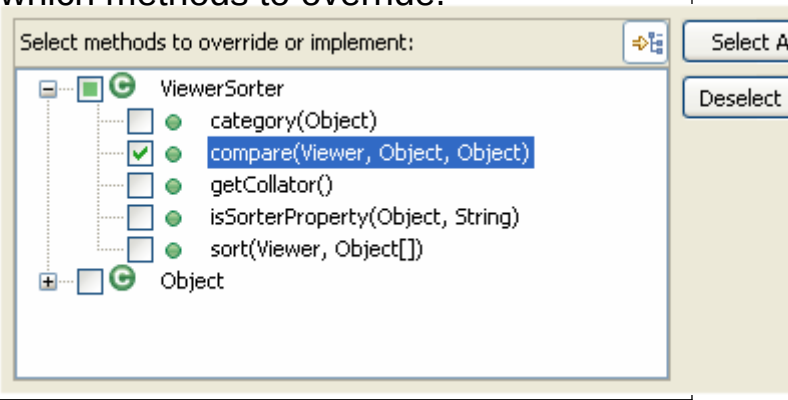
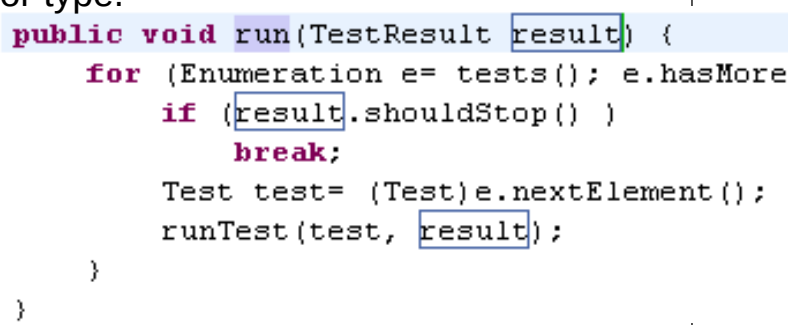
Create `hashCode()` and `equals()`	To create the methods `hashCode()` and `equals()` invoke **Source > Generate hashCode() and equals()**.
Use templates to create a method	Templates are shown together with the **Content Assist (Ctrl+Space)** proposals. There are existing templates, such as 'private_method', 'public_method', 'protected_method' and more, but you can also define new templates for method stubs. After applying a template, use the **Tab** key to navigate among the values to enter (return type, name and arguments). 

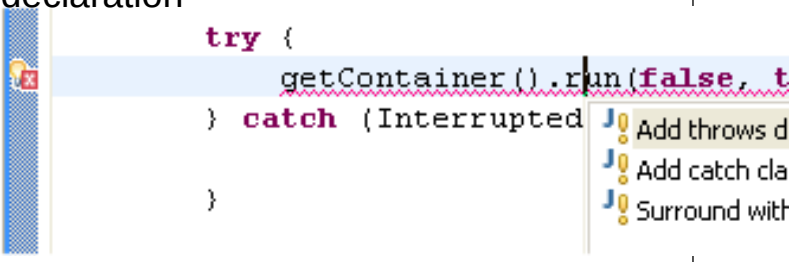
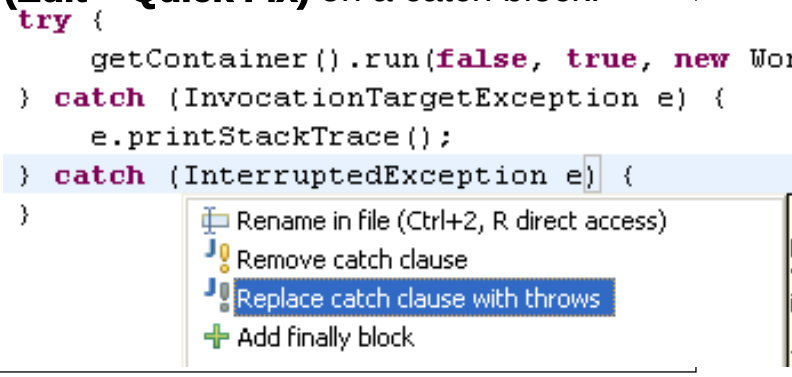
Use templates to create SWT widgets	On projects which have the SWT library on the classpath, you can create SWT widgets with **Content Assist (Ctrl+Space)**To add, for example, an SWT button, type *Button* and press **Ctrl+Space**, select the *Button* SWT template, and press **Enter**.  To see all available templates go to the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Templates** preference page or open the Templates view through **Window > Show View > Other...**.

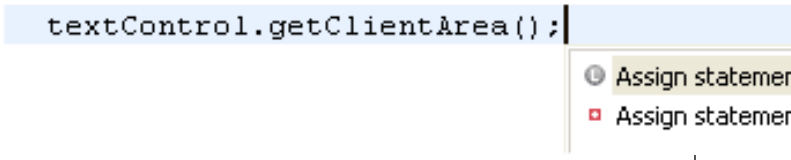
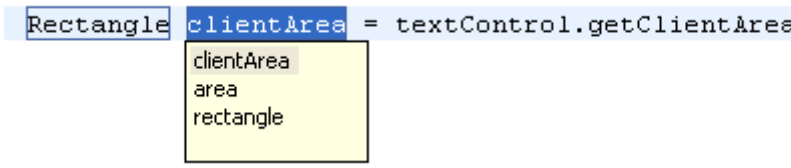
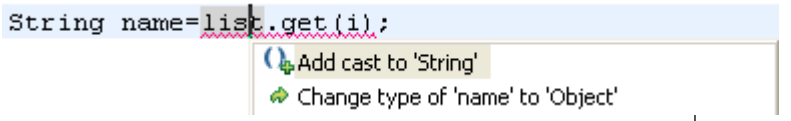
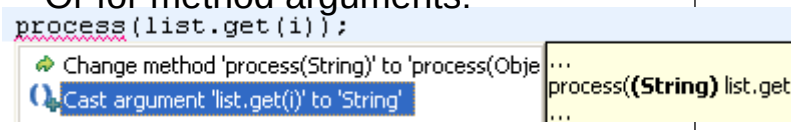
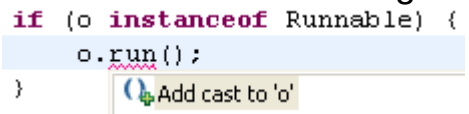
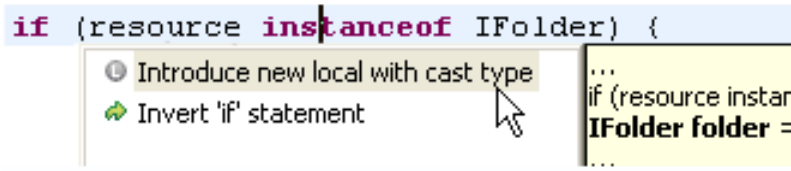
Create your own templates	To create your own templates, go to the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Templates** preference page and press the **New** button to create a template. For example, a template to iterate backwards in an array would look like this:
	`for (int ${index}= ${array}.length - 1;${index} >`
	`${cursor}`
	`}`

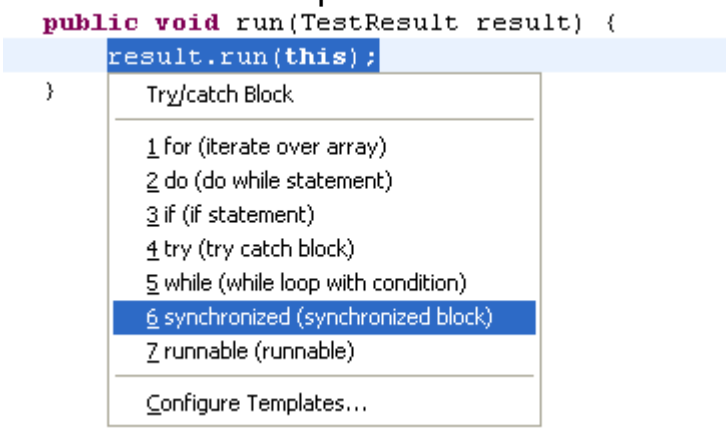
Use Quick Fix to create a new method	Start with the method invocation and use **Quick Fix (Ctrl+1)** to create the method. 
Use Quick Fix to change a method signature	Add an argument to a method invocation at a call site. Then use **Quick Fix (Ctrl+1)** to add the required parameter in the method declaration. 

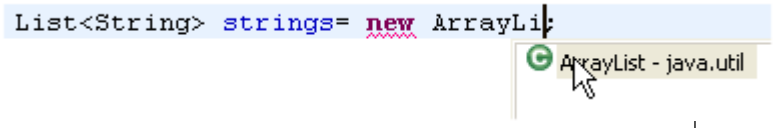
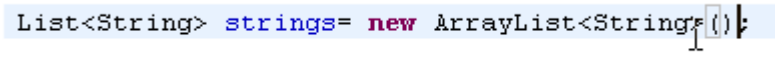
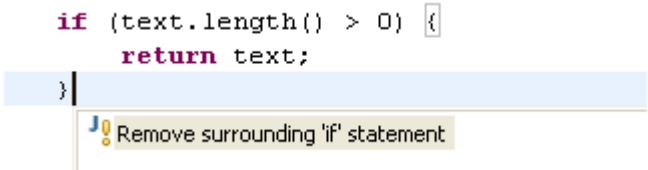
Use content assist to create a constructor stub	At the location where you want to add the new constructor, use content assist after typing the first letters of the constructor name. 
Create new fields from parameters	Do you need to create new fields to store the arguments passed in the constructor? Use **Quick Assist (Ctrl+1)** on a parameter to create the assignment and the field declaration and let Eclipse propose a name according to your Code Style preferences. 
Use content assist to override a method	Invoke **Content Assist (Ctrl+Space)** in the type body at the location where the method should be added. Content assist will offer all methods that can be overridden. A method body for the chosen method will be created. 
Use Quick Fix to add unimplemented methods	To implement a new interface, add the 'implements' declaration first to the type. Even without saving or building, the Java editor will underline the type to signal that methods are missing and will show the Quick Fix light bulb. Click on the light bulb or press **Ctrl+1 (Edit > Quick Fix)** to choose between adding the unimplemented methods or making your class abstract. 

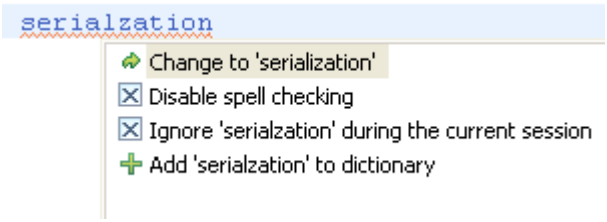
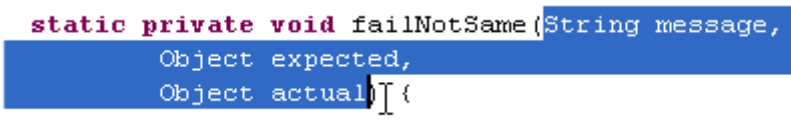
Use Clean Up to add unimplemented methods	When you add a new method to an interface or an abstract method to an abstract class, Eclipse can generate method stubs in all concrete subclasses at once. Invoke **Source > Clean Up...** on a set of Java elements, use a custom profile, and select on the **Configure...** dialog to **Add unimplemented methods** on the **Missing Code** tab.
Override a method from a base class	To create a method that overrides a method from a base class: Select the type where the methods should be added and invoke **Source > Override/Implement Methods**. This opens a dialog that lets you choose which methods to override. 
Rename in file	To quickly do a rename that doesn't require full analysis of dependencies in other files, use the 'Rename in file' Quick Assist. In the Java Editor, position the cursor in an identifier of a variable, method or type and press **Ctrl+1 (Edit > Quick Fix)** The editor is switched to the linked edit mode (like templates) and changing the identifier simultaneously changes all other references to that variable, method or type.  You can also use the direct shortcut **Ctrl+2 R**. Use the [Image: /topic/org.eclipse.help/command_link.png]**General > Keys** preference page to configure shortcuts (in the 'Source' category).

Use Quick Fix to handle exceptions	Dealing with thrown exceptions is easy. Unhandled exceptions are detected while typing and marked with a red line in the editor. Click on the light bulb or press **Ctrl+1** to surround the call with a try catch block. If you want to include more statements in the try block, select the statements and use **Source > Surround With > Try/catch Block**. You can also select individual statements by using **Edit > Expand Selection To** and selecting **Enclosing**, **Next** or **Previous**.If the call is already surrounded with a try block, Quick Fix will suggest adding the catch block to the existing block.If you don't want to handle the exception, let Quick Fix add a new thrown exception to the enclosing method declaration  At any time, you can convert a catch block to a thrown exception. Use **Ctrl+1 (Edit > Quick Fix)** on a catch block. 

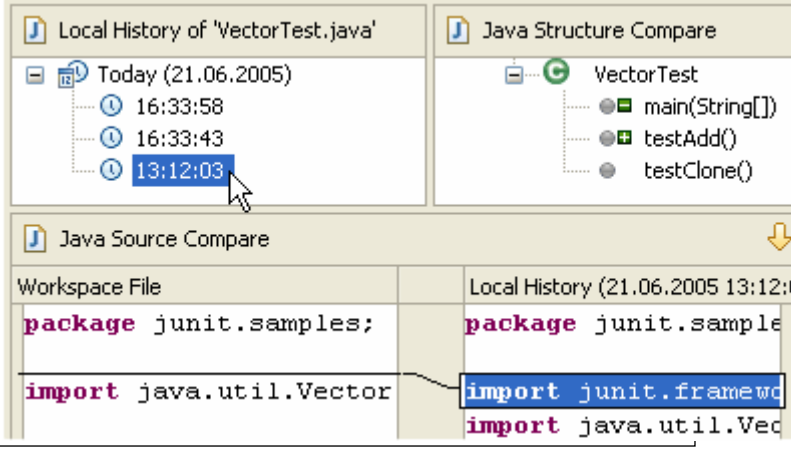
Less typing for assignments	Instead of typing an assignment, start with the expression that will be assigned.  Now use **Ctrl+1 (Edit > Quick Fix)** and choose '**Assign statement to new local variable**' and Quick Assist will guess a variable name for you. 
Less work with cast expressions	Don't spend too much time with typing casts. Ignore them first and use quick assist to add them after finishing the statement. For example on assignments:  Or for method arguments:  Or for method call targets: 
Assign a cast expression	After an 'instanceof' check, it is very common to cast the expression and assign it to a new local variable. Invoke **Quick Assist (Ctrl+1)** on the 'instanceof' keyword or at the beginning of the block body to save yourself some typing: 

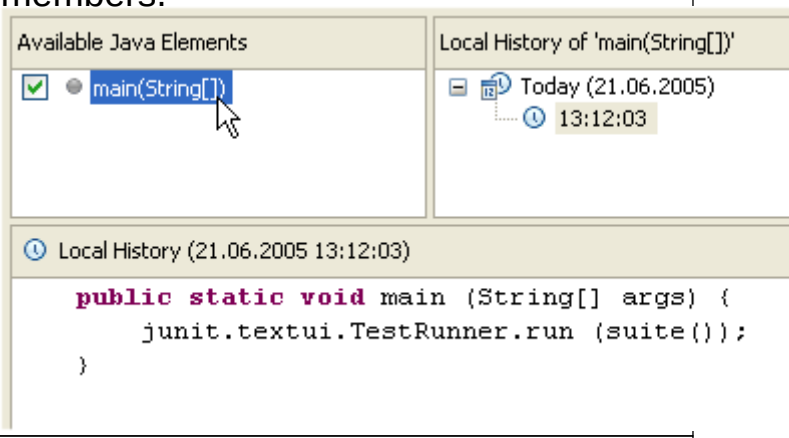
Surround lines	To surround statements with an if / while / for statement or a block, select the lines to surround and invoke **Source > Surround With** or press **Alt+Shift+Z**.  The entries in the menu are derived from the normal editor templates: All templates that contain the variable ${line_selection} will show up in the menu. Templates can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Templates** preference page. Edit the corresponding templates to customize the resulting code or define your own surround-with templates.
More Quick Assists and Fixes	Check out the Quick Assist page for a complete list of available code transformations. A list of quick fixes can be found here.
Shortcuts for Quick Fixes and Assists	Some of the popular quick assists like **Rename in file** and **Assign to local variable** can be invoked directly with **Ctrl+2 R** and **Ctrl+2 L**. Check the [Image: /topic/org.eclipse.help/command_link.png]**General > Keys** preference page for more quick fixes and quick assists that support direct invocation. Type "Quick Assist" or "Quick Fix" in the filter field:

Content assist can insert argument names automatically	You can have content assist insert argument names automatically on method completion. This behavior can be customized on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist** preference page (see the **Fill method arguments and show guessed arguments** checkbox.) For example, when you select the second entry here,  content assist will automatically insert argument names:  You can then use the **Tab** key to navigate between the inserted names. If you choose **Insert best guessed arguments**, the best guess will be filled in by default. The alternative proposals are still available.
Automatically insert type arguments	Enabling **Fill method arguments and show guessed arguments** on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Content Assist** preference page is also useful when working with parameterized types in J2SE 5.0.  results in: 
Remove surrounding statement	To remove a surrounding statement or block, position the cursor at the opening or closing bracket and press **Ctrl+1 (Edit > Quick Fix)**. 

How was that word spelled again?	You can enable spell-checking support in the Java editor on the [Image: /topic/org.eclipse.help/command_link.png]**General > Editors > Text Editors > Spelling** preference page. Spelling errors are displayed in the Java editor and corresponding Quick Fixes are available:  You can make the dictionary also available to the content assist. A Quick Fix allows you to add new words to the user dictionary on the fly.
Structured selections	The Structured Selection actions can be used to enlarge the current selection to the next enclosing element: Highlight some text and press **Alt+Shift+Arrow Up** or select **Edit > Expand Selection To > Enclosing Element** from the menu bar - the selection will be expanded to the smallest Java-syntax element that contains the selection. You can then further expand the selection by invoking the action again (or other actions from the Expand Selection To menu). This is for example helpful to select the enclosing identifier for renames, or to select adjacent statements for a subsequent Extract Method refactoring.
Find the matching bracket	To find a matching bracket select an opening or closing bracket and press **Ctrl+Shift+P** or select **Navigate > Go To > Matching Bracket**. You can also **double click** before an opening or after a closing bracket - this selects the text between the two brackets. 

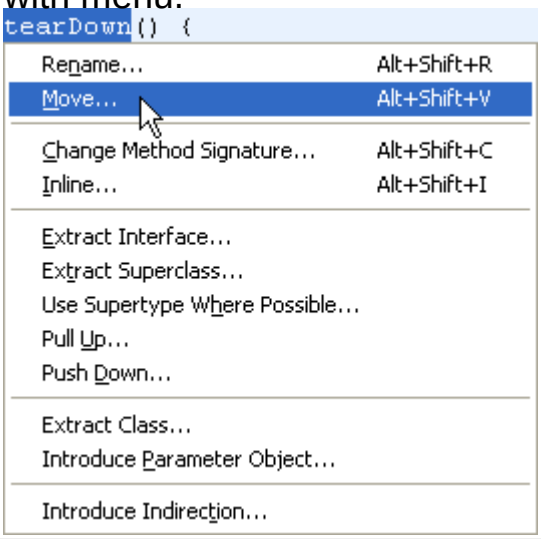
Smart Javadoc	Type '/**' and press **Enter**. This automatically adds a Javadoc comment stub containing the standard @param, @return and @exception tags.
	`/**`
	` * `
	` * @param condition`
	` */`
	`static public void assertTrue(boolean`
	`assertTrue(null, condition);`
	`}`
	The templates for the new comment can be configured in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Code Templates** preference page.

Use the local history to revert back to a previous edition of a method	Whenever you edit a file, its previous contents are kept in the local history. Java tooling makes the local history available for Java elements, so you can revert back to a previous edition of a single method instead of the full file. Select an element (e.g. in the Outline view) and use **Replace With > Local History** to revert back to a previous edition of the element. 

Use the local history to restore removed methods	Whenever you edit a file, its previous contents are kept in the local history. Java tooling makes the local history available for Java elements, so you can restore deleted methods selectively. Select a container (e.g. in the Outline view) and use **Restore from Local History** to restore any removed members. 
Customizable code generation	The [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Code Templates** preference page allows you to customize generated code and comments in a similar way to normal templates. These code templates are used whenever code is generated. Since 3.1, it is possible to use project specific Code templates, that will also be shared in the team if your project is shared. Open the **Properties** on a project to enable project specific settings.

Create comments in your code	Comments can be added explicitly with **Source > Generate Element Comment (Alt+Shift+J)** or automatically by various wizards, refactorings or quick fixes. Configure the comment templates on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Code Templates** preference page. Enable or disable the automatic generation of comments either directly on the wizard (e.g. using **'Generate Comment'** checkbox on the new Java type wizards) or by the **'Automatically add new comments for new methods and types'** checkbox of the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Code Templates** preference page. All these settings can also be configured on a per project basis. Open the **Properties** on a project to enable project specific settings.
Sort members	You can **Sort Members** of a Java compilation unit or a set of compilation units according to a category order defined in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Appearance > Members Sort Order** preference page. You'll find the action under **Source > Sort Members**.

Wrap strings	You can have String literals wrapped when you edit them. For example, if you have code like this: `String message= "This is a very long message.";` position your caret after the word "very" and press **Enter**. The code will be automatically changed to:
	`String message= "This is a very" +`
	`" long message.";`
	This behavior can be customized in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Typing** preference page.

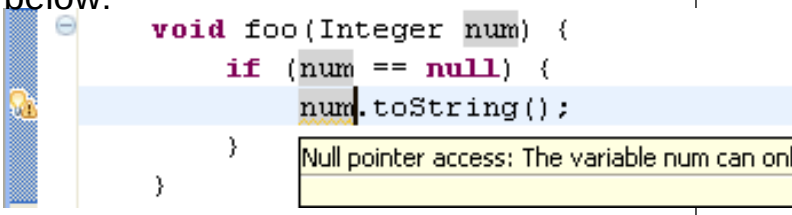
Smart Typing and how to control it	The Java editor's Smart Typing features ease your daily work. You can configure them on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Typing** preference page. When you enable **Automatically insert Semicolons at correct position**, typing a semicolon automatically positions the cursor at the end of the statement before inserting the semicolon. This saves you some additional cursor navigation. You can undo this automatic positioning by pressing backspace right afterwards.
Fix your code indentation with one key stroke	A useful feature is **Source > Correct Indentation** or **Ctrl+I**. Select the code where the indents are incorrect and invoke the action. If nothing is selected, the action indents the current line.
Fix your code indentation on save	Eclipse can correct the indentation of your code when you save the editor. Go to the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Save Actions** preference page and **Configure...Additional actions**, and select to **Correct indentation** on the **Code Organizing** tab.
Quick menus for source and refactoring actions	The refactoring and source actions can be accessed via a quick menu. Select the element to be manipulated in the Java editor or in a Java view and press **Alt+Shift+S** for the quick source menu, **Alt+Shift+T** for the quick refactoring menu and **Alt+Shift+Z** for the surround with menu. 

Find unused code	The Java compiler detects unreachable code, unused variables, parameters, imports and unused private types, methods and fields. Change the settings for the detection on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Error/Warnings** preference page. These settings can also be specified per project, use: **Project > Properties > Java Compiler > Error/Warnings**. These problems are detected as you type, and a quick fix is offered to remove the unneeded code. You can also use **Source > Clean Up...** to remove unused code.

Find problems with null

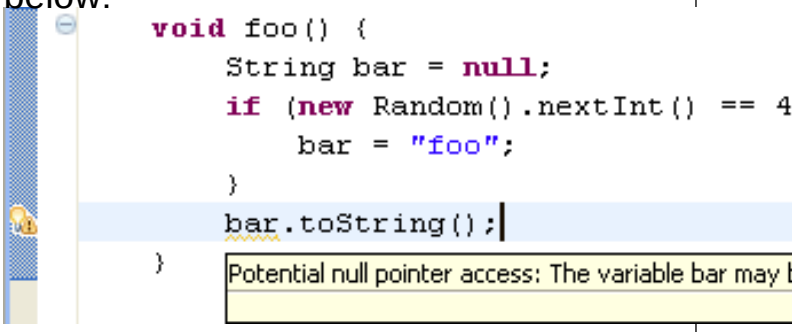
The compiler can help you find problems with `null` in your code. The [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page has three options to detect problems: **Null pointer access** (in 'Potential programming problems')

When this option is enabled, the compiler will issue an error or warning whenever a variable that is statically known to hold a null value is used to access a field or method, as shown in the example below:



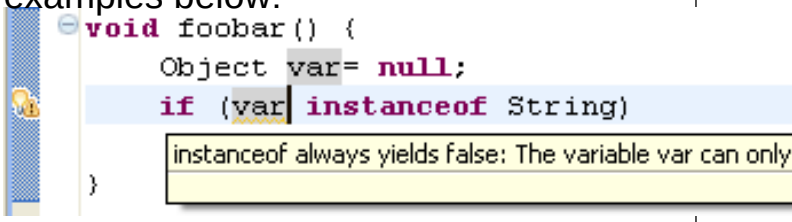
Potential null pointer access (in 'Potential programming problems')

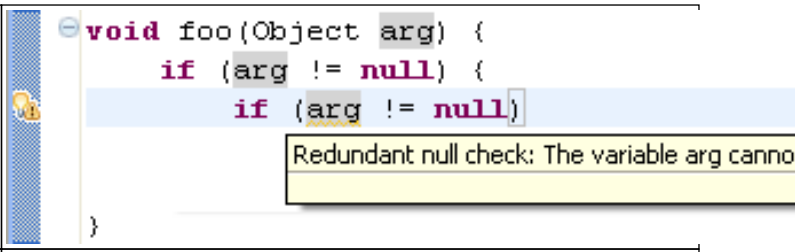
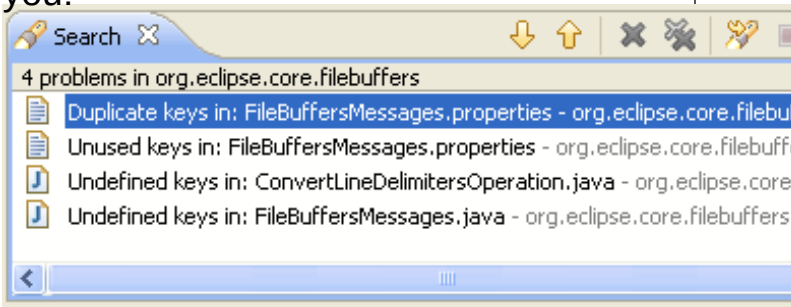
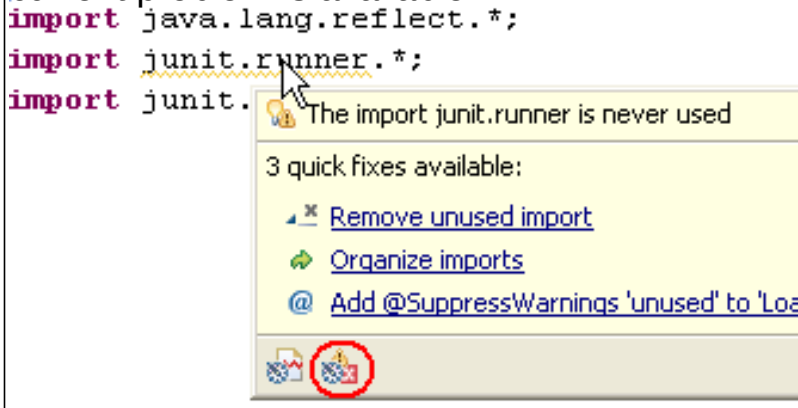
When this option is enabled, the compiler will issue an error or a warning whenever a variable is statically known to potentially hold a null value, as shown in the example below:

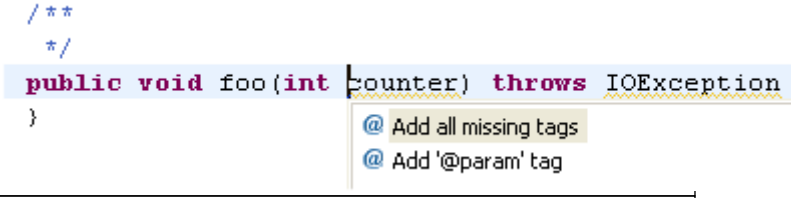
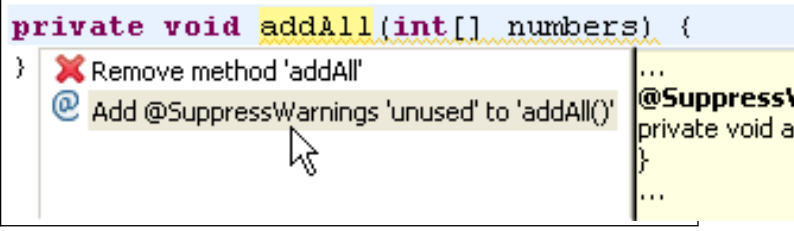


Redundant null check (in 'Unnecessary code')

When enabled, the compiler will issue an error or a warning whenever a variable that is statically known to hold a null or a non-null value is tested against null, as shown in the examples below:



	
Find problems with externalized strings	The **Source > Find Broken Externalized Strings** action finds undefined, unused and duplicate keys for you: 
Change problem severity in problem hover	When a problem hover has been enriched (by pressing F2 or moving the mouse into the hover), an action to change the severity of the current problem is available.  Click on the button in the toolbar to go to the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page.

Javadoc comment handling	The Eclipse Java compiler can process Javadoc comments. Search reports references in doc comments, and refactoring updates these references as well. This feature is controlled from the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Javadoc** preference page (or set for an individual project using **Project > Properties > Java Compiler > Javadoc**). When turned on, malformed Javadoc comments are marked in the Java editor and can be fixed using **Edit > Quick Fix (Ctrl+1)**: 
Suppress warnings	In J2SE 5.0 and later, you can suppress all optional compiler warnings using the `SuppressWarnings` annotation. In this example, `addAll()` is marked as an unused method. **Quick Fix (Ctrl+1)** is used to add a `SuppressWarnings` annotation so that the warning will not be shown for this method. 

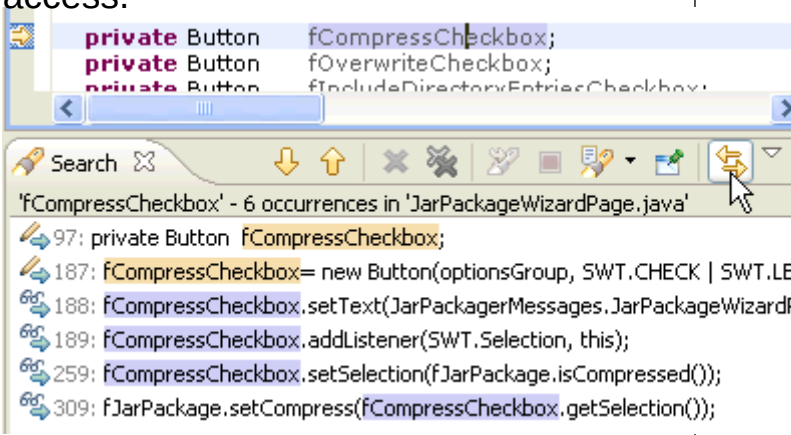
Clean Ups	**Source > Clean Up...** helps fixing multiple problems at once and helps to establish a consistent code style. For instance, you can: convert all `for` loops to enhanced `for` loops where possible. mark all overriding methods in a whole project with an `@Override` annotation. organize importsformat your coderemove unnecessary codeClean Ups are organized in Clean Up profiles. A profile can be attached to the workspace or to individual projects. Project settings can be shared in a team through a version control system. It is also possible to export and import each profile. Clean Ups can be executed as save actions on save. Go to the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Save Actions** preference page to configure which clean ups to invoke on save.

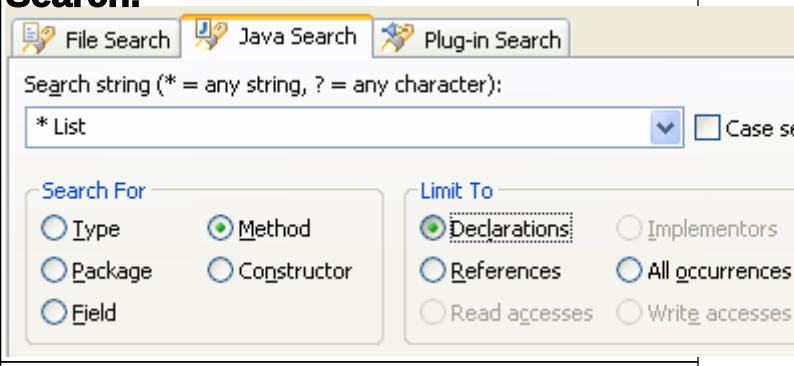
Refactoring

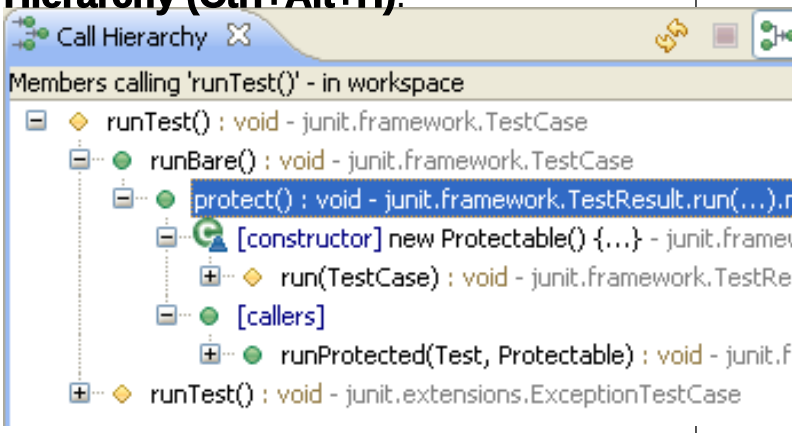
Scripting of refactorings	Most of the refactorings offered by JDT can not only be executed interactively, but also by a refactoring script. Create a refactoring script from the refactoring history using **Refactor > Create Script...**. A refactoring script can then be applied later on an arbitrary workspace using **Refactor > Apply Script...**. Such refactoring scripts can be used in different scenarios such as automatic fixing of breaking API changes between software layers or providing patches with rich semantics.

Safe JAR file migration	When exporting a JAR file from the workspace, the JAR Export Wizard offers the option to include refactoring information into the JAR file. Use **File > Export...** and select **JAR file**. On the first page of the JAR Export Wizard, select **Export refactorings for checked projects**. Click on the link to select the refactorings to include. Clients are then able to migrate an old version of the JAR file to a new one using the **Refactor > Migrate JAR File...** refactoring. This refactoring automatically updates all code which is dependent on the old version of the JAR file to use the new version of the JAR file.

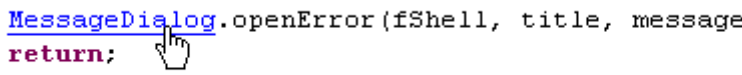
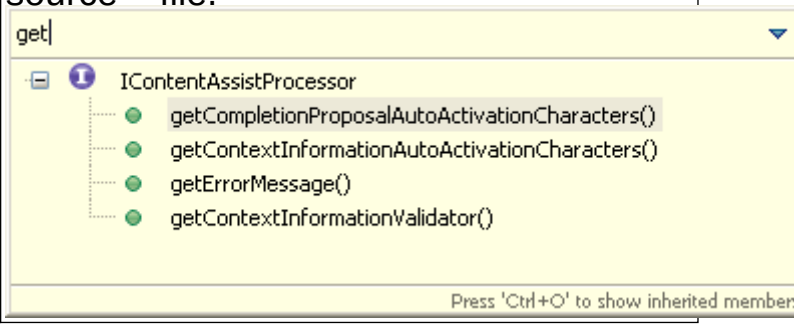
Searching

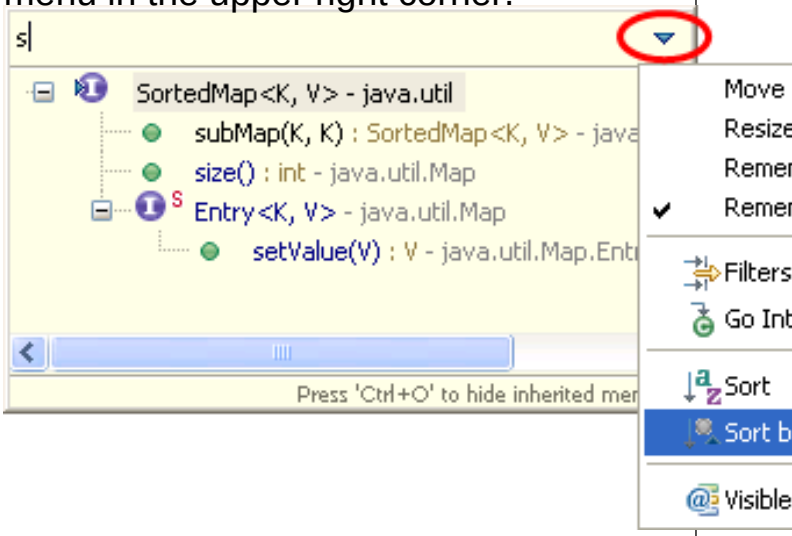
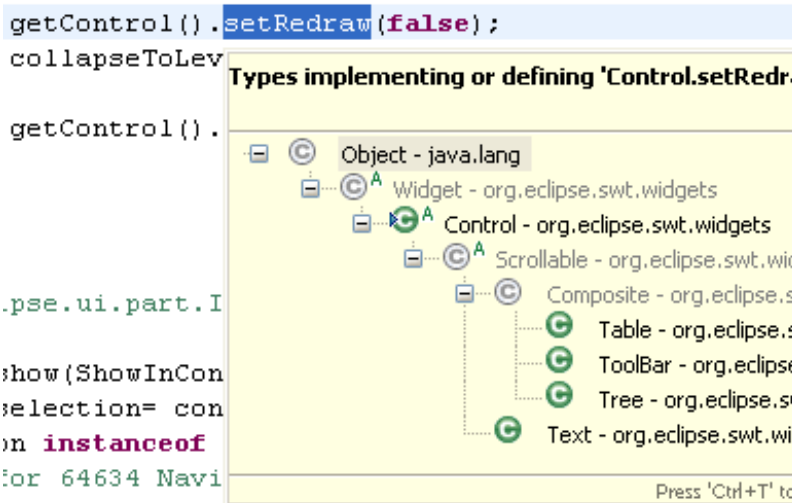
Locate variables and their read/write access	You can locate variables and see their read/write status by selecting an identifier (variable, method or type reference or declaration) and invoking **Search > Occurrences in File > Identifier**. This marks all references of this identifier in the same file. The results are also shown in the search view with different colors for read or write access, along with icons showing the variable's read or write access.  Alternatively, use the **Mark Occurrences** feature to dynamically highlight occurrences. You can search over several files by using the general search features (**Search > References**).

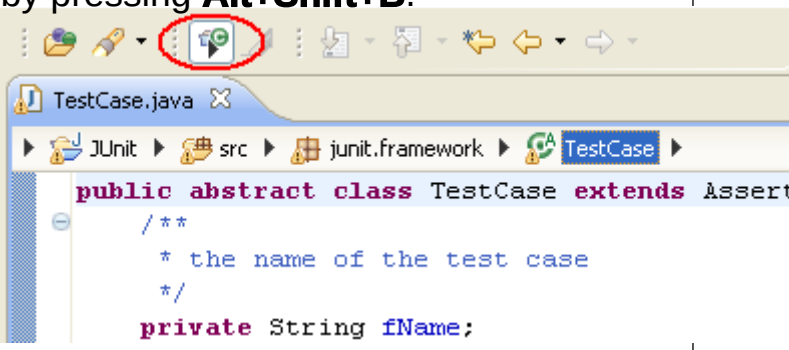
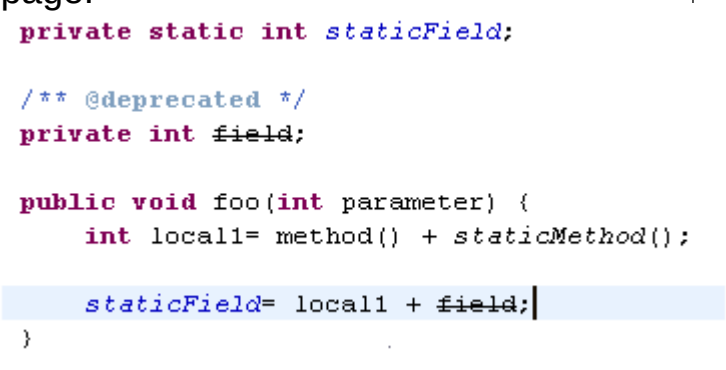
Search for methods with a specific return type	To search for methods with a specific return type, use "* <return type>" as follows: Open the search dialog and click on the **Java Search** tab.Type '*' and the return type, separated by a space, in the **Search string**.Select the **Case sensitive** checkbox.Select **Method** and **Declarations** and then click **Search.** 
Filter search matches in Javadoc	By default, **Java Search** finds references inside Java code and Javadoc. If you don't want to see the references inside Javadoc, you can filter these matches by enabling 'Filter Javadoc' in the view menu (triangle symbol) of the search view.
Filter potential search matches	Potential matches occur when a compile-time problem prevents the search engine from completely resolving the match. Filter these matches with **Filter Potential** in the search view menu (triangle symbol).

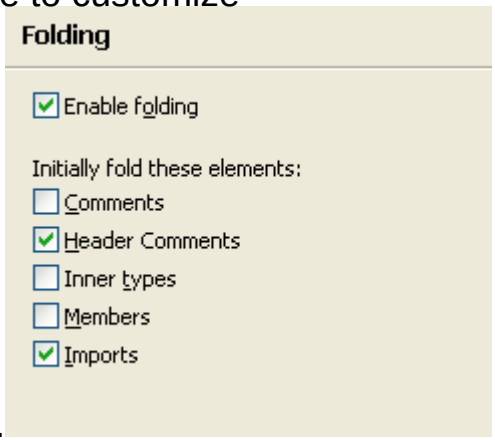
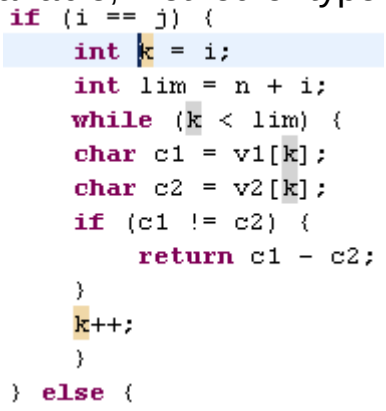
Trace method call chains with the Call Hierarchy	Use the Call Hierarchy to follow long or complex call chains without losing the original context: Just select a method and invoke **Navigate > Open Call Hierarchy (Ctrl+Alt+H)**.  You can also use drag and drop to replace the view input with the selected methods. **Ctrl+drag** adds the selected methods to the existing elements in the view. Methods from anonymous classes or special types like `Runnable` are by default expanded with constructors, since it is often more interesting to see where the object is created than where the method is invoked. **Expand with Constructors** from the context menu toggles this behavior. The defaults can be configured in the view menu.

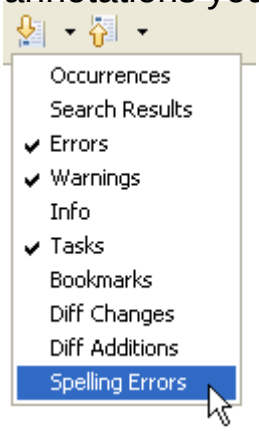
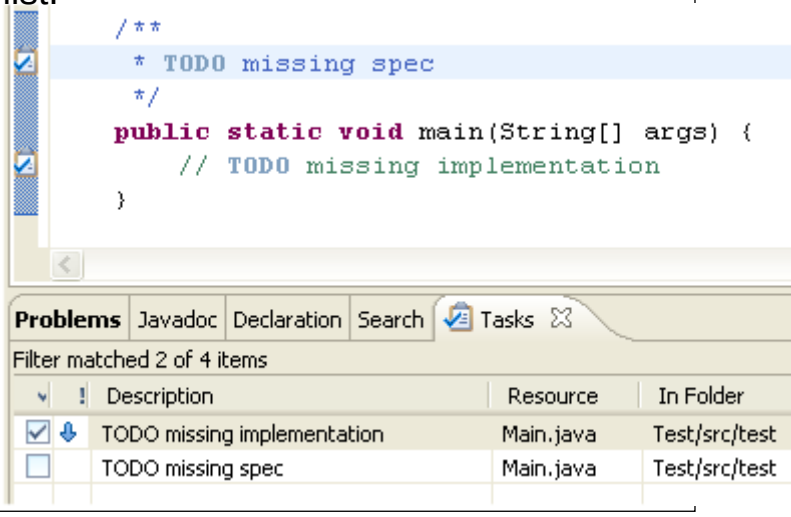
Code navigation and reading

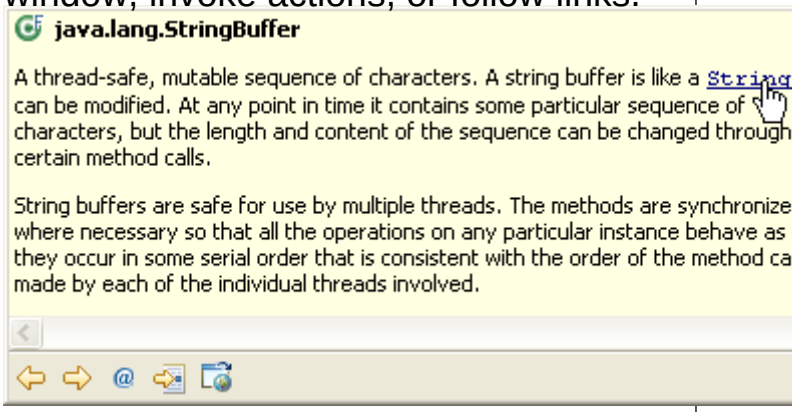
Open selection in Java editor	There are two ways how you can open an element from a reference in the Java editor. Select the reference in the code and press **F3 (Navigate > Open Declaration)**Hold **Ctrl**, move the mouse pointer over the reference, and click the hyperlink  Holding **Ctrl** on an overridable method shows a popup with an **Open Implementation** entry, which directly opens the implementation in case there's only one, or shows all the concrete implementations for that method in the hierarchy of its declaring type, using the quick type hierarchy. The hyperlink style navigation can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**General > Editors > Text Editors > Hyperlinking** preference page.
In-place outlines	Press **Ctrl+F3** in the Java editor to pop up an in-place outline of the element at the current cursor position. Or press **Ctrl+O (Navigate > Quick Outline)** to pop up an in-place outline of the current source file. 

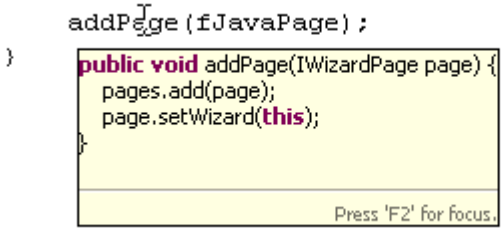
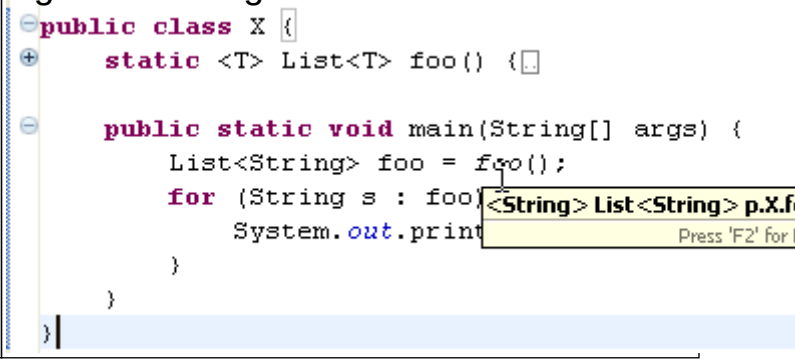
In-place outlines show inherited members	Press **Ctrl+O** or **Ctrl+F3** again to add inherited members to an open In-place outline. Inherited members have a blue label. Filter and sort the outline with the menu in the upper right corner.  The **Quick Outline (Ctrl+O)** also shows inherited members of the type that contains the current editor selection. The focus types that can show inherited members are marked with a triangle ( ).
In-place hierarchy	Find out which are the possible receivers of a virtual call using the **Quick Type Hierarchy**. Place the cursor inside the method call and press **Ctrl+T (Navigate > Quick Type Hierarchy)**. The view shows all types that implement the method with a full icon.  Press **Enter** to open the corresponding method in an editor. Press **Ctrl+T** again to switch to the Supertype hierarchy.

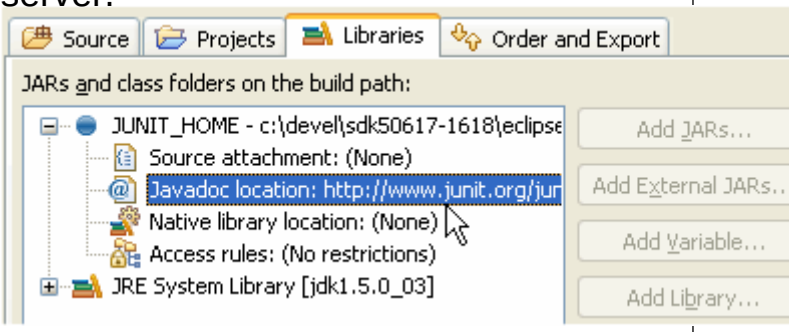
Java Editor Breadcrumb	The Java Editor contains a breadcrumb navigation bar which shows the path to the element at the cursor position. The breadcrumb can be enabled via the **Toggle Breadcrumb** tool bar button or by pressing **Alt+Shift+B**.  Each element in the breadcrumb can be selected and actions can be invoked through a context menu or keyboard short cuts. Furthermore the breadcrumb lets you navigate to other elements via drop-downs.
Advanced highlighting	The Java editor can highlight source code according to its semantics (for example: static fields, local variables, static method invocations). Have a look at the various options on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Syntax Coloring** preference page. 

Initially folded regions	You can specify which regions are folded by default when an editor is opened. Have a look at the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Folding** preference page to customize  this.
Mark occurrences	When working in the editor, turn on **Mark Occurrences** in the toolbar ( ) or press **Alt+Shift+O**. You'll see within a file, where a variable, method or type is referenced.  Different colors are used to mark read and write accesses. Selecting a return type shows you the method's exit points. Select an exception to see where it is thrown. Select a super class or interface to see the methods override or implement a method from the selected super type. Fine tune 'mark occurrences' on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Mark Occurrences** preference page. Change the color for to marker on the [Image: /topic/org.eclipse.help/command_link.png]**General > Editors > Text Editors > Annotations** preference page. To show occurrences in the Search view, use the **Search > Occurrences** feature.

Go to next / previous method	To quickly navigate to the next or previous method or field, use **Ctrl+Shift+Arrow Up (Navigate > Go To > Previous Member)** or **Ctrl+Shift+Arrow Down (Navigate > Go To > Next Member)**
Control your navigation between annotations	Use the Next / Previous Annotation toolbar buttons or **Navigate > Next Annotation** (**Ctrl+.**) and **Navigate > Previous Annotation** (**Ctrl+,**) to navigate between annotations in a Java source file. With the button drop-down menus, you can configure on which annotations you want to stop: 
Reminders in your Java code	When you tag a comment in Java source code with "TODO" the Java compiler automatically creates a corresponding task as a reminder. Opening the task navigates you back to the "TODO" in the code. Use the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Task Tags** preference page to configure any other special tags (like "FIXME") that you'd like to track in the task list. 

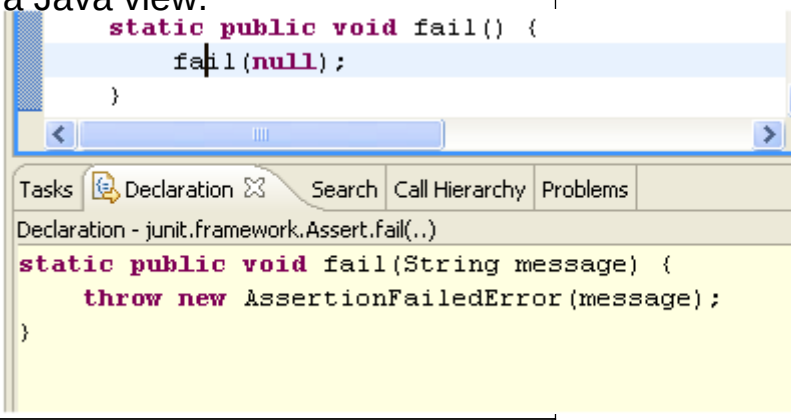
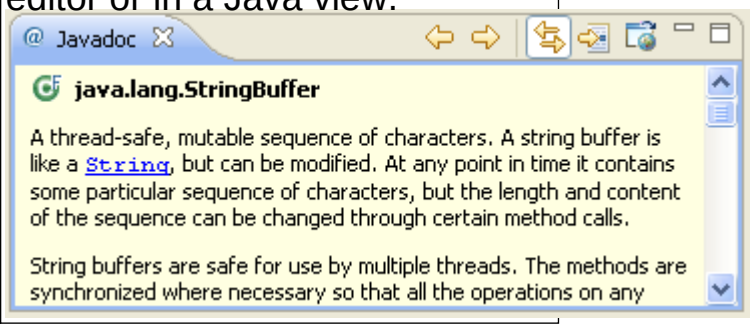
Tricks in the Open Type dialog	The Open Type dialog (**Navigate > Open Type** or toolbar button) helps you navigate to a type by its name. To find types quickly, only type the capital letters of the type name: `IOOBE` finds `IndexOutOfBoundsException`. Or type the first few characters of each name part: `NuPoiE` finds `NullPointerException`.To see all types ending with a given suffix, e.g. all Tests, use *Test< to not see all types containing Test somewhere else in the type name.To open multiple types at once, select them in the list and click OK.
Make hovers sticky	You can open the text from a hover in a scrollable window by pressing **F2 (Edit > Show Tooltip Description)** or by moving your mouse pointer into the hover. You can select and copy content from this window, invoke actions, or follow links.  You can configure how to enrich the hover on the [Image: /topic/org.eclipse.help/command_link.png]**General > Editors > Text Editors** preference page.

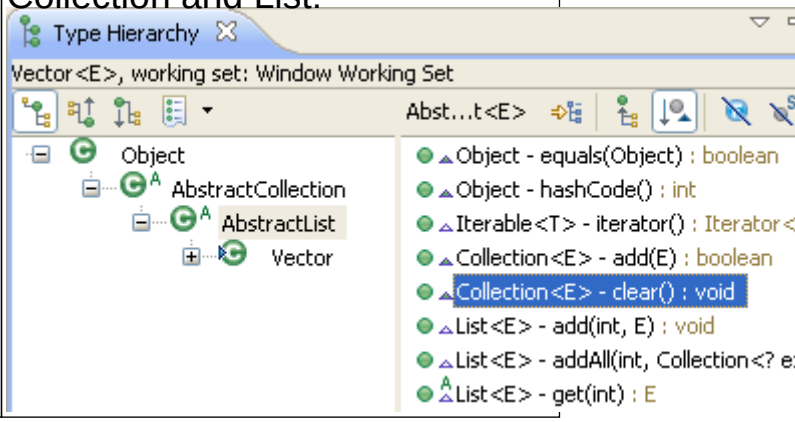
Hovers in the Java editor	You can see different hovers in the Java editor by using the modifier keys (**Shift**, **Ctrl**, **Alt**). When you move the mouse over an identifier that has no warning or problem annotation in the Java editor, by default a hover with the Javadoc extracted from the corresponding source of this element is shown. If there's a problem annotation on the identifier then the hover shows the corresponding message. Holding down the **Shift** key shows you the source code:  You can change this behavior and define the hovers for other modifier keys in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Hovers** preference page.
Generic method inferred signature	You can use hover to show the inferred signature of a generic method. 

Open and configure external Javadoc documentation	If you want to open the Javadoc documentation for a type, method or field with **Shift+F2 (Navigate > Open External Javadoc)**, you first have to specify the documentation locations to the elements parent library (JAR, class folder) or project (source folder). For libraries, open the build path page (**Project > Properties > Java Build Path**), go to the **Libraries**, and expand the node of the library where you can edit the 'Javadoc location' node. The documentation can be local on your file system in a folder or archive, or on a web server.  For types, methods or fields in source folders, go to the (**Project > Properties > Javadoc Location**).

Java views

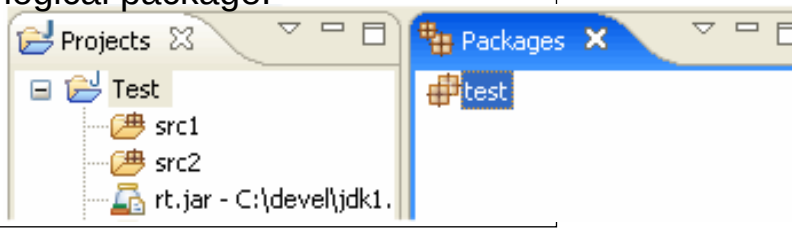
Organizing workspace with many projects	Use **Top Level Elements > Working Sets** in the Package Explorer's view menu to enable a new mode that shows working sets as top level elements. This mode makes it much easier to manage workspaces containing lots of projects. Use **Configure Working Sets...** from the Package Explorer's view menu to configure which working sets get shown. The dialog lets you create new Java working sets, define which working sets are shown and in what order. If **Sort Working Sets** is not enabled, working sets can also be rearranged directly in the Package Explorer using drag and drop.

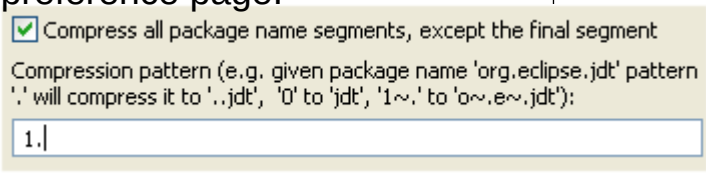
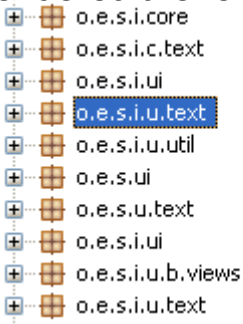
Assign working sets	To assign an element to a different working set, select the element in the Package Explorer and choose **Assign Working Sets...** from the context menu.
Declaration view	The Declaration view ([Image: /topic/org.eclipse.help/command_link.png]**Window > Show View > Other... > Java > Declaration**) shows the source of the element selected in the Java editor or in a Java view. 
Javadoc view	The Javadoc view ([Image: /topic/org.eclipse.help/command_link.png]**Window > Show View > Other... > Java > Javadoc**) shows the Javadoc of the element selected in the Java editor or in a Java view. 

Type Hierarchy view and method implementations / definitions	To find out which types in a hierarchy override a method, use the 'Show Members in Hierarchy' feature: Select the method to look at and press **F4 (Navigate > Open Type Hierarchy)**. This opens the Type Hierarchy view on the method's declaring type.With the method selected in the Type Hierarchy view, press the 'Lock View and Show Members in Hierarchy' tool bar button. The Type Hierarchy view now shows only types that implement or define the 'locked' method. You can for example see that `isEmpty()` is defined in `List` and implemented in `ArrayList` and `Vector` but not in `AbstractList`.
Type Hierarchy view supports grouping by defining type	The type hierarchy method view lets you sort the selected type's methods by its defining types. For example, for `AbstractList` you can see that it contains methods that were defined in Object, Collection and List: 

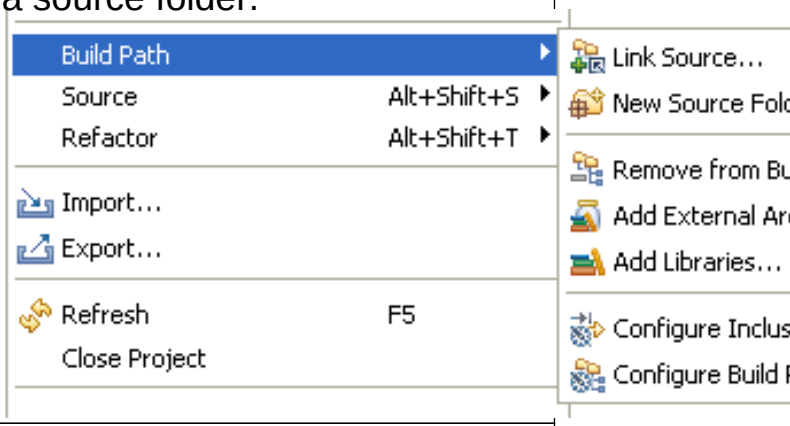
Tricks in the type hierarchy	Focus the type hierarchy on a new type by pressing **F4 (Navigate > Open Type Hierarchy)** on an element or a selected name.You can open the Type Hierarchy view not only on types but also on packages, source folders, JAR archives and Java projects. You can select multiple packages, source folders, or projects, and then open a hierarchy that contains all types in the selected containers.You can Drag & Drop an element onto the Type Hierarchy view to focus on that element.You can change the orientation (from the default vertical to horizontal) of the Type Hierarchy view from the view's toolbar menu.

Structural compare of Java source	A structural comparison of Java source ignores the textual order of Java elements like methods and fields and shows more clearly which elements were changed, added, or removed. For initiating a structural comparison of Java files you have two options: Select two Java compilation units and choose **Compare With** > **Each Other** from the view's context menu. If the files have differences, they are opened into a Compare Editor. The top pane shows the differing Java elements; double clicking on one of them shows the source of the element in the bottom pane.In any context where a file comparison is involved (e.g. a CVS Synchronization) a double click on a Java file not only shows the content of the file in a text compare viewer, but it also performs a structural compare and opens a new pane showing the results. You can even ignore formatting changes when performing the structural compare: turn on the **Ignore Whitespace** option via the Compare Editor's toolbar button, or a compare viewer's context menu.

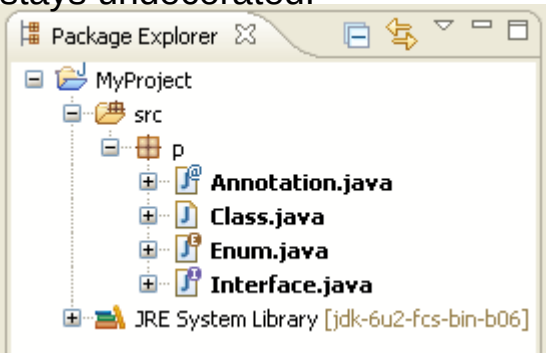
Structural compare of property files	A structural comparison of Java property files (extension: .properties) ignores the textual order of properties and shows which properties were changed, added, or removed. For initiating a structural comparison of property files you have two options: Select two files in the Package Explorer or Navigator and choose **Compare With** > **Each Other** from the view's context menu.In any context where a file comparison is involved (e.g. a CVS Synchronization) a double click on a property file not only shows the content of the file in a text compare viewer, but it also performs a structural compare and opens a new pane showing the results.
Hierarchical vs. flat presentation of packages	An option on the Java Packages (and Package Explorer) view menu allows you to change the way packages are displayed. **Package Presentation > Hierarchical** displays packages in a tree, with sub-packages below packages; **Package Presentation > Flat** displays them in the standard arrangement, as a flat list with all packages and sub-packages as siblings.
Logical packages	The Java Packages view (Java Browsing perspective) coalesces packages of the same name across source folders within a project. The screenshot shows the Packages view containing a logical package. 

Compress package names	If your package names are very long, you can configure a compressed name that appears in the viewers. Configuration of the compression pattern is done in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Appearance** preference page.  Using this example, packages are rendered the following way: 

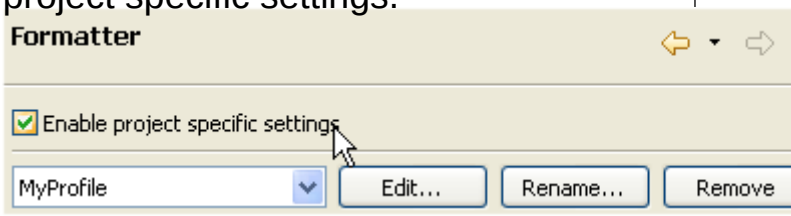
Package name abbreviations	Package names in Java views can be abbreviated with custom rules. For example, the following rules produce the rendering shown below:
	`org.eclipse.ui={UI}`
	`org.eclipse.ui.texteditor={T}`
	`org.eclipse.ui.internal.texteditor=[iT]`
	The abbreviation rules can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Appearance** preference page.

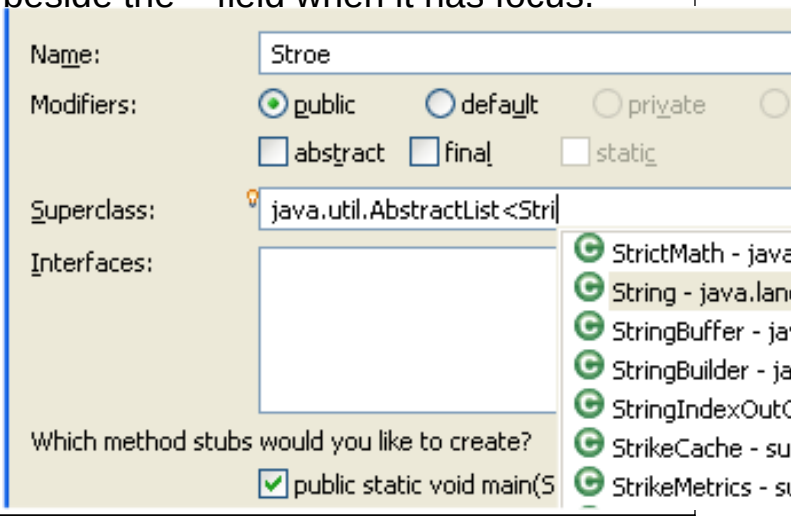
Manipulating the Java build path directly in the Package Explorer	Instead of manipulating the Java Build path on **Project > Properties > Java Build Path**, use the actions in the Package Explorer's context menu. You can for example add new source folders, archives and libraries to the build path or in- and exclude file and folders from a source folder. 

Paste code snippets to create a type	You can paste a snippet of code containing a Java type directly into a package or source folder to create a new compilation unit. For example, select and copy this source code:
	`package pack;`
	`public class HelloWorld {`
	`  public static void main(String[] args)`
	`{`
	`  System.out.println("Hello World");`
	`  }`
	`}`
	Then select a source folder in the Package Explorer and use **Ctrl+V (Edit > Paste)**. This automatically creates a new package 'pack' and file 'HelloWorld.java' with the copied content. You can also paste a method or a set of statements; Eclipse will create the required enclosing elements for you.
	If you paste while nothing is selected in the Package Explorer, Eclipse even creates a new Java Project and create the *.java file there. Furthermore, the clipboard can also contain multiple package and type declarations, in which case Eclipse will create all the necessary packages and *.java files on paste.

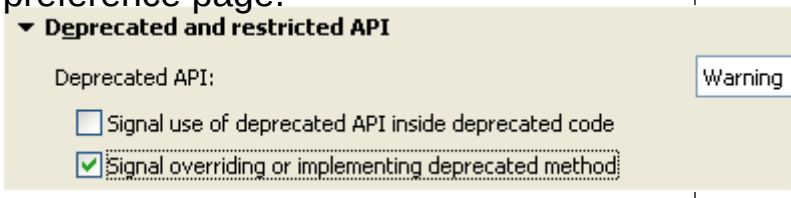
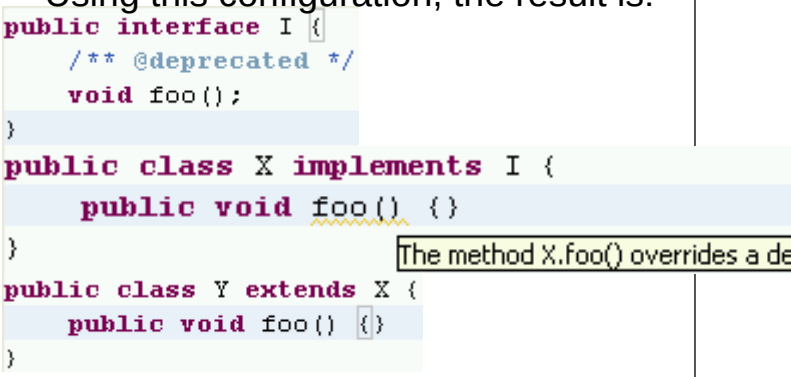
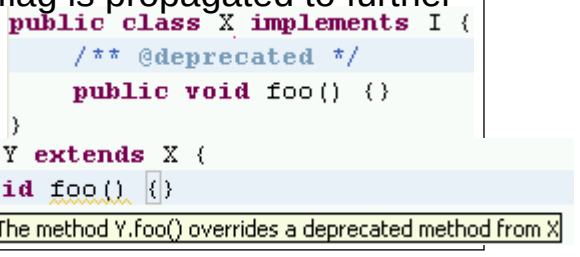
Paste patch into Package Explorer	What's the quickest way to apply a patch from Bugzilla? Just open the attachment, copy the patch to the clipboard and paste it into the Package Explorer.
Grouping Java problems	Configure the Problems view to group Java problems into categories with **Group by > Java Problem Type** in the view menu. You can control if a configurable error is intended to be fatal or not at the bottom of the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page.
Java type indicator	Enable the **Java Type Indicator** on the [Image: /topic/org.eclipse.help/command_link.png]**General > Appearance > Label Decoration** preference page to find out what the first type in a compilation unit or class file is. An adornment is shown for interfaces, annotations, and enums, while an ordinary class stays undecorated. 

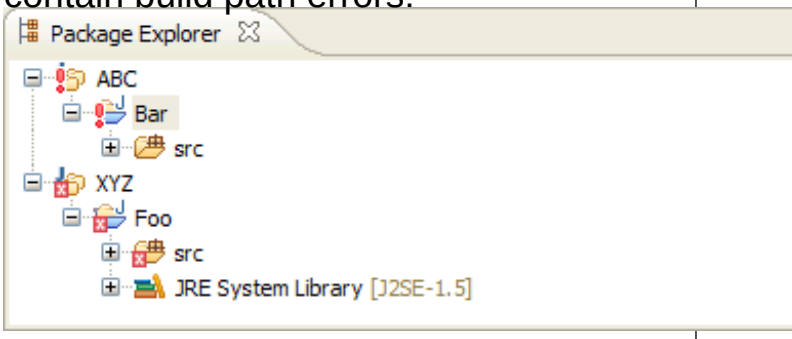
Miscellaneous

Project specific preferences	All code style and compiler options can be defined per project. Open the project property pages with **Project > Properties** on a project or use the link on the workspace preferences (e.g. the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style** preference page) to open a project property page and enable project specific settings.  The project specific settings are stored in a configuration file inside the project (in the '.settings' folder). When you share a project in a team, team members will also get these project specific settings.
Access rules	Access rules give you the possibility to enforce API rules for types from referenced libraries. On the Java build path page (**Project > Properties > Java Build Path**) edit the 'Access Rules' node available on referenced projects, archives, class folders and libraries. Packages or types in these references can be classified as:AccessibleDiscouragedForbiddenAccording to the settings on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page, the compiler will mark discouraged and forbidden references with warning or errors.
JUnit	Select a JUnit test method in a view and choose **Run > JUnit Test** from the context menu or **Run > Run As > JUnit Test** from the main menu. This creates a launch configuration to run the selected test.

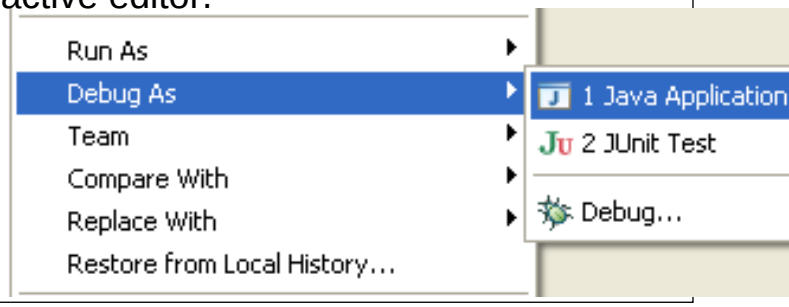
Hide JUnit view until errors or failures occur	You can configure the JUnit view to only open when there are errors or failures. To do so enable **Activate on Errors/Failures only** in the JUnit view menu. That way, you can have the view set as a fast view and never look at it when there are no failing tests. While there are no problems in your tests you will see this icon (the number of small green squares will grow, indicating progress) while running them and this icon after they are finished. If, however, errors or failures occur, the icon will change to (or if tests are finished) and the JUnit view will appear.
Content assist in dialog fields	Content assist (**Ctrl+Space**) is also available in input fields of various Java dialogs. Look for a small light bulb icon beside the field when it has focus. 
Smart caret positioning in dialogs showing Java names	Text fields for editing Java names support smart caret positioning. Use **Ctrl+Left** and **Ctrl+Right** to stop at camel case boundaries inside a name. Use **Ctrl+Shift+Left** and **Ctrl+Shift+Right** to extend the selection in small steps. Use **Ctrl+Delete** to delete the next and **Ctrl+Backspace** to delete the previous part of a name.
Define prefixes or suffixes for fields, parameters and local variables	You can configure the prefix or suffix for fields, static fields, parameters, and local variables. These settings on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style** preference page are used in content assist, quick fix, and refactoring whenever a variable name is computed.
Organize Imports works on more than single files	You can invoke **Source > Organize Imports** on sets of compilation units, packages, source folders or Java projects.

Organize imports on save	Eclipse can automatically organize imports whenever you save an editor. This feature can be enabled globally on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Editor > Save Actions** preference page. The save actions can also be configured per project, which makes it easy to enforce a project-wide standard by sharing the settings across a team: Besides organizing imports, save actions can also format code and perform other clean ups.
Format selection	Select multiple Java files to format and choose **Source > Format** to format them all. The format action is also available on packages, source folders or Java projects. In the Java Editor, **Source > Format (Ctrl+Shift+F)** formats just the selected region. If nothing is selected, the whole file gets formatted.
Select execution environment for Java project	When you create a Java project that you want to share with a team, it is a good idea to use an execution environment instead of a specific JRE. Execution environments are symbolic representations of JREs with standardized entries like 'J2SE-1.4', 'J2SE-1.5'. That means no file system path will go into the shared build path. JREs can be assigned to the environments on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Installed JREs > Execution Environments** preference page. To set a project specific JRE of an existing project, open the project's Java Build path property page **(Project > Properties > Java Build Path)**, then the **Libraries** page, select 'JRE System Library' and press **Edit**. In the 'Edit Library' dialog you can select either an Execution Environment, a project specific JRE, or the workspace default JRE.

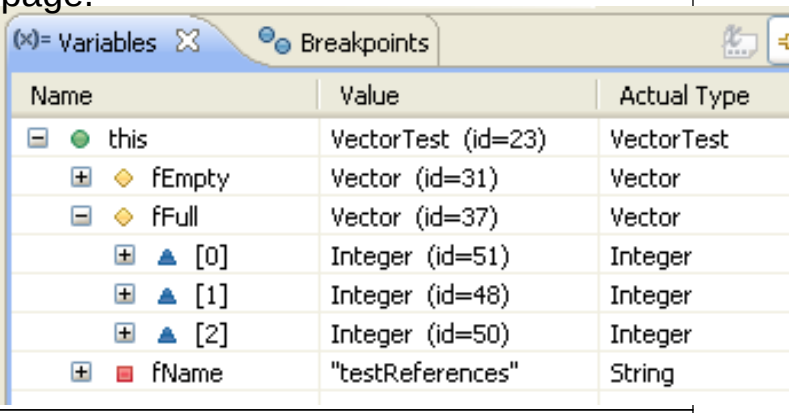
Use drag and drop	Drag and Drop is a handy replacement of **Edit > Cut** and **Edit > Paste** , or **Edit > Copy** and **Edit > Paste**. Hold down the **Ctrl** key while dropping to change from move to copy. Important to know: If you move Java compilation units between packages by Drag & Drop, this will behave like a refactoring move - all missing imports will be added and references updated If you drag and drop source elements like types or methods, this will only copy or move the corresponding source code - no references will be updated
Propagating deprecation tag	The Java compiler can be configured to diagnose deprecation using options on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page.  Using this configuration, the result is:  If you're unable to fix a usage of a deprecated construct, we recommend that you tag the enclosing method, field or type as deprecated. This way, you acknowledge that you did override a deprecated construct, and the deprecation flag is propagated to further dependents. 

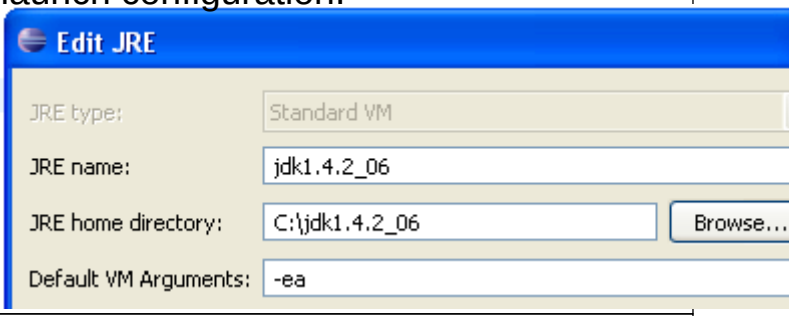
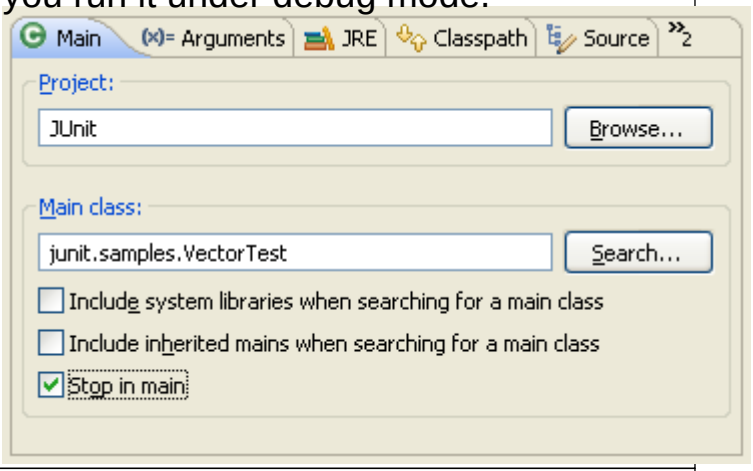
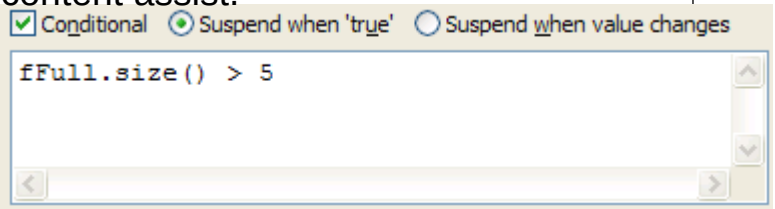
Recovering from abnormal inconsistencies	In the rare event of a dysfunction, JDT could reveal some inconsistencies such as: missing results in a **Java Search** or **Open Type**invalid items in Package Explorer To bring JDT into a consistent state again, the following actions need to be performed in this exact order: Close all projects using navigator **Close Project** menu actionExit and restart EclipseOpen all projects using navigator **Open Project** menu actionManually trigger a clean build of entire workspace (**Project > Clean...**)
Build path error decorator	The **Package Explorer** and the **Project Explorer** show this error decorator on Java projects and working sets if they contain build path errors:  In case of build path errors in the **Problems** view, first look in the **Package Explorer** for this icon since it better indicates where the problem is.

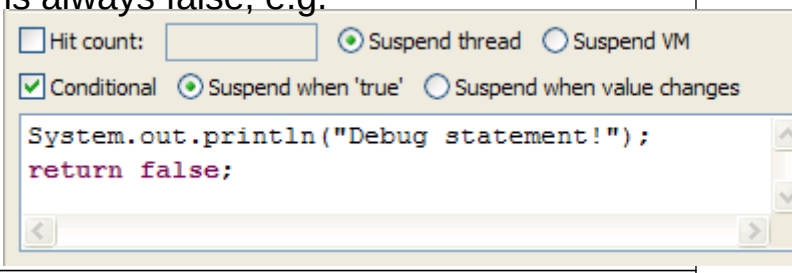
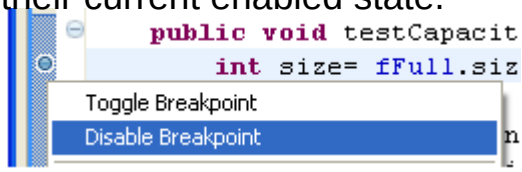
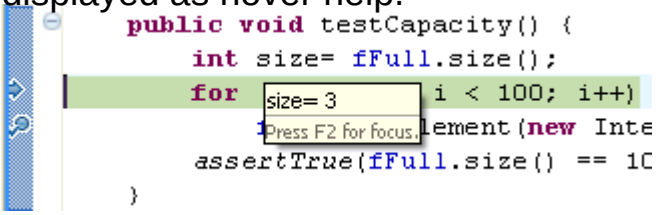
Debugging

Launching from the context menu	You can run and debug Java applications from the context menu. You can launch a source file, package, method, type, etc. by choosing **Run As** (or **Debug As**) **> Java Application** from the context menu in a view or editor. Alternatively, you can use the Java application launch shortcut key binding (**Alt+Shift+D, J**). The top level **Run As** (or **Debug As**) actions are also sensitive to the current selection or active editor. 

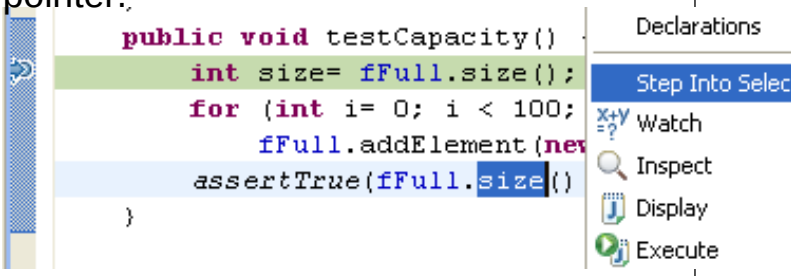
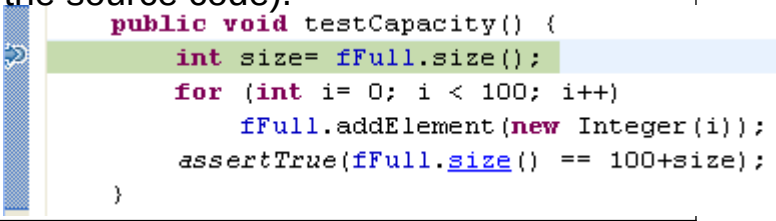
Evaluations in the debugger	Snippet evaluation is available from a number of places in the debugger. Choosing **Display** or **Inspect** from the context menu of the editor or **Variables view** will show the result in a pop-up whose result can be sent to the **Display** or **Expressions views**.
View management in non-Debug perspectives	The Debug view automatically manages debug related views based on the view selection (displaying Java views for Java stack frames and C views for C stack frames, for example). By default, this automatic view management only occurs in the Debug perspective, but you can enable it for other perspectives via the **View Management** preference page available from the **Debug view** toolbar pull down.
Environment variables	You can specify the environment used to launch Java applications via the **Environment** tab.
String substitutions	Variables are supported for many parameters of Java Application launch configurations. Most fields that support variables have a **Variables...** button next to them, such as the program and VM arguments fields. The **Main Type** field supports variables as well; the *${java_type_name}* variable allows you to create a configuration that will run the selected type.

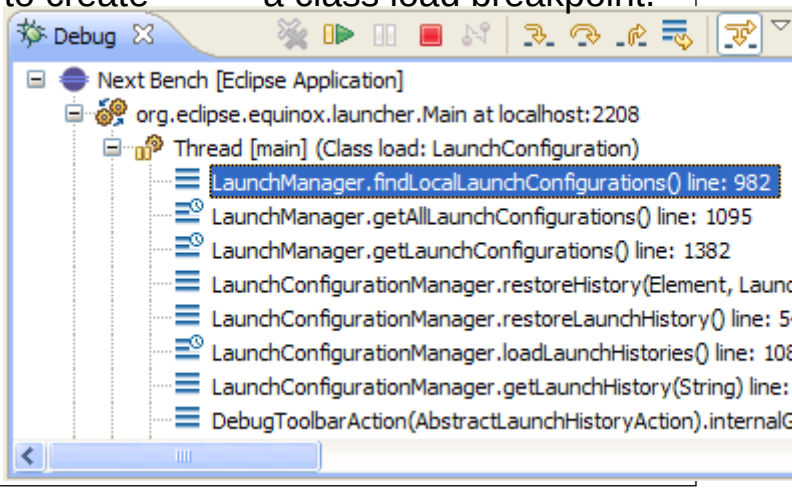
Logical structures	The **Logical Structures** toggle on the **Variables view** toolbar presents alternate structures for common types. JDT provides logical views for Maps, Collections, and SWT Composites. You can define logical structures for other types from the **Logical Structures** preference page. 
Variable columns	Columns in the **Variables view** can be configured by selecting **Layout > Select Columns...** in the view menu. A dialog allows you to select the columns to display. For example, a column can be added to display instance counts of classes (when debugging with JavaSE-1.6). Columns can be toggled on/off using the **Layout > Show Columns** action.
Show references	When debugging with JavaSE-1.6, references to objects can be displayed in the **Variables view** by selecting **Java > Show References** in the view menu. A "**referenced from**" entry will appear under each object that can be expanded to show all references to that object.
All instances	When debugging with JavaSE-1.6, all instances of a class can be inspected by selecting a class in an editor outline, variables view, compilation unit or class file editor and invoking **All Instances...** from the context menu. A pop-up dialog will display all instances of the selected class.

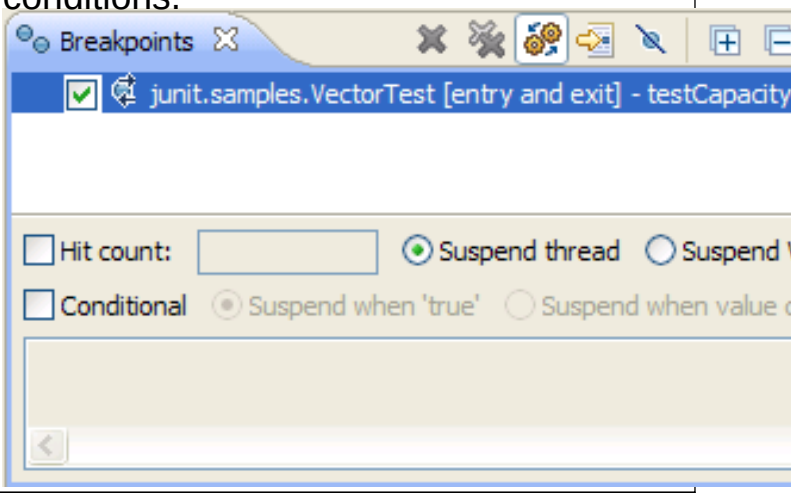
Default VM arguments	If you specify the same arguments to a certain VM frequently, you can configure **Default VM Arguments** in the **Installed JREs** preference page. This is more convenient than specifying them for each launch configuration. 
Stop in main	You can use **Stop in main** in a Java Application launch configuration to cause your program to stop at the first executable line of the main method when you run it under debug mode. 
Conditional breakpoints	You can use expressions to define conditional breakpoints using the **Breakpoint Properties...** dialog or detail pane in the breakpoints view. A condition controls when a breakpoint actually halts execution. You can specify whether you want the breakpoint to suspend execution only when the condition is true, or when the condition value changes. The breakpoint condition editor supports content assist. 

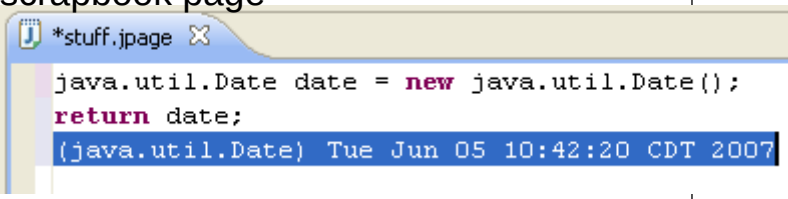
Debugging by writing to console	You can avoid inserting `System.out.println()` statements in your code for debugging by using conditional breakpoints to print to the Console view. To do so, set a conditional breakpoint with **Suspend when 'true'** option and a condition which is always false, e.g. 
Disabling breakpoints	If you find yourself frequently adding and removing a breakpoint in the same place, consider disabling the breakpoint when you don't need it and enabling it when needed again. This can be done using **Disable Breakpoint** in the breakpoint context menu or by unchecking the breakpoint in the **Breakpoints view**. You can also temporarily disable all breakpoints using the **Skip All Breakpoints** action in the **Breakpoints view** toolbar. This will tell the debugger to skip all breakpoints while maintaining their current enabled state. 
Changing variable values	When a thread is suspended in the debugger, you can change the values of Java primitives and Strings in the **Variables** view. From the variable's context menu, choose **Change Variable Value**. You can also change the value by typing a new value into the Details pane and using the **Assign Value** action in the context menu (**CTRL+S** key binding).
Variable values in hover help	When a thread is suspended and the cursor is placed over a variable in the Java editor, the value of that variable is displayed as hover help. 

Drop to frame	When stepping through your code, you might occasionally step too far, or step over a line you meant to step into. Rather than restarting your debug session, you can use the **Drop to Frame** action to quickly go back to the beginning of a method. Select the stack frame corresponding to the Java method you wish to restart, and select **Drop to Frame** from **Debug view** toolbar or the stack frame's context menu. The current instruction pointer will be reset to the first executable statement in the method. This works for non-top stack frames as well. Note that Drop to frame is only available when debugging with a 1.4 or higher VM, or the J9 VM. There are some situations where a JVM may be unable to pop the desired frames from the stack. For example, it is generally impossible to drop to the bottom frame of the stack or to any frame below a native method.
Hot code replace	The debugger supports Hot Code Replace when debugging with a 1.4 or higher VM, or the J9 VM. This lets you make changes to code you are currently debugging. Note that some changes such as new or deleted methods, class variables or inner classes cannot be hot swapped, depending on the support provided by a particular VM.

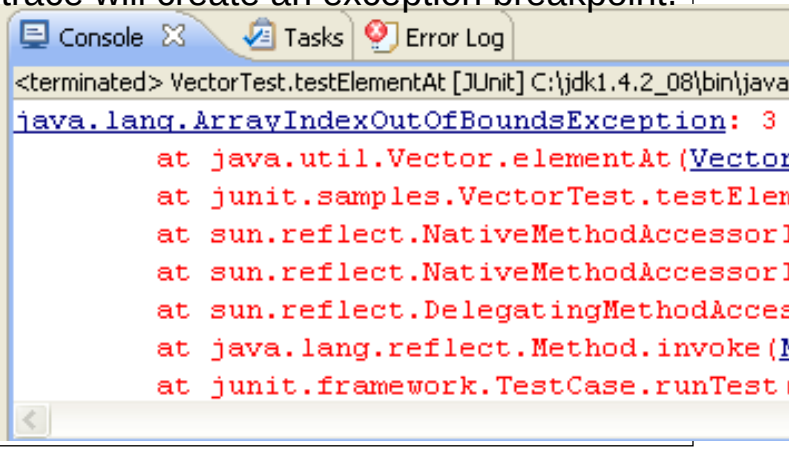
Stepping into selections and hyperlink debugging	The Java debugger allows you to step into a single method within a series of chained or nested method calls. Simply select the method you wish to step into and select **Step into Selection** from the Java editor context menu. This feature works in places other than the currently executing line. Try debugging to a breakpoint and step into a method a few lines below the current instruction pointer.  When the caret is not in a method name, the feature steps into the next method on the selected line. For example, when the caret is at the beginning of the line `assertTrue(fFull.size() == 100)`, then **Step into Selection** steps into `assertTrue(..)` (unlike **Step Into**, which would step into `size()`). You can also step into a method by using hyperlink navigation. Simply place the cursor over the method you wish to step into and use **Ctrl+Alt+Click** to step into the method (rather than **Ctrl+Click** which will navigate to the source code). 
Console pin and lock	Output displayed in the console can be locked to a specific process via the **Pin Console** action in the Console view toolbar. There's also a **Scroll Lock** action that stops the console from scrolling as new output is appended.
Creating watch items	A watch item is an expression in the **Expressions** view whose value is updated as you debug. You can create watch items from the Java editor by selecting an expression or variable and choosing **Watch** from its context menu or the top-level **Run** menu.

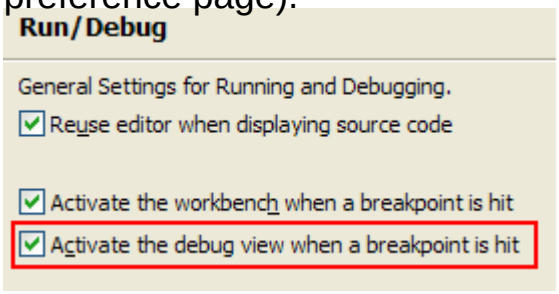
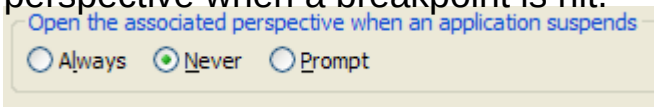
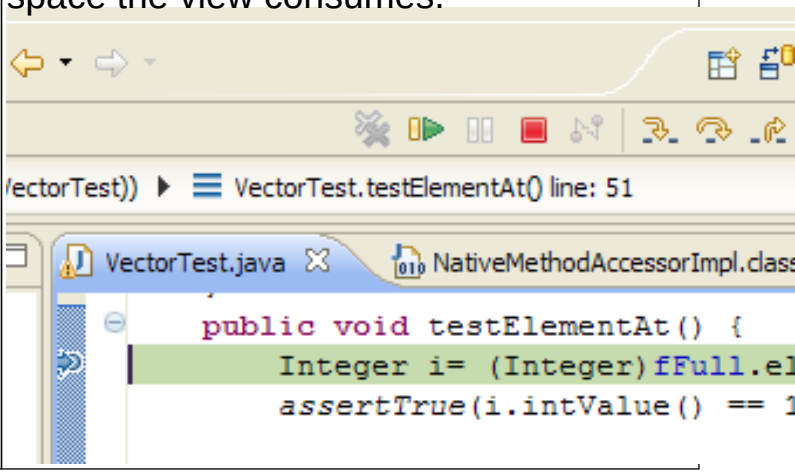
Watch points	A watch point is a breakpoint that suspends execution whenever a specified field is accessed or modified. To set a watchpoint, select a field in the Outline view and choose **Toggle Watchpoint** from its context menu. To configure a watchpoint, select the watchpoint in the **Breakpoints** view and choose **Properties...** from its context menu. The most important properties for this type of breakpoint are the **Access** and **Modification** checkboxes which control when the breakpoint can suspend execution.
Class load breakpoints	A class load breakpoint suspends execution when a specified class is loaded in the virtual machine. To set a class load breakpoint, select a class in the Outline view and choose **Toggle Class Load Breakpoint** from its context menu. You can also use the **Run > Add Class Load Breakpoint...** menu action to create a class load breakpoint. 

Method breakpoints	A method breakpoint suspends execution when a specific method is entered or exited. To set a method breakpoint, select a method in the Outline view and choose **Toggle Method Breakpoint** from its context menu. Alternatively, double click on a method declaration line in the editor ruler or use the **Run > Toggle Method Breakpoint** menu action to create a method breakpoint in the currently selected method. By default a method breakpoint only suspends execution when a method is entered. You can use the breakpoint detail pane or the breakpoint properties dialog to configure the breakpoint to suspend on exit. Method breakpoints also support conditions. 
Threads and monitors	The Java debugger optionally displays monitor information in the **Debug** view. Use the **Show Monitors** action in the Debug view drop down menu to show which threads are holding locks and which are waiting to acquire locks. Threads involved in a deadlock are rendered in red.

Step filters	Step filters prevent the debugger from suspending in specified classes and packages when stepping into code. Step filters are established with the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Step Filtering** preference page. When the **Use Step Filters** toggle (on the debug toolbar and menu) is on, step filters are applied to all step actions. In the Debug view, the selected stack frame's package or declaring type can be quickly added to the list of filters by selecting **Filter Type** or **Filter Package** from the stack frame's context menu.
Using the scrapbook	If you want to experiment with API or test out a new algorithm, it's frequently easier to use a Java scrapbook page than create a new class. A scrapbook page is a container for random snippets of code that you can execute at any time without a context. To create a scrapbook page, create a new file with a **.jpage** extension (or use the [Image: /topic/org.eclipse.help/command_link.png]**New Scrapbook Page** wizard). Enter whatever code you wish to execute and then select it. There are three ways to execute your code: Execute the selected code and place the returned result in an inspect popupExecute the selected code and place the String result right in the scrapbook page  Execute the selected code (and ignore any returned result)These actions are in the workbench toolbar and also in the scrapbook page's context menu.
Editing launch configurations	Holding down the **Ctrl** key and making a selection from the **Run** or **Debug** drop-down menu opens the associated launch configuration for editing. The launch configuration can also be opened from the context menu associated with any item in the Debug view.

Favorite launch configurations	Launch configurations appear in the Run/Debug drop-down menus in most recently launched order. However it is possible to force a launch configuration to always appear at the top of the drop-downs by making the configuration a 'favorite'. Use the **Organize Favorites...** action from the appropriate drop down menu to configure your favorite launch configurations.
Detail formatters	In the **Variables** & **Expressions** views, the detail pane shows an expanded representation of the currently selected variable. By default, this expanded representation is the result of calling `toString()` on the selected object, but you can create a custom detail formatter that will be used instead by choosing **New Detail Formatter** from the variable's context menu. This detail formatter will be used for all objects of the same type. You can view and edit all detail formatters in the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Detail Formatters** preference page.
Running code with compile errors	You can run and debug code that did not compile cleanly. The only difference between running code with and without compile errors is that if a line of code with a compile error is executed, one of two things will happen: If the 'Suspend execution on compilation errors' preference on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug** preference page is set and you are debugging, the debug session will suspend as if a breakpoint had been hit. Note that if your VM supports Hot Code Replace, you could then fix the compilation error and resume debuggingOtherwise, execution will terminate with a 'unresolved compilation' errorIt is important to emphasize that as long as your execution path avoids lines of code with compile errors, you can run and debug just as you normally do.
Word wrap in Variables view	The details area of the debugger's **Variables** and **Expressions** views supports word wrap, available from the view drop-down menu.

Code assist in the debugger	Code assist is available in many contexts beyond writing code in the Java editor: When entering a **breakpoint condition**In the **Details** pane of the **Variables** & **Expressions** viewWhen entering a **Detail Formatter** code snippetWhen entering a **Logical Structure** code snippetWhen entering code in a **Scrapbook** pageIn the **Display** view
Command line details	You can always see the exact command line used to launch a program in run or debug mode by selecting **Properties** from the context menu of a process or debug target, even if the launch has terminated.
Stack trace hyperlinks	Java stack traces in the console appear with hyperlinks. When you place the mouse over a line in a stack trace the pointer changes to the hand. Pressing the mouse button opens the associated Java source file and positions the cursor at the corresponding line. Pressing the mouse button on the exception name at the top of the stack trace will create an exception breakpoint. 

Debugging in the Java perspective	You can debug in the Java perspective by configuring the **Run/Debug > View Management** preferences. Select the perspectives you want to be debug from. Ensure the preference to activate the debug view when a breakpoint is hit is turned on (via the top level **Run/Debug** preference page).  You can disable the perspective switching prompt on the **Run/Debug > Perspectives** preference page. With these settings the Debug view will automatically open in the Java perspective when a breakpoint is hit.  By placing the Debug view at the top of the perspective and resizing its height to the size of one element, the view will switch to a bread crumb presentation. This minimizes the amount of screen space the view consumes. 

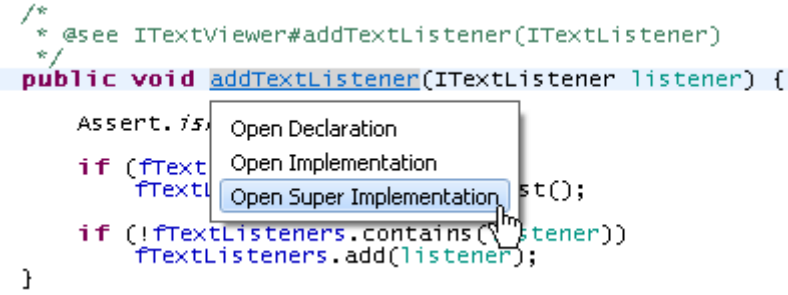
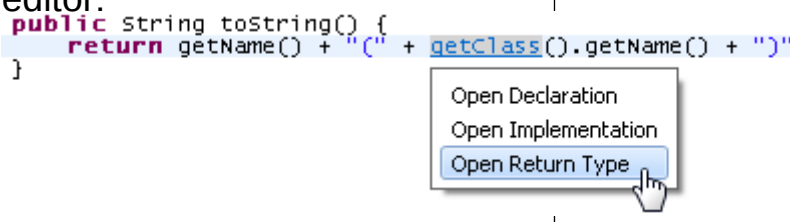
What's New in 3.7 (JDT)

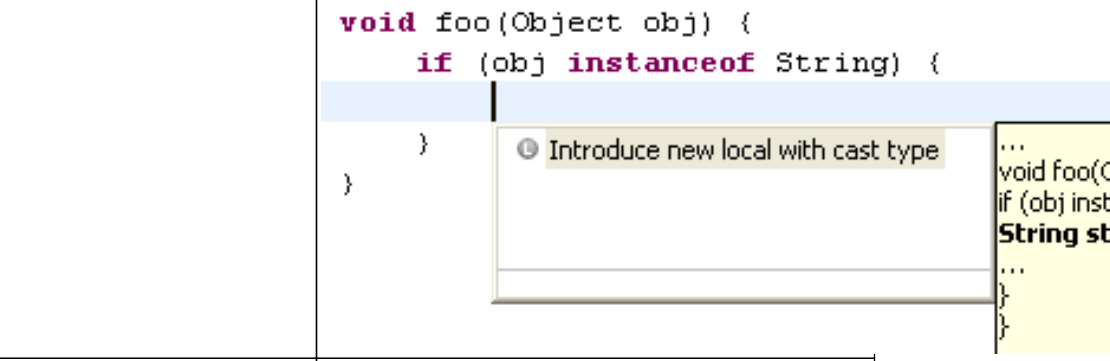
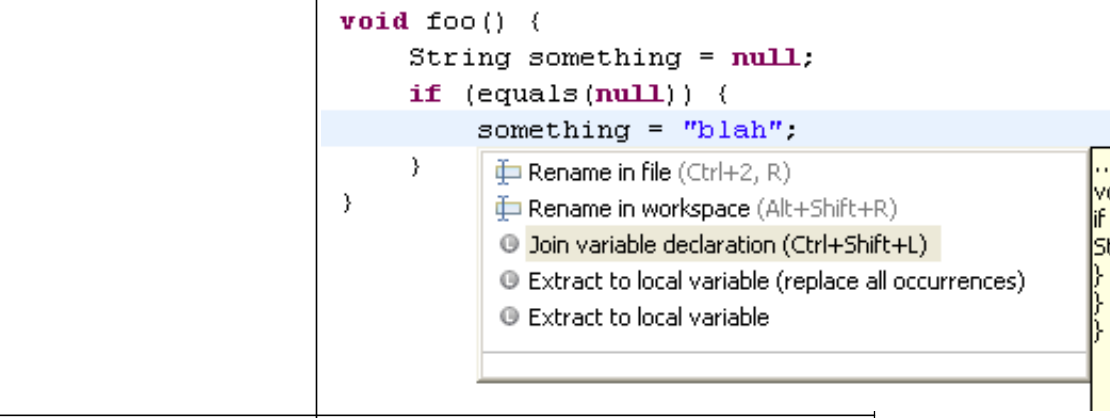
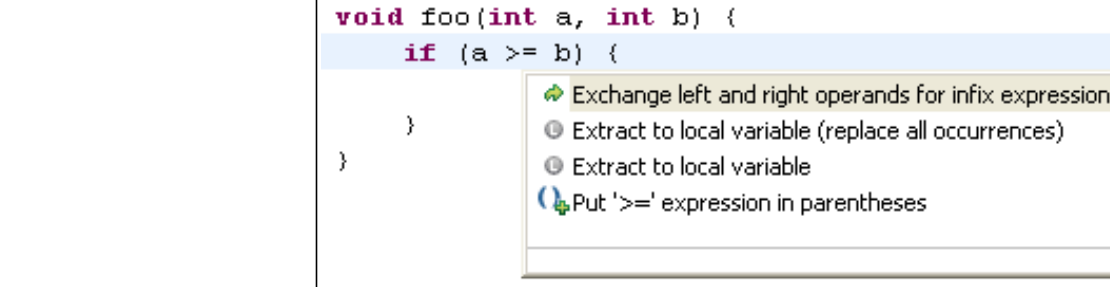
Here are descriptions of some of the more interesting or significant changes made to the Java development tools for the 3.7 release of Eclipse. They are grouped into:

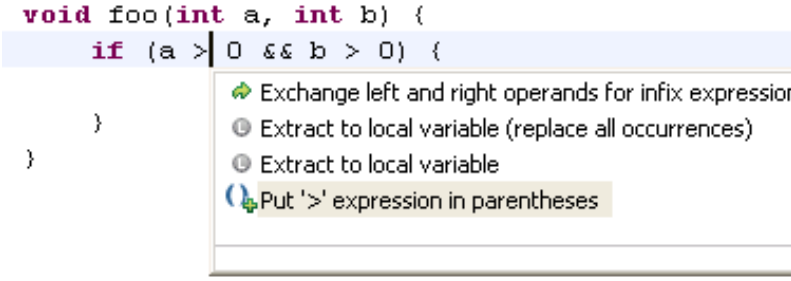
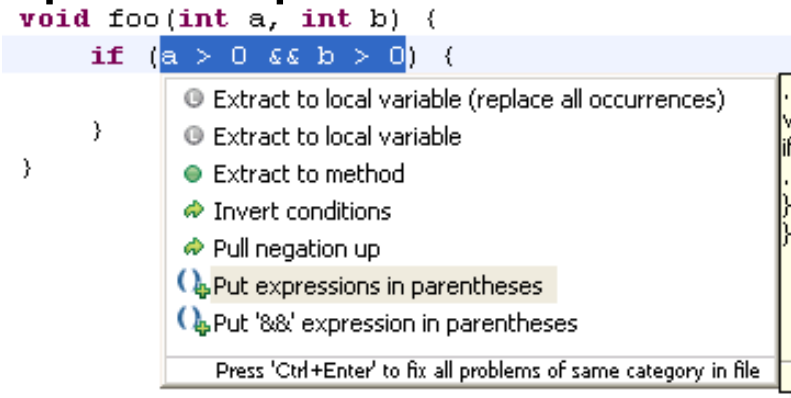
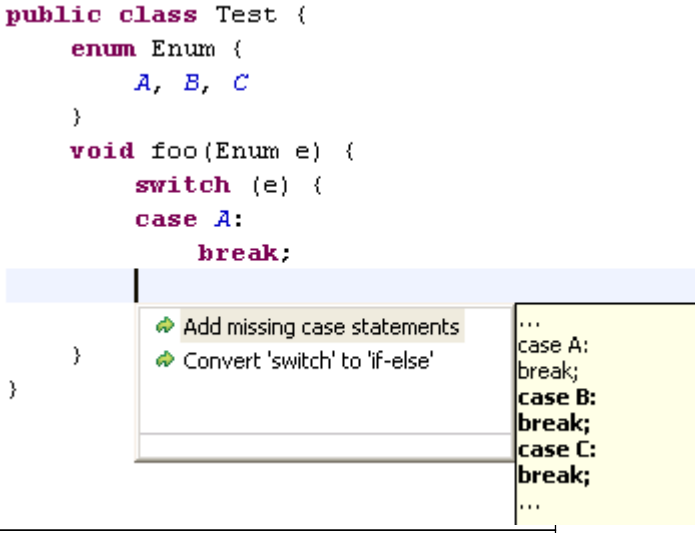
- Java 7
- Java Editor
- Java Formatter
- Java Compiler
- Java Views and Dialogs
- Properties File Editor
- JUnit

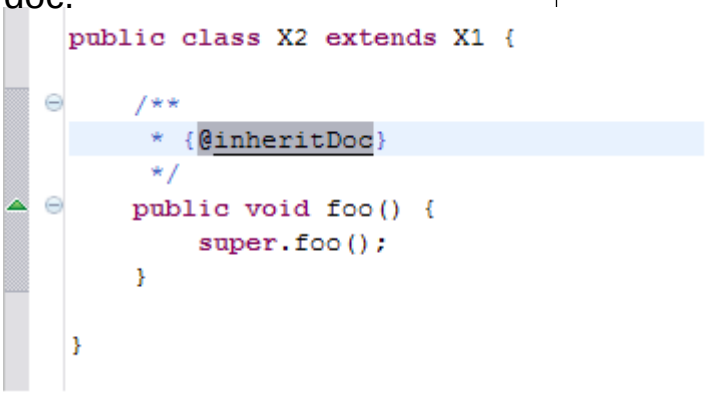
See also the Eclipse Platform What's New document for changes in the Platform.

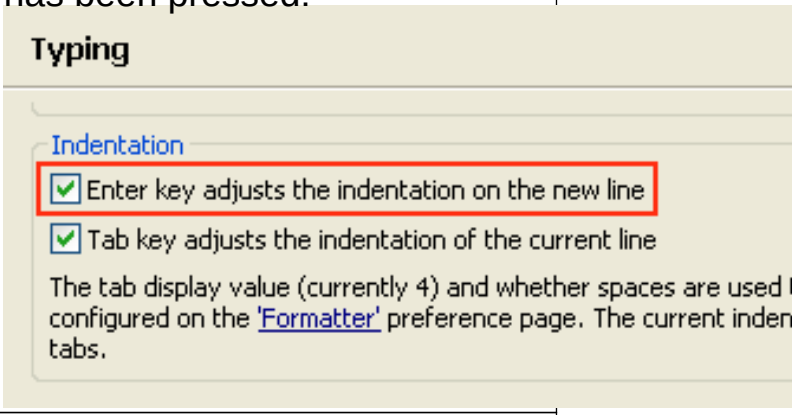
Java Editor	
Open Java element from clipboard	The new **Navigate > Open from Clipboard** command tries to open the matching Java element in the editor if the clipboard contains a single line. Otherwise it opens the contents in the Java Stack Trace Console. Examples: java.lang.StringStringString#getBytesString.getBytesjava.lang.String.getBytes(String)String.java:123at java.lang.String.matches(String.java:1550)java.lang.String.valueOf(char) line: 1456currentTimeMillis()Note: If the action is not enabled in a certain perspective, then you can enable the **Java Debug** action group in [Image: /topic/org.eclipse.help/command_link.png]**Window > Customize Perspective> Command Groups Availability**.

Hyperlinks	Three new hyperlinks have been added to the Java editor: **Open Super Implementation**: The action from the **Navigate** menu is now also available as a hyperlink. It is enabled for overridden methods and opens the super implementation of the selected method.  **Open Declared Type**: This link is enabled for local variables and fields. When invoked, it opens the declared type of the variable in an editor:  **Open Return Type**: This link is enabled for methods. When invoked, it opens the return type of the method in an editor:  By default, the hyperlink appears when you hold down the **Ctrl** key while hovering over an appropriate element, or when you use the **Navigate > Open Hyperlink** command. You can configure the modifier for the hyperlinks on the The hyperlink style navigation can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**General > Editors > Text Editors > Hyperlinking** preference page.

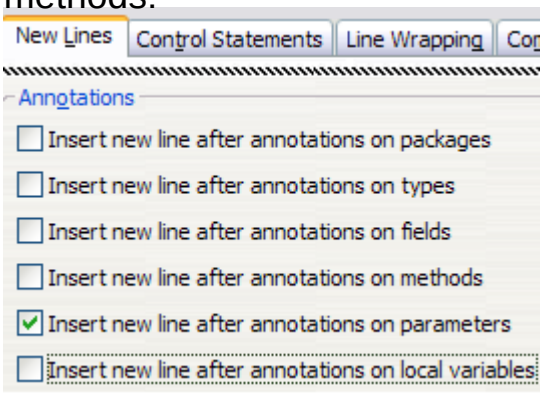
"Introduce new local with cast type" quick assist	The **Introduce new local with cast type** quick assist (**Ctrl+1**) is now not only available on the `instanceof` keyword, but also in the body of the conditional statement (before the first body statement). 
"Join variable declaration" quick assist for variables initialized to null	The **Join variable declaration** quick assist is now also available for variables initialized to `null`. 
"Exchange Operands" quick assist for comparison operators	The **Exchange left and right operands for infix expression** quick assist is now also available for the !=, <, <=, >, and >= operators. 

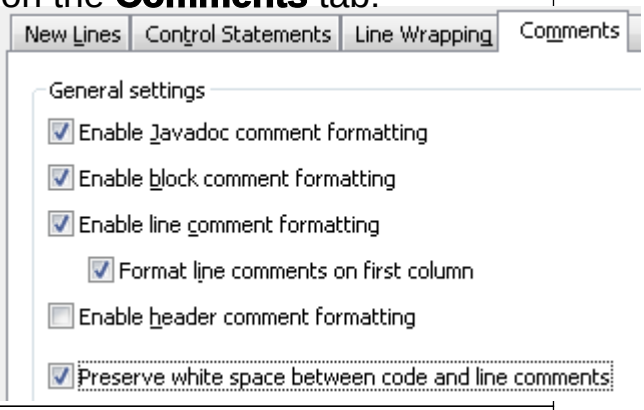
New 'Put expression in parentheses' quick assist	The Java editor now offers a new quick assist **Put expression in parentheses**.  Also the **Add paranoiac parentheses** quick assist has been renamed to **Put expressions in parentheses**. 
'Add missing case statements' quick assist	The **Add missing case statements** quick assist is now available in the body of the switch statement: 
Semantic coloring for abstract classes	Abstract classes can now be highlighted separately in the source code: The rendering can be configured on the **Java > Editor > Syntax Coloring** preference page.

Quick Outline shows inherited members for nested type	The **Quick Outline** shows inherited members of top-level types when **Ctrl+O** is pressed twice. Now, it also shows inherited members of the type that contains the current editor selection: The focus types that can show inherited members are marked with a triangle (▸).
Ctrl+drag to add to Call Hierarchy	**Ctrl+drag** now adds the selected methods to the existing elements in the **Call Hierarchy** view. Plain drag and drop continues to replace the view input with the selected methods.
Navigate to {@inheritDoc} target	To quickly navigate to the target of an {@inheritDoc} tag in a Javadoc comment, you can now **Ctrl+click** the tag or use **Navigate > Open Declaration** to jump to the method that defines the inherited doc.  In this example, the target of the `@inheritDoc` inline tag on X2.foo() is X1.foo().
Navigate to 'break' and 'continue' target	To quickly navigate to the target of a `break` or `continue` statement, you can now **Ctrl+click** or use **Open Declaration** (**F3**) on `break` or `continue` keywords or their labels. Navigation on... `break`: jumps to the end of the target statement`continue`: jumps to the beginning of the target statementa label: jumps to the label declaration

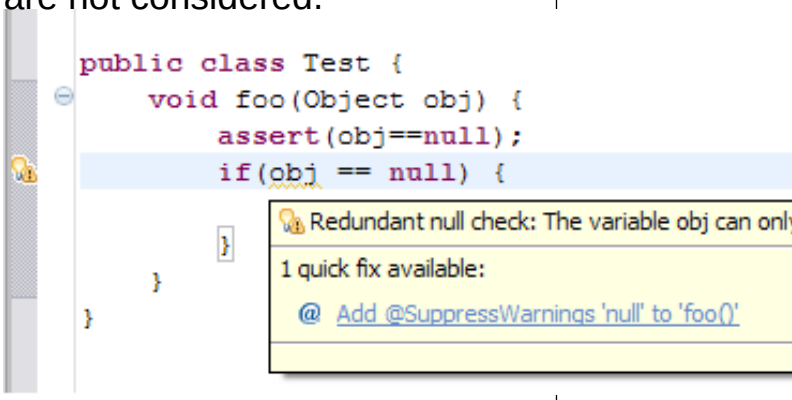
Format Java elements via Outline view	You can now format one or more selected elements in the Java Outline view by invoking **Source > Format Element** from the main menu.
Disable smart indentation on 'Enter'	A new preference has been added to the **Java > Editor > Typing** preference page that allows to disable smart indentation after the **Enter** key has been pressed: 

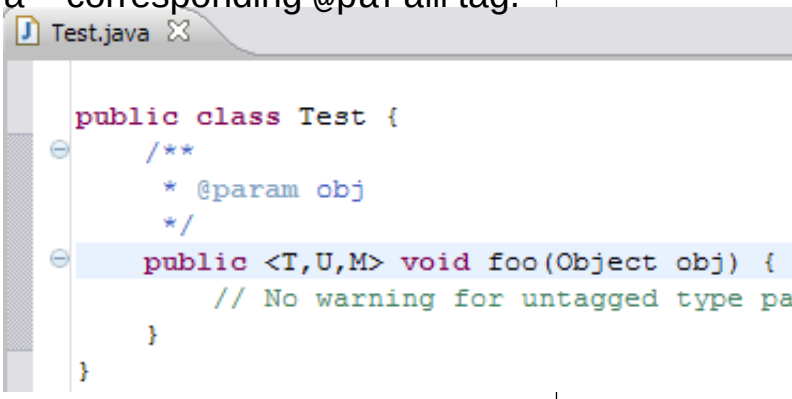
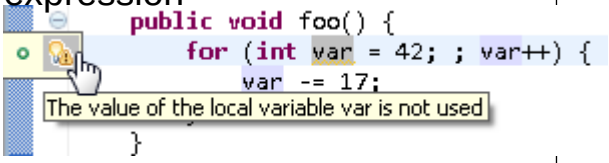
Java Formatter

New line after annotations	The **Insert new line after annotations on members** option on the '**New Lines**' tab of the formatter preferences has been split up to support packages, types, fields and methods:
	 
	
	For example, with this preference activated as shown above, the formatted code looks like this:
	`@Deprecated package pkg;`
	
	`@Deprecated public class Foo {`
	`  @Deprecated public int field;`
	
	`  @Deprecated public void bar(@SuppressWar`
	`  int param) {`
	`  @SuppressWarnings("unused") int loca`
	`  }`
	`}`
	
	Formatter profiles can be configured on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Formatter** preference page.

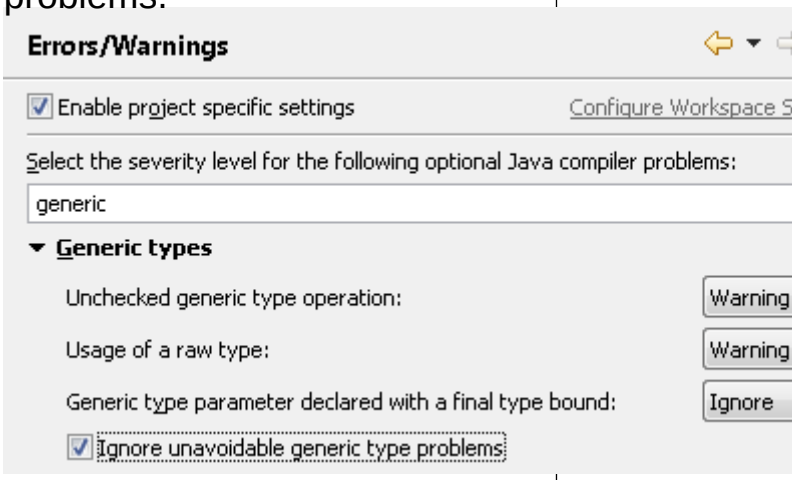
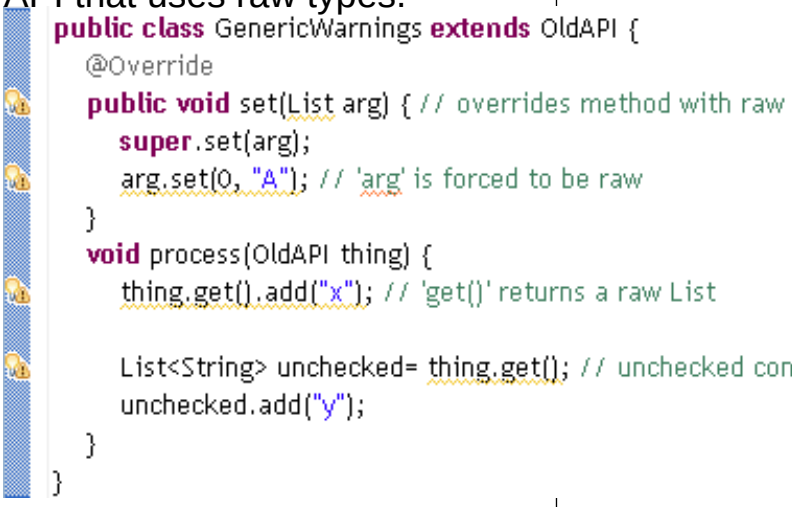
Preserve whitespace before line comment	A code formatter option has been added to preserve white space between code and line comments:
	`class Test {`
	`  void trailingCommented() {`
	`  System.out.println("indented");  // comment`
	`  System.out.println("indent");  // comment`
	`  }`
	This preference can be enabled on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Code Style > Formatter** preference page on the **Comments** tab:
	 

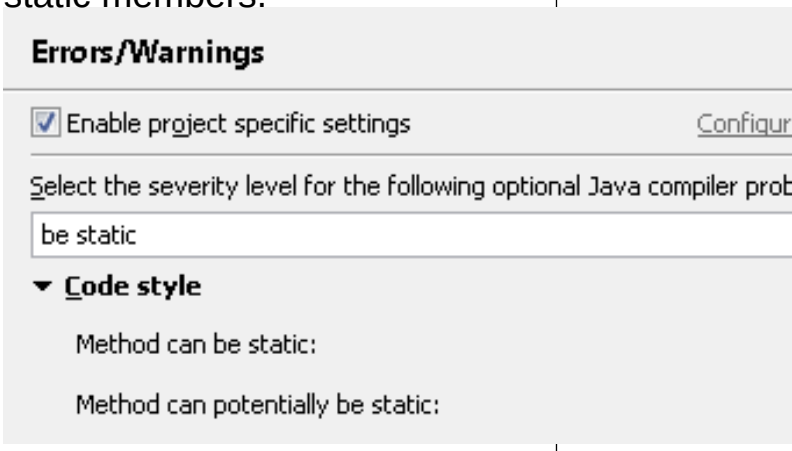
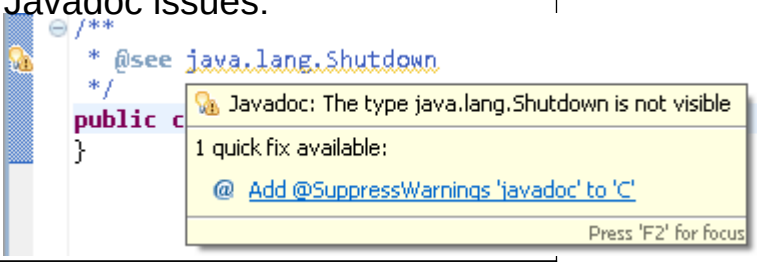
Java Compiler	

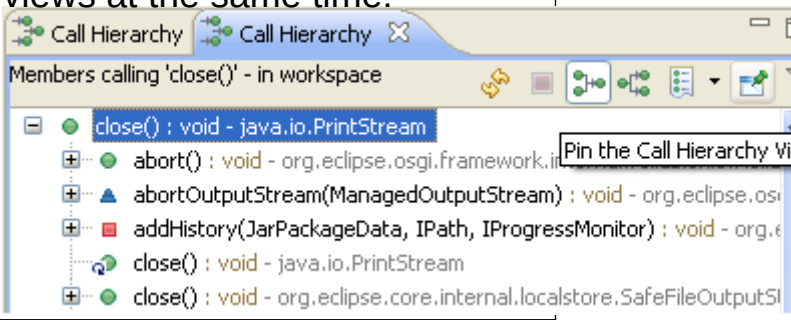
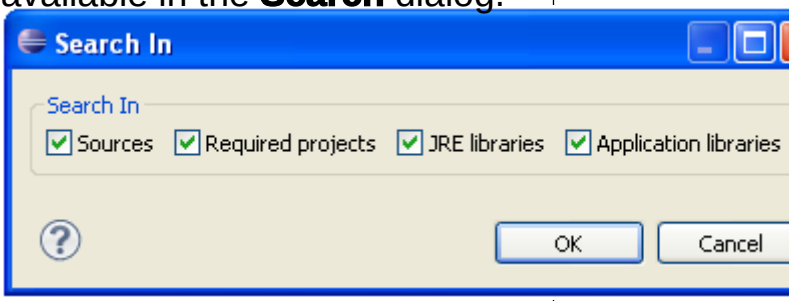
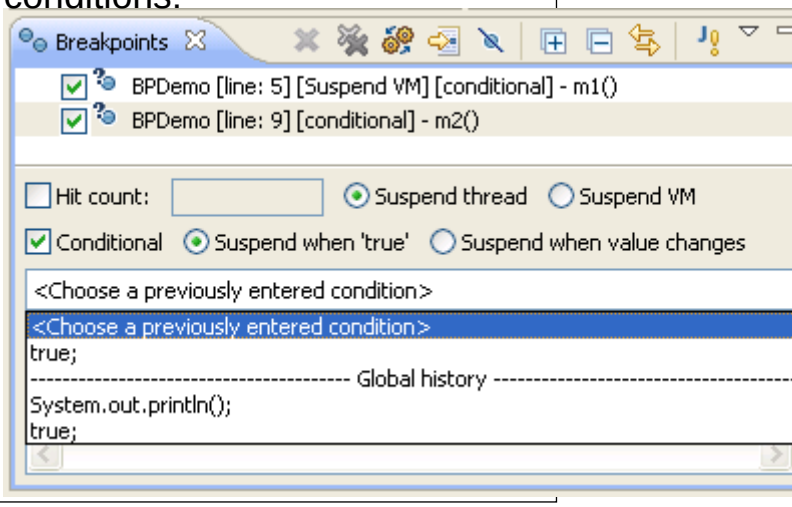
New option to include 'assert' in null analysis	JDT now gives the flexibility to enable or disable null-related errors/warnings on variables that got marked as potentially or definitely null inside an assert statement. If you have runtime asserts enabled, you can also enable the new warning to see the null-related problems being reported because of null-related checks inside the assert statement. If the option is disabled, null checks in asserts are not considered.  The new option **Include 'assert' in null analysis** is disabled by default and can be enabled on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page.

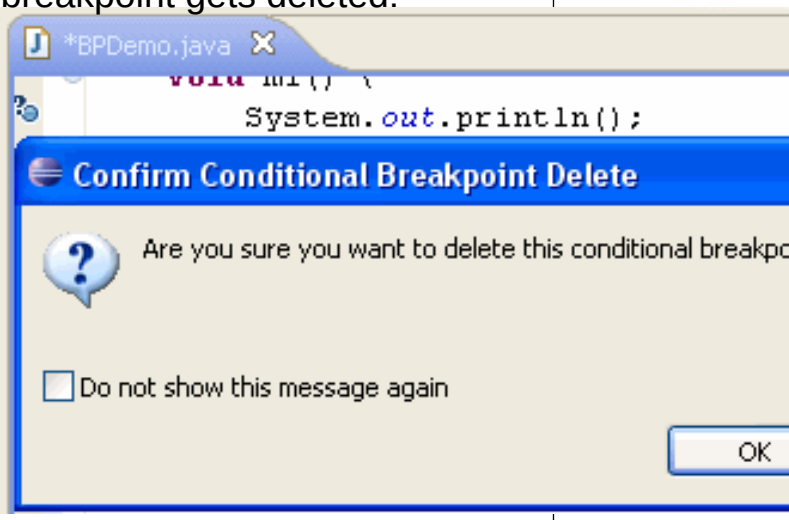
Missing Javadoc tags for method type parameters no longer reported by default	JDT now provides an option to enable or disable the missing Javadoc tag error or warning for a method type parameter without a corresponding `@param` tag.  The new **Ignore method type parameters** option is enabled by default and can be disabled on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page.
Improved detection of unused local variables, parameters and fields	JDT now reports local variables, parameters, and fields of a primitive type as unused in the following scenarios: compound assignmentprefix/postfix increment/decrement expression  In case the above expressions involve an unboxing operation, the variable is not reported as unused. We also changed the wording from "X has never been read" to "The value of X is not used."

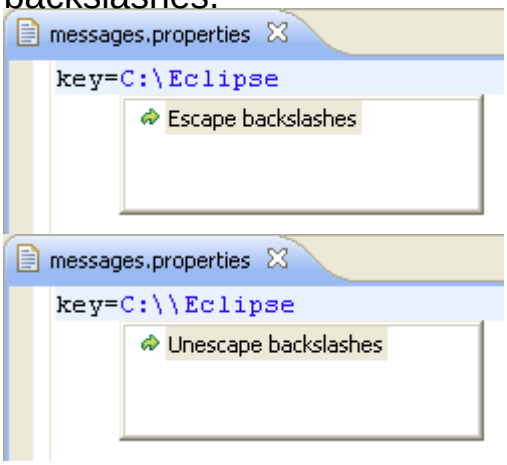
Filter preferences on Java > Compiler > Errors/Warnings page	You can now filter preferences on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** page by preference label text or by preference value. A word in the filter string preceded by '~' is used to filter on preference values. Examples: param~off~ignoreparam ~enabled~ignore param

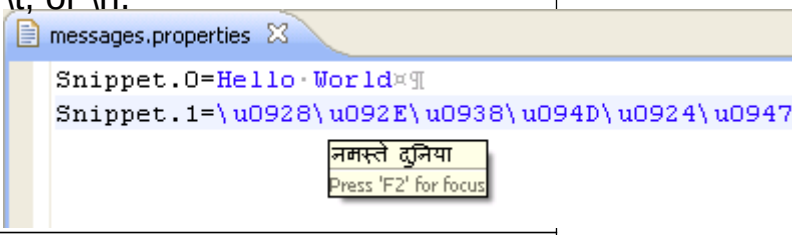
Unavoidable generic type problems	A new compiler option has been added that suppresses unavoidable generic type problems.  Those problems only show up because a type refers to an old API that uses raw types.  When the old API is eventually generified, then these problems either go away, or you will see a compile error because the type arguments you used are not correct. This option is disabled by default but can be enabled on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Compiler > Errors/Warnings** preference page.

Compiler detects methods that can be static	Two new compiler options have been added that mark methods which can be made static because they only refer to static members.  The first option marks private and final methods than can always be made static. The second option also marks other methods. Note that methods can be overridden in a subclass, so if you make a "potentially static" method static, this may break existing clients. These options are disabled by default.
On-disk JRE change detection	In the past, if a system update or another event changed a JRE used in Eclipse, those changes would not be reflected in Eclipse: In some cases, this caused the JRE to stop functioning. Now, these changes are accounted for, and any JRE that changed on-disk (external to Eclipse) will be updated accordingly when Eclipse restarts.
New "javadoc" token for `@SuppressWarnings` annotation	The `@SuppressWarnings` annotation can now also be used to suppress compiler warnings/errors related to Javadoc issues: 
Java Views and Dialogs	

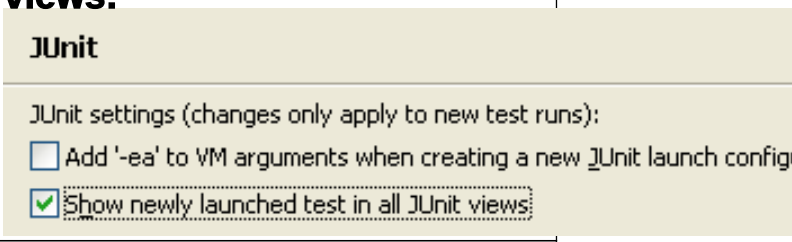
Open Type Hierarchy on multiple type containers	The **Open Type Hierarchy** action now also works on multiple type containers: You can select multiple packages, source folders, or projects, and then open a hierarchy that contains all types in the selected containers.
Pin the Call Hierarchy view	The **Call Hierarchy** view now allows to pin the current view, which allows you to open multiple Call Hierarchy views at the same time: 
Configure 'Search In...' in Call Hierarchy	A new **Search In...** action has been added to the **Call Hierarchy** view menu. The action shows the **Search In** dialog with the same options that are already available in the **Search** dialog. 
History for breakpoint conditions	There is now a history for recently used breakpoint conditions: 

Prompt before deleting conditional breakpoints	In the Java editor, if you delete a breakpoint that has a condition set on it, you will now be prompted before the breakpoint gets deleted:  You can specify that you would not like to be notified again when trying to remove a conditional breakpoint. This can later be changed again via **Prompt for confirmation when deleting a conditional breakpoint from editor** on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug** preference page.

Step filtering for bean-style getters and setters	When single stepping, you can now automatically step over simple getters and setters during **Step Into**: Consider single stepping into statements like: `computeEffect(meters.getCurrent(), meters.getVoltage());` Normally, this means first stepping into getCurrent, then into getVoltage, and finally into computeEffect. You can now skip the first two steps (provided they are simple bean-style getters), and jump directly into the computeEffect method, by selecting the **Filter simple getters** check box on the [Image: /topic/org.eclipse.help/command_link.png]**Java > Debug > Step Filtering** preference page. Similarly, selecting **Filter simple setters** avoids stopping inside simple bean-style setters when using **Step Into**.
Properties File Editor	
New "Escape backslashes" and "Unescape backslashes" quick assists in properties file editor	The properties file editor now offers two new quick assists (**Ctrl+1**) to escape or unescape backslashes.  The **Escape backslashes** quick assist is also offered automatically on paste if the pasted text contains backslashes that should be escaped.

Auto-escaped characters and editor hovers	The properties file editor now auto-escapes typed and pasted non-ISO-8859-1 characters and shows a hover containing the real text if the underlying text contains special characters like \uHHHH, \t, or \n. 
JUnit	

New JUnit Test Suite wizard supports JUnit 4	The **New > JUnit Test Suite** wizard can now also create JUnit 4 test suites:
	Result:
	`@RunWith(Suite.class)`
	`@SuiteClasses({ ListTest.class, SimpleTest.class })`
	`public class AllTests {`
	`}`

Show latest JUnit test in all windows	By default, a new JUnit test run only shows up in the window where the test was started. If you prefer to see the latest test in all open JUnit views, select [Image: /topic/org.eclipse.help/command_link.png]**Preferences > Java > JUnit > Show newly launched test in all JUnit views:** 
Paste URL into JUnit view	Apart from drag-and-drop or **Test Run History > Import from URL...**, you can now also paste a URL into the JUnit view to load test results from the web. Supported test run formats are XML files exported from the JUnit view or generated by the Ant JUnit task.

What's new for Java 7

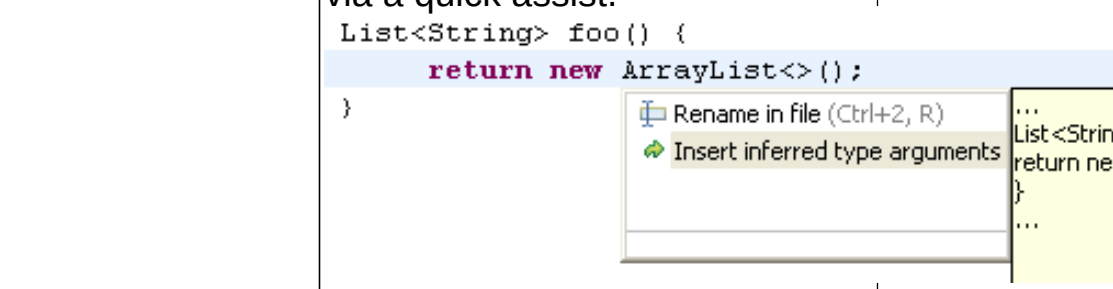
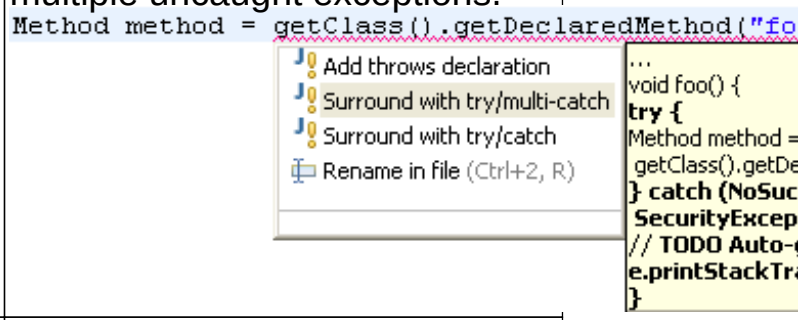
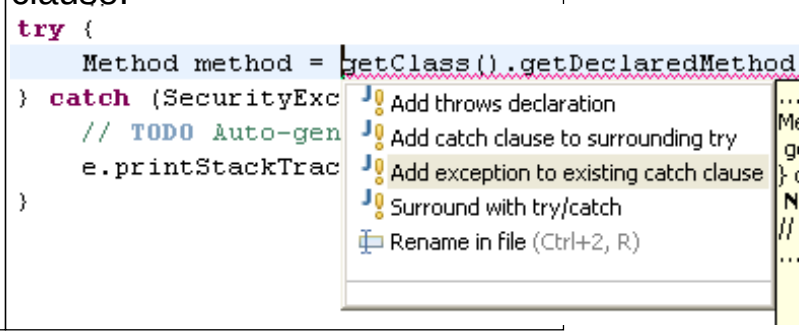
- Improved Type Inference for Generic Instance Creation (Diamond)
- Multi-catch
- `try`-with-resources statement
- Simplified Varargs Method Invocation
- Strings in switch
- Polymorphic Methods
- Miscellaneous

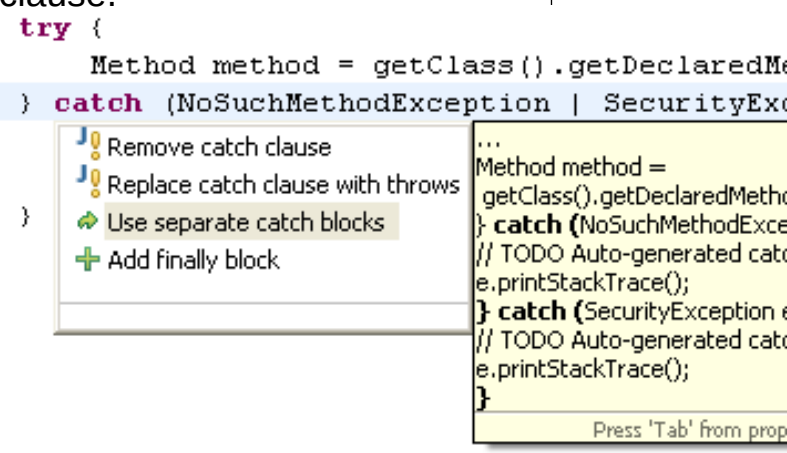
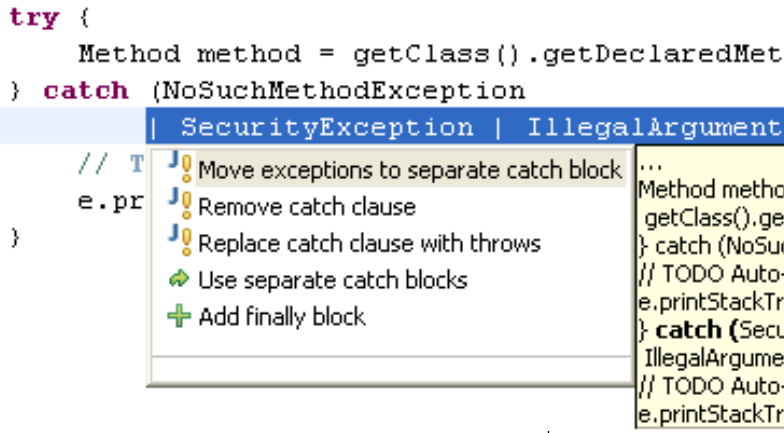
Java 7 comes with a set of small enhancements to the Java language (aka Project Coin), a new byte code to dynamically invoke methods, and many additions to the libraries. The Eclipse compiler implements all the new features of Java 7 and the IDE also supports them now.

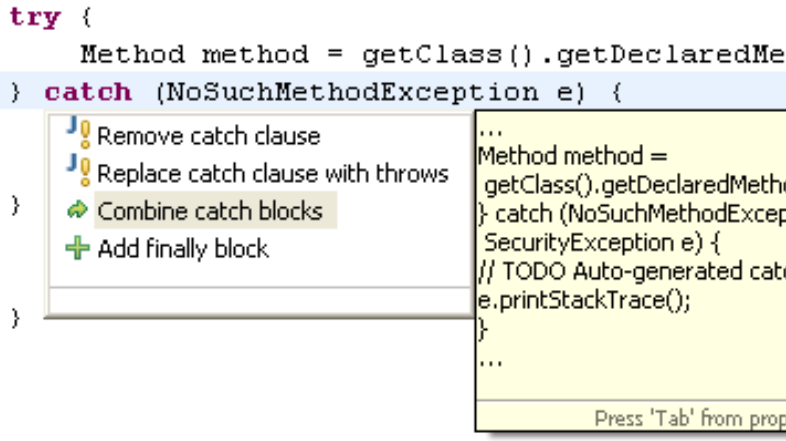
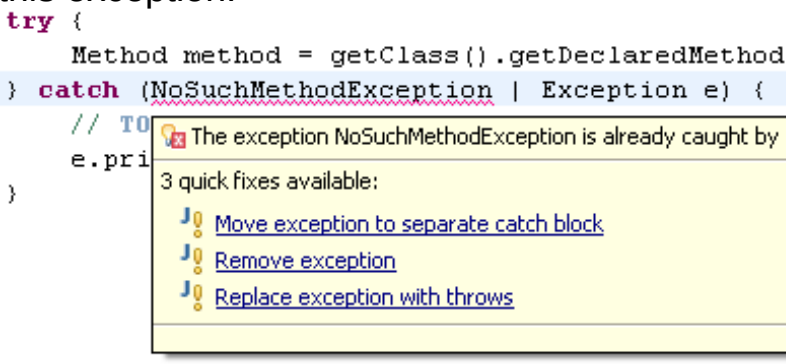
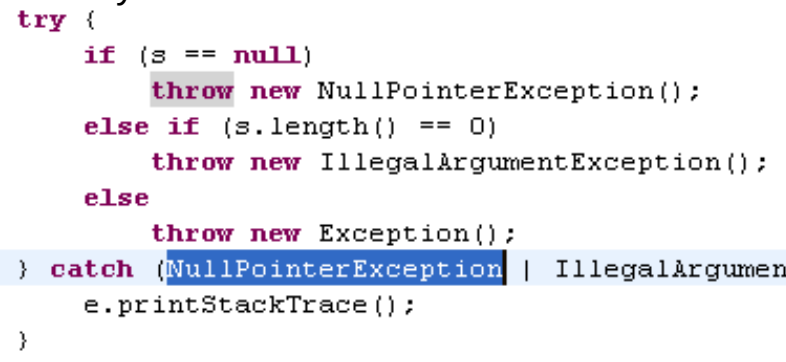
Improved Type Inference for Generic Instance Creation (Diamond)	

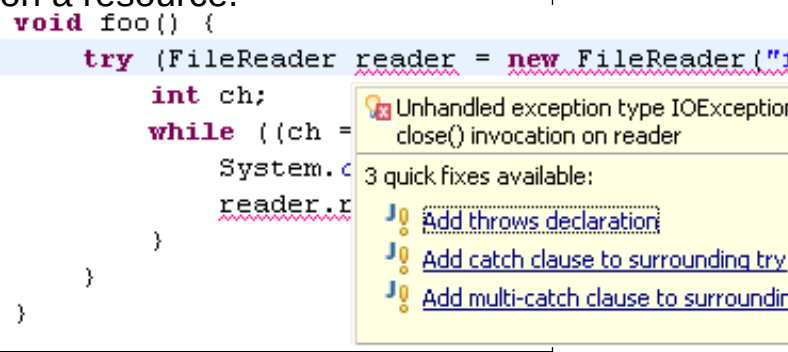
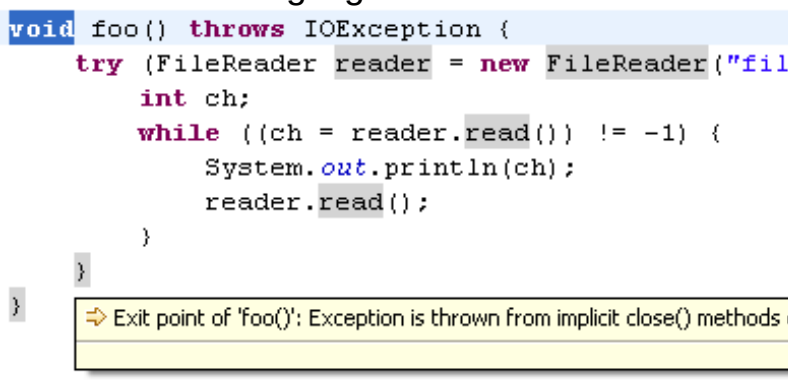
Content assist inserts diamond	Where possible, content assist for constructor invocations now inserts a diamond instead of explicit type arguments.
	`HashMap<Integer, List<String>> map= new HashMap`
	Result:
	`HashMap<Integer, List<String>> map= new HashMap<>()`

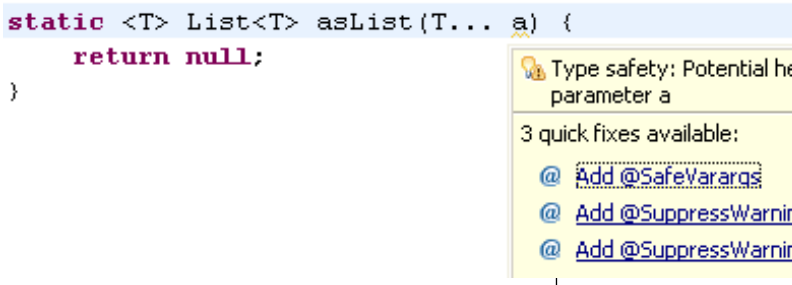
Detection of redundant type arguments	The compiler can now detect **redundant specification of type arguments**, which you can remove via the **Remove type arguments** quick fix.
	`void foo() {`
	`List<String> a = new ArrayList<String>();`
	`}`
	Redundant specification of ty
	3 quick fixes available:
	Remove type arguments
	Add @SuppressWarnings 'u
	Add @SuppressWarnings 'u
	This option is disabled by default but can be enabled on the **Java > Compiler > Errors/Warnings** preference page:

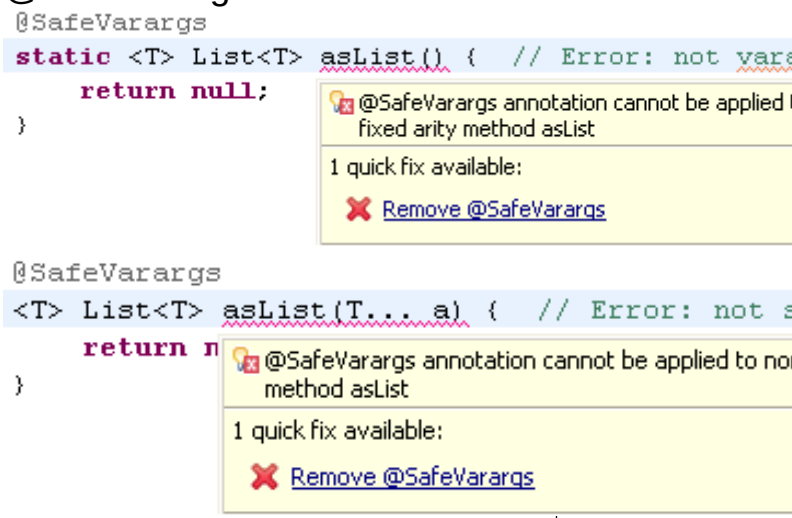
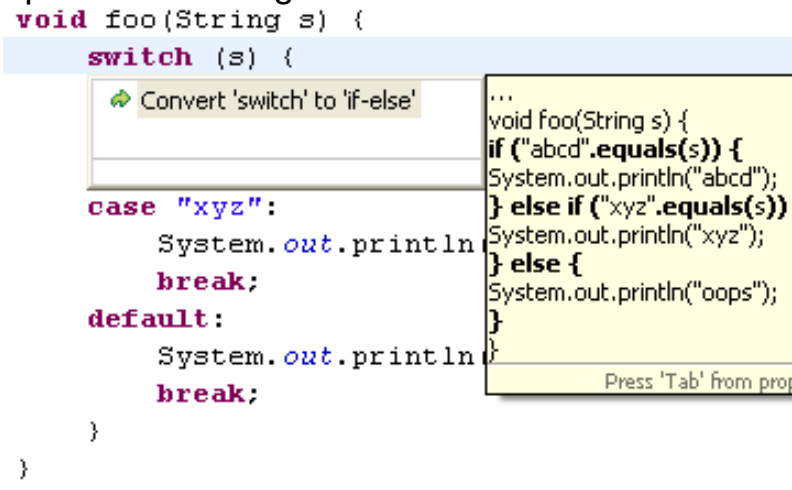
New Insert inferred type arguments quick assist	You can **insert inferred type arguments** of a diamond via a quick assist.  Hint: This also works as a quick fix in 1.5 and 1.6 code, where the diamond is a syntax error.
Multi-catch	
New Surround with try/multi-catch quick fix	The new action **Source > Surround With > Try/multi-catch Block** allows you to surround selected statements with a try/multi-catch block. This is also available as **Surround with try/multi-catch** quick fix in case there are multiple uncaught exceptions. 
New Add exceptions to existing catch clause quick fix	The new **Add exceptions to existing catch clause** quick fix allows you to add uncaught exceptions to an existing catch clause. 

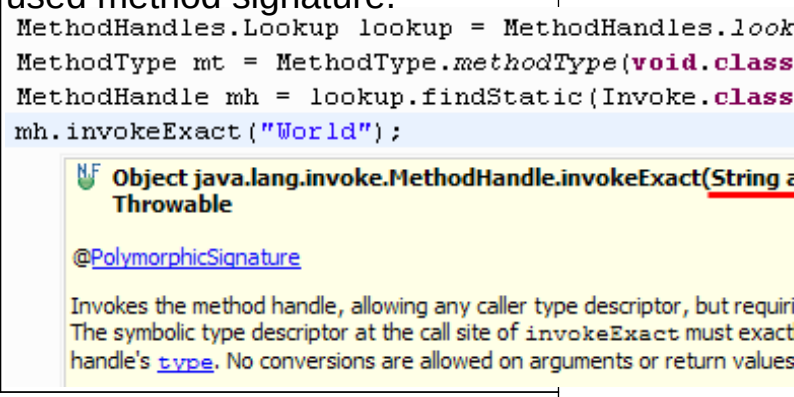
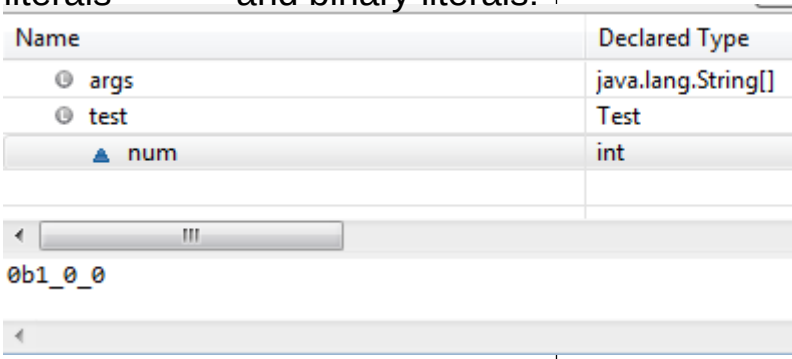
New Use separate catch blocks quick assist	The new **Use separate catch blocks** quick assist allows you to replace a multi-catch clause with individual catch blocks, one for each exception in the multi-catch clause. 
New Move exceptions to separate catch block quick assist	The new **Move exceptions to separate catch block** quick assist allows you to pick out one or more selected exceptions from a multi-catch clause. 

New Combine catch blocks quick assist	The new **Combine catch blocks** quick assist allows you to combine separate catch blocks into a single multi-catch block. The quick assist is offered only when bodies of all the catch blocks are same. 
New Remove exception quick fix	The compiler gives an error if an exception in a multi-catch clause is already caught by an alternative exception. The new **Remove exception** quick fix allows you to remove this exception. 
Mark Occurrences	**Mark Occurrences** has been updated to understand the multi-catch syntax. 

Formatter	There are new **Line Wrapping options** in the formatter for the multi-catch syntax. These can be configured on **Java > Code Style > Formatter** preference page under **Line Wrapping > Statements > 'multi-catch'**.
try-with-resources statement	
Detection of unhandled exceptions thrown by automatic close()	The compiler detects unhandled exceptions thrown by automatic `close()` invocation on a resource. 
Mark Occurrences	**Mark Occurrences** has been updated to understand the `try`-with-resources syntax. The closing '}' of a `try`-with-resources statement is marked as a method exit point if the implicit `close()` invocation throws an exception. The corresponding resource variable is also highlighted. 

Formatter	There are new **Line Wrapping and White Space options** in the formatter for the `try`-with-resources syntax. These can be configured on **Java > Code Style > Formatter** preference page under **Line Wrapping > Statements > 'try-with-resources'** and **White Space > Control Statements > 'try-with-resources'**.
Simplified Varargs Method Invocation	
New Add @SafeVarargs quick fix	The new **Add @SafeVarargs** quick fix is offered for potential heap pollution warnings on method declarations.  The quick fix is also offered from call sites. 

New Remove @SafeVarargs quick fix	The new **Remove @SafeVarargs** quick fix is offered for incorrect usage of @SafeVarargs annotation. 
Strings in switch	
Convert 'switch' to 'if-else' quick assist	The **Convert 'switch' to 'if-else'** quick assist has been updated for strings in switch.  Note that the quick assist avoids a NullPointerException in the resultant code.
Polymorphic Methods	

Polymorphic method signature in Javadoc hover	Javadoc hovers for references to polymorphic methods show the actually used method signature. 
Miscellaneous	
Evaluation support for Java 7	The evaluation engine in JDT debug has been updated to support Java 7. You can now use all of the new code structures from Java 7 in: Display / Inspect actionsDisplay viewAssign variable value operationsBreakpoint conditions For example, switch on string is supported in breakpoint conditions:
Execution environment for Java 7	When configuring JREs and execution environments, you now have the ability to configure a Java 7 execution environment:
Assign variable value support for Java 7	The feature for changing the value of a variable while debugging has been updated for Java 7 to support underscores in literals and binary literals: 
API Tools support for Java 7	The execution environment description for Java 7 is available from the corresponding update site. After installing it, one can detect invalid references to Java 7 system libraries:

Notices

Visually developing Java applications by using domain modeling diagrams

You can use domain modeling diagrams to visually represent and develop artifacts of Java™ applications in a single, tightly integrated development environment.

About this task

You can use domain modeling diagrams to represent and analyze an existing system to identify the system's components and interrelationships and to create representations of the system in another form. You can use domain modeling diagrams automatically abstract the system's structural information from code to a new form at a higher abstraction level. You can redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally developed. You can also modify the target system or develop and generate new systems.A domain modeling class diagram depicts some or all of the components or elements in an application. You can use class diagrams to visually represent and develop the structures and relationships for Java projects, packages, classes, interfaces, and enum types. You can create your own context to understand, collaborate, and develop an application by using a subset of its components, such as packages, classes, interfaces, and enum types. You can also develop Java elements directly from class diagrams.

You can use sequence diagrams to visually represent and develop behaviors and interactions of Java applications or to visually represent Java methods.

You can use temporary, non-editable browse diagrams to create quick static views and explore existing relationships in applications, and use non-editable topic diagrams to create dynamic views of applications based on context and queries.

You can also generate Javadoc HTML documentation with domain modeling diagram images to provide more information about the source code.

- **Setting preferences for handling older format diagrams**
  You can specify the default preference for handling domain modeling diagrams that are in an older format when you open them in the diagram editor. You can always keep diagrams in the old format, always convert them to the current format, or receive a prompt each time you open older format diagrams.
- **Visually developing structural features of Java applications by using domain modeling class diagrams**
  You can use domain modeling class diagrams to visually represent and develop structural features in Java applications.
- **Navigating to Java code elements from domain modeling diagrams**
  You can navigate from domain modeling diagrams to the source code of Java diagram elements or to Java elements (classes and interfaces) in the Package Explorer view. You can also start Java editors from the diagram editor to make modifications to the source code of Java elements.
- **Generating Javadoc HTML documentation with domain modeling diagram images**
  You can generate Javadoc HTML documentation that contains domain modeling diagram images to provide more information about the source code inside Java projects.

Setting preferences for handling older format diagrams

You can specify the default preference for handling domain modeling diagrams that are in an older format when you open them in the diagram editor. You can always keep diagrams in the old format, always convert them to the current format, or receive a prompt each time you open older format diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling**, expand **Java**, and click **Old Format Diagrams Handling**.
3. On the Old Format Diagrams Handling page, complete one of the following steps:
   - To keep diagrams in the old format, click **Leave in old format**.
   - To always convert old format diagrams to the current format, click **Convert to current format**.
   - To receive a prompt each time you open old format diagrams, click **Ask every time**.
4. Click **OK**.

Related tasks:

Visually developing structural features of Java applications by using domain modeling class diagrams

Navigating to Java code elements from domain modeling diagrams

Generating Javadoc HTML documentation with domain modeling diagram images

Visually developing structural features of Java applications by using domain modeling class diagrams

You can use domain modeling class diagrams to visually represent and develop structural features in Java™ applications.

About this task

A domain modeling class diagram depicts some or all the projects, packages, classes, and interfaces in an application. You can create your own context to visually represent an application by using a subset of its packages, classes, and interfaces. You can use domain modeling class diagrams to develop Java packages, classes, interfaces, and enum types and to design their structures and relationships.

- **Visually developing Java elements by using domain modeling class diagrams**
  You can use domain modeling class diagrams to visually develop and edit Java packages, classes, interfaces, and enum types in your applications.
- **Visually developing relationships between Java elements by using domain modeling class diagrams**
  You can use domain modeling class diagrams to develop relationships between Java elements to visually represent and develop your Java applications.

Related tasks:

Setting preferences for handling older format diagrams

Navigating to Java code elements from domain modeling diagrams

Generating Javadoc HTML documentation with domain modeling diagram images

Creating and managing domain modeling class diagrams for visual representation of applications

You can create new domain modeling class diagrams and then populate them with existing or new source elements to visually represent and develop structures and relationships in C/C++, Java™ 2 Platform, Standard Edition (J2SE), Java 2 Platform, Enterprise Edition (J2EE), and Enterprise JavaBeans 3.0 (EJB 3.0) applications and Web Services Definition Language (WSDL) services.

- **Creating domain modeling class diagrams for visual development of structural features in applications**
  You can create new domain modeling class diagrams and then populate them with existing or new source elements to visually represent and develop structures and relationships in applications.
- **Managing visual representation of source elements in domain modeling class diagrams**
  You can populate existing domain modeling class diagrams by adding source elements to diagrams to visually represent and develop structures and relationships in applications.
- **Managing visual representation of classifiers in domain modeling class diagrams**
  You can show or hide attributes and operations of classifiers in domain modeling class diagrams to manage visual representation of applications.
- **Deleting domain modeling elements from class diagrams and projects**
  You can delete domain modeling elements that you no longer require either from a class diagram and the project or from only the diagram.

Related reference:

UML modeling best practices

Creating domain modeling class diagrams for visual development of structural features in applications

You can create new domain modeling class diagrams and then populate them with existing or new source elements to visually represent and develop structures and relationships in applications.

About this task

Before you create new domain modeling class diagrams, you can set the default global preferences for attributes and operations, such as visibility styles, showing or hiding attributes and operations, showing or hiding operation signatures, and showing or hiding parent names of classifiers.Showing the parent name of a classifier displays the fully qualified package name in which the classifier is owned. It is useful to show parent names, for example, when you need to know the containing projects of classifiers or you have a class diagram that displays classifiers that belong to different packages.

- **Creating class diagrams to visually develop applications**
  You can create new domain modeling class diagrams to visually represent and develop structures and relationships in applications.
- **Creating class diagrams of source elements in applications**
  You can create domain modeling class diagrams of existing source elements, including projects, packages, entity beans, classes, interfaces, and data types, to visually represent and develop structures and relationships in applications.

Related concepts:

Domain modeling class diagrams

Related tasks:

Managing visual representation of source elements in domain modeling class diagrams
Managing visual representation of classifiers in domain modeling class diagrams
Deleting domain modeling elements from class diagrams and projects

Domain modeling class diagrams

In C/C++, Java™, Enterprise JavaBeans (EJB), and Web Services Description Language (WSDL) visual development, a domain modeling class diagram provides a graphical representation of the structures and relationships of components in an application.

For example, a class diagram can depict some or all of the components or elements in an application. You can use class diagrams to create your own context to understand, collaborate, and design using a subset of the components or elements in an application.

Class diagrams use multiple variations of association relationships to indicate which classifiers need to share data with other classifiers.

Related concepts:

Classifiers in domain modeling class diagrams

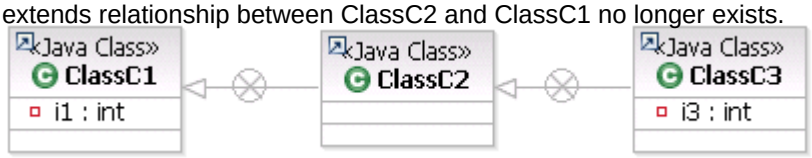
Adornments for unresolved references in domain modeling class diagrams

In domain modeling class diagrams, Java™ classes and interfaces or Enterprise JavaBeans (EJB) enterprise beans belonging to different applications or projects that are not currently loaded in the workspace can result in an unresolved reference.

Unresolved references also occur if classes or fields are deleted when the diagram update policy is disabled or if actions performed outside diagrams that result in broken references. Unresolved references are shown in class diagrams with different adornment icons to indicate the potential sources of the unresolved references.

For example, a classifier or a relationship in a class diagram that has an unresolved reference (that is, the enterprise bean or Java class or interface that the diagram element represents cannot be found.) has a special unresolved view adornment to indicate that this classifier or relationship has an unresolved reference.

As the following figure illustrates, the circled "x" on the supplier side of the generalization relationship between ClassC3 and ClassC2 indicates that the relationship resides in ClassC3, but the supplier diagram element (ClassC2) is unresolved. The generalization relationship between ClassC2 and ClassC1 is adorned with a circled "x", which indicates that the Java extends relationship between ClassC2 and ClassC1 no longer exists.



Creating class diagrams to visually develop applications

You can create new domain modeling class diagrams to visually represent and develop structures and relationships in applications.

Before you begin

You must be in the Package Explorer or Project Explorer view.

Procedure

1. Click **File** > **New** > **Class Diagram**.
2. In the New Class Diagram wizard, in the **Enter or select the parent folder** field, specify the parent folder.
3. In the **File name** field, type a file name.
4. Optional: To enable the associated capabilities, click **Next** and complete the following steps:
   A. Select the **Customize UI visibility for this diagram by selecting capabilities below** check box.
   B. From the**Capabilities** list, expand a category and select the capabilities.
5. Click **Finish**.

Results

A new class diagram is created and opens in the diagram editor. **Tip:** You can also create class diagrams from the Package Explorer or Project Explorer view: Right-click the **Diagrams** folder; then click **Create Diagram** > **Class Diagram**.

Related concepts:

Domain modeling class diagrams

Related tasks:

Creating class diagrams of source elements in applications

Creating class diagrams of source elements in applications

You can create domain modeling class diagrams of existing source elements, including projects, packages, entity beans, classes, interfaces, and data types, to visually represent and develop structures and relationships in applications.

Before you begin

You must be in the Package Explorer or Project Explorer navigation view.

Procedure

1. In the navigation view, select and right-click a source element or selected source elements; then click **Visualize** > **Add to New Diagram File** > **Class Diagram**.
2. In the New Class Diagram wizard, in the **Enter or select the parent folder** field, specify the parent folder.
3. In the **File name** field, type a file name.
4. Optional: To enable the associated capabilities, click **Next** and complete the following steps:
   A. Select the **Customize UI visibility for this diagram by selecting capabilities below** check box.
   B. From the **Capabilities** list, expand a category and select the capabilities.
5. Click **Finish**.

Results

A class diagram is created and opens in the diagram editor with the selected source elements in it.**Tip:** If any selected source elements are Java™ 2 Platform, Standard Edition (J2SE) 5.0 classes with annotations, the details of the annotations are shown in the annotation compartments of the classes in the diagram. You can also add or edit annotations on the Annotation tab in the Properties view.

Related tasks:

Creating class diagrams to visually develop applications

Managing visual representation of source elements in domain modeling class diagrams

You can populate existing domain modeling class diagrams by adding source elements to diagrams to visually represent and develop structures and relationships in applications.

About this task

You can also populate existing domain modeling class diagrams by showing source elements that are related to each other by specific types of relationships and by specifying the levels of depth to expand the related source elements.

- **Populating domain modeling class diagrams with source elements in applications**
  You can populate existing domain modeling class diagrams with source elements to create visual representation of applications.
- **Customizing queries for showing related source elements in domain modeling class diagrams**
  You can customize queries to show related source elements in domain modeling class diagrams to visually represent and develop applications.
- **Showing related elements in domain modeling diagrams**
  You can populate a domain modeling class diagram by showing existing source elements that are related by specific relationships to the selected source elements.
- **Showing and hiding relationships between source elements in domain modeling class diagrams**
  You can update an open domain modeling class diagram or selected classifiers in a domain modeling class diagram by showing or hiding relationships between source elements that match your selected criteria.

Related concepts:

Domain modeling class diagrams

Related tasks:

Creating domain modeling class diagrams for visual development of structural features in applications

Managing visual representation of classifiers in domain modeling class diagrams

Deleting domain modeling elements from class diagrams and projects

Populating domain modeling class diagrams with source elements in applications

You can populate existing domain modeling class diagrams with source elements to create visual representation of applications.

Before you begin

You must be in the Package Explorer or Project Explorer navigation view and have a class diagram open.

Procedure

To populate a class diagram with source elements, in the navigation view, right-click a source element or selected source elements; then click **Visualize** > **Add to Current Diagram**.

Related concepts:

Domain modeling class diagrams

Related tasks:

Customizing queries for showing related source elements in domain modeling class diagrams

Showing related elements in domain modeling diagrams

Showing and hiding relationships between source elements in domain modeling class diagrams

Customizing queries for showing related source elements in domain modeling class diagrams

You can customize queries to show related source elements in domain modeling class diagrams to visually represent and develop applications.

Before you begin

You must have a class diagram open.

Procedure

1. In the diagram editor, right-click a classifier; then click **Filters** > **Show Related Elements**.
2. In the Show Related Elements in Diagram window, click **Details** if the details page is not already open.
3. Under **Relationship Types**, select the relationships to show and specify the expansion directions and levels of depth; then click **Save As**.
4. In the Save As window, type a name for the new query and click **OK**.
5. Click **OK**.

Related concepts:

Domain modeling class diagrams

Related tasks:

Populating domain modeling class diagrams with source elements in applications

Showing related elements in domain modeling diagrams

Showing and hiding relationships between source elements in domain modeling class diagrams

Showing related elements in domain modeling diagrams

You can populate a domain modeling class diagram by showing existing source elements that are related by specific relationships to the selected source elements.

Before you begin

You must have a diagram open.

Procedure

1. In the diagram editor, right-click a source element or a group of selected source elements; then click **Filters** > **Show Related Elements**.
2. In the Show Related Elements in Diagram window, complete one of the following steps:
   - To perform an existing query, under **Custom Query**, click a query in the list.
   - To perform a new query, click **Details**, specify the details for the query; then click **Save As** and, in the Save As window, type a name for the new query and click **OK**.
3. click **OK**.

Related concepts:

Domain modeling class diagrams

Related tasks:

Populating domain modeling class diagrams with source elements in applications

Customizing queries for showing related source elements in domain modeling class diagrams

Showing and hiding relationships between source elements in domain modeling class diagrams

Showing and hiding relationships between source elements in domain modeling class diagrams

You can update an open domain modeling class diagram or selected classifiers in a domain modeling class diagram by showing or hiding relationships between source elements that match your selected criteria.

Before you begin

Relationships must already exist between source elements in an application before you can show or hide relationships.

Note: Hiding relationships removes the connectors from the diagram only. The underlying semantic relationships remain intact.

You must have a class diagram open.

Procedure

1. In the diagram editor, right-click selected source elements; then click **Filters** > **Show/Hide Relationships**.**Tip:** It is not necessary to select any source element if you want to show the relationships between all the elements in the diagram.
2. In the Show/Hide Relationships window, select the types of relationships to show or hide.
3. Click **OK**.

Related tasks:

Populating domain modeling class diagrams with source elements in applications

Customizing queries for showing related source elements in domain modeling class diagrams

Showing related elements in domain modeling diagrams

Managing visual representation of classifiers in domain modeling class diagrams

You can show or hide attributes and operations of classifiers in domain modeling class diagrams to manage visual representation of applications.

About this task

You can show operation signatures and the parent names of classifiers. Showing the parent name of a classifier displays the fully qualified package name in which the classifier is owned. It is useful to show parent names, for example, when you need to know the containing projects of classifiers or you have a class diagram that displays classifiers that belong to different packages.You can also show the fully qualified names of individual classifiers for type proposals in other packages.

- **Showing and hiding attributes and operations in domain modeling class diagrams**
  You can show or hide attributes and operations of individual classifiers so that only the elements that are important to you are displayed in a domain modeling class diagram for better visual representation of applications
- **Specifying the visibility style for attributes and operations in domain modeling class diagrams**
  You can specify the visibility style for attributes and operations in domain modeling class diagrams to meet your needs for visual representation of applications.
- **Showing operation signatures of classifiers in domain modeling class diagrams**
  You can show the operations of classifiers with the full signature of parameters along with the full package names in domain modeling class diagrams to visually represent structures and relationships in applications.
- **Showing parent names of classifiers in domain modeling class diagrams**
  You can show the names or qualified names of parents of individual classifiers in domain modeling class diagrams to visually represent relationships between source elements in applications.
- **Showing fully qualified names of classifiers in domain modeling class diagrams**
  In domain modeling class diagrams, you can show the fully qualified names of individual classifiers for type proposals in other packages to visually represent relationships between source elements in applications.

Related tasks:

Creating domain modeling class diagrams for visual development of structural features in applications

Managing visual representation of source elements in domain modeling class diagrams

Deleting domain modeling elements from class diagrams and projects

Classifiers in domain modeling class diagrams

In C/C++, Java™, Enterprise JavaBeans (EJB), and Web Services Description Language (WSDL) visual development, a classifier is the superclass of classes, interfaces, and data types that have similar syntax and, therefore, are all notated by using a rectangle with keywords in domain modeling class diagrams.

Because classes are most common in domain modeling class diagrams, a rectangle without a keyword represents a class, and the other subclasses of classifiers are indicated with keywords.

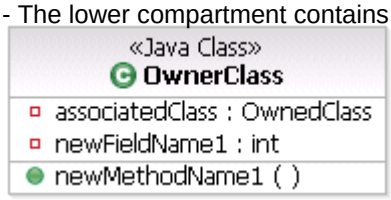
A class is a class diagram element that represents an application element. An application element can be represented by multiple instances of a class in one or more class diagrams.

You use classes to illustrate conceptually related application elements in diagrams. A class diagram element shows all or part of the underlying semantics. A class diagram element identifies the attributes, operations, relationships, and semantics that instances of the class possess. Every object that instantiates a class usually provides its own attribute values. Attributes are also called variables, member variables, properties, and fields, but are usually implemented as variables.

Each class diagram element has properties that govern its appearance and location in a diagram. Modifying properties of a diagram element only changes the appearance of the element and does not affect the underlying semantics or any other diagram element that represents that application element.

As the following figure illustrates, a class is displayed in a diagram as a rectangle with three compartments:

- The upper compartment contains the class name.
- The middle compartment contains the attributes.
- The lower compartment contains the operations.



You can show, hide, or collapse the attribute and operation compartments. You can use additional compartments to display other details such as constraints or signals that instances of the class can receive.

The classes in an application usually appear in class diagrams. You can add classes or instances of classes (objects or classifier roles) to diagrams to represent the following items:

- Workers and artifacts of a business
- Components or building blocks of a software system, such as Java classes and data types

Example

An e-commerce application might include a Cart class. The class defines an itemList attribute, and an addItem operation that belong to all objects of type Cart. At run time, multiple instances of the Cart class might be created, each possessing the attributes and operations that the class defines. The values of the attributes for each instance will differ if, for example, one class object calls the addItem operation to add videos to its itemList attribute, while another instance uses the same operation to add books.

Related concepts:

Domain modeling class diagrams

Related tasks:

Showing parent names of classifiers in domain modeling class diagrams

Showing fully qualified names of classifiers in domain modeling class diagrams

Attributes in domain modeling class diagrams

In domain modeling class diagrams, an attribute represents a data definition for an instance of a classifier. An attribute describes a range of values for that data definition.

A classifier can have any number of attributes or none at all. Attributes describe the structure and value of an instance of a class.

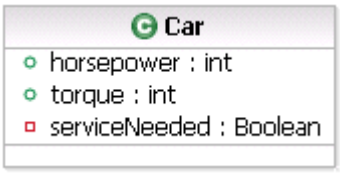
For example, a Client class might have a balance attribute that holds the amount of money in the client's account.

In domain modeling class diagrams, Java™ fields map to attributes.

Attributes are shown in the attribute compartment of a classifier in a class diagram. Attributes that are defined in the scope of the class, that is static, are shown as underlined. The visibility styles of attributes can be represented as text symbols (such as "-") or icons (such as).

The following figure illustrates how attributes are represented with visibility icons in UML class diagrams.

Java source code	UML visual representation

`public class Car {`
 	
`public int horsepower;`
`public int torque;`
`private Boolean serviceNeeded;`
 	
`}`
	

Related concepts:

Operations in domain modeling class diagrams

Operations in domain modeling class diagrams

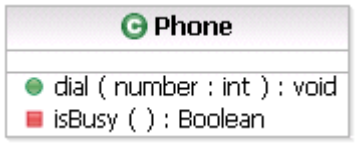
In domain modeling class diagrams, an operation requests a service that a classifier or an instance of a class is called to perform. Operations are contained by classes and interfaces. A classifier can have any number of operations or none at all. Operations are implementations of functions or queries that an object might be called to perform. A well-defined operation does only one thing.

For example, you can make a Cart class responsible for adding and removing merchandise that a client plans to buy. You can then add an addItem( ) operation that adds merchandise to the cart and a removeItem( ) operation that removes merchandise.

Operations are shown in the operation compartment of a classifier in a class diagram. Operations that are defined in the scope of the class, that is static, are shown as underlined. The visibility styles of operations can be represented as text symbols (such as "+") or icons (such as).

The following figure illustrates how operations are represented with visibility icons in class diagrams.

Java™ source code	UML visual representation

public class Phone {
public void dial(int number) {
}
private Boolean isBusy( ) {
//determine if the phone line busy
return (Boolean) false;
}
}
	

Related concepts:

Attributes in domain modeling class diagrams

Related tasks:

Showing operation signatures of classifiers in domain modeling class diagrams

Visibility in domain modeling class diagrams

In domain modeling class diagrams, visibility defines whether attributes and operations of specific classes can be seen and used by other classes.

For example, the attributes and operations in a class with public visibility can be seen and used by other classes, while the attributes and operations with private visibility can be seen and used only by the class that contains them.

You can use decoration icons or text symbols to show the level of visibility for attributes and operations. A text symbol appended to the name of an association end shows the visibility of that association end.

The following table illustrates how different levels of visibilities are represented for attributes and operations with visibility icons or text symbols in class diagrams.

Visibility level	Icon for attribute	Icon for operation	Text symbol	Description
Private			-	Only classes in the same container can see and use the classes.
Protected			#	Only classes in the same container or a descendent of the container can see and use the classes.
Public			+	Any class that can see the container can also see and use the classes.
Package			~	Only classes within the same package as the container can see and use the classes.

Related tasks:

Specifying the visibility style for attributes and operations in domain modeling class diagrams

Showing and hiding attributes and operations in domain modeling class diagrams

You can show or hide attributes and operations of individual classifiers so that only the elements that are important to you are displayed in a domain modeling class diagram for better visual representation of applications

Before you begin

You must have a class diagram with classifiers open.

Procedure

To show or hide attributes or operations, in the diagram editor, right-click a classifier or a group of classifiers; then click **Filters** > **Show/Hide Compartment** and click a type of compartment to show or hide.**Tip:** You can also filter out selected attributes or operations individually from their respective compartments by clicking **Filters** > **Sort/Filter Compartment Items**.

Related tasks:

Specifying the visibility style for attributes and operations in domain modeling class diagrams
Showing operation signatures of classifiers in domain modeling class diagrams
Showing parent names of classifiers in domain modeling class diagrams
Showing fully qualified names of classifiers in domain modeling class diagrams

Specifying the visibility style for attributes and operations in domain modeling class diagrams

You can specify the visibility style for attributes and operations in domain modeling class diagrams to meet your needs for visual representation of applications.

Before you begin

You must have a class diagram open.

Procedure

1. In the diagram editor, select a classifier or selected classifiers.
2. Click **Diagram** > **Filters** > **Stereotype and Visibility Style** and complete one of the following steps:
   - To display visibilities with decoration icons.**Visibility Style: Decoration**.
   - To display visibilities with text symbols.**Visibility Style: Text**.
   - Not to display any visibility style, click **Visibility Style: None**.

Results

The visibility style has changed as specified. **Note:** You can also right-click a classifier or selected classifier; then click **Filters** > **Stereotype and Visibility Style** to specify the visibility style for attributes and operations.

Related concepts:

Visibility in domain modeling class diagrams
Attributes in domain modeling class diagrams
Operations in domain modeling class diagrams

Related tasks:

Showing and hiding attributes and operations in domain modeling class diagrams
Showing operation signatures of classifiers in domain modeling class diagrams
Showing parent names of classifiers in domain modeling class diagrams
Showing fully qualified names of classifiers in domain modeling class diagrams

Showing operation signatures of classifiers in domain modeling class diagrams

You can show the operations of classifiers with the full signature of parameters along with the full package names in domain modeling class diagrams to visually represent structures and relationships in applications.

Before you begin

You must have a diagram open and the classifiers' operation compartments must be visible. Ensure that you have not set preferences to filter or hide operations.**Note:** If you do not show operation signatures, the parameters and return types do not appear in the diagram.

Procedure

To show the operation signature of a classifier, in the diagram editor, right-click a classifier; then click **Filters** > **Show Signature**.**Note:** If the **Show Parent Name** option is also enabled, the parameter types are displayed with the full package name.

Related concepts:

Operations in domain modeling class diagrams

Related tasks:

Showing and hiding attributes and operations in domain modeling class diagrams
Specifying the visibility style for attributes and operations in domain modeling class diagrams
Showing parent names of classifiers in domain modeling class diagrams
Showing fully qualified names of classifiers in domain modeling class diagrams

Showing parent names of classifiers in domain modeling class diagrams

You can show the names or qualified names of parents of individual classifiers in domain modeling class diagrams to visually represent relationships between source elements in applications.

Before you begin

You must have a class diagram open.

Procedure

1. In the diagram editor, right-click a classifier; then click **Filters** > **Parent Style**.
2. Select one of the following options:
   - Not to show the parent name, click **None**.
   - To show the parent name, click **Name**.
   - To show the qualified parent name, click **Qualified Name**

Related concepts:

Classifiers in domain modeling class diagrams

Related tasks:

Showing and hiding attributes and operations in domain modeling class diagrams
Specifying the visibility style for attributes and operations in domain modeling class diagrams
Showing operation signatures of classifiers in domain modeling class diagrams
Showing fully qualified names of classifiers in domain modeling class diagrams

Showing fully qualified names of classifiers in domain modeling class diagrams

In domain modeling class diagrams, you can show the fully qualified names of individual classifiers for type proposals in other packages to visually represent relationships between source elements in applications.

Before you begin

You must have a class diagram open.

Procedure

To show the fully qualified name of a classifier, in the diagram editor, right-click a classifier; then click **Filters** > **Show Qualified Name**.

Related concepts:

Classifiers in domain modeling class diagrams

Related tasks:

Showing and hiding attributes and operations in domain modeling class diagrams
Specifying the visibility style for attributes and operations in domain modeling class diagrams
Showing operation signatures of classifiers in domain modeling class diagrams
Showing parent names of classifiers in domain modeling class diagrams

Deleting domain modeling elements from class diagrams and projects

You can delete domain modeling elements that you no longer require either from a class diagram and the project or from only the diagram.

Before you begin

You must have a class diagram open.

About this task

If you delete domain modeling elements from a class diagram, you remove the elements only from the visual representation in the diagram; however, the elements remain in the project and are still displayed in the Project Explorer or Project Explorer view. If you delete elements from a project, you permanently remove the domain modeling elements from both the diagram and the project.To delete a domain modeling element from a class diagram or project:

Procedure

1. In the diagram editor, right-click a diagram element.
2. From the pop-up menu, complete one of the following steps:
   - To delete the element from only the diagram, click **Delete from Diagram**.
   - To delete the element from both the diagram and project, click **Delete from Project**.

Related tasks:

Creating domain modeling class diagrams for visual development of structural features in applications

Managing visual representation of source elements in domain modeling class diagrams

Managing visual representation of classifiers in domain modeling class diagrams

UML modeling best practices

Unified Modeling Language (UML) is the industry-standard notation for software architecture.

Three primary websites provide information on modeling techniques, best practices, and UML standards:

- The IBM® developerWorks®: Rational®
- The IBM UML Resource Center
- Object Management Group

These sites provide reliable, up-to-date information on UML and related topics, as well as links to other resources.

IBM developerWorks: Rational

The IBM developerWorks: Rational provides guidance and information that can help you implement and deepen your knowledge of Rational tools and best practices. This network includes access to white papers, artifacts, source code, discussions, training, and other documentation.

IBM UML Resource Center

At the IBM UML Resource Center, you can access a library of UML information and resources that IBM continues to build upon and update. In addition to current news and updates about UML, this website provides the following resources:

Resource	Description
Documentation	A series of documents that define UML.
Quick Reference	A quick reference to UML notation.
White Papers	Technical papers on UML.
Recommended Reading	A list and description of books on visual modeling for basic, intermediate, and advanced users.
Rational Unified Process	An overview of a full life cycle software engineering process that is designed to increase the quality of development while reducing time-to-market. It draws on UML as the basis for an iterative approach to component-based software development.

Object Management Group

The Object Management Group (OMG) website provides the following resources:

Resource	Description
Formal Specifications	Specifications on UML that have been adopted by the OMG and are available in either published or downloadable form.
Technical Submissions	Submissions on UML that have not been adopted.

Collected links

IBM developerWorks: Rational

IBM UML Resource Center

Object Management Group (OMG) website

Visually developing Java elements by using domain modeling class diagrams

You can use domain modeling class diagrams to visually develop and edit Java™ packages, classes, interfaces, and enum types in your applications.

- **Creating and populating domain modeling class diagrams with Java source elements**
  You can create domain modeling class diagrams with Java source elements to represent Java source elements and populate existing class diagrams with Java source elements to visually develop Java applications.
- **Creating Java elements in domain modeling class diagrams**
  You can use domain modeling class diagrams to create packages, classes, interfaces, and enum types to visually develop Java application code.
- **Editing Java classifiers in domain modeling class diagrams**
  You can use the pop-up code editor from within domain modeling class diagrams to view or directly edit Java classifiers, such as classes and interfaces.
- **Deleting Java elements from domain modeling class diagrams and projects**
  You can delete Java elements from a domain modeling class diagram to remove them from visual representation in the diagram or delete them from both the diagram and project permanently.

Related tasks:

Visually developing relationships between Java elements by using domain modeling class diagrams

Creating and populating domain modeling class diagrams with Java source elements

You can create domain modeling class diagrams with Java™ source elements to represent Java source elements and populate existing class diagrams with Java source elements to visually develop Java applications.

- **Creating class diagrams of Java source elements**

  You can create class diagrams of existing Java source elements, including projects, packages, classes, and interfaces, to visually represent and develop structures and relationships in applications.

- **Populating domain modeling class diagrams with Java source elements**

  You can populate existing domain modeling class diagrams with Java source elements, including projects, packages, classes, and interfaces, to visually represent and develop structures and relationships in applications.

Related tasks:

Creating Java elements in domain modeling class diagrams

Editing Java classifiers in domain modeling class diagrams

Deleting Java elements from domain modeling class diagrams and projects

Creating class diagrams of Java source elements

You can create class diagrams of existing Java™ source elements, including projects, packages, classes, and interfaces, to visually represent and develop structures and relationships in applications.

Before you begin

You must be in the Package Explorer view or have a class diagram open.

Procedure

1. In the Package Explorer view or in the diagram editor, right-click a source element or selected source elements; then click **Visualize** and complete one of the following steps:
   - To add the selected elements to the new diagram, click **Add to New Diagram File** > **Class Diagram**.
   - To add the selected Java project or packages and their contents to the new diagram, click **Add Elements and Contents to New Diagram** > **Class Diagram**.
2. In the New Class Diagram wizard, in the **Enter or select the parent folder** field, specify the parent folder.
3. In the **File name** field, type a file name.
4. Optional: To enable the associated capabilities, click **Next** and complete the following steps:
   A. Select the **Customize UI visibility for this diagram by selecting capabilities below** check box.
   B. From the **Capabilities** list, expand a category and select the capabilities.
5. Click **Finish**.

Results

A class diagram is created and opens in the diagram editor with the selected source elements and contents in it.**Tip:** If any selected source elements are Java 2 Platform, Standard Edition (J2SE) 5.0 classes with annotations, the details of the annotations are shown in the annotation compartments of the classes in the diagram. You can also add or edit annotations in the Properties view on the Annotation page .

Related tasks:

Populating domain modeling class diagrams with Java source elements

Populating domain modeling class diagrams with Java source elements

You can populate existing domain modeling class diagrams with Java™ source elements, including projects, packages, classes, and interfaces, to visually represent and develop structures and relationships in applications.

Before you begin

You must be in the Package Explorer view and have a domain modeling class diagram open.

Procedure

To populate a class diagram with Java source elements, in the Package Explorer view or in the diagram editor, right-click a source element or selected source elements; then click **Visualize** and complete one of the following steps:

- In the Package Explorer view, to add only the selected elements to the diagram, click **Add to Current Diagram**.
- In the Package Explorer view, to add the selected elements and their contents to the diagram, click **Add Elements and Contents to Current Diagram**.
- In the diagram editor, to add only the contained contents of the selected elements to the diagram, click **Add Only Contained Elements to Current Diagram**.

Results

The class diagram is populated with the selected elements and contents.**Tip:** If any selected source elements are Java 2 Platform, Standard Edition (J2SE) 5.0 classes with annotations, the details of the annotations are shown in the annotation compartments of the classes in the diagram. You can also add or edit annotations in the Properties view on the Annotation page.

Related tasks:

Creating class diagrams of Java source elements

Creating Java elements in domain modeling class diagrams

You can use domain modeling class diagrams to create packages, classes, interfaces, and enum types to visually develop Java™ application code.

- **Customizing the default settings for Java field and method creation in domain modeling class diagrams**
  You can set the default for using the Create Java Filed wizard or the Create Java Method wizard when you add new Java fields (attributes) or methods (operations) to classifiers in domain modeling class diagrams.
- **Creating Java packages in domain modeling class diagrams**
  You can use domain modeling class diagrams to create Java packages to visually develop Java applications.
- **Adding classes and interfaces to Java packages in domain modeling class diagrams**
  You can use domain modeling class diagrams to add classes and interfaces to Java packages.
- **Creating top-level Java classes in domain modeling class diagrams**
  You can use domain modeling class diagrams to create Java classes that are not enclosed in other types (top-level classes) to visually develop Java applications.
- **Creating nested Java classes in domain modeling class diagrams**
  You can domain modeling class diagrams to create Java classes that are enclosed in other types (nested classes) to visually develop Java applications.
- **Creating top-level Java interfaces in domain modeling class diagrams**
  You can use domain modeling class diagrams to create Java interfaces that are not enclosed in other types (top-level interfaces) Java application code.
- **Creating nested interfaces in domain modeling class diagrams**
  You can use domain modeling class diagrams to create Java interfaces that are enclosed in other types (nested interfaces).
- **Creating Java enum types in domain modeling class diagrams**
  You can use domain modeling class diagrams to create Java enumeration types to visually develop applications.

Related tasks:

Creating and populating domain modeling class diagrams with Java source elements
Editing Java classifiers in domain modeling class diagrams
Deleting Java elements from domain modeling class diagrams and projects

Customizing the default settings for Java field and method creation in domain modeling class diagrams

You can set the default for using the Create Java Filed wizard or the Create Java Method wizard when you add new Java™ fields (attributes) or methods (operations) to classifiers in domain modeling class diagrams.

About this task

If you clear the **Fields** and **Methods** selections, the Create Java Filed wizard or the Create Java Method wizard does not appear when you add fields or methods to Java classifiers. The default settings apply to new fields or methods, not existing ones.

Procedure

1. Click **Window** > **Preferences**
2. In the Preferences window, expand **Modeling**, expand **Java**, and then click **Field and Method Creation**.
3. On the Field and Method Creation page, select **Fields** and **Methods**.
4. Click **OK**.

Related tasks:

Creating Java packages in domain modeling class diagrams
Adding classes and interfaces to Java packages in domain modeling class diagrams
Creating top-level Java classes in domain modeling class diagrams
Creating nested Java classes in domain modeling class diagrams
Creating top-level Java interfaces in domain modeling class diagrams
Creating nested interfaces in domain modeling class diagrams
Creating Java enum types in domain modeling class diagrams

Creating Java packages in domain modeling class diagrams

You can use domain modeling class diagrams to create Java™ packages to visually develop Java applications.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the Palette, click **Package** and click an empty space inside the class diagram.
2. In the New Java Package wizard, in the **Source folder** field, specify the source folder for the new package.
3. In the **Name** field, type a name for the new Java package.
4. Click **Finish**.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams
Adding classes and interfaces to Java packages in domain modeling class diagrams
Creating top-level Java classes in domain modeling class diagrams
Creating nested Java classes in domain modeling class diagrams
Creating top-level Java interfaces in domain modeling class diagrams
Creating nested interfaces in domain modeling class diagrams
Creating Java enum types in domain modeling class diagrams

Adding classes and interfaces to Java packages in domain modeling class diagrams

You can use domain modeling class diagrams to add classes and interfaces to Java™ packages.

Before you begin

In the Java Perspective, open a domain modeling class diagram that contains Java packages.

About this task

You can also use the package action bars or the Palette to add classes and interfaces to Java packages.

Procedure

1. In the diagram editor, right-click a Java package; then click **Add Java** > **Class** or **Add Java** > **Interface**.
2. Follow the instructions in the New Java Class or New Java Interface wizard.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams
Creating Java packages in domain modeling class diagrams
Creating top-level Java classes in domain modeling class diagrams
Creating nested Java classes in domain modeling class diagrams
Creating top-level Java interfaces in domain modeling class diagrams
Creating nested interfaces in domain modeling class diagrams
Creating Java enum types in domain modeling class diagrams

Creating top-level Java classes in domain modeling class diagrams

You can use domain modeling class diagrams to create Java™ classes that are not enclosed in other types (top-level classes) to visually develop Java applications.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the Palette, click **Class** and click an empty space inside the class diagram.
2. In the New Java Class wizard, in the **Source Folder** field, specify the source folder for the new class.
3. In the **Package** field, specify the package to contain the new class. If you want the new class to be created in the default package, leave this field empty.
4. Clear the **Enclosing type** check box.
5. In the **Name** field, type a name for the new class.
6. Under **Modifiers**, select access modifiers for the new class.
7. In the **Superclass** field, specify the superclass for the new class.
8. Click **Add**.
9. In the Implemented Interfaces Selection window, select the interfaces for the new class to implement and click **OK**.
10. Select method stubs to create in the new class.
11. Click **Finish**.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams

Creating Java packages in domain modeling class diagrams

Adding classes and interfaces to Java packages in domain modeling class diagrams

Creating nested Java classes in domain modeling class diagrams

Creating top-level Java interfaces in domain modeling class diagrams

Creating nested interfaces in domain modeling class diagrams

Creating Java enum types in domain modeling class diagrams

Creating nested Java classes in domain modeling class diagrams

You can domain modeling class diagrams to create Java™ classes that are enclosed in other types (nested classes) to visually develop Java applications.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the Palette, click **Class** and click an empty space inside the class diagram.
2. In the New Java Class wizard, in the **Source Folder** field, specify the source folder for the new class.
3. In the **Package** field, specify the package to contain the new class or leave this field empty if you want the new class to be created in the default package.
4. Select the **Enclosing type** check box and type a name of the enclosing type in the field.
5. In the **Name** field, type a name for the new class.
6. Under **Modifiers**, select access modifiers for the new class.
7. In the **Superclass** field, specify the superclass for the new class.
8. Click **Add**; then in the Implemented Interfaces Selection window, select the interfaces for the new class to implement and click **OK**.
9. Select method stubs to create in the new class.
10. Click **Finish**.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams

Creating Java packages in domain modeling class diagrams

Adding classes and interfaces to Java packages in domain modeling class diagrams

Creating top-level Java classes in domain modeling class diagrams

Creating top-level Java interfaces in domain modeling class diagrams

Creating nested interfaces in domain modeling class diagrams

Creating Java enum types in domain modeling class diagrams

Creating top-level Java interfaces in domain modeling class diagrams

You can use domain modeling class diagrams to create Java™ interfaces that are not enclosed in other types (top-level interfaces) Java application code.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the Palette, click **Interface** and click an empty space inside the class diagram.
2. In the New Java Interface wizard, in the **Source Folder** field, specify the source folder for the new interface.
3. In the **Package** field, specify the package to contain the new interface or leave this filed empty if your want the new interface to be created in the default package.
4. Clear the **Enclosing type** check box.
5. In the **Name** field, type a name for the new interface.
6. Under **Modifiers**, select either the **public** or **default** access modifier.
7. Click **Add** beside the **Extended Interfaces** field.
8. In the Extended Interfaces Selection window, select the extended interfaces and click **Add**, and **OK**.
9. Click **Finish**.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams
Creating Java packages in domain modeling class diagrams
Adding classes and interfaces to Java packages in domain modeling class diagrams
Creating top-level Java classes in domain modeling class diagrams
Creating nested Java classes in domain modeling class diagrams
Creating nested interfaces in domain modeling class diagrams
Creating Java enum types in domain modeling class diagrams

Creating nested interfaces in domain modeling class diagrams

You can use domain modeling class diagrams to create Java™ interfaces that are enclosed in other types (nested interfaces).

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. On the diagram editor tool palette, click **Interface** and click an empty space inside the class diagram.
2. In the New Java Interface wizard, in the **Source Folder** box, select the source folder in which you want the new interface to reside.
3. Select the **Enclosing type** option and click **Browse**.
4. In the Enclosing Type Selection window, select the type in which to enclose the new interface, and then click **Add** and **OK**.
5. In the **Name** field, type the name for the new interface.
6. Select an access modifier (public, default, private, or protected).
7. Click **Add** next to the **Extended Interfaces** box and, in the Extended Interfaces Selection window, select the extended interfaces; then click **Add** and **OK**.
8. Click **Finish**.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams
Creating Java packages in domain modeling class diagrams
Adding classes and interfaces to Java packages in domain modeling class diagrams
Creating top-level Java classes in domain modeling class diagrams
Creating nested Java classes in domain modeling class diagrams
Creating top-level Java interfaces in domain modeling class diagrams
Creating Java enum types in domain modeling class diagrams

Creating Java enum types in domain modeling class diagrams

You can use domain modeling class diagrams to create Java™ enumeration types to visually develop applications.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the Palette, click **Enum** and click an empty space in the diagram editor where you want to place the enum type.
2. In the New Enum Type wizard, in the **Source Folder** field, specify the source folder for the new enum type.
3. Complete one of the following steps:
   - In the **Package** field, specify the package to contain the new enum type.
   - Select **Enclosing type** and specify a type in which to enclose the new enum type.
4. In the **Name** field, type a name for the new enum type.
5. In the **Modifiers** field, select one or more access modifiers for the new enum type. Private and protected access modifiers are available only if you specify an enclosing type.
6. In the **Interfaces** area, click **Add**; then, in the Implemented Interfaces Selection window, select the interfaces that the new enum type implements and click **Add**.
7. Click **Finish**.

Related tasks:

Customizing the default settings for Java field and method creation in domain modeling class diagrams
Creating Java packages in domain modeling class diagrams
Adding classes and interfaces to Java packages in domain modeling class diagrams
Creating top-level Java classes in domain modeling class diagrams
Creating nested Java classes in domain modeling class diagrams
Creating top-level Java interfaces in domain modeling class diagrams
Creating nested interfaces in domain modeling class diagrams

Editing Java classifiers in domain modeling class diagrams

You can use the pop-up code editor from within domain modeling class diagrams to view or directly edit Java™ classifiers, such as classes and interfaces.

About this task

You can also use domain modeling class diagrams to modify Java classifiers and to add fields or methods to classes or interfaces. When you add fields or methods to classes or interfaces, qualified names will be added for type proposals in packages if the Java Editor Code Assistant's default **Add import instead of qualified name** preference option is cleared. Otherwise, import declarations will be added automatically.

- **Setting preferences for showing the pop-up code editor in domain modeling class diagrams**
  You can customize the default settings to enable the pop-up code editor and modify code directly in domain modeling class diagrams.
- **Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams**
  You can customize the default settings to enable the visual and direct editing of Java classifiers in domain modeling class diagrams.
- **Customizing default setting for Java reference resolution in Java domain modeling**
  You can customize the default setting to resolve Java element references based on the build path of the container Java project.
- **Customizing the default settings for Java fields in domain modeling class diagrams**
  You can set the default preferences for new fields (that is, attributes in classifiers) that you add to Java classifiers in domain modeling class diagrams.
- **Customizing the default settings for Java methods in domain modeling class diagrams**
  You can set the default preferences for new methods that you add to Java classes and interfaces in domain modeling class diagrams.
- **Setting the default display style for Java methods in domain modeling class diagrams**
  You can set the Java or UML notation style as the default display style for methods of Java classifiers in domain modeling class diagrams.
- **Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams**
  You can customize the default settings for showing or hiding annotation compartments of classifiers in domain modeling class diagrams.
- **Editing Java classes and interfaces in domain modeling class diagrams**
  You can Edit names, fields (attributes), and methods (operations) of Java classes and interfaces from domain modeling class diagrams.
- **Adding fields to Java classes and interfaces in domain modeling class diagrams**
  You can use domain modeling class diagrams to add fields to Java classes and interfaces.
- **Adding methods to Java classes and interfaces in domain modeling class diagrams**
  You can use domain modeling class diagrams to add a method to a Java class or interface.

Related tasks:

Creating and populating domain modeling class diagrams with Java source elements
Creating Java elements in domain modeling class diagrams
Deleting Java elements from domain modeling class diagrams and projects

Setting preferences for showing the pop-up code editor in domain modeling class diagrams

You can customize the default settings to enable the pop-up code editor and modify code directly in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** and click **Java**.
3. On the Java page, under **Code view**, select **Show pop-up code editor**.
4. Click **OK**.

Related tasks:

Automatically navigating to the source code of Java diagram elements
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams

You can customize the default settings to enable the visual and direct editing of Java™ classifiers in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** and click **Java**.
3. On the Java page, under **Direct editing**, select **Refactor Java fields and methods when renaming using direct editing**.
4. Click **OK**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Customizing default setting for Java reference resolution in Java domain modeling

You can customize the default setting to resolve Java™ element references based on the build path of the container Java project.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** and click **Java**.
3. On the Java page, under **Java reference resolution**, select the **Resolve Java element references based on the build path of the container Java project** option.
4. Click **OK**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Customizing the default settings for Java fields in domain modeling class diagrams

You can set the default preferences for new fields (that is, attributes in classifiers) that you add to Java™ classifiers in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling**, expand **Java**, expand **Field and Method Creation**, and click **Field Defaults**.
3. On the Field Defaults page, under **Field name**, type a name, name prefix, and name suffix for new fields.
4. Under **Type and dimensions**, select the type and specify the dimensions for new fields.
5. Under **Visibility**, click a visibility style for new fields.
6. Under **Modifiers**, click modifiers for new fields.
7. In the **Initial value** field, type an initial value expression for new fields.
8. Click **OK**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Customizing the default settings for Java methods in domain modeling class diagrams

You can set the default preferences for new methods that you add to Java™ classes and interfaces in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling**, expand **Java**, expand **Field and Method Creation**, and click **Method Defaults**.
3. On the Method Defaults page, under **Method name**, type a name, name prefix, and name suffix.
4. Under **Return type and Dimensions**, click a return type and specify the dimensions.
5. Under **Visibility**, click a method visibility style.
6. Under **Modifiers**, click method modifiers.
7. Click **Browse** beside the **Throws** field and, in the Browse types window, click an exception type or types, and then click **OK**.
8. Click **OK**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Setting the default display style for Java methods in domain modeling class diagrams

You can set the Java™ or UML notation style as the default display style for methods of Java classifiers in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** and click **Java**.
3. On the Java page, under **Java operation display style**, select the **Java** or **UML notation** style.
4. Click **OK**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams

You can customize the default settings for showing or hiding annotation compartments of classifiers in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling**, expand **Java**, and click **Appearance**.
3. On the Appearance page, under **Compartments**, select the **Show annotation compartments** check box.
4. Click **OK**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Editing Java classes and interfaces in domain modeling class diagrams

You can Edit names, fields (attributes), and methods (operations) of Java™ classes and interfaces from domain modeling class diagrams.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the diagram editor, click the name, an attribute, or an operation of a Java class or interface and press F2.
2. Type your modifications and press Enter.

Results

The changes are saved to the source code file of the Java class or interface.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Adding fields to Java classes and interfaces in domain modeling class diagrams

You can use domain modeling class diagrams to add fields to Java™ classes and interfaces.

Before you begin

You must have a domain modeling class diagram with Java classes or interfaces open.

About this task

When you add fields to classes or interfaces, fully qualified names will be added for type proposals in packages if the Java Editor Code Assist's default **Add import instead of qualified name** preference option is cleared. Otherwise, import declarations will be added automatically. You can also use the action bar to add fields to Java classes and interfaces.

Procedure

1. In the diagram editor, right-click a Java class or interface; then click **Add Java** > **Field**.
2. In the Create Java Field wizard, in the **Name** field, type a name for the new field.
3. In the **Initial Value** field, specify the initial value expression.
4. Under **Type and Dimensions**, click a type and specify dimensions.
5. Under **Visibility**, click a visibility style.
6. Under **Modifiers**, click modifiers.
7. Click **Finish**.

Related tasks:

Creating association relationships between Java classifiers in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Showing Java attributes as associations in domain modeling class diagrams
Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding methods to Java classes and interfaces in domain modeling class diagrams

Adding methods to Java classes and interfaces in domain modeling class diagrams

You can use domain modeling class diagrams to add a method to a Java™ class or interface.

Before you begin

You must have a domain modeling class diagram with Java classes or interfaces open.

About this task

When you add methods to classes or interfaces, fully qualified names will be added for type proposals in packages if the Java Editor Code Assist's default **Add import instead of qualified name** preference option is cleared. Otherwise, import declarations will be added automatically. You can also use the action bar or the Methods tab in the Properties view to add methods to Java classes and interfaces.

Procedure

1. In the diagram editor, right-click a Java class or interface; then click **Add Java** > **Method**.
2. In the Create Java Method wizard, in the **Name** field, type a name for the new method.
3. To make the method a constructor, select the **Constructor** check box.
4. To throw exceptions, in the **Throws** field, enter an exception.
5. Under **Type** and **Dimensions**, click a method return type and specify dimensions.
6. Under **Visibility**, click a visibility style.
7. Under **Modifiers**, click modifiers.
8. Under **Parameters**, click **Add**; then in the Create Parameter window, type a name, select a parameter type, specify dimensions, and click **OK**.
9. Click **Finish**.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Setting preferences for visual and direct editing of Java classifiers in domain modeling class diagrams
Customizing default setting for Java reference resolution in Java domain modeling
Customizing the default settings for Java fields in domain modeling class diagrams
Customizing the default settings for Java methods in domain modeling class diagrams
Setting the default display style for Java methods in domain modeling class diagrams
Setting preferences for showing or hiding annotation compartments in domain modeling class diagrams
Editing Java classes and interfaces in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams

Deleting Java elements from domain modeling class diagrams and projects

You can delete Java™ elements from a domain modeling class diagram to remove them from visual representation in the diagram or delete them from both the diagram and project permanently.

Before you begin

You must have a class diagram open.

About this task

When you delete Java elements from class diagrams, the underlying semantics remain intact. When you delete them from a project, the Java elements are deleted from both the diagram and the project permanently.

Procedure

To delete Java elements from a project, in the diagram editor, right-click the selected Java elements; then complete one of the following steps:

- To delete them from the diagram, click **Delete from Diagram**.
- To delete them from both the diagram and project permanently, click **Delete from Project**.

Related tasks:

Creating and populating domain modeling class diagrams with Java source elements

Creating Java elements in domain modeling class diagrams

Editing Java classifiers in domain modeling class diagrams

Visually developing relationships between Java elements by using domain modeling class diagrams

You can use domain modeling class diagrams to develop relationships between Java™ elements to visually represent and develop your Java applications.

- **Visually creating relationships between Java elements by using domain modeling class diagrams**

  You can use domain modeling class diagrams to create extends and implements relationships between Java elements. You can also create outgoing (from source to target) and incoming (from target to source) relationships between Java elements by using the relationship handles.

- **Managing relationships of classifiers in domain modeling class diagrams**

  You can show or hide the existing relationships between Java classifiers in domain modeling class diagrams for visual representation and development of applications.

Related tasks:

Visually developing Java elements by using domain modeling class diagrams

Relationships in domain modeling class diagrams

In domain modeling class diagrams, a relationship is the connection between Java™ classes or interfaces or between Enterprise JavaBeans (EJB) enterprise beans.

You can use several relationships to define the structure between Java classes or interfaces or between enterprise beans:

- Association relationships imply that instances of one class connect to instances of another class.
- Dependency relationships imply that a change to one class might affect another class.
- Extends relationships imply that one class is a specialization of another class.
- Implements relationships imply that one class provides a specification and the other class implements the specification.

You can also use note attachment relationships to provide additional information about diagram elements.

Related tasks:

Managing relationships of classifiers in domain modeling class diagrams

Visually creating relationships between Java elements by using domain modeling class diagrams

Association relationships in domain modeling class diagrams

In domain modeling class diagrams, an association is a structural relationship that indicates that objects of one classifier (such as a class and interface) are connected and can navigate to objects of another classifier.

An association connects two classifiers: the supplier classifier and the client classifier. Associations can help you make design decisions about the structure of your data. You can make decisions about not only the classes that are needed to contain the data, but also about which classes are needed to share the data with other classes. An association supports data sharing between classes or, in the case of a self-association, between objects of the same class.

For example, a Client class may have a single association (1) to an Account class, which indicates that each Account instance is owned by one Client instance. If you have an Account, you can locate the owning Client of that account. And with a given Client, you can find the Account of that client. The association between the Client class and the Account class is important because it shows the structure between the two classifiers.

Multiplicity information can be linked to an association to show how many instances of class A are linked with instances of class B. Multiplicity information can be linked to both ends of association relationships.

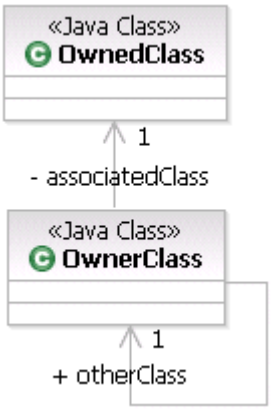
In domain modeling class diagrams, association relationships in a Java™ application represent the following things:

- A semantic relationship between two or more classes that specifies connections between their instances
- A structural relationship that specifies that objects of one class are connected to objects of a second (possibly the same) class

In visual representation, instance variables in a Java application become attributes in classifiers in domain modeling class diagrams. By default, all Java and container-managed persistence (CMP) entity bean fields are shown as attributes.

As the following figure illustrates, an association relationship is represented as a solid line between two classifiers.

Java source code	UML visual representation

OwnedClass.Java:
`public class OwnedClass {`
`// <<class body>>`
`}`
 	
OwnerClass.Java:
`public class OwnerClass {`
`private OwnedClass associatedClass;`
`public OwnerClass otherClass;`
`// <<class body>>`
`}`
	

Related concepts:

Extends relationships in domain modeling class diagrams

Implements relationships in domain modeling class diagrams

Related tasks:

Showing related Java elements in domain modeling class diagrams based on outgoing relationships

Showing related Java elements in domain modeling class diagrams based on incoming relationships

Populating domain modeling class diagrams with Java source elements based on type

Showing Java associations as attributes in domain modeling class diagrams

Showing Java attributes as associations in domain modeling class diagrams

Setting preferences for showing or hiding Java association type labels in domain modeling class

diagrams

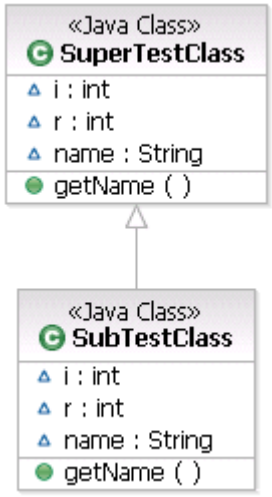
Creating association relationships between Java classifiers in domain modeling class diagrams

Extends relationships in domain modeling class diagrams

In domain modeling class diagrams, an extends relationship (also called an inheritance or an is-a relationship) implies that a specialized (child) class is based on a general (parent) class.

In domain modeling class diagrams, extends relationships apply only to container-managed persistence (CMP) entity beans. They do not apply to session or message-driven beans.

As the following figures illustrates, an extends relationship is displayed as a solid line with an unfilled arrowhead that points from the specialized (child) Java™ class or Enterprise JavaBeans (EJB) enterprise bean to the general (parent) Java class or EJB enterprise bean. You can also visually represent and develop extends (inheritance) relationships between CMP entity beans.

Java source code	UML visual representation
	

Related concepts:

Association relationships in domain modeling class diagrams

Implements relationships in domain modeling class diagrams

Related tasks:

Showing related Java elements in domain modeling class diagrams based on outgoing relationships

Showing related Java elements in domain modeling class diagrams based on incoming relationships

Populating domain modeling class diagrams with Java source elements based on type

Creating extends relationships between Java classes in domain modeling class diagrams

Implements relationships in domain modeling class diagrams

In domain modeling class diagrams, an implements relationship exists between two classes when one of them must implement, or realize, the behavior specified by the other.

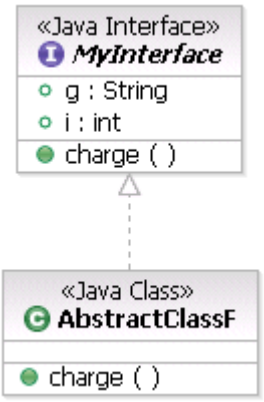
The class that specifies the behavior is called the supplier, and the class that implements the behavior is called the client.

An implements relationship can include those between interfaces and classes.

For example, an implements relationship connects an interface to a subsystem. The interface specifies the behaviors, and the subsystem implements the behaviors.

In domain modeling class diagrams, an implements relationship represents a class that implements the operations in a Java™ interface.

As the following figure illustrates, an implements relationship is displayed as a dashed line with an unfilled arrowhead. The connector points from the client (that realizes the behavior) to the supplier (that specifies the behavior).

Java source code	UML visual representation
	

Related concepts:

Association relationships in domain modeling class diagrams

Extends relationships in domain modeling class diagrams

Related tasks:

Showing related Java elements in domain modeling class diagrams based on outgoing relationships

Showing related Java elements in domain modeling class diagrams based on incoming relationships

Populating domain modeling class diagrams with Java source elements based on type

Creating implements relationships between Java classes and interfaces in domain modeling class diagrams

Owned element association relationships in domain modeling class diagrams

In domain modeling class diagrams, an owned element association relationship is a type of association that dictates ownership.

The owned element association represents classes that are owned by a package. This relationship is used to represent nested classes in a domain modeling class diagram where a class is declared in the scope of another (outer) class. A nested class belongs to the namespace of the outer class and can only be used in the outer class. That is, the parent class declares the inner (or nested) class. This construct is primarily used for implementation reasons and for hiding information.

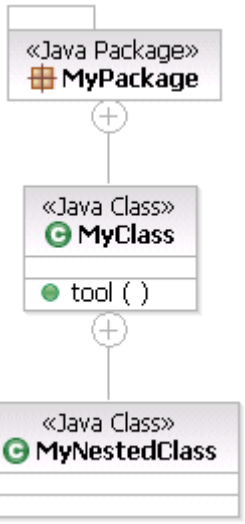
A declaring class and a class in its namespace are connected by an anchor line (referred to as owned association in the UML), with an anchor icon on the end connected to a declaring class. An anchor icon appears as a plus sign inside a circle.

The contents of the package are declared in the class and belong to its namespace.

For example, if Class B is attached to Class A by an anchor line with the anchor symbol on Class A, Class B is declared in the namespace of Class A. That is, the relationship between Class A and Class B is the namespace-owned element association.

The following figure illustrates how owned element associations are represented in domain modeling class diagrams.

Java™ source code	UML visual representation

`/* Package that owns MyClass and M`
 	
`package MyPackage;`
 	
`public class MyClass {`
`public void tool (Util util)`
`}`
 	
`/* Nested class defined insi`
 	
`public class MyNestedClass {`
`}`
 	
`}`
	

Related tasks:

Showing related Java elements in domain modeling class diagrams based on outgoing relationships

Showing related Java elements in domain modeling class diagrams based on incoming relationships

Populating domain modeling class diagrams with Java source elements based on type

Visually creating relationships between Java elements by using domain modeling class diagrams

You can use domain modeling class diagrams to create extends and implements relationships between Java™ elements. You can also create outgoing (from source to target) and incoming (from target to source) relationships between Java elements by using the relationship handles.

- **Creating extends relationships between Java classes in domain modeling class diagrams**
  You can use domain modeling class diagrams to create extends relationships between Java classes.
- **Creating implements relationships between Java classes and interfaces in domain modeling class diagrams**
  You can use domain modeling class diagrams to create implements relationships between Java classes and interfaces.
- **Creating association relationships between Java classifiers in domain modeling class diagrams**
  You can use domain modeling class diagrams to create association relationships between Java™ classifiers to visually represent Java fields, or attributes, as associations. That is, you create Java fields and place them outside of classifiers as associations in class diagrams.
- **Creating Java relationships from sources to targets in domain modeling class diagrams**
  You can use domain modeling class diagrams to create outgoing relationships from sources to targets between classifiers to visually develop Java applications.
- **Creating Java relationships from sources to unspecified targets in domain modeling class diagrams**
  You can use domain modeling class diagrams to create outgoing relationships from sources to unspecified targets between classifiers to visually develop Java applications.
- **Creating relationships from targets to sources in domain modeling class diagrams**
  You can use domain modeling class diagrams to create incoming relationships from targets to sources between classifiers to visually develop Java applications.
- **Creating relationships from targets to unspecified sources in domain modeling class diagrams**
  You can use domain modeling class diagrams to create incoming relationships from targets to unspecified sources between classifiers to visually develop Java applications.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Managing relationships of classifiers in domain modeling class diagrams

Creating extends relationships between Java classes in domain modeling class diagrams

You can use domain modeling class diagrams to create extends relationships between Java™ classes.

Before you begin

You cannot create extends relationships between Java classes in the following situations:

- Between two Java classes where the sub Java class already extends another class (multiple extends or inheritance relationships are not supported in Java)
- Between two Java classes where the super class is a final class
- When the source type is in a named project and the target type is in a default package.

You must have a domain modeling class diagram open.

About this task

You can also use the modeling assistant to create extends relationships.

Procedure

1. In the Palette, click **Extends**.
2. In the diagram editor, click the sub (client) Java class, drag it to the super (supplier) Java class, and release the mouse button.

Related concepts:

Extends relationships in domain modeling class diagrams

Related tasks:

Creating implements relationships between Java classes and interfaces in domain modeling class diagrams
Creating association relationships between Java classifiers in domain modeling class diagrams
Creating Java relationships from sources to targets in domain modeling class diagrams
Creating Java relationships from sources to unspecified targets in domain modeling class diagrams
Creating relationships from targets to sources in domain modeling class diagrams
Creating relationships from targets to unspecified sources in domain modeling class diagrams

Creating implements relationships between Java classes and interfaces in domain modeling class diagrams

You can use domain modeling class diagrams to create implements relationships between Java™ classes and interfaces.

Before you begin

You cannot create an implements relationship in the following situations:

- Between a class and an interface if such a relationship already exists between the two
- If the source class is in a named package and the target interface is in a default package
- If both the source class and the target interface are in default packages but in different projects or plug-ins

You must have a domain modeling class diagram with Java classes and interfaces open.

About this task

You can also use the modeling assistant to create implements relationships.

Procedure

1. In the Palette, click **Implements**.
2. In the diagram editor, click a Java class, drag it to a Java interface, and release the mouse button.

Related concepts:

Implements relationships in domain modeling class diagrams

Related tasks:

Creating extends relationships between Java classes in domain modeling class diagrams
Creating association relationships between Java classifiers in domain modeling class diagrams
Creating Java relationships from sources to targets in domain modeling class diagrams
Creating Java relationships from sources to unspecified targets in domain modeling class diagrams
Creating relationships from targets to sources in domain modeling class diagrams
Creating relationships from targets to unspecified sources in domain modeling class diagrams

Creating association relationships between Java classifiers in domain modeling class diagrams

You can use domain modeling class diagrams to create association relationships between Java™ classifiers to visually represent Java™ fields, or attributes, as associations. That is, you create Java fields and place them outside of classifiers as associations in class diagrams.

Before you begin

You must have a domain modeling class diagram with Java classifiers open.

Procedure

1. In the Palette, click **Association**.
2. In the diagram editor, click a Java classifier, drag it to another Java classifier, and release the mouse.
3. In the Create Association wizard, in the **Name** field, type a name for the new field.
4. In the **Initial Value** field, specify the initial value expression.
5. Under **Type and Dimensions**, specify dimensions.
6. Under **Visibility**, click a visibility style.
7. Under **Modifiers**, click modifiers.
8. Click **Finish**.

Results

A Java field is created and visually represented as an association relationship in the class diagram.

Related concepts:

Association relationships in domain modeling class diagrams

Related tasks:

Adding fields to Java classes and interfaces in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Showing Java attributes as associations in domain modeling class diagrams
Creating extends relationships between Java classes in domain modeling class diagrams
Creating implements relationships between Java classes and interfaces in domain modeling class diagrams
Creating Java relationships from sources to targets in domain modeling class diagrams
Creating Java relationships from sources to unspecified targets in domain modeling class diagrams
Creating relationships from targets to sources in domain modeling class diagrams
Creating relationships from targets to unspecified sources in domain modeling class diagrams

Creating Java relationships from sources to targets in domain modeling class diagrams

You can use domain modeling class diagrams to create outgoing relationships from sources to targets between classifiers to visually develop Java™ applications.

Procedure

1. In the diagram editor, place the cursor over the source classifier until the connector handles appear.
2. Click the outgoing connector handle, drag it to the target classifier, and release the mouse button.
3. In the pop-up window, click a relationship type.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Creating extends relationships between Java classes in domain modeling class diagrams
Creating implements relationships between Java classes and interfaces in domain modeling class diagrams
Creating association relationships between Java classifiers in domain modeling class diagrams
Creating Java relationships from sources to unspecified targets in domain modeling class diagrams
Creating relationships from targets to sources in domain modeling class diagrams
Creating relationships from targets to unspecified sources in domain modeling class diagrams

Creating Java relationships from sources to unspecified targets in domain modeling class diagrams

You can use domain modeling class diagrams to create outgoing relationships from sources to unspecified targets between classifiers to visually develop Java™ applications.

Procedure

1. In the diagram editor, place the cursor over the source classifier until the connector handles appear.
2. Click the outgoing connector handle, drag it to an empty space in the diagram editor, and release the mouse button.
3. In the pop-up window, click a relationship type and click a new Java element to create or click an existing one as the target.
4. Follow the instructions in the wizard.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Creating extends relationships between Java classes in domain modeling class diagrams
Creating implements relationships between Java classes and interfaces in domain modeling class diagrams
Creating association relationships between Java classifiers in domain modeling class diagrams
Creating Java relationships from sources to targets in domain modeling class diagrams
Creating relationships from targets to sources in domain modeling class diagrams
Creating relationships from targets to unspecified sources in domain modeling class diagrams

Creating relationships from targets to sources in domain modeling class diagrams

You can use domain modeling class diagrams to create incoming relationships from targets to sources between classifiers to visually develop Java™ applications.

Procedure

1. In the diagram editor, place the cursor over the target classifier until the connector handles appear.
2. Click the incoming connector handle, drag it to the source Java element, and release the mouse button.
3. In the pop-up window, click a relationship type.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Creating extends relationships between Java classes in domain modeling class diagrams
Creating implements relationships between Java classes and interfaces in domain modeling class diagrams
Creating association relationships between Java classifiers in domain modeling class diagrams
Creating Java relationships from sources to targets in domain modeling class diagrams
Creating Java relationships from sources to unspecified targets in domain modeling class diagrams
Creating relationships from targets to unspecified sources in domain modeling class diagrams

Creating relationships from targets to unspecified sources in domain modeling class diagrams

You can use domain modeling class diagrams to create incoming relationships from targets to unspecified sources between classifiers to visually develop Java™ applications.

Procedure

1. In the diagram editor, place the cursor over the target classifier until the connector handles appear.
2. Click the incoming connector handle, drag it to an empty space in the diagram editor, and release the mouse button.
3. In the pop-up window, click a relationship type and click a new Java element to create or click an existing one as the source.
4. Follow the instructions in the wizard.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Creating extends relationships between Java classes in domain modeling class diagrams
Creating implements relationships between Java classes and interfaces in domain modeling class diagrams
Creating association relationships between Java classifiers in domain modeling class diagrams
Creating Java relationships from sources to targets in domain modeling class diagrams
Creating Java relationships from sources to unspecified targets in domain modeling class diagrams
Creating relationships from targets to sources in domain modeling class diagrams

Managing relationships of classifiers in domain modeling class diagrams

You can show or hide the existing relationships between Java™ classifiers in domain modeling class diagrams for visual representation and development of applications.

About this task

You can change Java field and association collection types to show attributes as associations or show associations as attributes. By default, Java attributes are shown as associations in domain modeling class diagrams. For example, if an attribute type is a collection, it is shown as an association to the collection type (vector) in a class diagram. In most cases, this might not be useful. Therefore, it is useful to show the type that the collection contains.

- **Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams**
  You can customize the default setting for showing or hiding type labels for Java association relationships in domain modeling class diagrams.
- **Defining default Java collection types for visual representation in domain modeling class diagrams**
  You can define default Java collection types by adding them to the Collection Types preferences for visual representation of Java applications in domain modeling class diagrams.
- **Customizing the default Java types to be shown or filtered out in domain modeling class diagrams**
  You can customize the default Java types to be shown or filtered out in domain modeling class diagrams when you populate diagrams.
- **Populating domain modeling class diagrams with Java source elements based on type**
  You can populate domain modeling class diagrams with existing Java classes and interfaces based on type to create visual representation of applications.
- **Showing Java attributes as associations in domain modeling class diagrams**
  You can show Java attributes, or fields, as association relationships in domain modeling class diagrams.
- **Showing Java attributes of collection type as associations in domain modeling class diagrams**
  You can show Java attributes of collection type, such as array lists, vectors, and so on, as association relationships to the collection type in domain modeling class diagrams. It is also useful to show the type of elements that the collection contains.
- **Showing Java associations as attributes in domain modeling class diagrams**
  You can show Java association relationships as attributes in Java classifiers in domain modeling class diagrams.
- **Showing Java elements in domain modeling class diagrams based on relationships**
  You can show related Java elements in domain modeling class diagrams based on their relationships to visually represent and develop Java applications.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Visually creating relationships between Java elements by using domain modeling class diagrams

Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams

You can customize the default setting for showing or hiding type labels for Java™ association relationships in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling**, expand **Java**, and click **Appearance**.
3. On the Appearance page, under **Connectors**, select **Show type label for Java association connectors**.
4. Click **OK**.

Related concepts:

Association relationships in domain modeling class diagrams

Related tasks:

Defining default Java collection types for visual representation in domain modeling class diagrams
Customizing the default Java types to be shown or filtered out in domain modeling class diagrams
Populating domain modeling class diagrams with Java source elements based on type
Showing Java attributes as associations in domain modeling class diagrams
Showing Java attributes of collection type as associations in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Showing Java elements in domain modeling class diagrams based on relationships

Defining default Java collection types for visual representation in domain modeling class diagrams

You can define default Java™ collection types by adding them to the Collection Types preferences for visual representation of Java applications in domain modeling class diagrams.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** , expand **Java** and click **Collection Types**.
3. On the Collection Types page, click **New**.
4. In the Select Java Type window, from the **Matching types** list, select a class or an interface, and from the **Qualifier** list, select the qualifier for the collection type, and click **OK**.
5. Click **Apply** and click **OK**.

Related tasks:

Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams

Customizing the default Java types to be shown or filtered out in domain modeling class diagrams

Populating domain modeling class diagrams with Java source elements based on type

Showing Java attributes as associations in domain modeling class diagrams

Showing Java attributes of collection type as associations in domain modeling class diagrams

Showing Java associations as attributes in domain modeling class diagrams

Showing Java elements in domain modeling class diagrams based on relationships

Customizing the default Java types to be shown or filtered out in domain modeling class diagrams

You can customize the default Java™ types to be shown or filtered out in domain modeling class diagrams when you populate diagrams.

Before you begin

Note: This preference setting only affects **Show Related Elements** actions and topic and browse diagrams. It does not prevent you from visualizing a binary type in the following situation:

- By using the **Visualize Existing** > **Java type** menu
- By using the **Visualize** menu on the selected binary type in the Package or Project Explorer view
- As part of visualizing attributes as associations
- By dragging and dropping from the Package or Project Explorer view into diagrams

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling**, expand **Java**, and click **Show Related elements Filters**.
3. On the Java Show Related Elements Filters page, complete one of the following steps:
   - To show all Java types, click **Show all Java types**.
   - To filter all binary Java types, click **Filter all binary Java types (defined in referenced libraries)**.
   - To filter specific Java types, click **Filter specific Java types**; then add the specific **Libraries**, **Packages**, or **Types**.
4. Click **OK**.

Related tasks:

Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams
Defining default Java collection types for visual representation in domain modeling class diagrams
Populating domain modeling class diagrams with Java source elements based on type
Showing Java attributes as associations in domain modeling class diagrams
Showing Java attributes of collection type as associations in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Showing Java elements in domain modeling class diagrams based on relationships

Populating domain modeling class diagrams with Java source elements based on type

You can populate domain modeling class diagrams with existing Java™ classes and interfaces based on type to create visual representation of applications.

Before you begin

You must have a domain modeling class diagram open.

Procedure

1. In the diagram editor, right-click an empty space; then click **Visualize Existing** > **Java Type**.
2. In the Visualize Java Type window, select a Java type available in your workspace and a qualifier.
3. Click **OK**.

Related concepts:

Association relationships in domain modeling class diagrams
Extends relationships in domain modeling class diagrams
Implements relationships in domain modeling class diagrams
Owned element association relationships in domain modeling class diagrams

Related tasks:

Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams
Defining default Java collection types for visual representation in domain modeling class diagrams
Customizing the default Java types to be shown or filtered out in domain modeling class diagrams
Showing Java attributes as associations in domain modeling class diagrams
Showing Java attributes of collection type as associations in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Showing Java elements in domain modeling class diagrams based on relationships

Showing Java attributes as associations in domain modeling class diagrams

You can show Java™ attributes, or fields, as association relationships in domain modeling class diagrams.

Before you begin

You must have a class diagram open.

About this task

You can also drag an attribute outside the classifier to show the attribute as an association.

Procedure

To show a Java attribute as an association in a domain modeling class diagram, in the diagram editor, right-click an attribute; then click **Filters** > **Show as Association**.

Results

The attribute is removed from the classifier and displayed as an association relationship.

Related concepts:

Association relationships in domain modeling class diagrams

Related tasks:

Creating association relationships between Java classifiers in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams
Defining default Java collection types for visual representation in domain modeling class diagrams
Customizing the default Java types to be shown or filtered out in domain modeling class diagrams
Populating domain modeling class diagrams with Java source elements based on type
Showing Java attributes of collection type as associations in domain modeling class diagrams
Showing Java elements in domain modeling class diagrams based on relationships

Showing Java attributes of collection type as associations in domain modeling class diagrams

You can show Java™ attributes of collection type, such as array lists, vectors, and so on, as association relationships to the collection type in domain modeling class diagrams. It is also useful to show the type of elements that the collection contains.

Before you begin

You must have a class diagram open.

About this task

The Select Type window opens only when the field does not already have a collection content type specified. In Java 5, collection content types are already specified, hence you no longer see the Select Type window. You can also drag an attribute outside the classifier to show the attribute as an association.

Procedure

1. In the diagram editor, right-click an attribute of collection type; then click **Filters** > **Show as Association**.
2. In the Select Type window, from the **Matching type** list, specify the type of elements that the collection contains and click **OK**.

Results

The attribute is removed from the classifier and displayed as an association relationship along with the element type that the collection contains.

Related tasks:

Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams
Defining default Java collection types for visual representation in domain modeling class diagrams
Customizing the default Java types to be shown or filtered out in domain modeling class diagrams
Populating domain modeling class diagrams with Java source elements based on type
Showing Java attributes as associations in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams
Showing Java elements in domain modeling class diagrams based on relationships

Showing Java associations as attributes in domain modeling class diagrams

You can show Java™ association relationships as attributes in Java classifiers in domain modeling class diagrams.

Before you begin

You must have a domain modeling class diagram open.

Procedure

To show a Java association as an attribute, in the diagram editor, right-click an association relationship; then click **Filters** > **Show as Attribute**.**Tip:** You can also drag an association onto the attribute compartment of the source of the association or right-click an association connector and click **Delete from Diagram** to show the association as an attribute.

Results

The association is removed from the diagram and is added to the source classifier as an attribute.

Related concepts:

Association relationships in domain modeling class diagrams

Related tasks:

Creating association relationships between Java classifiers in domain modeling class diagrams
Adding fields to Java classes and interfaces in domain modeling class diagrams
Showing Java attributes as associations in domain modeling class diagrams
Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams
Defining default Java collection types for visual representation in domain modeling class diagrams
Customizing the default Java types to be shown or filtered out in domain modeling class diagrams
Populating domain modeling class diagrams with Java source elements based on type
Showing Java attributes of collection type as associations in domain modeling class diagrams
Showing Java elements in domain modeling class diagrams based on relationships

Showing Java elements in domain modeling class diagrams based on relationships

You can show related Java™ elements in domain modeling class diagrams based on their relationships to visually represent and develop Java applications.

Before you begin

You must have a domain modeling class diagram open.

About this task

This procedure expands the related elements of that relationship type to only one level. However, you can select the **Show Related Elements** option and specify the levels of depth to expand the related elements.

Procedure

1. In the diagram editor, place the cursor over a source or target Java classifier until the outgoing and incoming connector handles appear; then double-click the outgoing or incoming connector handle.
2. In the pop-up window, select a relationship.

Related concepts:

Relationships in domain modeling class diagrams

Related tasks:

Setting preferences for showing or hiding Java association type labels in domain modeling class diagrams
Defining default Java collection types for visual representation in domain modeling class diagrams
Customizing the default Java types to be shown or filtered out in domain modeling class diagrams
Populating domain modeling class diagrams with Java source elements based on type
Showing Java attributes as associations in domain modeling class diagrams
Showing Java attributes of collection type as associations in domain modeling class diagrams
Showing Java associations as attributes in domain modeling class diagrams

Navigating to Java code elements from domain modeling diagrams

You can navigate from domain modeling diagrams to the source code of Java™ diagram elements or to Java elements (classes and interfaces) in the Package Explorer view. You can also start Java editors from the diagram editor to make modifications to the source code of Java elements.

- **Automatically navigating to the source code of Java diagram elements**

  You can automatically navigate from the diagram editor to the source code of Java diagram elements, including classes, interfaces, enumerations, fields, and methods, to make modifications.

- **Navigating from class diagrams to Java elements in the Package Explorer or Project Explorer view**

  You can navigate from domain modeling class diagrams to Java elements in the Package Explorer or Project Explorer view.

- **Starting the default Java editor from domain modeling class diagrams**

  You can start the default Java editor from domain modeling class diagrams to make modifications to the source code of Java classes and interfaces.

Related tasks:

Setting preferences for handling older format diagrams

Visually developing structural features of Java applications by using domain modeling class diagrams

Generating Javadoc HTML documentation with domain modeling diagram images

Automatically navigating to the source code of Java diagram elements

You can automatically navigate from the diagram editor to the source code of Java™ diagram elements, including classes, interfaces, enumerations, fields, and methods, to make modifications.

Procedure

1. In the diagram editor, right-click a Java diagram element.
2. On the pop-up menu, click **Navigate** > **Navigate to Source**

Results

The source code of the selected Java element opens in a separate editor. **Remember:** You can always use the pop-up code editor from within class diagrams to view or directly edit Java classifiers.

Tip: To navigate to the source code of Java diagram elements, you can also click the **Auto-Navigate to Source** button on the tool bar and, in the diagram editor, click a Java diagram element. As your selection of the element in the diagram changes, the corresponding source code is shown in the source editor.

Related tasks:

Setting preferences for showing the pop-up code editor in domain modeling class diagrams
Navigating from class diagrams to Java elements in the Package Explorer or Project Explorer view
Starting the default Java editor from domain modeling class diagrams

Navigating from class diagrams to Java elements in the Package Explorer or Project Explorer view

You can navigate from domain modeling class diagrams to Java™ elements in the Package Explorer or Project Explorer view.

Before you begin

You must have a domain modeling class diagram open.

Procedure

To navigate to a Java element in the Package Explorer view, in the diagram editor, right-click a Java package, class, or interface; then click **Navigate** > **Show in** > **Package Explorer** or **Project Explorer**.

Related tasks:

Automatically navigating to the source code of Java diagram elements

Starting the default Java editor from domain modeling class diagrams

Starting the default Java editor from domain modeling class diagrams

You can start the default Java™ editor from domain modeling class diagrams to make modifications to the source code of Java classes and interfaces.

Before you begin

You must be in the Java perspective and have a class diagram open.

About this task

Changes that you make to the source code are automatically reflected in the corresponding diagram.

Procedure

To start the default Java editor from a class diagram, in the diagram editor, right-click a Java class or interface; then click **Navigate** > **Open**.**Tip:** To open the default Java editor, you can also double-click a Java classifier or relationship in the diagram editor.

Related tasks:

Automatically navigating to the source code of Java diagram elements

Navigating from class diagrams to Java elements in the Package Explorer or Project Explorer view

Generating Javadoc HTML documentation with domain modeling diagram images

You can generate Javadoc HTML documentation that contains domain modeling diagram images to provide more information about the source code inside Java™ projects.

About this task

You can integrate domain modeling diagram images into Javadoc HTML documentation in parallel with the source code inside Java projects by including the *@viz.diagram* tag. This tag can define a specific diagram image to appear in Javadoc HTML documentation. There is a specific format for the tags, for example, *@viz.diagram class_diagram.dnx* that inserts an image of the contents of *class_diagram.dnx* into Javadoc HTML documentation.For example, you can specify that the tool produces Javadoc HTML documentation for particular classes, shows the author of the code in a prominent way, and displays a simple domain modeling diagram with some aspects of the inheritance hierarchy of the classes inside a Java project. Some of the usage relationships with other classes might help increase your understanding of the source code inside the project.

You can generate diagram images in the bitmap format (.bmp), the Graphic Interchange Format (.gif), or the Joint Photographic Experts Group format (.jpeg or .jpg).

- **Automatically generating Javadoc HTML documentation with domain modeling diagram images**
  You can automatically generate Javadoc HTML documentation that contains domain modeling diagram images by using the default Javadoc options and formats.
- **Generating Javadoc HTML documentation with domain modeling diagram images from existing tags**
  You can generate Javadoc HTML documentation that contains domain modeling diagram images by using the existing *@viz.diagram* tags in your comments for the top-level type of a Java source code file (a class or an interface) or for a package.html file inside a Java project.

Related tasks:

Setting preferences for handling older format diagrams

Visually developing structural features of Java applications by using domain modeling class diagrams

Navigating to Java code elements from domain modeling diagrams

Automatically generating Javadoc HTML documentation with domain modeling diagram images

You can automatically generate Javadoc HTML documentation that contains domain modeling diagram images by using the default Javadoc options and formats.

Before you begin

You must have installed Oracle Java™ Development Kit (JDK) Version 1.4.2 or later that contains a Javadoc executable file.

Procedure

1. In the Package Explorer view, select a Java project and click **Project** > **Generate Javadoc with Diagrams** > **Automatically**.
2. In the Generate Javadoc wizard, under **Javadoc command**, select the Javadoc command (an executable file).**Note:** Only Oracle JDK Version 1.4.2 or later is fully supported and ensure that you select the appropriate Javadoc executable file that comes with the Oracle JDK.
3. Under **Diagrams**, complete one of the following steps:
   - To include diagram images for packages, click **For packages**. Each package summary page contains a diagram that shows the contents of the packages.
   - To include diagram images for Java types, click **For types**. Each Java type (class or interface) summary page contains a diagram that shows extends, implements, usages, and inner types to one level of depth.
   - Select an **Image type**. Available formats are GIF, JPG, and BMP.
4. To embed the diagram image tags in the source code, click **Contribute diagrams and diagram tags to source**.
5. Click **Finish**.

What to do next

Note: This will insert UML diagram image tags into your source code. Now you can generate Javadoc HTML documentation by clicking **Project** > **Generate Javadoc with Diagrams** > **From Existing Tags**.

Related tasks:

Generating Javadoc HTML documentation with domain modeling diagram images from existing tags

Generating Javadoc HTML documentation with domain modeling diagram images from existing tags

You can generate Javadoc HTML documentation that contains domain modeling diagram images by using the existing *@viz.diagram* tags in your comments for the top-level type of a Java™ source code file (a class or an interface) or for a package.html file inside a Java project.

Before you begin

You must have installed Oracle Java Development Kit (JDK) Version 1.4.2 or later that contains a Javadoc command (an executable file). You also have a Java project with Java types (classes and interfaces) or packages that have *@viz.diagram* tags inserted. You must be in the Java perspective.

Procedure

1. In the Package Explorer view, select a Java project and click **Project** > **Generate Javadoc with Diagrams** > **From Existing Tags**.
2. In the Generate Javadoc wizard, on the Select types for Javadoc generation page, select an image type for the diagrams.
3. To generate an automated ant script, specify a destination for the Doclet.
4. Click **Finish**.

What to do next

Note: If this is the first time you generate a Javadoc HTML documentation with diagram images for your Java project and have not inserted *@viz.diagram* tags in the source code, you can automatically generate such documentation by clicking **Project** > **Generate Javadoc with Diagrams** > **Automatically**.

Related tasks:

Automatically generating Javadoc HTML documentation with domain modeling diagram images

Adding file artifacts to diagrams

You can represent existing file artifacts or create new ones in UML diagrams either to represent actual files in your projects or to represent UML modeling artifact elements. You can place UML modeling artifact elements in diagrams that have .dnx or .emx as a file name extension. In UML diagrams, file artifacts are shown as UML artifacts with the <<file>> stereotype.

- **Representing existing file artifacts in UML diagrams**

  In UML diagrams that have .dnx as a file name extension, you can visually represent existing project files as artifacts.

- **Creating new file artifacts in UML diagrams**

  You can use UML diagrams that have .dnx as a file name extension to create new file artifacts in your projects.

Representing existing file artifacts in UML diagrams

In UML diagrams that have .dnx as a file name extension, you can visually represent existing project files as artifacts.

Procedure

To represent an existing file artifact, in the navigation view, select the file artifact and drag it into an open diagram.

Results

The file artifact is created and displayed as a UML artifact in the diagram.

Related tasks:

Creating new file artifacts in UML diagrams

Creating new file artifacts in UML diagrams

You can use UML diagrams that have .dnx as a file name extension to create new file artifacts in your projects.

Before you begin

You must have a diagram open that has .dnx as a file name extension.

Procedure

1. In the Palette, click **File**; then click in an empty space in the diagram.
2. Follow the instructions in the New File wizard.

Results

A new file artifact is created and displayed as a UML artifact in the diagram with the same name as the file that you just created in the wizard, including the file name extension.

Related tasks:

Representing existing file artifacts in UML diagrams

Exploring relationships in applications

You can use topic and browse diagrams to explore and navigate through an application and to view the details of its elements and relationships.

About this task

In a topic diagram, you can query a specific topic in an application, and then you can explore the details of a selected context element and its relationships in a browse diagram.You can refresh topic and browse diagrams to reflect the latest changes to the source code

You cannot save browse diagrams and cannot edit topic or browse diagrams. However, you can convert them to editable Unified Modeling Language (UML) class diagrams, with a .dnx file name extension, that you can use to visually represent and develop applications.

- **Querying elements and relationships in applications**

  You can create a topic diagram to dynamically explore existing relationships between elements in an application. You can customize, refresh, and save a topic diagram.
- **Browsing elements and relationships in applications**

  A browse diagram is a temporary diagram in which you can explore the details of an application and its underlying elements and relationships. A browse diagram provides a view of a context element that you specify and is similar in functionality to a Web browser. You cannot add or modify the individual diagram elements or save a browse diagram in an application; however, you can convert a browse diagram to a UML class diagram or save it in an image file that captures the inheritance hierarchy, which you can send in an email message or publish on a website.

Querying elements and relationships in applications

You can create a topic diagram to dynamically explore existing relationships between elements in an application. You can customize, refresh, and save a topic diagram.

About this task

For example, you can create topic diagrams to visually represent entire inheritance hierarchies or associations of Java™ classes by using the predefined queries **Inherited Java Classes (N levels)** or **Java Field Types**. You can customize a query for an existing topic diagram to change its context or level of detail.

- **Creating topic diagrams of application elements**
  You can use topic diagrams to create dynamic views of relationships between elements in applications, including packages with external JAR files in Java applications.
- **Customizing queries for existing topic diagrams**
  You can customize a query for an existing topic diagram if you want to change its context or level of detail. When you customize the topic query, you cannot change individual diagram elements; you must change the query for the entire diagram.
- **Refreshing topic and browse diagrams**
  When you open a topic diagram, the diagram elements in a project are refreshed automatically. However, if you make changes to the underlying elements when a topic or browse diagram is open, you must manually refresh the diagram to see all the changes.
- **Saving topic diagrams**
  You can save a topic diagram as an image file or a save as topic diagram or convert it to an editable UML diagram so that you can work with it further.

Related concepts:

Topic diagrams

Related tasks:

Creating topic diagrams of application elements
Customizing queries for existing topic diagrams
Refreshing topic and browse diagrams
Saving browse diagrams

Topic diagrams

A topic diagram is a non-editable, query-based diagram that you can use to view existing relationships between elements. A topic diagram displays a specific set of relationships between a group of given elements. The set of relationships, or the topic, defines the parameters of the query for the diagram.

The query and the context that you specify persists in the topic diagram. Each time that you open a topic diagram, the system queries the underlying elements and automatically populates the diagram with the moat recent results. However, if you make changes to the underlying elements when a topic diagram is open, the diagram might not represent the current status of the elements until you refresh the diagram manually.

You can select from the Properties page, a graph, link, or label layout format for the topic diagram. The default layout shows all diagram elements in a hierarchy that can span several levels.

A topic diagram is similar to a browse diagram except that you can save a topic diagram and reopen it at any time. Unless you save a topic diagram as an editable diagram, you cannot add or modify the diagram elements.

Related tasks:

Exploring relationships in UML models
Querying elements and relationships in UML models
Creating topic diagrams of UML elements
Customizing queries for existing topic diagrams
Refreshing topic and browse diagrams
Saving topic diagrams
Querying elements and relationships in applications
Creating topic diagrams of application elements
Saving browse diagrams
Browsing elements and relationships in applications

Creating topic diagrams of application elements

You can use topic diagrams to create dynamic views of relationships between elements in applications, including packages with external JAR files in Java™ applications.

Before you begin

You must have created or imported a project.

Procedure

1. To specify the context element for the topic query, in the diagram editor, right-click a diagram element; then click **Visualize** > **In Topic Diagram**.
2. In the Topic Diagram window, on the Topic Diagram Location page, specify a parent folder, type a name, and click **Next**.
3. On the Topics page, select a predefined query and click **Finish**.

Related concepts:

Topic diagrams

Related tasks:

Querying elements and relationships in applications
Customizing queries for existing topic diagrams
Refreshing topic and browse diagrams
Saving browse diagrams

Customizing queries for existing topic diagrams

You can customize a query for an existing topic diagram if you want to change its context or level of detail. When you customize the topic query, you cannot change individual diagram elements; you must change the query for the entire diagram.

Procedure

1. In the diagram editor, right-click an empty space; then click **Customize Query**.
2. In the Configure Query window, specify the relationship types, the expansion directions, the levels of depth to query, and the layout type for the diagram.
3. Click **Finish**.

Related concepts:

Topic diagrams

Related tasks:

Querying elements and relationships in applications

Creating topic diagrams of application elements

Refreshing topic and browse diagrams

Saving browse diagrams

Refreshing topic and browse diagrams

When you open a topic diagram, the diagram elements in a project are refreshed automatically. However, if you make changes to the underlying elements when a topic or browse diagram is open, you must manually refresh the diagram to see all the changes.

About this task

To refresh a topic or browse diagram, in the diagram editor, right-click an empty space; then click **Refresh**.

Related concepts:

Topic diagrams

Browse diagrams

Related tasks:

Creating browse diagrams of UML model elements

Navigating in browse diagrams

Saving topic diagrams

Querying elements and relationships in applications

Creating topic diagrams of application elements

Customizing queries for existing topic diagrams

Saving browse diagrams

Browsing elements and relationships in applications

Creating browse diagrams of application elements

Saving topic diagrams

You can save a topic diagram as an image file or a save as topic diagram or convert it to an editable UML diagram so that you can work with it further.

Procedure

1. In the diagram editor, right-click an empty space; then click **File**.
2. Complete one of the following:
   - To save as an editable diagram, click **Save as Diagram File**.
   - To save as an editable diagram in a model, click **Save as Diagram in Model**.
   - To save as an image, click **Save as Image File**.
3. In the Save As window, specify a folder, type a name, and click **OK**.

Related concepts:

Topic diagrams

Browse diagrams

Related tasks:

Creating browse diagrams of UML model elements

Navigating in browse diagrams

Refreshing topic and browse diagrams

Browsing elements and relationships in applications

A browse diagram is a temporary diagram in which you can explore the details of an application and its underlying elements and relationships. A browse diagram provides a view of a context element that you specify and is similar in functionality to a Web browser. You cannot add or modify the individual diagram elements or save a browse diagram in an application; however, you can convert a browse diagram to a UML class diagram or save it in an image file that captures the inheritance hierarchy, which you can send in an email message or publish on a website.

- **Creating browse diagrams of application elements**

  You can create browse diagrams of application elements to visually represent and explore elements and relationships in applications.

- **Navigating in browse diagrams**

  In browse diagrams, you can explore a diagram element and its relationships. You can specify the relationships and number of levels of relationships that you want to show in a browse diagram. You can navigate individual diagram elements to view the history of the diagram elements and their relationships.

- **Refreshing topic and browse diagrams**

  When you open a topic diagram, the diagram elements in a project are refreshed automatically. However, if you make changes to the underlying elements when a topic or browse diagram is open, you must manually refresh the diagram to see all the changes.

- **Saving browse diagrams**

  You can save a browse diagram as an image file or convert it to an editable UML diagram so that you can work with it further. You cannot directly save a browse diagram convert.

Related concepts:

Browse diagrams

Topic diagrams

Related tasks:

Creating browse diagrams of application elements

Navigating in browse diagrams

Refreshing topic and browse diagrams

Saving browse diagrams

Browse diagrams

A browse diagram is a temporary, non-editable diagram that shows the results of a query on a context diagram element. You can use browse diagrams to navigate elements and relationships in a project. For example, you can create a browse diagram to show a dynamic view of a class and its related elements to understand how it fits into the overall project structure.

Browse diagrams provide a view of a context element that you specify and are similar in functionality to a Web browser. A browse diagram retains the history of what you view. You can navigate backward and forward to view previously viewed context elements and their relationships without creating another diagram. For example, you can browse a specific class to see its relationships to other elements. You can double-click an element in a browse diagram to make it the context element and to update the diagram to show the new context.

You cannot control the format of a browse diagram. The elements in the diagram can be arranged using the Properties page to select a graph, link, or label layout.

Browse diagrams do not persist, but you can refresh them to reflect the changes to their underlying elements.

A browse diagram is not editable; you cannot change its underlying model elements. However, you can convert a browse diagram to an editable UML diagram so that you can add and modify its elements. You can also save a browse diagram as an image file.

Related tasks:

Creating browse diagrams of UML model elements
Navigating in browse diagrams
Refreshing topic and browse diagrams
Saving topic diagrams
Browsing elements and relationships in UML models
Browsing elements and relationships in applications
Creating browse diagrams of application elements
Saving browse diagrams

Creating browse diagrams of application elements

You can create browse diagrams of application elements to visually represent and explore elements and relationships in applications.

Before you begin

You must have created or imported a project.

About this task

To create a browse diagram of application elements, in the navigation view or diagram, right-click an element; then click **Visualize** > **Explore in Browse Diagram**.

Results

The browse diagram opens in a radial layout, with the selected application element as the center, or in an inheritance tree-view layout if the application elements contain only inheritance relationships.

Related concepts:

Browse diagrams

Related tasks:

Browsing elements and relationships in applications
Navigating in browse diagrams
Refreshing topic and browse diagrams
Saving browse diagrams

Navigating in browse diagrams

In browse diagrams, you can explore a diagram element and its relationships. You can specify the relationships and number of levels of relationships that you want to show in a browse diagram. You can navigate individual diagram elements to view the history of the diagram elements and their relationships.

About this task

To navigate in a browse diagram, in the diagram editor, complete any of the following steps:

- To explore the details of a diagram element, double-click it. This element becomes the context diagram element.
- To filter the relationships that you want to view for the context element, in the toolbar, click a relationship button and click **Apply**.
- To change the number of levels of relationships that you want to view for the context element, in the toolbar, specify a number and click **Apply**
- To navigate between diagram elements that you opened in a browse diagram, click the Down arrow to select an element from the history list or click the Forward or Back arrow to move between consecutive context elements.
- To view the inheritance structure of a diagram element, right-click the diagram element; then use the Navigate menu.

Related concepts:

Browse diagrams

Related tasks:

Creating browse diagrams of UML model elements
Refreshing topic and browse diagrams
Saving topic diagrams
Browsing elements and relationships in applications
Creating browse diagrams of application elements
Saving browse diagrams

Saving browse diagrams

You can save a browse diagram as an image file or convert it to an editable UML diagram so that you can work with it further. You cannot directly save a browse diagram convert.

Procedure

1. In the diagram editor, right-click an empty space; then click **File**.
2. Complete one of the following:
   - To save as an editable diagram, click **Save as**.
   - To save as an image, click **Save as Image File**.
3. In the Select a Container window, specify a folder, type a name, and click **OK**.

Related concepts:

Topic diagrams

Browse diagrams

Related tasks:

Querying elements and relationships in applications

Creating topic diagrams of application elements

Customizing queries for existing topic diagrams

Refreshing topic and browse diagrams

Browsing elements and relationships in applications

Creating browse diagrams of application elements

Navigating in browse diagrams

Exploring methods by using static sequence diagrams

You can create static sequence diagram views of methods (operations), including constructs and signatures, in classifiers to visually represent and explore behaviors and interactions in applications.

About this task

Static sequence diagrams show views of interaction operators. A static sequence diagram is a topic diagram, which has a .tpx file name extension. You can refresh static sequence diagram views of methods to reflect the latest changes in the source code. You can convert them to editable UML sequence diagrams, which have a .dnx file name extension, that you can use to understand and develop behaviors and interactions between classifiers.

- **Customizing settings to filter Java types in new static method views**
  You can customize settings to specify which Java™ types to show in new static sequence diagram views of Java methods.
- **Creating views of methods by using static sequence diagrams**
  You can create static sequence diagram to present views of methods (operations), including constructs and signatures, in classifiers to visually represent and explore behaviors and interactions in applications.
- **Refreshing views of methods in static sequence diagrams**
  You can refresh views of methods in static sequence diagrams to reflect the latest changes in the source code.
- **Converting static sequence diagrams to editable UML sequence diagrams**
  You can convert static sequence diagrams to editable UML sequence diagrams, which have a .dnx file name extension, that you can use to visually represent and develop behaviors and interactions between classifiers.

Customizing settings to filter Java types in new static method views

You can customize settings to specify which Java™ types to show in new static sequence diagram views of Java methods.

Before you begin

You must have the Java domain modeling capabilities enabled.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** > **Java** and click **Static Method Sequence Diagram Filters**.
3. On the Static Method Sequence Diagram Filters page, select one of the following options:
   - **Show all Java types**
   - **Filter all binary Java types (defined in referenced libraries)**
   - **Filter specific Java types**
4. If you select **Filter specific Java types**, follow the instructions on the page to add libraries, types, and packages.
5. Click **Apply** and then click **OK**

Related tasks:

Creating views of methods by using static sequence diagrams

Refreshing views of methods in static sequence diagrams

Converting static sequence diagrams to editable UML sequence diagrams

Creating views of methods by using static sequence diagrams

You can create static sequence diagram to present views of methods (operations), including constructs and signatures, in classifiers to visually represent and explore behaviors and interactions in applications.

Procedure

1. In the navigation view, right-click a method; then click **Visualize** > **Add to New Diagram File** > **Static Method Sequence Diagram**.
2. In the New Static Method Sequence Diagram wizard, in the **Enter or select the parent folder** field, specify the parent folder.
3. In the **File name** field, type a file name.
4. Click **Finish**.

What to do next

Tip: You can also create static views of methods from class, topic, or browse diagrams: Right-click a method in a classifier; then click **Visualize** > **In Static Method Sequence Diagram**.

Related tasks:

Customizing settings to filter Java types in new static method views

Refreshing views of methods in static sequence diagrams

Converting static sequence diagrams to editable UML sequence diagrams

Refreshing views of methods in static sequence diagrams

You can refresh views of methods in static sequence diagrams to reflect the latest changes in the source code.

Before you begin

You must have a static sequence diagram open.

Procedure

To refresh a view of methods in a static sequence diagram, in the diagram editor, right-click an empty space; then click **Refresh**.

Related tasks:

Customizing settings to filter Java types in new static method views

Creating views of methods by using static sequence diagrams

Converting static sequence diagrams to editable UML sequence diagrams

Converting static sequence diagrams to editable UML sequence diagrams

You can convert static sequence diagrams to editable UML sequence diagrams, which have a .dnx file name extension, that you can use to visually represent and develop behaviors and interactions between classifiers.

Before you begin

You must have a static sequence diagram open in the diagram editor.

Procedure

1. In the diagram editor, right-click an empty space; then click **File** > **Save as Diagram File**.
2. In the Save As window, specify a parent folder and type a file name.
3. Click **OK**.

Related tasks:

Customizing settings to filter Java types in new static method views

Creating views of methods by using static sequence diagrams

Refreshing views of methods in static sequence diagrams

Managing sequence diagrams

You can use Unified Modeling Language (UML) sequence diagrams to create a graphical representation of the source elements in a system or application to understand and develop behaviors and interactions between classes and data types.

About this task

You can create new sequence diagrams, populate existing sequence diagrams with source elements, and add lifelines, messages, and combined fragments to sequence diagrams.

- **Setting preferences for sequence and communication diagrams**

  You can set several default preferences for sequence and communication diagrams. For example, you can specify the appearance of lifelines, messages, and execution occurrences that are displayed in sequence and communication diagrams when you create them.

- **Creating sequence diagrams from existing diagram elements**

  You can create sequence diagrams in the typical way that you create any UML diagram, or you can create them from existing elements in your project. You can create a sequence diagram either within a model, which also creates a collaboration and interaction, or as a standalone diagram, which becomes a DNX file.

- **Populating sequence diagrams**

  You can create sequence diagrams and populate them with elements to understand and develop behaviors and interactions between objects in systems. You can add lifelines to represent objects or actors, messages to represent communication between the objects, and combined fragments to represent the control structures.

- **Organizing sequence diagrams**

  You can organize sequence diagrams to better represent a system. You can reorder messages, delay asynchronous messages, as well as manage combined fragments and lifelines.

- **Deleting messages from sequence diagrams**

  You can delete messages from your project that you no longer need. When you delete a message, you have different results depending on the type of message that you delete.

- **Deleting lifelines from UML diagrams**

  You can delete lifelines from UML diagrams, such as sequence or communication diagrams, to remove an object from an interaction. When you delete a lifeline, all send and receive messages that originate from the deleted lifeline are also deleted. When you delete a lifeline from a sequence diagram, all interaction fragments that cover the lifeline are deleted, but the combined fragments and interaction uses that cover other lifelines are not.

- **Example of a sequence diagram**

  The following example shows a sequence diagram taken from the PiggyBank sample that is available in the Samples Gallery, which describes the DisplayBalance use case from an online banking scenario.

Sequence diagrams

A sequence diagram is a Unified Modeling Language (UML) diagram that illustrates the sequence of messages between objects in an interaction.A sequence diagram consists of a group of objects that are represented by lifelines, and the messages that they exchange over time during the interaction.

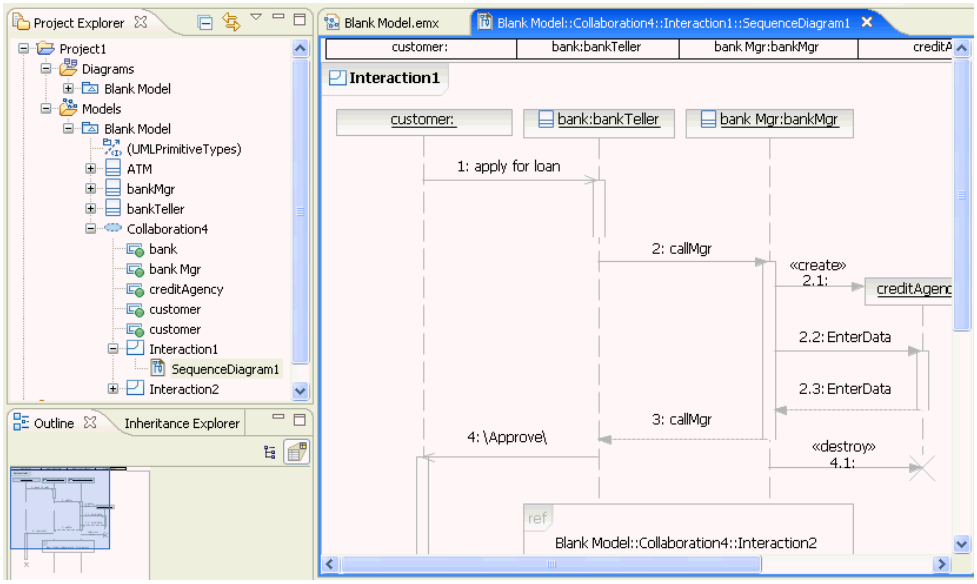
A sequence diagram shows the sequence of messages passed between objects. Sequence diagrams can also show the control structures between objects. For example, lifelines in a sequence diagram for a banking scenario can represent a customer, bank teller, or bank manager. The communication between the customer, teller, and manager are represented by messages passed between them. The sequence diagram shows the objects and the messages between the objects.

Development process and sequence diagrams

As the following table illustrates, you can use sequence diagrams at different stages during the development process to describe interactions between objects in a system.

Phase	Description
Analysis	During the analysis phase, you can use sequence diagrams to illustrate the interactions of class instances to realize a use case. In the analysis phase, sequence diagrams can help identify the classes that a system needs and what class objects do in interactions.
Design	You can refine sequence diagrams to show how a system completes interactions. In the design phase, sequence diagrams explain how the system works to accomplish interactions.
Construction	During the construction of a system architecture, you can use sequence diagrams to show the behavior of design patterns and mechanisms that the system uses.

As the following figure illustrates, when you create a sequence diagram, the collaboration and the interaction appear in the Project Explorer view, and an interaction frame appears in the diagram editor.



In the interaction frame, you position instances that participate in the interaction in any order from left to right, and then you position the messages between the participants in sequential order from top to bottom. Execution specifications appear on the lifelines and show the start and finish of the flow of control.

The following topics describe the elements in sequence diagrams:

- **Interaction frames**

  In sequence diagrams and communication diagrams, an interaction frame provides a context or boundary to the diagram in which you create diagram elements, such as lifelines or messages, and in which you observe behavior.
- **Lifelines in UML diagrams**

  In UML diagrams, such as sequence or communication diagrams, lifelines represent the objects that participate in an interaction. For example, in a banking scenario, lifelines can represent objects such as a bank system or customer. Each instance in an interaction is represented by a lifeline.
- **Messages in UML diagrams**

  A message is an element in a Unified Modeling Language (UML) diagram that defines a specific kind of communication between instances in an interaction. A message conveys information from one instance, which is represented by a lifeline, to another instance in an interaction.
- **Combined fragments in sequence diagrams**

  In sequence diagrams, combined fragments are logical groupings, represented by a rectangle, which contain the conditional structures that affect the flow of messages. A combined fragment contains interaction operands and is defined by the interaction operator.
- **Interaction uses in sequence diagrams**

  In sequence diagrams, interaction uses enable you to reference other existing interactions. You can construct a complete and complex sequence from smaller simpler interactions.

Related concepts:

Communication diagrams

Related tasks:

Modeling the interactions between objects in UML

Creating sequence diagrams

Creating lifelines in UML diagrams

Creating messages in UML diagrams

Creating combined fragments in sequence diagrams

Related reference:

Example of a sequence diagram

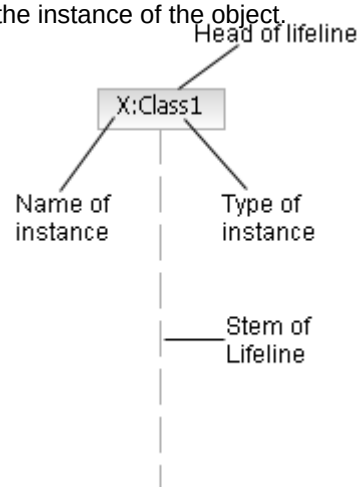
Example of a communication diagram

Lifelines in UML diagrams

In UML diagrams, such as sequence or communication diagrams, lifelines represent the objects that participate in an interaction. For example, in a banking scenario, lifelines can represent objects such as a bank system or customer. Each instance in an interaction is represented by a lifeline.

Lifelines in sequence diagrams

As the following figure illustrates, a lifeline in a sequence diagram is displayed with its name and type in a rectangle, which is called the head. The head is located on top of a vertical dashed line, called the stem, which represents the timeline for the instance of the object.



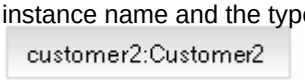
Messages, which are sent and received by the instance, appear on the lifeline in sequential order. You can create new lifelines, create lifelines from existing elements, or assign element types to existing lifelines.

As the following table illustrates, lifelines can indicate several actions in sequence diagrams.

Behavior	Description
Creation	You can create an instance in an interaction by using a create message. The create message enables an object to create new objects in the sequence diagram.
Communication	You indicate messages with arrows between instances. The arrow originates from the source lifeline that sends the message and ends at the target lifeline that receives it.
Execution	An execution specification shows the length of a behavior of an operation directly or through a subordinate operation.
Destruction	You can destroy an instance during an interaction by using a destroy message or a stop node. A destroy message is a message that ends the target lifeline. A stop node, represented by an X, marks the end of the stem of the lifeline to indicate the lifeline has ended.

Lifelines in communication diagrams

As the following figure illustrates, a lifeline in a communication diagram is represented by a rectangle that contains the instance name and the type.



Related tasks:

Creating lifelines in UML diagrams

Deleting lifelines from UML diagrams

Messages in UML diagrams

A message is an element in a Unified Modeling Language (UML) diagram that defines a specific kind of communication between instances in an interaction. A message conveys information from one instance, which is represented by a lifeline, to another instance in an interaction.

Types of messages

A message specifies a sender and receiver, and defines the kind of communication that occurs between lifelines. For example, a communication can invoke, or call, an operation by using a synchronous call message or asynchronous call message, can raise a signal using an asynchronous signal, and can create or destroy a participant.

You can use the five types of messages that are listed in the following table to show the communication between lifelines in an interaction.

Message type	Description	Example
Create	A create message represents the creation of an instance in an interaction. The create message is represented by the keyword «create». The target lifeline begins at the point of the create message.	In a banking scenario, a bank manager might start a credit check on a client by sending a create message to the server.
Destroy	A destroy message represents the destruction of an instance in an interaction. The destroy message is represented by the keyword «destroy». The target lifeline ends at the point of the destroy message, and is denoted by an X.	A bank manager, after starting a credit check, might close or destroy the credit program application for a customer.
Synchronous call	Synchronous calls, which are associated with an operation, have a send and a receive message. A message is sent from the source lifeline to the target lifeline. The source lifeline is blocked from other operations until it receives a response from the target lifeline.	A bank teller might send a credit request to the bank manager for approval and must wait for a response before further serving the customer.
Asynchronous call	Asynchronous calls, which are associated with operations, typically have only a send message, but can also have a reply message. In contrast to a synchronous message, the source lifeline is not blocked from receiving or sending other messages. You can also move the send and receive points individually to delay the time between the send and receive events. You might choose to do this if a response is not time sensitive or order sensitive.	A bank customer could apply for credit but can receive banking information over the phone or request money from an ATM, while waiting to hear about the credit application.
Asynchronous signal	Asynchronous signal messages are associated with signals. A signal differs from a message because no operation is associated with the signal. A signal can represent an interrupt condition or error condition. To specify a signal, you create an asynchronous call message and change the type in the message properties view.	A credit agency could send an error signal message to the bank manager that states a failure to connect to the credit bureau.
Lost and found	A lost message is a message that has a known sender but the receiver is not known. A found message is a message that does not have a known sender but has a receiver.	An outside actor sends a message to a bank manager. The actor is outside the scope of the sequence diagram and is therefore a found message. A lost message can occur when a message is sent to an element outside the scope of the UML diagram.

An asynchronous message is the only message type for which you can individually move the sending and receiving points. You can move the points of an asynchronous message to manipulate the time delay between the sending event and the

receiving event; the result is called a skewed message. You can create an asynchronous message with or without a behavior execution specification.

A self-directed message is a message that is sent from the source lifeline to itself. A self-directed message could be a recursive call or a call to another operation or signal that belongs to the same object.

The message that the source lifeline sends to the target lifeline represents an operation or a signal that the target lifeline implements. You can name and order messages. The appearance of the line or arrowhead reflects the properties of the message. The following table shows the graphics that represent messages in UML diagrams.

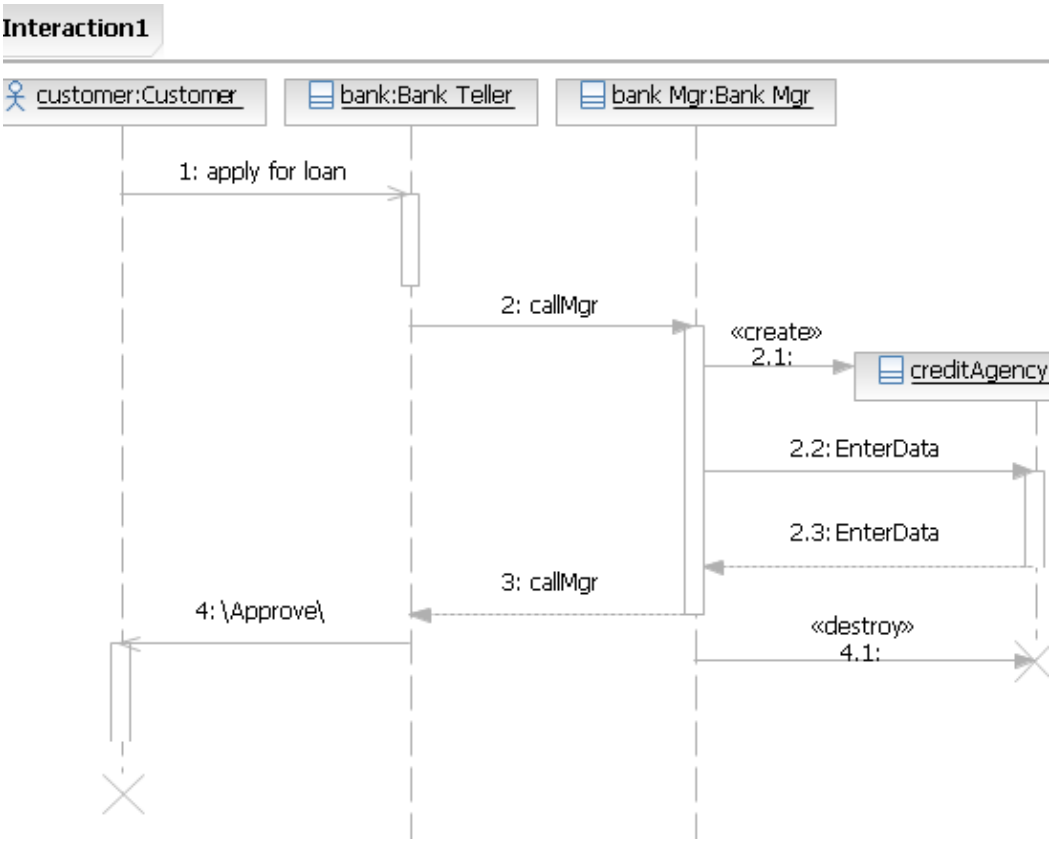
Message type	Graphic	Description	Representation
Asynchronous		A line with an open arrowhead	This graphic represents an asynchronous signal or an asynchronous call in which the source object sends the message and immediately continues with the next step.
Synchronous		A line with a solid arrowhead that points toward the receiving lifeline	This graphic represents a synchronous call operation in which the source object sends a message and waits for a return message from the target before the source can continue.
Synchronous return		A dashed line with a solid arrowhead that points toward the originating lifeline	This graphic represents a return message from a call to a procedure. When you create a synchronous message, a return message is created by default. You can change this default in the Preferences window.
Lost and found		A line with an open arrowhead and that contains a dot at either end.	This graphic represents a lost or found message. A lost message contains a dot at the end of the arrowhead to indicate the destination is unknown. A dot at the source of the message indicates a found message with an unknown sender.

A message represents either an operation call or the sending and receiving of a signal. When the message represents an operation, the operation name identifies the message. The arguments of the message are passed to the target source. The return message contains the arguments of the resulting operation call. When a message represents a signal, the arguments of the message are the signal itself. If the message is a synchronous call, a return message occurs from the called lifeline to the calling lifeline before the calling lifeline can proceed.

You can identify messages by using a name or an operation signature. A name identifies only the name of the message that is not associated with an operation. When an operation is associated with a message, the operation name replaces the name. An operation signature is displayed to identify the name of the operation. You can use signatures in diagrams during the design phase to provide details to the developers who code the design.

As the following figure illustrates, messages are displayed as a line with an arrow that points in the direction in which the message is sent; that is, from the sending message end to the receiving message end. The following example shows how messages are displayed in a sequence diagram that represents a banking scenario in which a bank customer applies for a loan by following this process.

- A customer gives an application for the loan to a bank teller.
- The bank teller sends the application to the bank manager to processed and waits for the manager to finish.
- The bank manager starts the credit check program, enters the data, and waits for the credit agency to send the results.
- The bank manager receives a response from the credit agency and sends a message to the bank teller that states the decision.
- The bank teller sends a message to the customer that states whether the loan was approved.
- The bank manager closes the credit agency program and the customer completes the transaction.



Related concepts:

Message pathways in communication diagrams

Related tasks:

Creating and editing messages on lifelines in sequence diagrams
Assigning operations to messages in sequence diagrams
Binding operations and parameters to variables in sequence diagrams
Setting preferences for sequence and communication diagrams
Showing message signatures
Creating messages in communication diagrams
Creating message pathways in communication diagrams
Creating interaction uses in sequence diagrams
Creating interaction uses from existing elements
Customizing the default settings for messages

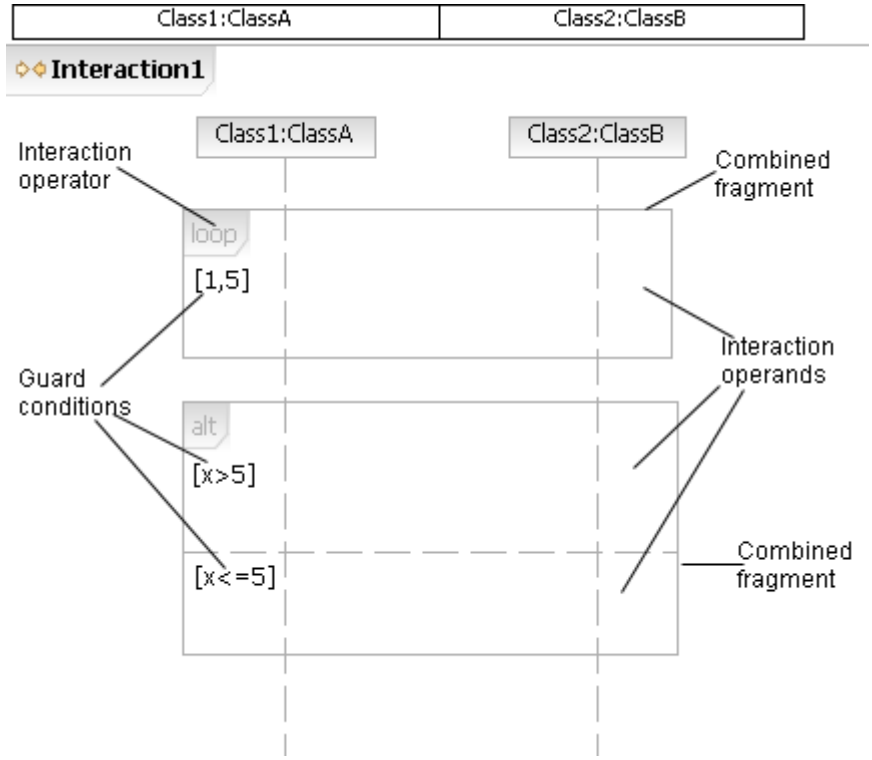
Combined fragments in sequence diagrams

In sequence diagrams, combined fragments are logical groupings, represented by a rectangle, which contain the conditional structures that affect the flow of messages. A combined fragment contains interaction operands and is defined by the interaction operator.

The type of combined fragment is determined by the interaction operator. You can use combined fragments to describe several control and logic structures in a compact and concise manner. The interaction operator identifies the type of logic or conditional statement that defines the behavior of the combined fragment.

A combined fragment can also contain nested combined fragments or interaction uses containing additional conditional structures that represent more complex structures that affect the flow of messages.

As the following figure illustrates, a combined fragment is displayed as a frame that covers lifelines and that contains interaction operands. A combined fragment separates the contained interaction operands with a dashed horizontal line between each operand.



Interaction operators

An interaction operator defines the semantics of a combined fragment and determines how to use the interaction operands in the combined fragment. The operator defines the type of logical conditions to apply to the operands. For example, a combined fragment with the alternative (alt) interaction operator acts like an if-then-else statement. In the previous figure, a loop and alternative (alt) interaction operator define the two combined fragments.

Interaction operands

In sequence diagrams, an interaction operand is a container that groups the interaction fragments and messages that run if the guard condition is met. If there is no guard condition, the block always runs.

Each interaction operand is a fragment of an interaction and covers the lifelines in the combined fragment. An interaction operand contains a guard condition, which contains an interaction constraint. The interaction operand runs only if the guard condition tests `true`.

Depending on the type of interaction operator, you can have one or more interaction operands in a combined fragment. Each operand must have a guard condition.

In the previous figure, the alternative (alt) combined fragment contains two interaction operands; one with a guard condition

x>5 and the other interaction operand with the guard condition x<=5. If x=6, the x<=5 guard condition and corresponding operand and messages run. The x>5 guard condition and corresponding operand and contained messages do not run.

Guard conditions

In sequence diagrams, a guard condition contains an interaction constraint, which is a Boolean conditional expression or, in the case of a loop, an expression that designates the minimum and maximum number of times the loop runs and, optionally, the incremental value.

A guard condition is a semantic condition or restriction that is set in square brackets in an interaction operand in a combined fragment. When you create a combined fragment, a guard condition is created automatically. You can also manually create a guard condition in an interaction operand that does not have an existing guard condition.

A guard condition appears at the start of the interaction and contains all the information that is required to make the decision about whether to run the interaction operand. If the guard condition tests `true`, the interaction fragment runs.

As the following figure illustrates, an interaction operand is displayed as a rectangle in a combined fragment. The rectangle can contain messages that run if the guard condition is `true`. The code on the right is what pseudocode would like for the alternative (alt) combined fragment.

	`// This section is a combined fragment`
	
	`//if else represents the interaction operator`
	`//if-else is the Alternative(alt) combined fragment`
	`if ( value is greater than 5)`
	`// the condition in the () is the guard condition`
	`// The code in the {} is the interaction operand`
	`// It runs if the guard condition is true`
	`{ A send a message to B; B sends a return message to A; } else if(x is less than or equal to 5) { B sends a message to A; }`

Related tasks:

Managing combined fragments in sequence diagrams

Creating combined fragments in sequence diagrams

Managing interaction operands in sequence diagrams

Adding guard conditions to sequence diagrams

Related reference:

Interaction operators in sequence diagrams

Interaction uses in sequence diagrams

In sequence diagrams, interaction uses enable you to reference other existing interactions. You can construct a complete and complex sequence from smaller simpler interactions.

An interaction use is represented by a frame, similar to a combined fragment. The "ref" tag is placed inside the frame and the name of the interaction being referenced is placed inside the frame's content area, along with parameters and return types.

In the Properties view, on the Arguments page, you can specify how arguments are acquired and other properties that the interaction use might require.

Example

In the following example, an interaction use references an interaction called Interaction2. The example references Interaction2, which is an interaction between the bank teller and the bank manager. Interaction2 shows the messages from the bank teller and the bank manager while the customer completes the loan application information in Interaction1.

Referencing interaction
Referenced interaction

Related tasks:

Creating interaction uses in sequence diagrams

Setting preferences for sequence and communication diagrams

You can set several default preferences for sequence and communication diagrams. For example, you can specify the appearance of lifelines, messages, and execution occurrences that are displayed in sequence and communication diagrams when you create them.

About this task

You can specify the default settings for showing or hiding message numbering, message and operation signatures, or names and type names of lifelines.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** > **UML Diagrams**, and click **Sequence and Communication**.
3. On the **Sequence and Communication Diagrams** page, specify the appropriate preferences and click **OK**.

Related concepts:

Messages in UML diagrams

Related tasks:

Binding operations and parameters to variables in sequence diagrams

Creating sequence diagrams from existing diagram elements

You can create sequence diagrams in the typical way that you create any UML diagram, or you can create them from existing elements in your project. You can create a sequence diagram either within a model, which also creates a collaboration and interaction, or as a standalone diagram, which becomes a DNX file.

Procedure

1. In the Project Explorer view, right-click one or more elements and complete one of the following steps:
   - To add the sequence diagram to a model, click **Visualize** > **Add to New Diagram in Model File** > **Sequence Diagram** .
   - To create a standalone diagram, click **Visualize** > **Add to New Diagram File** > **Sequence Diagram**.
2. In the New Sequence Diagram wizard, specify a parent folder and file name.
3. Click **Finish**.

Results

Tip: You can also populate an existing sequence diagram with source elements. In the Project Explorer view, right-click one or more classes, interfaces, and data types; then click **Visualize** > **Add to Current Diagram**.

Related tasks:

Creating UML diagrams in projects

Creating UML diagrams in models

Populating sequence diagrams

You can create sequence diagrams and populate them with elements to understand and develop behaviors and interactions between objects in systems. You can add lifelines to represent objects or actors, messages to represent communication between the objects, and combined fragments to represent the control structures.

- **Creating lifelines in UML diagrams**

  In UML diagrams, such as sequence and communication diagrams, you can create lifelines to represent objects in a system.

- **Creating messages in UML diagrams**

  In sequence diagrams, you can add create, destroy, synchronous and asynchronous messages to lifelines to represent communication between objects. You can also specify the signatures of messages.

- **Creating combined fragments in sequence diagrams**

  In sequence diagrams, you can create combined fragments to visually represent control structures, such as a for-loop or if-else statements, in interactions. Combined fragments can contain interaction operands, guard conditions and other combined fragments. Combined fragments contain procedural logic that control the messages inside the combined fragment.

- **Managing interaction operands in sequence diagrams**

  In sequence diagrams, an interaction operand is a container that groups interaction fragments, such as messages, that runs if the guard condition is `true`. An interaction operand is created automatically when you create a combined fragment. You can add new interaction operands to combined fragments that allow multiple operands, such as alternative, parallel, strict, or weak combined fragments. You can also remove and reposition interaction operands in a combined fragment to reflect changes in your system.

- **Adding guard conditions to sequence diagrams**

  In sequence diagrams, a guard condition contains an interaction constraint. An interaction constraint is a condition or restriction. A guard condition is created automatically when you create a combined fragment. You can also manually add a guard condition to an interaction operand that does not have an existing guard condition.

- **Creating interaction uses in sequence diagrams**

  In sequence diagrams, you can create an interaction use by specifying a new unspecified occurrence or by creating a new interaction.

- **Binding operations and parameters to variables in sequence diagrams**

  You can bind operations and parameters to variables to reuse an operation or parameter or to change operation and parameter names withoutchanging each instance of the parameter or operation.

- **Adding state invariants to sequence diagrams**

  State invariants are runtime constraints, such as values of variables, attributes, and states, that are placed on the objects or capsules of an interaction. You can add state invariants to sequence diagrams to specify values or states.

- **Creating gates in sequence diagrams**

  You can add formal and actual gates to sequence diagrams. Gates are messages that have parameters that pass information. Formal gates can end or start from the diagram edge and extend to a lifeline. An actual gate can start or end from an interaction use.

- **Creating interaction uses from existing elements**

  You can create an interaction use from existing elements to simplify and improve the readability of a sequence diagram.

Creating lifelines in UML diagrams

In UML diagrams, such as sequence and communication diagrams, you can create lifelines to represent objects in a system.

Before you begin

You must have a sequence or communication diagram open.

Procedure

1. In the Palette, click **Lifeline**.
2. In the editor, click in an interaction frame where you want to add the new lifeline.
3. In the dialog box, click a type of lifeline:

Lifeline	Details
Unspecified	This option creates a lifeline without a specified type.
Create Class	This option creates a lifeline that represents a class. You type a name for the new lifeline and then for the object.
Create Actor	This option creates a lifeline that represents an actor. You type a name for the new lifeline and then for the object.
Create Component	This option creates a lifeline that represents a component. You type a name for the new lifeline and then for the object.
Select Existing Element	This option enables you to select an existing element in the model.

Results

When you create a lifeline in a sequence diagram, the lifeline is aligned vertically with other lifelines in the interaction, and extends downward. You can resize a lifeline by clicking it and dragging the anchors vertically or horizontally.**Tip:** You can also assign an existing element type to a lifeline: In the navigation view, click a class, interface, or data type and drag it onto a lifeline in the interaction frame.

Related concepts:

Lifelines in UML diagrams

Sequence diagrams

Creating messages in UML diagrams

In sequence diagrams, you can add create, destroy, synchronous and asynchronous messages to lifelines to represent communication between objects. You can also specify the signatures of messages.

Procedure

1. In the Palette, click a message type.
2. In the diagram editor, in an interaction frame, click a source lifeline and drag the cursor to a target lifeline.
3. Type a name for the message.

Results

If you create a synchronous message, a return message is created automatically. If you start the message at the diagram frame with no source, a found message is created. If you create a message without a target, a lost message is created. A lost or found message is indicated by a dot at the source or target of the message.

What to do next

Note: To delay an asynchronous message, drag the asynchronous message first horizontally and then vertically across existing messages to the target lifeline where you want to terminate the message. You can also create an asynchronous message without a behavior execution specification.

Tip: To assign an operation to a message in an existing diagram, in the navigation view, click an operation and drag it onto a message in the interaction frame. The operation must belong to the target instance of the object or be visible to and in scope of the operation in an object.

Related concepts:

Sequence diagrams

Lifelines in UML diagrams

Creating combined fragments in sequence diagrams

In sequence diagrams, you can create combined fragments to visually represent control structures, such as a for-loop or if-else statements, in interactions. Combined fragments can contain interaction operands, guard conditions and other combined fragments. Combined fragments contain procedural logic that control the messages inside the combined fragment.

Before you begin

You must have a sequence diagram open that contains lifelines.

Procedure

1. In the Palette, click a type of combined fragment.
2. In the editor, click in the interaction frame.
3. Drag the combined fragment across the elements that you want it to include.

Results

The covered messages and interaction fragments are displayed in the first interaction operand of the combined fragment. When you create a combined fragment, the interaction operand can be collapsed to hide the operand and its associated messages and interaction fragments to minimize the size of the combined fragment in the interaction frame.
You can create nested combined fragments by adding one combined fragment inside another. To nest a combined fragment, the source combined fragment cannot contain a create or a destroy message.

Related concepts:

Combined fragments in sequence diagrams
Sequence diagrams
Lifelines in UML diagrams

Managing interaction operands in sequence diagrams

In sequence diagrams, an interaction operand is a container that groups interaction fragments, such as messages, that runs if the guard condition is `true`. An interaction operand is created automatically when you create a combined fragment. You can add new interaction operands to combined fragments that allow multiple operands, such as alternative, parallel, strict, or weak combined fragments. You can also remove and reposition interaction operands in a combined fragment to reflect changes in your system.

Before you begin

You must have a sequence diagram open that contains lifelines and combined fragments.

About this task

To add, remove, or reposition interaction operands in a combined fragment, complete one of the following steps:**Note:** You can not remove the last operand of a combined fragment.

Procedure

- To add a new interaction operand, right-click a combined fragment; then click **Add Interaction Operand**.
- To remove an interaction operand, right-click one; then click **Interaction Operand** > **Remove Operand**.
- To reposition an interaction operand, right-click one; then click **Interaction Operand** and click the appropriate menu item.

What to do next

Tip: To create an interaction operand above or below an interaction operand, select an operand and click **Interaction Operand** > **Add before** or **Add after**.

Related concepts:

Interaction operands in sequence diagrams

Combined fragments in sequence diagrams

Related reference:

Interaction operators in sequence diagrams

Adding guard conditions to sequence diagrams

In sequence diagrams, a guard condition contains an interaction constraint. An interaction constraint is a condition or restriction. A guard condition is created automatically when you create a combined fragment. You can also manually add a guard condition to an interaction operand that does not have an existing guard condition.

Before you begin

You must have a sequence diagram open that contains at least one combined fragment with multiple lifelines.

Procedure

1. In the diagram editor, right-click an interaction operand that has no existing guard condition; then click **Interaction Operand** > **Add a guard condition**.
2. Click in the field, type the guard condition, and press Enter.

What to do next

Note: In a loop combined fragment, you can edit the guard condition using the textual syntax `[ '(' <minint> [, <maxint> ] ')' ]`, where `minit` and `maxint` are non-negative integers, and state the minimum and maximum number of times that a loop can occur.

Related concepts:

Interaction operands in sequence diagrams

Guard conditions in sequence diagrams

Combined fragments in sequence diagrams

Related reference:

Interaction operators in sequence diagrams

Creating interaction uses in sequence diagrams

In sequence diagrams, you can create an interaction use by specifying a new unspecified occurrence or by creating a new interaction.

Procedure

1. In the Palette, click **Interaction Use**.
2. Click inside the interaction frame and, in the window that opens, complete one of the following steps:
   - To create an interaction use with an unspecified interaction that you will create later, click **Unspecified**.
   - To create an interaction use with a new interaction that you want to create now, click **Create New Interaction**.
   - To create an interaction use from an existing interaction, click **Create from Existing Interaction**.
3. In the Add Covered Lifelines window, click the lifelines to cover and click **OK**.
4. Type a name for the interaction use and press Enter.

Results

If you select **Create New Interaction**, a new interaction is created in the Project Explorer view and a new sequence diagram opens with the selected covered lifelines.**Note:** You can create a nested interaction use in a combined fragment or move an interaction use from a combined fragment by clicking the interaction use and dragging the combined fragment to the new location.

Related concepts:

Messages in UML diagrams

Interaction uses in sequence diagrams

Related tasks:

Creating interaction uses from existing elements

Binding operations and parameters to variables in sequence diagrams

You can bind operations and parameters to variables to reuse an operation or parameter or to change operation and parameter names withoutchanging each instance of the parameter or operation.

Before you begin

You must have the following preferences set:

- You must select the sequence diagram preference to always show signatures.
- You must clear the preference to show operation signatures instead of message signatures.
- You must have a sequence diagram open with at least two lifelines.

Procedure

1. In the Palette, click a type of message.
2. In the interaction frame, click a source lifeline and drag the cursor to a target lifeline.
3. Type a name for the message operation.
4. In the Project Explorer view, right-click the newly created message; then click **Add UML** > **Parameter**.
5. Type a name for the parameter.
6. In the Project Explorer view, right-click the newly created message; then click **Add UML** > **Return Parameter**.
7. Type a name for the return parameter.
8. Complete the following steps for both the message and its return message:
   A. In the interaction frame, right-click the message that you created in step 2; then click **UML Properties**.
   B. On the Argument page, select **Opaque Expression** as the argument type.
9. For each opaque expression, complete the following steps:
   A. On the Argument page, in the **body** field, type a name for the value.

What to do next

In the message signature, the synchronous message parameter is assigned the variable argument that you created. In the return message, the second variable argument that you created is assigned to the operation.

Related concepts:

Messages in UML diagrams

Related tasks:

Setting preferences for sequence and communication diagrams

Adding state invariants to sequence diagrams

State invariants are runtime constraints, such as values of variables, attributes, and states, that are placed on the objects or capsules of an interaction. You can add state invariants to sequence diagrams to specify values or states.

Procedure

1. In the Palette, click **State Invariant**.
2. In the interaction frame, click the lifeline to cover.
3. In the dialog box that opens, verify the selected lifelines and click **OK**.
4. Type a name for the state invariant and click **OK**.**Tip:** To change constraints to state invariants, in the diagram editor, right-click a constraint; then click **Refactor** > **Refactor into State Invariant**.

Creating gates in sequence diagrams

You can add formal and actual gates to sequence diagrams. Gates are messages that have parameters that pass information. Formal gates can end or start from the diagram edge and extend to a lifeline. An actual gate can start or end from an interaction use.

Before you begin

You must have a lifeline or an interaction use in a sequence diagram.

Procedure

1. In the Palette, click a message type.
2. In the interaction frame, click the edge of the frame and drag the cursor to a lifeline or select the lifeline as the source and drag the cursor to the interaction frame. An actual gate is created if the source or target is an interaction use.
3. Type a name for the message. A message is created with one end that contains an arrow in a square around it, which represents a gate.

Organizing sequence diagrams

You can organize sequence diagrams to better represent a system. You can reorder messages, delay asynchronous messages, as well as manage combined fragments and lifelines.

- **Reordering lifelines and messages in sequence diagrams**

  In sequence diagrams, you can reorder lifelines or move a message across other messages and interaction fragments on the same lifeline or on another lifeline to reorder the sequence in which the messages occur. After you reorder a message, the message number automatically indicates the updated position on the lifeline.

- **Adding and removing covered lifelines in sequence diagrams**

  In sequence diagrams, you can add and remove lifelines from combined fragments or interaction uses.

- **Creating interaction uses from existing elements**

  You can create an interaction use from existing elements to simplify and improve the readability of a sequence diagram.

Reordering lifelines and messages in sequence diagrams

In sequence diagrams, you can reorder lifelines or move a message across other messages and interaction fragments on the same lifeline or on another lifeline to reorder the sequence in which the messages occur. After you reorder a message, the message number automatically indicates the updated position on the lifeline.

Before you begin

Restriction:

- You cannot directly reorder a skewed asynchronous message. For the message renumbering to occur, you must move the other messages around it to change the skewed order.
- You cannot change the order between a send message and a return message.

You must have a sequence diagram open that contains at least two lifelines with messages.

Procedure

1. Right-click a message or lifeline; then click **Select Message Set/Reorder**.
2. Drag the message or lifeline to move it to the new location. The grid guideline indicates where the message will be located in the diagram.

What to do next

Tip: You can reorder lifelines and messages by holding the **Alt** key and dragging the element to the new location.

Note: To connect a message to a different lifeline, click a message handle and drag it to another lifeline.

Adding and removing covered lifelines in sequence diagrams

In sequence diagrams, you can add and remove lifelines from combined fragments or interaction uses.

Before you begin

You must have a sequence diagram open that contains lifelines and combined fragments or interaction uses.

Procedure

1. In the diagram editor, right-click a combined fragment or interaction use; then click **Covered Lifelines** and complete one of the following steps:
   - To add lifelines, click **Add Covered Lifeline(s)**.
   - To remove lifelines, click **Remove Covered Lifeline(s)**.
2. In the window, select the lifelines and click **OK**.

What to do next

Note: When you remove a covered lifeline from a combined fragment, you also delete all the messages and interaction fragments that the combined fragment contains and that the lifeline covers.

Related concepts:

Combined fragments in sequence diagrams

Lifelines in UML diagrams

Creating interaction uses from existing elements

You can create an interaction use from existing elements to simplify and improve the readability of a sequence diagram.

Before you begin

A sequence diagram that contains at least one element must be open.

Procedure

1. In the diagram editor, select the elements to include in the interaction use. Selecting a message also creates lifelines in the interaction use.
2. Right-click in the diagram editor, then click **Refactor** > **Refactoring Into New Interaction**.
3. Type a name for the interaction use.

Results

An interaction with the name you specified is created in the diagram. Double-click the interaction use to open a new diagram which contains the elements in the interaction use.

Related concepts:

Messages in UML diagrams

Related tasks:

Creating interaction uses in sequence diagrams

Deleting messages from sequence diagrams

You can delete messages from your project that you no longer need. When you delete a message, you have different results depending on the type of message that you delete.

Before you begin

You must have a sequence diagram open.

About this task

To delete a message, in the diagram editor, right-click a message; then click **Delete from Model**.

What to do next

When you delete a message, the result differs depending on the type of message.

- If you delete a send message from a synchronous message, the corresponding execution specification and return message are also deleted.
- If you delete a return message from a synchronous message, the corresponding send message and execution specification remain intact.
- If you delete an asynchronous message, the send message and corresponding execution specifications are deleted.
- If you delete a create message, the lifeline extends vertically to the top of the interaction frame.
- If you delete a destroy message, the lifeline extends vertically to the bottom of the interaction frame.

Related concepts:

Lifelines in UML diagrams

Messages in UML diagrams

Deleting lifelines from UML diagrams

You can delete lifelines from UML diagrams, such as sequence or communication diagrams, to remove an object from an interaction. When you delete a lifeline, all send and receive messages that originate from the deleted lifeline are also deleted. When you delete a lifeline from a sequence diagram, all interaction fragments that cover the lifeline are deleted, but the combined fragments and interaction uses that cover other lifelines are not.

Before you begin

You must have a UML diagram open that contains lifelines.

About this task

To delete a lifeline, in the diagram editor, right-click a lifeline; then click **Delete from Model**. All send and receive messages that originate from the deleted lifeline and all interaction fragments that cover the lifeline are deleted. The combined fragments and interaction uses that cover other lifelines are not deleted.

Related concepts:

Lifelines in UML diagrams

Interaction operators in sequence diagrams

In sequence diagrams, an interaction operator defines the semantics of a combined fragment and determines how to use the interaction operands in the combined fragment.

The following table lists the most commonly used interaction operators:

Interaction operator	Description
Alternative (alt)	An alternative interaction operator represents the logic equivalent of an if-then-else statement. Only one of the offered alternatives runs on any pass through the interaction. However, as for any operand, the selected operand in the alternative structure runs only if the guard condition tests `true`. If there is no guard, the operand always runs when it is selected. The else clause of the alternative combined fragment runs when no other option is selected.
Option (opt)	An option interaction operator represents the logic equivalent of an if statement. To run,, the guard condition must be satisfied. If the guard condition fails, the behavior is ignored. The graphic representation of an option combined fragment looks like an alternative that offers only one alternative.
Loop	A loop interaction operator indicates that the interaction fragment runs repeatedly. The number of times the fragment runs is determined by the `minint` and `maxint` parameters of the operator. The syntax of the loop operator is loop (`minint`, `maxint`) where `maxint` can also be infinity (*). After the minimum number of iterations is satisfied, a Boolean expression is tested on each pass. When the Boolean expression tests `false`, the loop ends.
Parallel (par)	A parallel interaction operation indicates that the interaction fragments run concurrently with each other.
Critical Region (critical)	A critical region operator indicates that the fragment can have only one thread that runs it at any time. The fragment must complete before another thread can run. For example, if an operation `d()` is within the critical region and is invoked, no other operations can be invoked until the completion of `d()`.
Negative (neg)	A negative interaction operator shows invalid interactions that should not be allowed to happen.
Break	The break interaction operator is similar to the break mechanism in other programming languages. When the guard condition is `true` , the current interaction run is abandoned and the clause in the break interaction operand runs.
Strict sequencing (strict)	The strict interaction operator explicitly defines the order of execution of the interaction fragments. The strict operator forces the completion of the interaction before running nested or additional interactions.
Weak sequencing (seq)	The weak sequencing interaction operator adds order to the interactions in the fragment based on their placement. The ordering is based on the UML 2.0 specification. Weak sequencing allows partial parallel execution, but controls the order of events on the same lifeline from different interactions.
Ignore	The ignore interaction operator indicates messages that the interaction fragment should not respond to. The ignore operator is usually paired with the consider operator.

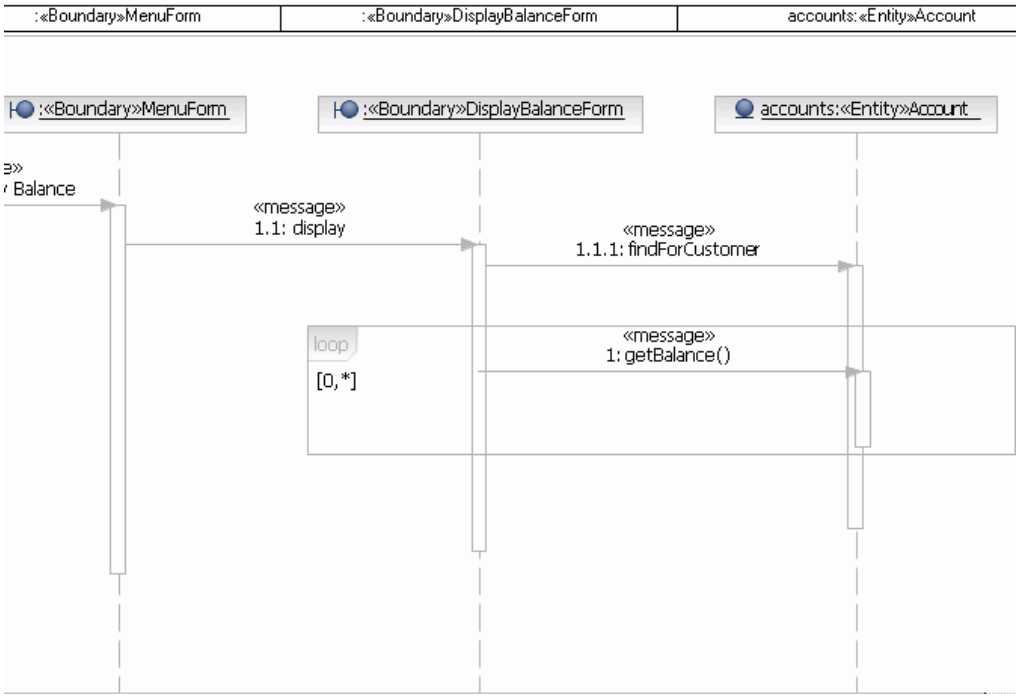
Consider	The consider interaction operator indicates messages that the interaction fragment should respond to. The consider operator is usually paired with the ignore operator.
Assert	The assert interaction operator indicates that an interaction operand is the next sequence in the interaction fragment.

Example of a sequence diagram

The following example shows a sequence diagram taken from the PiggyBank sample that is available in the Samples Gallery, which describes the DisplayBalance use case from an online banking scenario.

The sequence diagram shows the sequence of actions that occur when customers view the balance of their bank accounts. There are four participants in the interaction between the customers and the online banking system: Customer, MenuForm, DisplayBalanceForm, and Account. Each participant is represented by a lifeline in the interaction frame.

Each action that occurs between the lifelines is represented by a message and the messages are in a specific, numbered sequence.



Related concepts:

Sequence diagrams

Communication diagrams

Related tasks:

Modeling the interactions between objects in UML

Creating sequence diagrams

Managing UML diagrams

A Unified Modeling Language (UML) diagram provides a visual representation of a specific aspect or behavior of a system.

About this task

UML diagrams illustrate the quantifiable aspects of a system that can be described visually, such as relationships, behavior, structure, and functionality. The visual representation of a system that UML diagrams provide can offer both low-level and high-level insight into the concept and design of an application.

In IBM® Rational® Software Architect, you work with and manage all UML diagrams in a similar manner. For example, you state default settings for UML diagrams by specifying preferences. To create diagrams in the diagram editor, and to add elements and relationships to them, you use the **Create** tab of the Palette. To explore a model from a diagram, you use the **Explore** tab of the Palette.

The following topics provide the general tasks for managing UML diagrams, from setting preferences and creating diagrams to editing, printing, and saving them.

- **Opening UML diagrams**
  When you open a UML diagram, it is displayed in the diagram editor. You can have multiple diagrams open at the same time and use the tabs at the top of the editor to navigate between them.
- **Editing UML diagrams**
  You can edit, organize, and resize a UML diagram to optimize the diagram for viewing, printing, or for presenting. You can also add a stereotype and specify its style to give further meaning to your diagram.
- **Showing related elements in diagrams**
  You can populate a diagram by showing existing source elements that are related by specific relationships to the selected source elements.
- **Creating URLs to elements in UML models**
  A URL is a stereotyped comment object that you can use to navigate to a specified URL, file system element, or an element in the workspace. A URL has a type property as a further specialization of the URL. The traceability feature treats URLs as specifications.
- **Linking UML diagrams**
  You can create links between related UML diagrams in the same project or in different projects, so that you can navigate from an open diagram to other linked diagrams.
- **Saving UML diagrams as images**
  You can save UML diagrams as images so that you can use them in other media such as documents, presentations, or HTML pages. You can save diagrams in GIF, BMP, JPEG, JPG, or SVG format.
- **Managing the printing of UML diagrams**
  UML diagrams can become quite large, spanning several pages. You can control how your diagrams are printed by manipulating the page breaks. You can then preview and print UML diagrams.
- **Deleting UML diagram elements**
  You can delete UML elements that you no longer require from either a specific diagram or from an entire project.
- **Deleting diagrams from UML models**
  You can delete from UML models the diagrams that you no longer require.
- **Setting workspace preferences for UML diagrams**
  You can change the default display settings for the Palette and other diagram views. The default display settings provide simplified views that are appropriate for each type of diagram that you create. You can add items to or remove items from the user interface. You can also change the appearance of the user interface.
- **Creating and populating UML diagrams**
  You can create UML diagrams to visually represent and develop the elements of a system or application. You can create

diagrams and add elements and relationships to them. You can also add supplemental items to the diagram to provide additional information to the diagram.

- **Deleting diagrams from projects**

  You can delete from projects the diagrams that you no longer require.

UML diagrams

A Unified Modeling Language (UML) diagram provides a visual representation of an aspect of a system.

UML diagrams illustrate the quantifiable aspects of a system that can be described visually, such as relationships, behavior, structure, and functionality. For example, a class diagram describes the structure of the system or the details of an implementation, while a sequence diagram shows the interaction between objects over time.

In a UML diagram, the diagram elements visually represent the classifiers in a system or application. These classifiers are the diagrammatic representation of a source element. UML diagrams provide views of source elements; however, diagram elements do not have semantic value.

UML diagrams can help system architects and developers understand, collaborate on, and develop an application. High-level architects and managers can use UML diagrams to visualize an entire system or project and separate applications into smaller components for development.

System developers can use UML diagrams to specify, visualize, and document applications, which can increase efficiency and improve their application design. UML diagrams can also help identify patterns of behavior, which can provide opportunities for reuse and streamlined applications.

The visual representation of a system that UML diagrams provide can offer both low-level and high-level insight into the concept and design of an application.

Setting workspace preferences for UML diagrams

You can change the default display settings for the Palette and other diagram views. The default display settings provide simplified views that are appropriate for each type of diagram that you create. You can add items to or remove items from the user interface. You can also change the appearance of the user interface.

- **Setting preferences for UML diagrams**

  You can set several default preferences for UML diagrams. For example, you can specify the appearance style, font, and colors that are displayed in UML diagrams when you create them. You might want to change the font and the colors to add visual interest or to convey conceptual groupings between elements in a new diagram.

- **Changing the appearance of the Palette for UML diagrams**

  When you create UML diagrams, you can change the visual appearance of the Palette, which you use to add items to diagrams. For example, you can change the layout of the Palette to display items in columns, as a list, as icons only, or as detailed items.

- **Changing capabilities for UML diagrams**

  You can customize capabilities to show only the items that you want to display in UML diagrams. This function can help simplify the user interface. You can change the diagram capabilities to include more or fewer items. You can add more user interface icons to the diagram when you create new diagrams.

Setting preferences for UML diagrams

You can set several default preferences for UML diagrams. For example, you can specify the appearance style, font, and colors that are displayed in UML diagrams when you create them. You might want to change the font and the colors to add visual interest or to convey conceptual groupings between elements in a new diagram.

About this task

You can specify the default visibility style for attributes and operations and the default operation signature format for new diagrams. You might also want to change the default settings for showing or hiding attributes and operations, operation signatures, or parent names of classifiers. After you specify preferences, the settings apply to any new diagrams or elements that you create, but do not affect existing diagrams or elements.

Procedure

1. Click **Window** > **Preferences**.
2. In the Preferences window, expand **Modeling** > **UML Diagrams**.
3. On the Diagrams page, and the pages beneath it, specify the appropriate preferences and click **OK**.

Related tasks:

Resizing elements in UML diagrams
Changing zoom levels in UML diagrams
Making elements the same size in UML diagrams
Changing the appearance of elements in UML diagrams
Aligning elements in UML diagrams
Opening UML diagrams
Selecting elements in UML diagrams
Specifying aliases for UML diagram elements
Showing and hiding aliases in UML diagrams
Changing the appearance of the Palette for UML diagrams
Specifying parent name styles in UML diagrams

Changing the appearance of the Palette for UML diagrams

When you create UML diagrams, you can change the visual appearance of the Palette, which you use to add items to diagrams. For example, you can change the layout of the Palette to display items in columns, as a list, as icons only, or as detailed items.

Procedure

1. Right-click in the Palette.
2. Complete one of the following steps:
   - To change how items are displayed, click **Layout** and select the display type.
   - To change which items are displayed, click **Customize**, select the items that you want to suppress and select **Hide**.
   - To change the appearance and behavior of the Palette, click **Settings** and specify the options.

Related tasks:

Specifying aliases for UML diagram elements
Showing and hiding aliases in UML diagrams
Setting preferences for UML diagrams
Changing the appearance of elements in UML diagrams

Related reference:

Supplemental information in UML diagrams

Changing capabilities for UML diagrams

You can customize capabilities to show only the items that you want to display in UML diagrams. This function can help simplify the user interface. You can change the diagram capabilities to include more or fewer items. You can add more user interface icons to the diagram when you create new diagrams.

Procedure

1. In the diagram editor, click inside a diagram.
2. In the Properties view, on the Diagram Capabilities page, select the capabilities to display.

Related reference:

Supplemental information in UML diagrams

Creating and populating UML diagrams

You can create UML diagrams to visually represent and develop the elements of a system or application. You can create diagrams and add elements and relationships to them. You can also add supplemental items to the diagram to provide additional information to the diagram.

Before you begin

Rational® Software Architect supports UML 2.2. For detailed information about UML 2.2, see the specification that is available on the Object Management Group (OMG) website.

- **Creating UML diagrams in models**
  You can create UML diagrams to represent different views of a system or application. when you model a system using UML, you create the diagrams within a UML model.
- **Adding elements to UML diagrams**
  You can add several items to UML diagrams to populate them with meaningful information. For example, you can add diagram elements to represent objects, and add connectors to represent the relationships between elements. You can also add several additional items to provide supplemental information about the diagrams.
- **Linking notes to UML diagrams**
  You can link a note in one diagram to a related diagram. This link is useful if you want to provide more information about a particular diagram element.
- **Creating UML diagrams in projects**
  You can create new diagrams in a project to represent different views of a system or application.

Related concepts:

UML diagram elements

Related reference:

Supplemental information in UML diagrams

Related information:

Tutorial: Understanding the UML model hierarchy

Tutorial: Introduction to the Modeling perspective

Diagramming techniques and diagramming assistance

Diagramming assistance is available through context sensitive help, diagram assist or content assist. The context sensitive assistance help ease the work and helps promote proper diagramming techniques.

IBM® Rational® Software Architect provide context-sensitive tools, such as diagram assist, action toolbars, connector handles, and content assist, that guide you as you create UML diagrams.

Diagram assist

The diagram assist features help you to create complex UML diagrams in less time, using fewer keystrokes and mouse movements. As you create a UML diagram, the diagram assist features prompt you with visual cues that form a simplified, but very powerful context-sensitive user interface. This simplified user interface can increase the effective size of your screen by allowing you to remove elements of the traditional user interface, such as the file menu or the palette.

Action toolbars

Action toolbars appear in the diagram editor when you hover over a spot in the diagram. The action toolbar is a hovering toolbar that provides access to only the set of elements that are specific to the context of the current diagram. For example, when you hover over a class diagram, an action tool bar appears that allows you to add a class, an enumeration, an artifact, a signal, a stereotyped class, an interface, or a package.

Connector handles

Similar to action bars, connector handles appear when you hover over a UML element in a diagram. Connector handles appear as arrowheads that you can drag onto another element in the diagram to create a relationship. You can use connector handles to model the relationship between two UML elements, or to create a relationship to a new UML element.

Content assist

Content assist is a built-in Eclipse modeling aid that provides suggestions, displays valid code listings, and attempts to complete your code as you type it. You can use content assist to edit property types and method signatures in a class diagram. When you type the name of a property, and then type a colon (:), you will see a list of all possible return value types. To select an item in the list, use the scroll bar to view the value types and double-click a return type and complete the signature.

Related information:

Create diagrams using IBM Rational UML Modeling tools

UML diagram elements

A diagram element is a textual and graphical element that represents a component of an application or a system in a Unified Modeling Language (UML) diagram.

The items that you add to populate UML diagrams are called diagram elements. Diagram elements, which can also be called shapes, graphically and textually represent the components in the application or real-world system that the diagram illustrates. Different types of UML diagrams contain different diagram elements to illustrate the objects or components that the diagram represents. For example, class diagrams, which show the static structure of a system, contain diagram elements that represent classes, whereas sequence diagrams, which show how objects interact with each other, contain diagram elements that illustrate objects and the messages that they send to and receive from each other.

You can change how diagram elements appear. For example, you can change the colors for lines, borders, backgrounds, fonts, and drop shadows. You can set default preferences for all the diagram elements that you create, or you can change the settings for individual elements in a diagram.

Compartments

A compartment is a divided section within a UML diagram element. A compartment organizes the items in a diagram element so that it is easy to differentiate between them. Individual compartments group similar items together. An element contains a name compartment, and can contain other compartments. For example, the UML representation of a class is a rectangle that is divided into three compartments. The top compartment shows the class name, the middle compartment lists the class attributes, and the bottom compartment lists the operations of the class.

You can specify whether to show or hide compartments and compartment titles within diagram elements.

Connectors

Connectors are specific elements in UML diagrams that represent the relationships between diagram elements. A connector appears as a line and can show all or part of the semantic information about the represented relationship.

You can change the properties for how connectors appear in a diagram. For example, you can change their color, set whether they should avoid obstacles or link elements along the shortest distance, or specify the smoothness of the lines. By default, connectors have an oblique line style which makes them appear as slanted or angled lines. Oblique lines can be easier to drag and more space-efficient in complex diagrams. You can also change this setting to a rectilinear style that shows horizontal and vertical lines. You can add bend points to connectors to make right angles in them to better organize a diagram.

Similar to diagram elements, you can set default preferences for all connectors that you add to diagrams, or for individual connectors in a specific diagram.

Related tasks:

Creating and populating UML diagrams

Related reference:

Supplemental information in UML diagrams

Related information:

Tutorial: Understanding the UML model hierarchy

Tutorial: Introduction to the Modeling perspective

Creating UML diagrams in projects

You can create new diagrams in a project to represent different views of a system or application.

Procedure

1. In the Project Explorer view, right-click a project; then click **Add Diagram** and click a diagram type.
2. Type a name for the diagram and press Enter.

Related reference:

Supplemental information in UML diagrams

Adding elements to UML diagrams

You can add several items to UML diagrams to populate them with meaningful information. For example, you can add diagram elements to represent objects, and add connectors to represent the relationships between elements. You can also add several additional items to provide supplemental information about the diagrams.

Procedure

1. Open a UML diagram.
2. In the Palette, on the **Create** tab, double-click an item. The item appears in the diagram.

Results

Tip: You can also right-click in the diagram editor to add elements.

Related reference:

Supplemental information in UML diagrams

Linking notes to UML diagrams

You can link a note in one diagram to a related diagram. This link is useful if you want to provide more information about a particular diagram element.

Procedure

1. Create a note in an open diagram so that it is displayed in the diagram editor.
2. In the Project Explorer view, click the target diagram and drag it into the note.

Results

Tip: To open the linked diagram, double-click the note.

Related tasks:

Documenting UML model elements

Related reference:

Supplemental information in UML diagrams

Supplemental information in UML diagrams

In UML diagrams, you can add text, notes, comments, images, geometric shapes, and URLs to provide additional information about the elements or the diagram.

The following table describes the elements that you can add to a UML diagram to provide more information about it.

Item	UML element	Description
Comments and comment attachments	Yes	Comments contain textual information and appear in both the Project Explorer view and the diagram editor. Comment attachments connect comments to diagram elements. You can attach comments to a diagram to provide review feedback or observations about a particular diagram element. You can apply documentation, URLs, and shortcut stereotypes to a comment.
Geometric shapes	No	You can add geometric shapes, such as triangles, to UML diagrams to add visual interest and provide additional information. These shapes do not have any UML semantics but can provide visual differentiation.
Notes and Note Attachments	No	A note contains textual information. You can use a note attachment to connect a note to one or many diagram elements.
Text	No	You can use text to add additional information to diagrams and diagram elements. Text is not attached to a particular element in the diagram.
URLs	Yes	A URL is a stereotyped comment. When you add a URL to your diagram, it appears in both the Project Explorer view and the diagram. In the diagram editor, you can use a comment attachment to connect a URL to another diagram element. You can add a URL or file path to point to supporting information about a diagram element. You can attach a single URL to multiple diagram elements. You can also add images to your diagrams by adding URLs that reference the image files.

Related concepts:

UML diagram elements

Related tasks:

Creating UML diagrams in models

Adding elements to UML diagrams

Changing the appearance of the Palette for UML diagrams

Creating and populating UML diagrams

Documenting UML model elements

Linking notes to UML diagrams

Creating UML diagrams in projects

Changing capabilities for UML diagrams

Related information:

Tutorial: Understanding the UML model hierarchy

Tutorial: Introduction to the Modeling perspective

Opening UML diagrams

When you open a UML diagram, it is displayed in the diagram editor. You can have multiple diagrams open at the same time and use the tabs at the top of the editor to navigate between them.

Procedure

To open a diagram, in the Project Explorer view, navigate to a diagram in either the Models folder or the Diagrams folder and double-click it.

What to do next

Tip: You can also open UML diagrams in the Model Editor view on the **Details** page.

Related tasks:

Aligning elements in UML diagrams
Setting preferences for UML diagrams
Selecting elements in UML diagrams
Arranging elements in UML diagrams
Resizing elements in UML diagrams

Editing UML diagrams

You can edit, organize, and resize a UML diagram to optimize the diagram for viewing, printing, or for presenting. You can also add a stereotype and specify its style to give further meaning to your diagram.

About this task

You can change the appearance of selected items to add visual interest to them. You can also organize a diagram and resize it to better present or print the diagram.

- **Selecting elements in UML diagrams**
  In UML diagrams, you can select elements in an open diagram so that you can perform actions on them. For example, after you select diagram elements, you can arrange, align, or resize them.
- **Grouping elements in UML diagrams**
  In UML diagrams, you can put elements into a group so you can move those units as a unit or apply changes to all of them at once.
- **Changing the appearance of elements in UML diagrams**
  In an open UML diagram, you can change the appearance of selected elements to add visual interest or convey conceptual groupings. For example, you can change the size and style of fonts or the colors of diagram elements.
- **Organizing UML diagrams**
  In UML diagrams, you can organize various elements to optimize the diagram for viewing, printing, or presenting.
- **Changing names in UML diagrams**
  You can rename UML diagrams to reflect changes in their contents or purpose. You can also define aliases for diagram elements. You can then choose to show or hide the aliases.
- **Resizing in UML diagrams**
  In UML diagrams, you can change the size of elements to improve the presentation of the diagram. You can resize diagram elements, make them the same size, or use the zoom feature to make more or less detail visible.
- **Specifying stereotype styles in UML diagrams**
  In UML diagrams, you can specify the stereotype style for diagram elements or for the attribute and operation compartments in diagram elements. A stereotype is a UML extension mechanism that broadens the UML vocabulary and gives more specific meaning to a diagram element.
- **Renaming UML diagrams**
  You can rename UML diagrams to reflect changes in their contents or purpose.

Selecting elements in UML diagrams

In UML diagrams, you can select elements in an open diagram so that you can perform actions on them. For example, after you select diagram elements, you can arrange, align, or resize them.

About this task

To select elements in a UML diagram, click **Diagram** > **Select All**; then complete one of the following steps:

- To select all diagram elements, click **All**.
- To select only shapes, click **All Shapes**.
- To select only connectors, click **All Connectors**.

Results

Tip: To select a group of diagram elements, in the Palette, click **Select**; then, in the diagram editor, click outside the diagram elements and drag the selection border around them.

Related tasks:

Aligning elements in UML diagrams
Setting preferences for UML diagrams
Opening UML diagrams
Arranging elements in UML diagrams
Resizing elements in UML diagrams
Organizing UML diagrams
Grouping elements in UML diagrams

Grouping elements in UML diagrams

In UML diagrams, you can put elements into a group so you can move those units as a unit or apply changes to all of them at once.

Procedure

1. Select the elements that you want to include in the group. To select elements, hold Ctrl or Shift while you click the elements, use the mouse to draw a rectangle around the elements, or see Selecting elements in UML diagrams for other methods of selecting many elements at once.
2. Right-click one of the elements and then click **Format** > **Group**.
3. To ungroup the elements, right-click one of the elements in the group and then click **Format** > **Ungroup**.

Results

Now you can move the elements around the diagram as a group. You can also apply formatting options to all of the elements in the group at once.

Related tasks:

Arranging elements in UML diagrams
Selecting elements in UML diagrams
Organizing UML diagrams

Changing the appearance of elements in UML diagrams

In an open UML diagram, you can change the appearance of selected elements to add visual interest or convey conceptual groupings. For example, you can change the size and style of fonts or the colors of diagram elements.

Procedure

1. In the diagram editor, select an element or group of elements.
2. Click **Diagram** and click the item whose appearance you want to change.
3. Specify your appearance preferences.

Results

Tip: To change how an element appears in all diagrams that you create, click **Window** > **Preferences** > **Modeling** > **Appearance**.

Related tasks:

Setting preferences for UML diagrams

Changing the appearance of the Palette for UML diagrams

Organizing UML diagrams

In UML diagrams, you can organize various elements to optimize the diagram for viewing, printing, or presenting.

About this task

You can organize your diagrams by arranging the elements in them or by aligning the elements with each other or with rulers and grids. You can also organize elements in a diagram by sorting them alphabetically or by choosing to show or hide certain ones.

You can organize several elements at once by adding them to a group and then arranging, aligning, or performing other organizational tasks. To add units to a group, see Grouping elements in UML diagrams.

- **Layers in diagrams**
  A diagram can contain many shapes and elements and with increased complexity, the diagram can become hard to follow. Layers can hide or show particular elements in the diagram and can focus on a particular aspect of the diagram.
- **Adding layers to diagrams**
  You can add layers to new or existing diagrams in the model.
- **Modifying layers in UML diagrams**
  You can modify layers to add, remove, or replace elements in the current layer.
- **Selecting elements in a layer**
  In complex diagrams with multiple layers, it is possible to select elements belonging to a particular layer.
- **Arranging UML diagrams by using ILOG JViews Graph Layouts technology**
  An UML diagram can be reorganized automatically to a variety of templates to visually change the look of the diagram. Using ILOG® JViews Graph Layouts technology, the diagrams are reorganized based on the template selected. The template organizes the layout of the elements, links and labels to enhance the diagram visually and promotes finding relationships and patterns in the diagram.
- **Arranging elements in UML diagrams**
  You can arrange all the elements in a UML diagram hierarchically for viewing purposes. You can also arrange only selected diagram elements.
- **Aligning elements in UML diagrams**
  In UML diagrams, you can better organize the diagram layout by aligning elements in various ways.
- **Sorting compartment items in UML diagrams**
  In UML diagrams, for diagram elements that have compartments, you can sort the compartment items to reduce the complexity or increase the level of abstraction in the diagram. In this context, sorting means putting the items in alphabetical order.
- **Filtering compartment items in UML diagrams**
  In UML diagrams, several diagram elements have compartments that display textual and graphical information. You can reduce visual clutter and complexity in diagrams by filtering the contents of these compartments. Filtering hides information based on criteria that you specify.
- **Showing and hiding connectors in UML diagrams**
  In UML diagrams, you can show or hide connectors to reduce visual clutter or to provide more information about the relationship that they represent.
- **Showing and hiding connector labels in UML diagrams**
  In UML diagrams, you can show or hide connector labels to reduce visual clutter or to provide more information about the relationships that they represent.
- **Changing the order of stacked elements in UML diagrams**
  The elements in UML diagrams can appear stacked on top of each other in layers. You can change the order of diagram elements in stacks to organize the diagram visually. You can move a diagram element up or down a layer, bring it to the

front, or send it to the back of a stack.

- **Specifying parent name styles in UML diagrams**

  In UML diagrams, you identify a classifier in the diagram by giving it a unique name. In addition to the classifier name, you can display the classifier's fully qualified name, the classifier's parent name, the classifier's fully qualified parent name, or a combination of these options.

Related tasks:

Arranging elements in UML diagrams

Selecting elements in UML diagrams

Grouping elements in UML diagrams

Arranging elements in UML diagrams

You can arrange all the elements in a UML diagram hierarchically for viewing purposes. You can also arrange only selected diagram elements.

About this task

To arrange diagram elements, complete one of the following steps:

- To arrange all diagram elements, click **Diagram** > **Arrange** > **All**.
- To arrange selected diagram elements, in the diagram editor, select the diagram elements that you want to arrange; then click **Diagram** > **Arrange** > **Selection**.

Results

The diagram elements are automatically arranged in a hierarchical manner.

Related tasks:

Selecting elements in UML diagrams
Opening UML diagrams
Aligning elements in UML diagrams
Sorting compartment items in UML diagrams
Organizing UML diagrams
Grouping elements in UML diagrams

Aligning elements in UML diagrams

In UML diagrams, you can better organize the diagram layout by aligning elements in various ways.

About this task

You can align elements vertically or horizontally. When you align selected elements, they align to the diagram element that you select last and that has filled sizing handles. Grouped diagram elements, that you select by using the Marquee selection tool in the Palette, are aligned by the boundary of the group, not by the individual diagram elements in the group. You can also use rulers and guidelines or a grid coordinate system to align the elements in a diagram.

Procedure

1. In the diagram editor, select the diagram elements that you want to align.
2. Right-click **Format** > **Align** and click a type of alignment.
3. Optional: To align elements by using the ruler, right-click an empty space in the diagram; then click **View** > **Rulers**. Move the diagram elements to align with the guidelines.
4. Optional: To align elements with the grid, right-click an empty space in the diagram; then click **View** > **Grid** and complete one of the following steps:
   - To manually align elements with the grid, move the elements into place.
   - To automatically align elements with the grid, right-click an empty space in the diagram; then click **View** > **Snap to Grid**.

   Tip: To set a default preference so that all diagram elements snap to the grid automatically click **Window** > **Preferences** > **Modeling** > **Diagrams** > **Rulers and Grid**.

Related tasks:

Setting preferences for UML diagrams
Opening UML diagrams
Selecting elements in UML diagrams
Arranging elements in UML diagrams
Sorting compartment items in UML diagrams

Sorting compartment items in UML diagrams

In UML diagrams, for diagram elements that have compartments, you can sort the compartment items to reduce the complexity or increase the level of abstraction in the diagram. In this context, sorting means putting the items in alphabetical order.

About this task

In compartments, you can sort attributes or operations by name, visibility style, or return type in ascending or descending alphabetical order. You can also filter compartment items and sort them at the same time.

Procedure

1. In the diagram editor, right-click a diagram element; then click **Filters** > **Sort/Filter Compartment Items**.
2. In the Sort/Filter Compartment Items window, in the list on the left, select which compartment type to sort: **Attribute** or **Operation**.
3. In the table on the right, click the appropriate column heading once to display the contents in ascending order. Click the column heading again to display the contents in descending order.
4. Click **Apply**.
5. Click **OK**.

Related tasks:

Aligning elements in UML diagrams

Arranging elements in UML diagrams

Filtering compartment items in UML diagrams

Filtering compartment items in UML diagrams

In UML diagrams, several diagram elements have compartments that display textual and graphical information. You can reduce visual clutter and complexity in diagrams by filtering the contents of these compartments. Filtering hides information based on criteria that you specify.

About this task

You can filter compartment items by visibility or by selecting specific items that you want to hide from view.

Procedure

1. In the diagram editor, right-click a diagram element; then click **Filters** > **Sort/Filter Compartment Items**.
2. In the Sort/Filter Compartment Items window, complete one of the following steps:
   - To filter items by visibility, in the list on the left, select **All Compartments,Attribute,** or **Operation**. On the right, move the appropriate filter criteria into the **Filter Items Containing** list.
   - To filter items by selection, in the list on the left, select **Attribute** or **Operation**. In the table on the right, select the check boxes for the items that you want to show.

   Note: The changes you make in one set of filter controls will override any changes you make in the other. For example, if you move private visibility from the **Filter Criteria** list to the **Filter Items Containing** list, and then select the **private** check box in the table, the private visibility will move back into the **Filter Criteria** list.
3. Click **Apply**.
4. Click **OK**.

Related tasks:

Sorting compartment items in UML diagrams

Specifying parent name styles in UML diagrams

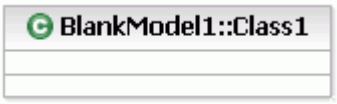
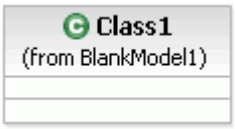
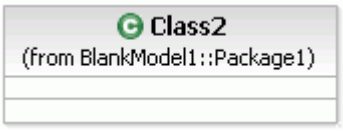
In UML diagrams, you identify a classifier in the diagram by giving it a unique name. In addition to the classifier name, you can display the classifier's fully qualified name, the classifier's parent name, the classifier's fully qualified parent name, or a combination of these options.

Before you begin

Before you can complete this task, you must have a diagram open in the Modeling perspective, and the diagram must contain at least one classifier.

About this task

The following table illustrates the different parent name style notations:

Style	Description	Example
Fully qualified name	The classifier's name compartment displays the concatenation of the parent name and the name of the classifier itself.	
Parent name	The classifier's name compartment displays the name of the classifier, and below it, the name of the namespace that contains the classifier.	
Fully qualified parent name	The classifier's name compartment displays the name of the classifier, and below it, the concatenation of the parent names.	

To specify the parent name style for an individual classifier, or a set of classifiers, complete one of the following steps:

Procedure

- To show the fully qualified name for a single classifier, in the diagram editor, right-click a classifier; then click **Filters** > **Show Qualified Name**.
- To show the parent name or the fully qualified parent name for an individual classifier, in the diagram editor, right-click a classifier; then click **Filters** > **Parent Style** and select the appropriate option.

Results

Tip: To set default preferences for parent name styles click **Window** > **Preferences** > **Modeling** > **Appearance** > **Shapes** .

Related tasks:

Setting preferences for UML diagrams

Showing and hiding connectors in UML diagrams

In UML diagrams, you can show or hide connectors to reduce visual clutter or to provide more information about the relationship that they represent.

About this task

In individual diagrams, you can specify whether to show or hide all the connectors or only the ones that you specify.

Procedure

1. In a diagram with connected diagram elements, complete one of the following steps:
   - To show or hide all the connectors in a diagram, right-click an empty space in the diagram.
   - To show or hide the connectors for a specific diagram element, right-click that element.
2. Click **Filters** > **Show/Hide Relationships**.
3. In the Show/Hide Relationships window, click the relationships to show or hide.
4. Click **OK**.

Showing and hiding connector labels in UML diagrams

In UML diagrams, you can show or hide connector labels to reduce visual clutter or to provide more information about the relationships that they represent.

About this task

In individual diagrams, you can specify whether to show or hide all the connector labels or only the ones that you specify. You can also set a default preference to show or hide all connector labels in the new diagrams that you create.

Procedure

1. In a diagram with connected diagram elements, complete one of the following steps:
   - To show or hide all connector labels in a diagram, right-click an empty space in the diagram.
   - To show or hide the labels for a specific connector, right-click that connector.
2. Click **Filters** > **Show/Hide Connector Labels** and click the appropriate item.

Results

Tip: To set default preferences to show or hide connector labels in the new diagrams that you create, click **Windows** > **Preferences** > **Modeling** > **Appearance** > **Connectors**.

Changing the order of stacked elements in UML diagrams

The elements in UML diagrams can appear stacked on top of each other in layers. You can change the order of diagram elements in stacks to organize the diagram visually. You can move a diagram element up or down a layer, bring it to the front, or send it to the back of a stack.

About this task

To change the order of a diagram element in a stack, in the diagram editor, right-click a diagram element; then click **Format** > **Order** and complete one of the following steps:

- To bring it to the front of the stack, click **Bring to Front**.
- To move it up a layer in the stack, click **Bring Forward**.
- To send it to the back of the stack, click **Send to Back**.
- To move it down a layer in the stack, click **Send Backward**.

Renaming UML diagrams

You can rename UML diagrams to reflect changes in their contents or purpose.

Procedure

1. In the Project Explorer view, right-click a diagram; then click **Refactor** > **Rename**.
2. Type a name and press Enter.

Results

Tip: You can also rename a diagram in the Properties view.

Resizing in UML diagrams

In UML diagrams, you can change the size of elements to improve the presentation of the diagram. You can resize diagram elements, make them the same size, or use the zoom feature to make more or less detail visible.

- **Making elements the same size in UML diagrams**

  In UML diagrams, you can make elements the same height and width, height, or width to improve the appearance or readability of the diagrams.

- **Resizing elements in UML diagrams**

  You can manually resize diagram elements in UML diagrams to accommodate a long name, to show more details, to optimize viewing and printing, or to emphasize elements.

- **Changing zoom levels in UML diagrams**

  You can change the zoom level in an open UML diagram. With a higher zoom level, you can see a diagram in more detail. A lower zoom level is helpful for large diagrams so that you can see a larger area of a diagram. You can change the animation effects for zoom levels by modifying the preferences for UML diagrams.

Making elements the same size in UML diagrams

In UML diagrams, you can make elements the same height and width, height, or width to improve the appearance or readability of the diagrams.

About this task

When you make diagram elements the same size, the elements assume the size of the primary selected element, which has filled sizing handles. Your selection method determines which element is primary. The following table shows which element's size is applied, based on the selection method.

Selection method	Element whose size is applied
Select All menu item	Last element that you added to the diagram
Marquee selection tool in the Palette	Nearest element to the starting point of the selection border
Ctrl or Shift	Last element that you select

Procedure

1. In the diagram editor, select the diagram elements to resize.
2. Click **Diagram** > **Make Same Size** and complete one of the following steps:
   - To make the elements the same height and width, click **Both**.
   - To make the elements the same height, click **Height**.
   - To make the elements the same width, click **Width**.

Related tasks:

Resizing elements in UML diagrams

Changing zoom levels in UML diagrams

Setting preferences for UML diagrams

Resizing elements in UML diagrams

You can manually resize diagram elements in UML diagrams to accommodate a long name, to show more details, to optimize viewing and printing, or to emphasize elements.

Procedure

1. In the diagram editor, select a diagram element or a group of diagram elements.
2. Complete one of the following steps:
   - To resize the diagram element in a specific direction, drag the appropriate sizing handle.
   - To resize the diagram element while maintaining the diagram element's center of view, press the Shift key and drag a sizing handle.
   - To resize the diagram element and maintain aspect ratio, press Ctrl and drag a sizing handle.
   - To resize the diagram element and maintain both the aspect ratio and center of view, press Ctrl+Shift and drag a sizing handle.
   - To automatically resize the diagram elements, click **Diagram** > **Auto Size**.
   - To equally resize the diagram elements, click **Diagram** > **Make Same Size** and select the appropriate menu choice.

Results

Note: If you resize a diagram element manually, the auto-size default is turned off. You can turn on the auto-size default by clicking **Diagram** > **Auto Size**.

Related tasks:

Changing zoom levels in UML diagrams
Making elements the same size in UML diagrams
Setting preferences for UML diagrams
Selecting elements in UML diagrams
Opening UML diagrams

Changing zoom levels in UML diagrams

You can change the zoom level in an open UML diagram. With a higher zoom level, you can see a diagram in more detail. A lower zoom level is helpful for large diagrams so that you can see a larger area of a diagram. You can change the animation effects for zoom levels by modifying the preferences for UML diagrams.

About this task

To change the zoom level of a UML diagram, click **Diagram** > **Zoom** and click a zoom level.

Results

The diagram is resized in the diagram editor according to the zoom level that you select.

Related tasks:

Resizing elements in UML diagrams

Making elements the same size in UML diagrams

Setting preferences for UML diagrams

Specifying stereotype styles in UML diagrams

In UML diagrams, you can specify the stereotype style for diagram elements or for the attribute and operation compartments in diagram elements. A stereotype is a UML extension mechanism that broadens the UML vocabulary and gives more specific meaning to a diagram element.

About this task

For example, you might change how a stereotype for a class is displayed by adding a decorative icon. Specifically, you can choose whether to display stereotypes with decoration icons, with text in double angle brackets (« »), or not to display them at all.

Procedure

1. In the diagram editor, select a diagram element or group of elements.
2. Click **Diagram** > **Filter** > **Stereotype and Visibility Style** and complete one of the following steps:
   - To specify a setting for a diagram element, click a style that is prefaced with **Stereotype**.
   - To specify a setting for compartments, click a style that is prefaced with **Compartment Stereotype**.

Results

Note: You can also specify the stereotype styles for diagram elements in the Properties view.

Related concepts:

UML stereotypes

Showing related elements in diagrams

You can populate a diagram by showing existing source elements that are related by specific relationships to the selected source elements.

Procedure

1. In the diagram editor, right-click a source element or a group of selected source elements; then click **Filters** > **Show Related Elements**.
2. In the Show Related Elements in Diagram window, complete one of the following steps:
   - To perform an existing query, under **Custom Query**, click a query in the list.
   - To perform a new query, click **Details** and specify the details for the query.
3. Click **OK**.

Creating URLs to elements in UML models

A URL is a stereotyped comment object that you can use to navigate to a specified URL, file system element, or an element in the workspace. A URL has a type property as a further specialization of the URL. The traceability feature treats URLs as specifications.

Procedure

1. In the Project Explorer view, right-click an element; then click **Add UML** > **URL**. You can add a URL from in the Palette, under **UML Common**, click **URL** and click in the diagram editor.
2. Specify a URL by typing one, browsing to a file, or browsing the workspace.
3. Optional: Type a display name for the URL.**Tip:** By default, URL image is displayed in the UML link on a diagram. To display the name of the URL instead of the image, go to **Windows** > **Preferences** > **Modeling** > **UML Diagrams** > **URL presentation**, and then select **Show URL link**. Now, you can create a URL link on the diagram, set the URL path to local image file, and show the display name on the diagram.
4. Optional: Specify a type for the URL.
5. Optional: Specify an icon for the **URL**.
6. Click **OK**.**Tip:** To navigate to a link from an element, in the diagram editor, right-click the element and click **Navigate** > **To Link**. The element that was referenced in the element link is selected in the Project Explorer view.

What to do next

To add URLs to elements from the Properties window, on the Documentation page, click the **Add Link** icon ().

Linking UML diagrams

You can create links between related UML diagrams in the same project or in different projects, so that you can navigate from an open diagram to other linked diagrams.

Procedure

1. Open a diagram that you want to link to another diagram.
2. In the Project Explorer view, locate the diagram that you want to link to.
3. Click and drag the diagram file to an empty space in the open diagram where you want to insert a link icon.

Results

A link, or a shortcut, is created between the two diagrams and a link icon appears in the open diagram. You can double-click the link icon to open the linked diagram in the diagram editor.

Related concepts:

Link relationships

Related tasks:

Specifying relationships in diagrams

Linking UML model elements to external files

Saving UML diagrams as images

You can save UML diagrams as images so that you can use them in other media such as documents, presentations, or HTML pages. You can save diagrams in GIF, BMP, JPEG, JPG, or SVG format.

Procedure

1. Right-click in the diagram and click **File** > **Save as Image File**.
2. In the Save as Image File window, in the **Folder** field, specify where to save the image file.
3. In the **File Name** field, type a file name.
4. From the **Image Format** list, select a file format.
5. Optional: To automatically replace an image with the same name when you save, select the **Overwrite existing file without warning** check box.
6. Click **OK**.

Managing the printing of UML diagrams

UML diagrams can become quite large, spanning several pages. You can control how your diagrams are printed by manipulating the page breaks. You can then preview and print UML diagrams.

- **Managing page breaks in UML diagrams**

  In UML diagrams, you can display page breaks and store the page break information with the diagrams. If you make changes to the print parameters, such as page size, margins, or orientation, the page breaks are updated in the diagrams. You can also recalculate page breaks after you make changes to diagrams and center diagrams on a minimum number of pages for printing.

- **Setting printing preferences for UML diagrams**

  You can define the printing preferences for an individual UML diagram or for all new UML diagrams that you create.

- **Printing UML diagrams**

  Standard printing features are available for printing UML diagrams; however, you can also specify which diagrams you want to print and how to scale them to fit on the paper.

Results

Note: You can print or preview diagrams in Windows environments only.

Managing page breaks in UML diagrams

In UML diagrams, you can display page breaks and store the page break information with the diagrams. If you make changes to the print parameters, such as page size, margins, or orientation, the page breaks are updated in the diagrams. You can also recalculate page breaks after you make changes to diagrams and center diagrams on a minimum number of pages for printing.

Before you begin

Note: You can print or preview diagrams in Windows environments only.

The open diagram must contain elements that span several pages. You must also have a printer installed on your system so that information about the printer and its settings are available to the application for calculating page breaks.

About this task

To manage the page breaks in a UML diagram, in the diagram editor, right-click an empty space; then click **View** and complete one of the following steps:

- To view the page breaks, click **Page Breaks**.
- To recalculate the page breaks, click **Recalculate Page Breaks**.

Results

The diagram elements are displayed within the boundaries for the updated page breaks.

Related tasks:

Printing UML diagrams

Setting printing preferences for UML diagrams

Setting printing preferences for UML diagrams

You can define the printing preferences for an individual UML diagram or for all new UML diagrams that you create.

Before you begin

Note: You can print or preview diagrams in Windows environments only.

Procedure

1. Click anywhere in the diagram and click **File** > **Page Setup**.
2. In the Page Setup window, complete one of the following steps:
   - To set printing preferences for the diagram that is open, click **Use diagram settings** and modify the page settings.
   - To set printing preferences for all new diagrams, click **Use workspace settings** > **Configure workspace settings**. In the Preferences window, modify the settings.
3. Click **OK**.

Related tasks:

Managing page breaks in UML diagrams

Printing UML diagrams

Printing UML diagrams

Standard printing features are available for printing UML diagrams; however, you can also specify which diagrams you want to print and how to scale them to fit on the paper.

Before you begin

Note: You can print or preview diagrams in Windows environments only.

Procedure

1. Optional: To preview a diagram before you print it, click **File** > **Print Preview**.
2. To print a diagram, click **File** > **Print**.
3. Under **Diagram print range**, complete one of the following steps:
   - To print all the diagrams in the project, click **All Diagrams**.
   - To print the open diagram, click **Current Diagram**.
   - To print specific diagrams, click **Selected Diagrams** and then click the diagram names.
4. Optional: Under **Scaling**, complete one of the following steps:
   - To scale the diagram by a specific percentage, click **By percent** and type a percent value.
   - To scale the diagram to fit on a specific number of pages, click **By pages** and type the width and height values. The system maintains the aspect ratio of the diagram.
5. Click **OK**.

Related tasks:

Managing page breaks in UML diagrams

Setting printing preferences for UML diagrams

Deleting diagrams from projects

You can delete from projects the diagrams that you no longer require.

About this task

To delete a diagram from a project, in the Project Explorer view, right-click a diagram; then click **Delete**.

Results

The diagram is removed from the entire project.

Related tasks:

Deleting UML diagram elements

Developing SIP applications

Session Initiation Protocol (SIP) is an application-layer protocol that you can use to initiate, modify, or terminate communication and collaborative sessions over Internet Protocol (IP) networks. It is typically used for instant messaging, telephony, and other real-time collaboration activities. A SIP application is a Java™ program that uses at least one servlet that runs on a SIP-enabled application server. The workbench includes tools to help you create and develop SIP applications.

Using the workbench wizards, you can create a project and then add SIP servlets to the project. SIP projects use the Java EE perspective and support the servlet archive (SAR) format.

SIP 1.0 is defined in the JSR-116 specification and SIP 1.1 is defined in the JSR-289 specification. Both specifications describe the convergence of SIP with Java EE components; SIP 1.0 includes support for providing deployment descriptors, and SIP 1.1 projects can use both deployment descriptors and annotations.

Specifications

SIP 1.0 is defined in the JSR-116 specification and SIP 1.1 is defined in the JSR-289 specification. Both specifications describe the convergence of SIP with Java EE components; SIP 1.0 includes support for providing deployment descriptors, and SIP 1.1 projects can use both deployment descriptors and annotations.

If you are new to SIP application development, you should review the SIP specifications:

- JSR-116 SIP Servlet 1.0 API
- JSR-289 SIP Servlet 1.1API

Getting started

If you are already familiar with SIP application technology, these topics guide you through the development process.

- **Creating SIP projects**
- **Adding SIP capability to an existing web project**
- **Creating SIP servlets**
- **Adding SIP annotations**
- **Editing SIP deployment descriptors**

Tools for SIP development

Tools to help you create Session Initiation Protocol (SIP) applications include several wizards and a deployment descriptor editor.

The following tools are provided to assist you in creating SIP applications:

- A wizard for creating a new SIP project
- A wizard for creating SIP 1.1 or 1.0 servlets
- A graphical SIP deployment descriptor editor
- Wizards for importing and exporting Servlet Archive (SAR) files
- Support for annotations and content assist

Samples

SIP samples are included with the product. The sample versions for SIP 1.1 are the same as the corresponding SIP 1.0 version; however, the deployment descriptor was updated for SIP 1.1.

To view the sample and tutorials in this product, click **Help** > **Help Contents** and expand the Samples and Tutorials sections.**Note:** The links to the samples work only if you are viewing this topic from the product help system.

- **Call Blocking sample**
  - This sample demonstrates the use of the SIP Servlet 1.0 application programming interfaces (API). This sample checks a list to determine whether the caller is valid. If the caller is not valid, the call is blocked. If the caller is valid, the call is forwarded.
    - SIP version 1.1.
    - SIP version 1.0.
- **Call Forwarding sample**
  - This sample demonstrates the use of the SIP Servlet 1.0 application programming interfaces (API). This sample checks to determine if the caller is in a forwarding list and forwards the call. The sample is available for these versions:
    - SIP version 1.1.
    - SIP version 1.0.

\

Resources for learning available on the Web

In addition to the information found in this information center, the following links provide additional learning material.

- JSR-116 SIP Servlet 1.0 API
- JSR-289 SIP Servlet 1.1 API
- IBM® Education Assistant: SIP Overview
- IBM Education Assistant: JSR 289 Overview
- IBM Education Assistant: Annotations for SIP Applications

Latest developerWorks articles and tutorials about SIP

- **SIP overview**
  SIP (Session Initiation Protocol) is a peer-to-peer protocol used to establish, modify, and terminate multimedia internet protocol (IP) sessions between two endpoints including telephony and instant messaging.
- **Creating SIP projects**
  Use the New Project wizard to create a Session Initiation Protocol (SIP)/Hypertext Transfer Protocol (HTTP) Project. You can develop both SIP and HTTP servlets. On the Project Definition page, you can define the new project name, location, and target server runtime environment.
- **Adding SIP capability to an existing web project**
  You can add a SIP facet to an existing web project so that it becomes a SIP application.
- **Creating SIP servlets**
  Use the New SIP Servlet wizard to create a Session Initiation Protocol (SIP) servlet. A SIP servlet is a Java-based application component that performs SIP signaling and is run by a Java-enabled SIP application server.
- **Editing SIP deployment descriptors**
  The Session Initiation Protocol (SIP) deployment descriptor file, sip.xml, specifies deployment information, Multipurpose Internet Mail Extension (MIME) types, session configuration details and other settings for a SIP application. This information is used in building a servlet archive (SAR) file from a project. The sip.xml file is located in the WEB-INF directory of the project.
- **Adding SIP annotations**
  Add annotations to SIP 1.1 servlets to embed data directly in a SIP application and to inject resources, such as EJB or SIP utility classes.
- **SIP workbench preferences**
  You can set the Session Initiation Protocol (SIP) preference page to control how the Web deployment descriptors are updated with SIP deployment descriptor content.
- **Deployment configurations for SIP applications**
  To deploy SIP applications to WebSphere® Application Server, the target server must have a configuration that is

supported by the SIP version that you are using.

- **Importing servlet archive (SAR) files**

  A servlet archive (SAR) file is a portable, packaged Session Initiation Protocol (SIP) application that you can import into your workspace.

- **Exporting servlet archive (SAR) files**

  A servlet archive (SAR) file is a packaged Session Initiation Protocol (SIP) application. You can export this file to test, publish, and deploy the resources developed within the SIP project.

- **Importing SIP applications in EAR or WAR files**

  If you import EAR or WAR files that contain a SIP application, then you must add the SIP facet to the project.

Related concepts:

SIP overview

Differences between SIP 1.0 and SIP 1.1

SIP overview

SIP (Session Initiation Protocol) is a peer-to-peer protocol used to establish, modify, and terminate multimedia internet protocol (IP) sessions between two endpoints including telephony and instant messaging.
A SIP application is a Java™ program that uses at least one SIP servlet where a SIP servlet is a Java-based application component that is managed by a SIP servlet container (for example, WebSphere® Application Server).
Here are some common uses of SIP in telecommunications-based applications:
- Voice-over-IP (VoIP)
- Instant messaging
- Click to call
- Call notification, forwarding, blocking

SIP servlet specifications were developed under the Java Community Process:
- JSR-289 SIP Servlet 1.1 API
- JSR-116 SIP Servlet 1.0 API

Both the SIP 1.1 and SIP 1.0 specifications are based on the Java servlet application programming interface (API). SIP Servlet v1.1 (JSR 289) is a revision of the initial SIP Servlet v1.0 (JSR 116) Specification with additional enhancements. It defines the API for the SIP servlet programming model, clarifies the roles and responsibilities of the SIP servlet container, and describes the convergence of SIP with Java EE components.

Converged SIP applications

A converged SIP application is an application that uses both HTTP Servlet API and Java EE components. A converged SIP application can be either of the following combinations:
- SIP and HTTP converged applications, hosting SIP and HTTP servlets
- SIP and Java EE converged applications, hosting SIP, HTTP, and Java EE components such as web services

Converged SIP applications created in the workbench are packaged as a WAR file in an EAR file.

- **Differences between SIP 1.0 and SIP 1.1**
  There are some key differences between Session Initiation Protocol (SIP) 1.0 and SIP 1.1 in areas such as the deployment descriptor, annotations, and servlet specifications.

Related concepts:
SIP workbench preferences
Differences between SIP 1.0 and SIP 1.1
Developing SIP applications

Related tasks:
Adding SIP capability to an existing web project
Creating SIP servlets
Editing SIP deployment descriptors
Synchronizing the Web and SIP deployment descriptors
Adding SIP annotations
Importing servlet archive (SAR) files
Exporting servlet archive (SAR) files
Importing SIP applications in EAR or WAR files

Creating SIP projects

Related reference:

Deployment configurations for SIP applications

SIP 1.1 annotations

Differences between SIP 1.0 and SIP 1.1

There are some key differences between Session Initiation Protocol (SIP) 1.0 and SIP 1.1 in areas such as the deployment descriptor, annotations, and servlet specifications.

Table 1. Comparison of some differences between SIP 1.0 and SIP 1.1

Function	SIP 1.0	SIP 1.1
Deployment descriptor	The deployment descriptor is described by a DTD.	The deployment descriptor is described by an XML schema. Some other changes in the deployment descriptor are:**app-name** – Mandatory configuration to denote the SIP application name**servlet-selection:** You select either a Main-servlet or Servlet-mappings but not both.**Main-servlet:**Process initial requestForwards requests to other servletsOnly one Main-servlet per application**Servlet-mappings:** - Backward compatibility with SIP1.0
Annotations	No annotations support	Support for annotations defined by Java™ EE 5 within SIP servlets and listeners; defines custom annotations representing SIP interfaces.You can use annotations to perform these tasks:Embed metadata directly in an application.Inject resources, such as an enterprise bean, into an application. There are four supported annotations in SIP 1.1:@SipServlet@SipApplication@SipListener@SipApplicationKey

New in SIP 1.1 servlet specification

SIP 1.1 servlet specification adds some features:

- Introduces a new entity called an Application Router
  - Sits alongside the SIP Container for routing SIP requests
  - Has logic to select which SIP applications to call.
  - WebSphere® Application Server provides a default application router. For more information, see the topic on SIP application router in the WebSphere Application Server information center.
- o The SIP Servlet API was extended to include two new methods:
  - doRefer() : forwards request to a third party.
  - doUpdate(): updates a session without changing the dialog state.

Related concepts:

SIP overview

Developing SIP applications

Related tasks:

Creating SIP servlets
Adding SIP annotations

Related reference:
SIP 1.1 annotations

Creating SIP projects

Use the New Project wizard to create a Session Initiation Protocol (SIP)/Hypertext Transfer Protocol (HTTP) Project. You can develop both SIP and HTTP servlets. On the Project Definition page, you can define the new project name, location, and target server runtime environment.

Procedure

To create a SIP project:

1. Click **File** > **New** > **Project**.
2. Open the **SIP** folder, click **SIP Project** and then click **Next**.
3. On the first page of the wizard, follow these steps:
   A. Enter a name for the project.
   B. Enter a location for the project, or select to use the default location.
   C. Select the SIP module version for the project.
   D. If no target runtime environment is already selected, select one. Do **not** select **None**. The target runtime that you select must support your SIP module version.
   E. If it is not already set, in the **Configuration** field, select either SIP 1.0 Project or SIP 1.1 Project according to the SIP module version that you selected.
   F. Optional: To facilitate deployment, associate your new SIP application with a new or existing Enterprise Application project (EAR Project) by selecting **Add Project to an EAR**. Click **Next**.
4. On the Java page, add folders for your Java source files and specify a location for your output Java classes, or keep the default settings.
5. On the Web Module page, configure the Web module settings or accept the default settings and then click **Finish**.

Results

This SIP/HTTP project is a web project and can be used for development of both SIP and HTTP servlets. The project contains the following files in the WebContent\WEB-INF folder:

- A SIP deployment descriptor file (sip.xml)
- A Web deployment descriptor file (web.xml)

What to do next

You can define a server to run the SIP project by clicking **Project** > **Properties** > **Server** and adding a server definition from the list.**Tip:** The server that you add must have a profile that supports the SIP version that you added. To view the profiles that are available for WebSphere® Application Server, click **Window** > **Preferences** > **Server** > **WebSphere Application Server**. To view or change a profile configuration, select a profile and click **Run profile management tool**.

Related concepts:

SIP workbench preferences

SIP overview

Related tasks:

Adding SIP capability to an existing web project

Creating SIP servlets

Editing SIP deployment descriptors

Synchronizing the Web and SIP deployment descriptors

Related reference:

Deployment configurations for SIP applications

Related information:

Creating a profile on a local WebSphere Application Server V7.0

Adding SIP capability to an existing web project

You can add a SIP facet to an existing web project so that it becomes a SIP application.

Procedure

To add a SIP facet to an existing web project:

1. In Project Explorer, right-click the web project and select **Properties**.
2. Select **Project Facets** and click **Modify Project**.
3. Select the SIP Module check box, select an appropriate SIP version, and then click **Finish**. When you are prompted to update the Web deployment descriptor with SIP deployment descriptor content, click **Yes**.

Results

The Web project is updated to include a SIP deployment descriptor (sip.xml) file in the WebContent folder. You can now add SIP components to the project.

Related concepts:

SIP overview

SIP workbench preferences

Related tasks:

Creating SIP projects

Creating SIP servlets

Editing SIP deployment descriptors

Synchronizing the Web and SIP deployment descriptors

Related reference:

Deployment configurations for SIP applications

Creating SIP servlets

Use the New SIP Servlet wizard to create a Session Initiation Protocol (SIP) servlet. A SIP servlet is a Java-based application component that performs SIP signaling and is run by a Java-enabled SIP application server.

Before you begin

Before you create a SIP servlet, you must complete these tasks:

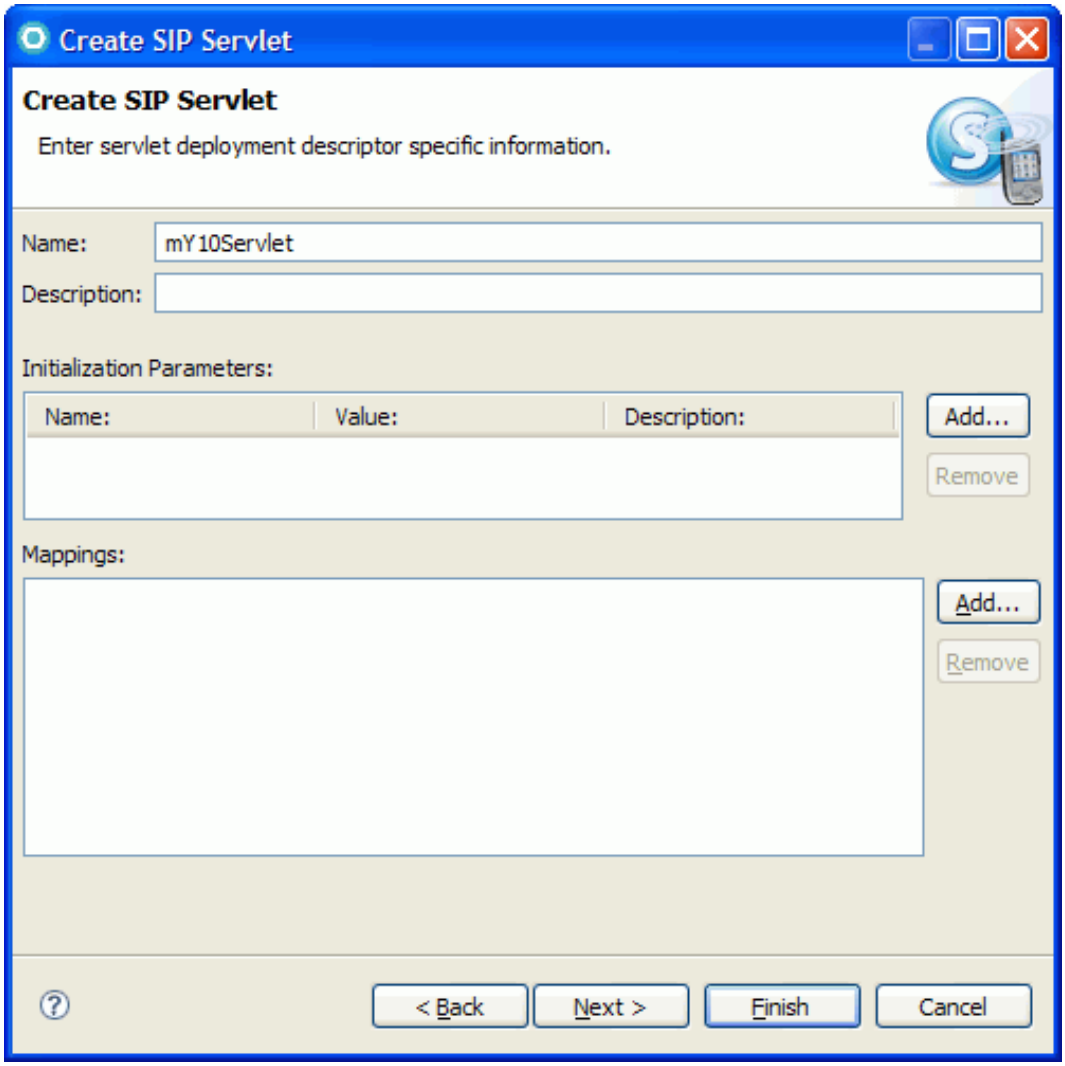
- Create a SIP project.
- Change to the Web development perspective.

Tip: On the SIP preferences page (**Window** > **Preferences** > **SIP**), ensure that **Always update the deployment descriptors is selected** so that the Web deployment descriptor (web.xml) is updated with the new servlet that you create.

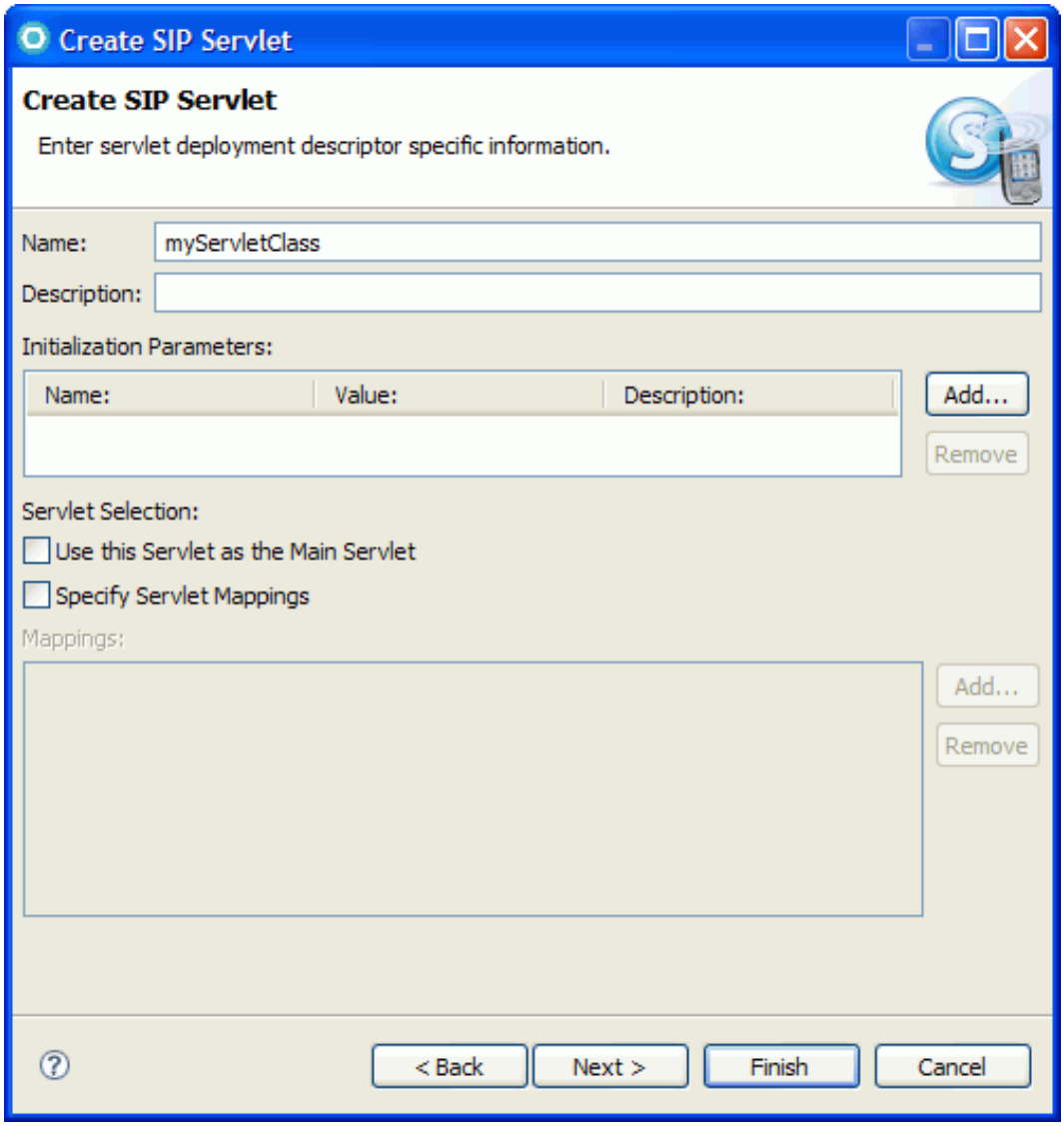
Procedure

To create a SIP servlet:

1. In the Enterprise Explorer view, right-click the project where you want to add a SIP Servlet, and then click **New** > **Other**.
2. Open the **SIP** folder, click **SIP Servlet** and then click **Next**.
3. On the first page of the wizard, either enter a Java package and class name, or select an existing servlet class or JSP, and click **Next**. Typically, the superclass is javax.servlet.sip.SipServlet.
4. On the second page of the wizard, enter deployment information such as the description, initialization parameters, and any servlet mappings.



For SIP 1.1 servlets, you have the option to select the servlet as the main servlet.



Click **Next**.**Tip:** You can change deployment information later by using the deployment descriptor editor.

5. On the final wizard page, specify the modifiers, interfaces, and method stubs you want to generate and click **Finish**.
6. If you are prompted to update the Web deployment descriptor, click **Yes**.**Tip:** If you click **No**, or if you turned off the preference to always update the deployment descriptors, then you can update the Web deployment descriptor (web.xml) file later. (Right-click the sip.xml file and click **SIP Operations** > **Update web.xml**).

Results

The wizard creates the new servlet in the specified source folder and opens it in the Java editor.

Related concepts:

SIP overview

SIP workbench preferences

Differences between SIP 1.0 and SIP 1.1

Related tasks:

Creating SIP projects

Adding SIP capability to an existing web project

Editing SIP deployment descriptors
Synchronizing the Web and SIP deployment descriptors

Related reference:
Deployment configurations for SIP applications
SIP 1.1 annotations

Editing SIP deployment descriptors

The Session Initiation Protocol (SIP) deployment descriptor file, sip.xml, specifies deployment information, Multipurpose Internet Mail Extension (MIME) types, session configuration details and other settings for a SIP application. This information is used in building a servlet archive (SAR) file from a project. The sip.xml file is located in the WEB-INF directory of the project.

Procedure

To edit the SIP deployment descriptor:

1. In Project Explorer, open the **WebContent** and **Web-INF** folders of a SIP project.
2. Double-click the **sip.xml** file.

What to do next

The SIP deployment descriptor editor opens for the SIP version of your project: SIP 1.1 or SIP 1.0.

With the **SIP 1.1 deployment descriptor editor**, you configure deployment information in a visual tree structure by adding and configuring nodes of properties for the deployment. If you click a node on the left side of the editor, you can edit the attributes for the node on the right side of the editor. You can add child nodes (properties) by clicking **Add**.**Tip:** For SIP 1.1, if your application has multiple servlets, then you should add a servlet-selection node with either a main servlet or servlet mapping. However, the editor does not require that you do this.

The **SIP 1.0 deployment descriptor editor** is organized in pages with tabs:

- **Overview:**
  - View a quick summary of the contents in the SIP deployment descriptor. You can add, remove, or change the contents.
- **Servlets:**
  - Add a new servlet to the deployment descriptor, modify an existing servlet or remove the selected servlet from the deployment descriptor.
- **Security:**
  - Manage security roles and security constraints. You can specify the following data constraints on user data:
    - None: The application requires no transport guarantees.
    - o Integral: Data cannot be changed in transit between client and server.
    - o Confidential: Data content cannot be observed while it is in transit.
- **Variables:**
  - Manage a list of listeners, context parameters and environment variables in the project.
- **References:**
  - Manage the project resource references. The following types of references are supported:
    - EJB Local
    - EJB Remote
    - Resource
    - Resource Environment
- **Source:**
  - Edit the sip.xml source directly.

- **Synchronizing the Web and SIP deployment descriptors**

  You can manually trigger synchronization of the Web deployment descriptor in your converged SIP/HTTP project with the

Session Initiation Protocol (SIP) deployment descriptor (sip.xml).

Related concepts:

SIP overview

SIP workbench preferences

Related tasks:

Creating SIP projects

Adding SIP capability to an existing web project

Creating SIP servlets

Synchronizing the Web and SIP deployment descriptors

Related reference:

Deployment configurations for SIP applications

SIP 1.1 annotations

Synchronizing the Web and SIP deployment descriptors

You can manually trigger synchronization of the Web deployment descriptor in your converged SIP/HTTP project with the Session Initiation Protocol (SIP) deployment descriptor (sip.xml).

Procedure

1. In the Enterprise Explorer view, right-click the sip.xml file (located in the WebContent/WEB-INF directory).
2. Click **SIP Operations** > **Update web.xml**.

Results

The web.xml file is updated with the SIP servlet metadata from the sip.xml file.

Related concepts:

SIP overview

SIP workbench preferences

Related tasks:

Creating SIP projects

Adding SIP capability to an existing web project

Creating SIP servlets

Editing SIP deployment descriptors

Related reference:

Deployment configurations for SIP applications

SIP 1.1 annotations

Adding SIP annotations

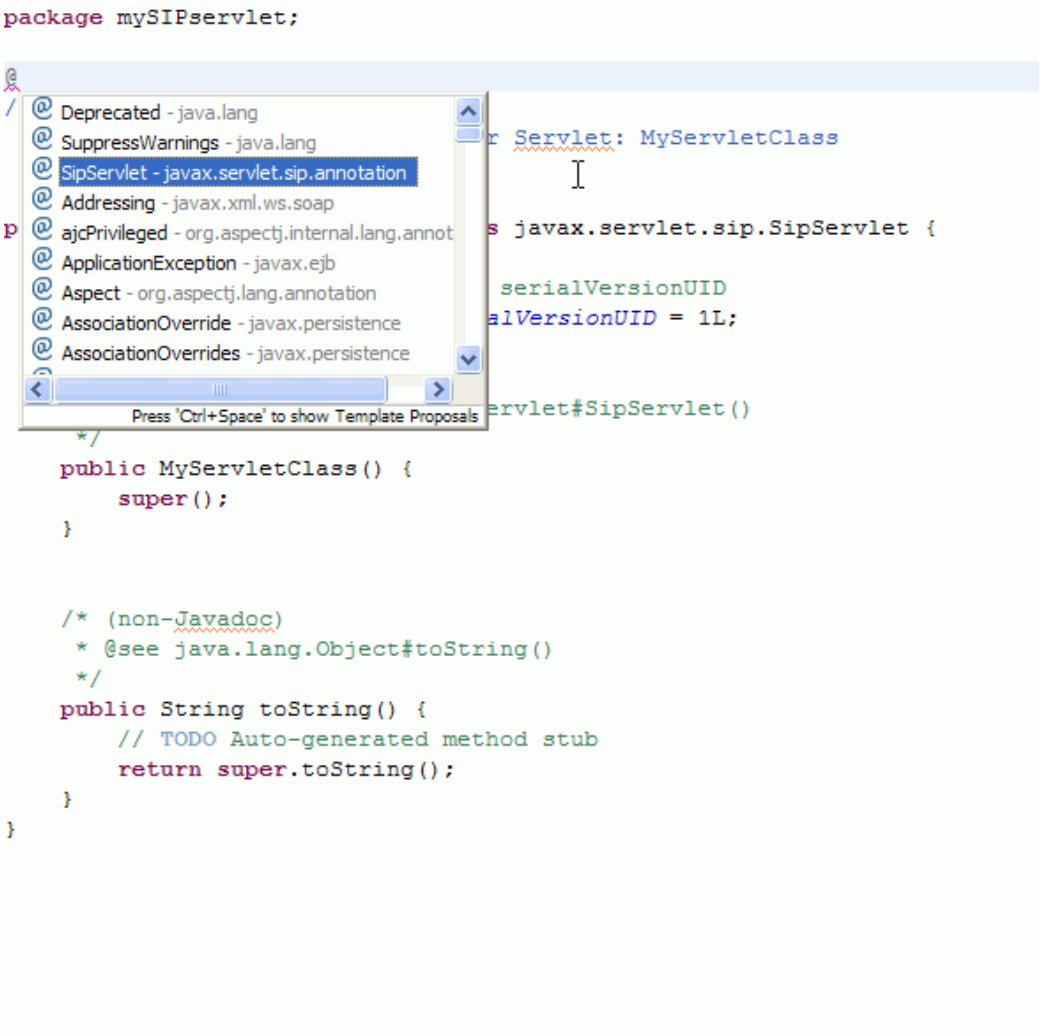
Add annotations to SIP 1.1 servlets to embed data directly in a SIP application and to inject resources, such as EJB or SIP utility classes.

Before you begin

You must create a SIP 1.1 servlet.

Procedure

1. Open your SIP servlet class open in the Java™ editor.
2. Click an appropriate location in your code and type the annotation (for example, @SipServlet).**Tip:** Type the @ character and click **CTRL+Spacebar** to see a list of annotations that you can add.



- **SIP 1.1 annotations**

  Starting in SIP 1.1, you can use annotations in SIP servlet applications. You can use annotations to embed data directly in an application instead of using the deployment descriptor.

Related concepts:

SIP overview

Differences between SIP 1.0 and SIP 1.1

Related reference:

SIP 1.1 annotations

SIP 1.1 annotations

Starting in SIP 1.1, you can use annotations in SIP servlet applications. You can use annotations to embed data directly in an application instead of using the deployment descriptor.

Annotations available in SIP 1.1

The following annotations are available in SIP 1.1:

- @SipApplication
- @SipServlet
- @SipListener
- @SipApplicationKey

Tip: You can still choose to use deployment descriptors instead of annotations. Deployment descriptors will override settings described in the annotations.

@SipApplication annotation

The @SipApplication annotation defines common configuration information about an application. It is a package-level annotation and must be located in a file named package-info.java. All servlets within the package belong to the same application.**Important:** Only one SIP application can be registered with the container for each .war or .sar file, regardless of whether you use a deployment descriptor or annotations.

Table 1. Properties for @SipApplication and mapping to corresponding deployment descriptor elements

Annotation property	Deployment descriptor element	Description	Default annotation value
name	<app-name>	The name of the SIP application	Required field
displayName	<display-name>	Displayed name of the application	Application name
description	<description>	Describes the application	Empty string
smallicon	<small-icon>	Path with the location of the small icon	Empty string
largeicon	<large-icon>	Path with the location of the large icon	Empty string
distributable	<destributable>	Indicates if the application can function in a distributed environment	False (boolean)
proxyTimeout	<proxy-timeout>	Default timeout for all proxy operations	Three minutes, in seconds
sessionTimeout	<session-timeout>	Default timeout for all application session operations	Three minutes, in minutes
mainServlet	<main-servlet>	Indicates the SIP servlet that is designed as the Main Servlet	Empty string

@SipServlet annotation

The @SipServlet annotation indicates that a class is a SIP servlet.

Table 2. Properties for @SipServlet and mapping to corresponding deployment descriptor elements

Annotation property	Deployment descriptor element	Description	Default annotation value

@SipServletServlet	<servlet-class>	Indicates that the class is a SIP servlet	The annotation is declared on the class.
name	<servlet-name>	Name of the servlet	Short name of the annotated class
applicationName	<app-name>	Name of the application	Optional; it can also be defined either in the deployment descriptor or by using @SipApplication
loadOnStartup	<load-on-startup>	Defines the starting order of the servlet application	A negative number (Causes the container to choose when to start this servlet)

@SipListener annotation

The @SipListener annotation provides an alternative to the <listener> deployment descriptor element.

The @SipListener annotation does not have any required fields, but can take an optional application name and description.

The class annotated by @SipListener must implement at least one listener interface.

@SipApplicationKey annotation

The @SipApplicationKey annotation marks the method that associates an incoming request and SipSession with a specific SipApplicationSession. You use this annotation with session key-based targeting.

Related concepts:

SIP overview

Differences between SIP 1.0 and SIP 1.1

Related tasks:

Editing SIP deployment descriptors

Synchronizing the Web and SIP deployment descriptors

Creating SIP servlets

Adding SIP annotations

Related information:

IBM Education Assistant: Annotations for SIP Applications

SIP workbench preferences

You can set the Session Initiation Protocol (SIP) preference page to control how the Web deployment descriptors are updated with SIP deployment descriptor content.

To access the SIP preference page, click **Window** > **Preferences** > **SIP**.

The SIP preference page has the following options for both SIP 1.1 and SIP 1.0 projects. These options are selected by default.

- **Always update the deployment descriptors**
  - If you select this option, the Web deployment descriptor (web.xml file) is automatically updated with SIP deployment descriptor (sip.xml) metadata. During Session Initiation Protocol project creation, the SIP deployment descriptor (sip.xml) and Web deployment descriptor (web.xml) files are created. To enable deployment to WebSphere® Application Server, add the SIP application to an EAR project and modify the web.xml file to include the SIP content. If the web.xml file is not updated, a warning message is displayed in the **Problems View**.
    Tip: To manually trigger an update of the Web deployment descriptor (web.xml) file later, right-click the sip.xml file and click **SIP Operations** > **Update web.xml**.

- **Prompt before updates**
  - If this option is selected, each time the sip.xml file is updated, you are prompted to choose whether to update the web.xml file.

Related concepts:

SIP overview

Related tasks:

Creating SIP projects

Adding SIP capability to an existing web project

Creating SIP servlets

Editing SIP deployment descriptors

Synchronizing the Web and SIP deployment descriptors

Related reference:

Deployment configurations for SIP applications

Deployment configurations for SIP applications

To deploy SIP applications to WebSphere® Application Server, the target server must have a configuration that is supported by the SIP version that you are using.

Supported server configurations for SIP 1.0

You can deploy SIP 1.0 applications to a WebSphere Application Server with any of the following configurations:

- Any version of WebSphere Application Server V6.1
- WebSphere Application Server V7.0 and earlier than V7.0.0.5 with a base server profile.
- WebSphere Application Server V7.0.0.5 or later, either with a base profile or a profile augmented to be compatible with the Feature Pack for Communications Enabled Applications (CEA).

Supported server configurations for SIP 1.1

To deploy SIP 1.1 applications to WebSphere Application Server, the server must have WebSphere Application Server V7.0.0.5 or later with the CEA feature pack installed. The server profile must be augmented to be compatible with the CEA feature pack.

Related concepts:

SIP overview

SIP workbench preferences

Related tasks:

Creating SIP projects

Adding SIP capability to an existing web project

Creating SIP servlets

Editing SIP deployment descriptors

Synchronizing the Web and SIP deployment descriptors

Related information:

Creating a profile on a local WebSphere Application Server V7.0 or V8.0

Augmenting profiles for remote WebSphere Application Server with the Feature Pack for Communications Enabled Applications

Importing servlet archive (SAR) files

A servlet archive (SAR) file is a portable, packaged Session Initiation Protocol (SIP) application that you can import into your workspace.

Before you begin

Before importing a SAR file, determine if the SAR file contains required Java source files. When you import a SAR file into an existing SIP project, the imported SIP deployment descriptor files are either not changed or are overwritten by the ones that are included in the imported SAR file, based on your response to the prompt. In either case, this action does not represent a merging of the two sets of deployment descriptors.

Procedure

To import the SIP project resources in a SAR file into your workspace:

1. Click **File** > **Import**.
2. In the Import dialog box, select **SAR file** and click **Next**.
3. Browse to the SAR file that you want to import.
4. By default, the wizard creates a new SIP project with the same name as the SAR file. If you accept this choice, the project is created with the same servlet version that is specified by the SAR file, in the same location.
5. To facilitate deployment, associate your imported SIP application with a new or existing Enterprise Application project (EAR Project) by selecting **Add Project to an EAR** file.
6. Click **Finish** to populate the SIP project.

Related concepts:

SIP overview

Related tasks:

Exporting servlet archive (SAR) files

Importing SIP applications in EAR or WAR files

Exporting servlet archive (SAR) files

A servlet archive (SAR) file is a packaged Session Initiation Protocol (SIP) application. You can export this file to test, publish, and deploy the resources developed within the SIP project.

Procedure

To export a SIP project into a SAR file:

1. Right-click a SIP project folder and select **Export** from the pop-up menu.
2. Select **SAR file** in the Export window and click **Next**.
3. Complete the fields and click **Finish**.

Related concepts:

SIP overview

Related tasks:

Importing servlet archive (SAR) files

Importing SIP applications in EAR or WAR files

Importing SIP applications in EAR or WAR files

If you import EAR or WAR files that contain a SIP application, then you must add the SIP facet to the project.

About this task

If you import a Web archive (WAR) file into your workspace, a web application project is created, not a SIP project. The Session Initiation Protocol (SIP) facet is not added automatically and you must add it manually.

Procedure

1. Click **File** > **Import**.
2. In the Import window:
   - To import and EAR file, click **Java EE** > **EAR file** and click **Next**.
   - To import a WAR file, click **Web** > **WAR file** and click **Next**.
3. Browse to the EAR or WAR file that you want to import.
4. Add the SIP facet to your project.

Related concepts:

SIP overview

Related tasks:

Importing servlet archive (SAR) files

Exporting servlet archive (SAR) files

Developing enterprise applications

The workbench provides the tools you need to develop enterprise applications. You can use the Java™ EE tools and features to create applications that are structured around modules with different purposes, such as Web sites and Enterprise Java beans (EJB) applications. When you use EJB 3.1 components, you can create a distributed, secure application with transactional support. When you develop applications that access persistent data, you can use the new Java Persistence API (JPA). This standard simplifies the creation and use of persistent entities, as well as adding new features. For developing presentation logic, you can use technologies such as JavaServer Pages (JSP) or JavaServer Faces (JSF). You can use the workbench tools to create applications that connect to enterprise information systems (EIS), using JCA compliant connectors (J2C resource adapters).

Overview

You can read the following topics before creating a enterprise application. They provide planning and technology overview information that may be useful if you are new to enterprise applications or developing enterprise applications in this development environment.

- **Overview of Java EE: Overview**
- **Developing EJB 3.1 Applications**
- **Developing JPA applications**
- **JPA architecture**
- **Connecting to enterprise information systems**
- **Resource Adapters**

Getting started

If you are already familiar with enterprise applications technology the following topics will help you set up your workspace for enterprise applications development, and guide you through the development process.

- **Tools for Java EE development**
- **Creating Java EE projects using wizards**
- **Creating and configuring Java EE modules using annotations**
- **Creating EJB 3.1 applications**
- **Creating JPA applications**
- **Context and dependency injection (CDI) Overview**
- **Creating J2C Applications**

Samples and tutorials

The following enterprise applications samples and tutorials are included with this product:

- **Sample: EJB 3.1 Counter**
  - This sample creates simple counter application that runs on WebSphere® Application Server v7.0.
- **Tutorial: EJB 3.1 counter application**
  - This tutorial creates simple counter application that runs on WebSphere Application Server v8.0.

Web resources for learning

In addition to the information found in this information center, the following links provide additional learning material.

Rational® Rational Application Developer for WebSphere Software V8 Programming Guide

Experience JEE Using Rational® Application Developer V7.5

Latest developerWorks® articles and tutorials about enterprise applications

Latest developerWorks articles and tutorials about Java beans

Latest developerWorks articles and tutorials about J2C applications

Developing Java EE Applications

The Java™ EE programming model simplifies the process of creating Java applications.

Java Enterprise applications (Java EE applications) are applications that conform to the Java Platform, Enterprise Edition (Java EE) specification. Prior to Java EE, the specification name was Java 2 Platform, Enterprise Edition (J2EE). The term Java EE includes Java EE and J2EE specifications.

In the Java EE specifications, programming requirements have been streamlined, and XML deployment descriptors are optional. Instead, you can specify many details of assembly and deployment with Java annotations. Java EE provides default values in many situations so that explicit specification of these values is not required.

Code validation, content assistance, Quick Fixes, and refactoring simplify working with your code. Code validators check your projects for errors. When an error is found, you can double-click it in the Problems view in the product workbench to go to the error location. For some error types, you can use a Quick Fix to correct the error automatically. For both Java source and Java annotations, you can rely on content assistance to simplify your programming task. When you refactor source code, the tools automatically update the associated metadata.

For additional information about Java EE, see the official specification:

- **Java EE 5**: JSR 244: Java Platform, Enterprise Edition 5 (Java EE 5) Specification
- **Java EE 6**: JSR 316: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification

Related concepts:

Java EE: Overview
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications
Securing enterprise applications

Related tasks:

Setting Java EE preferences
Creating and configuring Java EE projects using wizards
Validating code in enterprise applications
Deploying Java EE applications
Migrating the specification level of Java EE projects

Java EE: Overview

Using the Java™ Platform, Enterprise Edition (Java EE) architecture, you can build distributed web and enterprise applications. This architecture helps you focus on presentation and application issues, rather than on systems issues.
You can use the Java EE tools and features to create applications that are structured around modules with different purposes, such as web sites and Enterprise Java bean (EJB) applications. When you use EJB 3.1 components, you can create a distributed, secure application with transactional support. When you develop applications that access persistent data, you can use the Java Persistence API (JPA). This standard simplifies the creation and use of persistent entities. For developing presentation logic, you can use technologies such as JavaServer Pages (JSP) or JavaServer Faces (JSF).
Using the Java EE Platform Enterprise Edition (Java EE), you can develop applications more quickly and conveniently than in previous versions. Java EE significantly enhances ease of use providing
- Reduced development time
- Reduced application complexity
- Improved application performance

Java EE provides a simplified programming model, including the following tools:
- Inline configuration with annotations, making deployment descriptors now optional
- Dependency injection, hiding resource creation, and lookup from application code
- Java persistence API (JPA) allows data management without explicit SQL or JDBC
- Use of plain old Java objects (POJOs) for Enterprise Java beans and web services

Java EE provides simplified packaging rules for enterprise applications:
- Web applications use .WAR files
- Resource adapters use RAR files
- Enterprise applications use .EAR files
- The `lib` directory contains shared .JAR files
- A .JAR file can be specified in the application.xml of the EAR either as an application client or as an EJB module
- A .JAR file not specified by the application.xml of the EAR is defined as follows:
  - A .JAR file with an application-client.xml implies an application client
  - A .JAR file with an ejb-jar.xml implies an EJB module
  - A .JAR file with `META-INF/MANIFEST.MF` specified Main-Class implies an application client
  - A .JAR file with any `@Stateless`, `@Stateful`, or `@MessageDriven` annotations implies an EJB application
- A .JAR file with `Main-Class` implies an application client
- A .JAR file with `@Stateless` annotation implies an EJB application
- Many simple applications no longer require deployment descriptors, including
  - EJB applications (.JAR files)
  - Web applications that use JSP technology only
  - Application clients

  - Enterprise applications (.EAR files)

Java EE provides simplified resource access using dependency injection:
- In the Dependency Injection pattern, an external entity automatically supplies an object's dependencies.
  - The object need not request these resources explicitly
- In Java EE, dependency injection can be applied to all resources that a component needs
  - Creation and lookup of resources are hidden from application code
- Dependency injection can be applied throughout Java EE technology:
  - EJB containers
  - Web containers
  - Clients
  - Web services

Related concepts:
Developing Java EE Applications
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications
Securing enterprise applications

Related tasks:
Setting Java EE preferences
Creating and configuring Java EE projects using wizards
Validating code in enterprise applications
Deploying Java EE applications
Migrating the specification level of Java EE projects

Java EE: What's new in this release

This release adds support for Java™ EE 6, and includes support for the new specifications such as integrated support for scripting languages and web services, improved JDBC features, and an integrated Derby database (in the SDK release), as well as some management features and performance enhancements.
Java EE 6 provides a number of new or enhanced features, which this release supports:

Web services

- Web services offer support for writing XML web service client applications.
- You can expose your APIs as .NET interoperable web services with a simple annotation.
- Java SE 6 adds new parsing and XML to Java object-mapping APIs, previously only available in Java EE platform implementations or the Java Web Services Pack.

Support for Scripting Languages

- Java EE 6 comes with built-in support for scripting languages. You can embed scripts in various scripting languages into your Java applications, passing parameters, evaluating expressions, and retrieving results.
- You use the `ScriptEngine` object to contain your scripts:

```
ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("js");
```

Here `js` refers to JavaScript.

JDBC Enhancements

- The Java SE 6 development kit includes the all-Java JDBC database, Java DB based on Apache Derby.
- JDBC includes the updated JDBC 4.0 API, which includes many important improvements, including special support for XML as an SQL datatype and better integration of Binary Large OBjects (BLOBs) and Character Large OBjects (CLOBs)
- In Java 6, you no longer need to explicitly load the JDBC class. The line: `Class.forName("oracle.jdbc.driver.OracleDriver")` has been replaced with a simple declaration: `Connection conn = DriverManager.getConnection("jdbc:derby:TestDB");`

Monitoring and Management

- Java EE 6 improves the Java Monitoring and Management Console, or JConsole.
- The graphical look of JConsole has been improved; you can now monitor several applications in the same JConsole instance, and the summary screen has been redesigned.
- In Java 6, you can monitor any application that is running in a Java 6 virtual machine.
- Java 6 comes with sophisticated thread-management and monitoring features as well.

- The `OutOfMemoryError` will not just leave you guessing; it will print out a full stack trace so that you can have some idea of what might have caused the problem.

Managing the File System

- Java 6 gives you much finer control over your local file system.
- The `java.io.File` class has the following three new methods to determine the amount of space available (in bytes) on a given disk partition:
- The java.io.File class now has a set of functions allowing you to set the readable, writable, and executable flags on files in your local file system, as you would with the Unix `chmod` command.

Security

- This release adds the XML-Digital Signature (XML-DSIG) APIs for creating and manipulating digital signatures. It has also added various new ways to access platform-native security services, including
  - Native Public Key Infrastructure (PKI) and cryptographic services on Microsoft Windows for secure authentication and communication,
  - Java Generic Security Services (Java GSS) and Kerberos services for authentication,
  - Access to LDAP servers for authenticating users.

Changes in the Java EE 6 APIs

WebBeans 1.0: One of the most groundbreaking API developed in Java EE 6, WebBeans fills a number of gaps in Java EE. WebBeans

- Unifies the JSF, JPA and EJB 3 programming models into a single, well-integrated platform. WebBeans registers EJB 3 beans, JPA entities, and JavaBeans as WebBeans components, which are accessible via EL and are injectable into each other.
- Brings a robust set of dependency injection features to the platform.
- Enhances the Java EE Interceptor model by adding the ability to bind interceptors to annotations instead of having to bind interceptors to target object classes themselves.

JSF 2.0: JSF 2.0 focuses on ease-of-use, on innovation, and on increasing the feature set. JSF 2.0

- Adds Facelets as a new technology.
- JSF 2.0 brings Java EE 5- style annotation-driven configuration to the table using annotations such as `@ManagedBean`and `@ManagedProperty`.
- Provides full AJAX support including partial page processing to handle AJAX events.
- Provides a built-in capability to handle resources such as images, JavaScript files, and CSS.

EJB 3.1: EJB 3.1 continued to make EJB as simple as possible while adding meaningful business component services. EJB 3.1

- Made business interfaces optional, even for Session Beans.
- Adds the concept of Singleton Beans. Because they are intended for managing shared application state, they are completely thread-safe by default; at the same

time, EJB 3.1 adds declarative concurrency controls for greater flexibility.
- Adds support for cron-style scheduling.
- Adds the capability to invoke Session Bean methods asynchronously via the @Asynchronous annotation.
- Introduces the concept of EJB 3.1 Lite to add a smaller subset of the EJB API geared towards the Web Profile. While EJB Lite includes features like transactions and security, it does not include features like messaging, remoting and scheduling.
- Introduces a standard JNDI name: `java:global/app/module/beah#interface`.

JPA 2.0: In Java EE 6, JPA has been officially separated from EJB as a distinct API in its own right. JPA 2.0
- Adds a number of ORM mapping enhancements, such as the ability to model collections, maps, and lists using the @ElementCollection annotation and the ability to map unidirectional one-to-many relationships.
- Enhances both the EntityManager and Query APIs to support things like retrieving the first result, specifying the maximum size of query results, getting access to the underlying vendor-specific entity manager and query objects, and pessimistic locking.
- Enhanced JPQL with SQL-like CASE, NULLIF, COALESCE
- Adds a Criteria API.

Servlet 3.0: Servlet 3.0 embraces the Java EE 5 model. Servlet 3.0
- Introduces annotations such as @WebServlet, @WebFilter and @WebListener. This reduces web.xml configuration to the point that it can be eliminated altogether.
- Introduces the idea of web fragments. For more information, see: Creating web fragment projects.
- Adds the ability to programmatically add Servlets, Filters and Listeners through the ServletContext.
- Follows EJB 3.1 in making deployment descriptors completely optional.
- Provides support for EJB 3.0 and 3.1 beans in web 3.0 projects.

JAX-RS 1.1: JAX-RS 1.1 is the REST counterpart of JAX-WS. JAX-RS 1.1
- Uses the @Path annotation to determine the URL that a JAX-RS resource can be accessed.
- Maps input from sources like URL query parameters, parts of the URL, cookies, and HTTP header values.
- Uses the @Produces annotation to tell JAX-RS what the content type of returned values are such as text/xml and text/json.
- Integrates with Servlets, WebBeans and EJB

Tools for Java EE development

The Java™ EE development tools enable you to create enterprise applications including enterprise application projects, EJBs, and Java persistent API (JPA) applications.

The workbench provides a rich set of tools to create, develop, test, debug, and deploy enterprise applications. Some of the Java EE tools include:

- **Enterprise explorer view**: The Enterprise explorer view allows you to manage and maintain your enterprise applications in one location.
- **Annotations view**: The Annotations view provides a way for you to create, edit, browse, and generally keep track of the annotations that you use in your applications.
- **Project and object creation wizards**: You can use Java EE wizards to create enterprise applications, including application client projects, Enterprise Java bean projects, connector projects, web projects, web fragment projects, and web Services applications.
- **Export and import wizards**: You can import existing projects and modules into your current project; you can use export wizards to build, package, and export projects with a single click.
- **Code validation, content assistance, Quick Fixes, and refactoring**: These tools simplify working with your code. Code validators check your projects for errors. When one is found, you can double-click it, in the Problems view in the product workbench, to go to the error location. For some types of error, you can also choose a Quick Fix, which automatically corrects the error. For both Java™ source and Java annotations, you can rely on content assistance to simplify your programming task. When you refactor source code, the tools automatically update the associated metadata.
- **Tools for managing projects**: The tools for Java EE allow you to manage projects easily. To share applications outside of a version-control system, you can import and export projects as archive files. To migrate your projects to the current version of the Java EE specification, you can take advantage of a migration wizard.

Related concepts:

Developing Java EE Applications
Java EE: Overview
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications
Securing enterprise applications
Enterprise explorer view in the Java EE perspective
Java EE perspective

Related tasks:

Setting Java EE preferences
Creating and configuring Java EE projects using wizards
Validating code in enterprise applications

Java EE perspective

While developing enterprise applications in the Java™ EE perspective, the enterprise explorer view is your main view of your Java EE projects and resources. The Java EE perspective contains a selection of views and editors that are customized to be most useful to a Enterprise developer.

Views

The Java EE perspective includes workbench views that you can use when developing resources for enterprise applications, EJB modules, web modules, web fragment modules, application client modules, and connector projects or modules:

- **Enterprise explorer view**: The Enterprise Explorer view provides an integrated view of your projects and their artifacts related to Java EE development. You can show or hide your projects based on working sets. This view displays navigable models of Java EE deployment descriptors, Java artifacts (source folders, packages, and classes), navigable models of the available web services, and specialized views of web modules to simplify the development of web applications. In addition, EJB database mapping and the configuration of projects for a Java EE application server are made readily available.
- **Annotations view**: The Annotations view provides a way for you to create, edit, browse, and generally keep track of the annotations that you use in your applications.
- **Outline view**: The Outline view in the Java EE perspective shows the outline of the file that you are editing. For example, if you are editing an enterprise bean in the Java editor, the Outline view shows the outline for the Java class.
- **Tasks view:**The Tasks view lists the to-do items that you have entered.
- **Problems view**: The Problems view displays problems, warnings, or errors associated with the selected project. You can double-click an item to address the specific problem in the appropriate resource.
- **Properties view**: The Properties view provides a tabular view of the properties and associated values of objects in files you have open in an editor.
- **Servers view**: The Servers view shows all the created server instances. You can start and stop each server from this view, and you can launch the test client.
- **Snippets view**: The Snippets view provides categorized pieces of code that you can insert into appropriate places in your source code.
- **Data Source Explorer:**The Data Source Explorer provides a list of configured connection profiles. If categories are enabled, you can see the list grouped into categories. Use the Data Source Explorer to connect to, navigate, and interact with resources associated with the selected connection profile. It also provides import and export capabilities to share connection profile definitions with other Eclipse Workbenches.
- **Status bar**:The Status bar provides a description of the location of selected objects in the Enterprise Explorer views in the left side. When file and deployment descriptors are open, the status bar shows the read-only state of the files and the line and column numbers when applicable. Sometimes when long operations run, a status monitor appears in the status bar, along with a button with a stop sign icon. Clicking the stop sign stops the operation when the operation can be canceled.

Editors

Editors are workplace tools that allow you to edit the various types of files contained in your project. Depending on the type of file that is being edited, the appropriate editor is displayed in the editor area. For example, if a .TXT file is being edited, a text editor is displayed in the editor area. The following list contains some of the editors available to you in the Java EE development environment:

- **Deployment Descriptor Editors**: The editor includes editors for managing your various Java EE projects types.
- **Java Editor**: The editor includes the following features:
  - Syntax highlighting
  - Content and code assist
  - Code formatting
  - Import assistance
  - Quick fix
  - Integrated debugging features
- **Ant editor**: The editor includes the following features:
  - Syntax highlighting
  - Content and code assist (including Ant-specific templates)
  - Annotations
- **XML editor**: The XML editor is a tool for creating and viewing XML files. You can use it to perform various tasks, such as:
  - Creating new, empty XML files, or generating them from existing DTDs or existing XML schemas
  - Editing XML files
  - Importing existing XML files for structured viewing
  - Associating XML files with DTDs or XML schemas
- **Properties files editor**: The Properties files editor is a tool for creating and viewing Properties files. The editor includes the following features:
  - Syntax highlighting
  - Code formatting

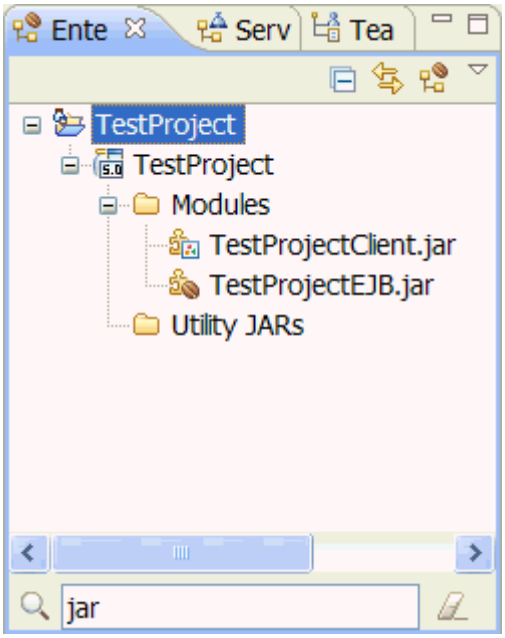
Related concepts:
Tools for Java EE development
Enterprise explorer view in the Java EE perspective

Related tasks:
Select working sets

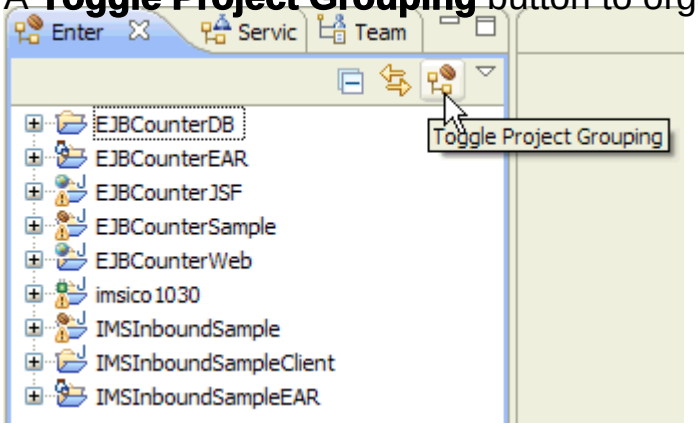
Enterprise explorer view in the Java EE perspective

While developing enterprise applications in the Java™ EE perspective, the enterprise explorer view is the main view of your Java™ EE projects and resources. The enterprise explorer view in the Java EE perspective is a view that allows you to manage and maintain your enterprise applications in one location. The enterprise explorer view provides you with

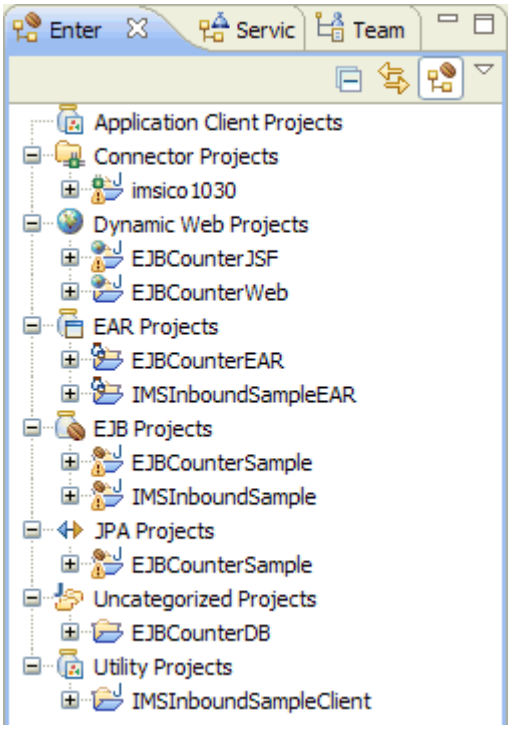
- An integrated view of all project resources, including models of Java EE deployment descriptors (if you are using deployment descriptors), Java artifacts, resources, web services, databases, and web project artifacts.
- A way to view all the modules associated with your project.
- Search bar capabilities that allow you to filter the files in your project folders according to the search parameter you type in the search bar. For example, if you type `xml` in the search bar and click **Search**, only the xml files show up in the Enterprise Explorer. If you type `jar` in the search bar and click **Search**, only the jar files in your project (excluding the jars on the classpath) show up:



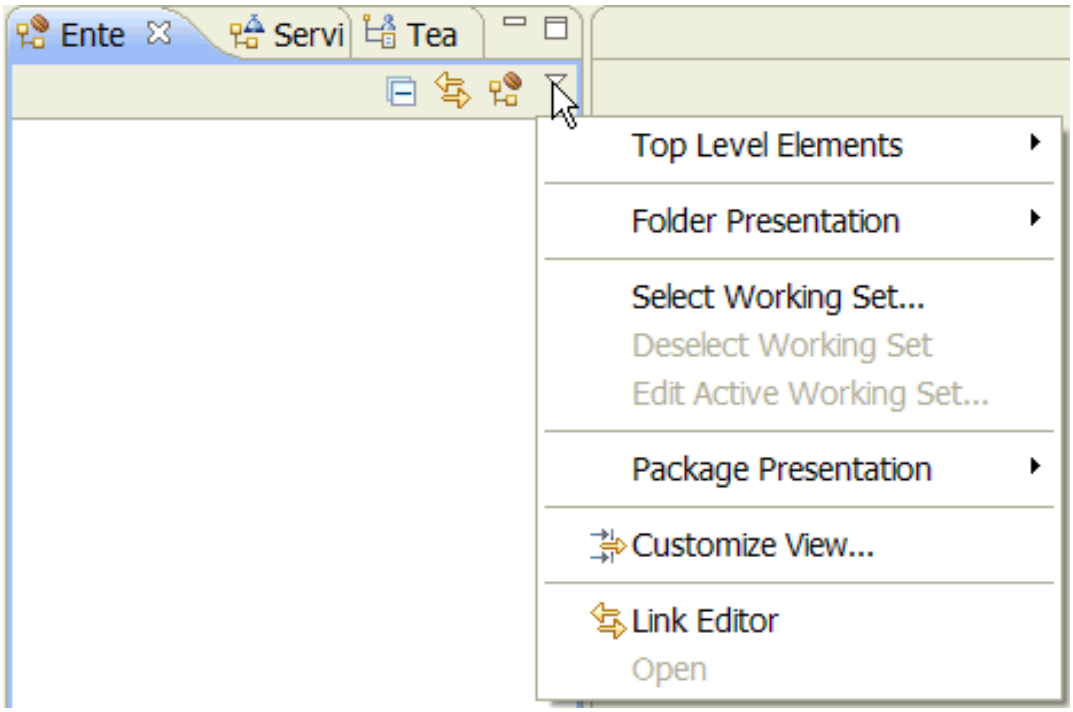
- A **Toggle Project Grouping** button to organize your projects by project type:



The result is a tree organized by project type:



- Customization options, so that you can select what functionality you want available in the enterprise explorer view. Click the down arrow at the top right of the Enterprise Explorer view, and select **Customize View...**:



- Version control management information (VCM) can be turned on and off from the Preferences page: **Window** > **Preferences** > **General** > **Appearance** > **Label decorations**.
- Classpath module dependencies can be modified by selecting **Window** > **Preferences** > **General** > **Appearance** > **Label decorations**
- View filtering is supported by clicking the down arrow at the top right of the Enterprise Explorer view, selecting **Customize View...** > **Filters**. Resources can be filtered by name, project type, or content type. Files beginning with a period are

filtered out by default.
- The status line shows the full path of the selected resource.
- Errors and warnings on resources (including Java, HTML/JSP, and Links Builder errors and warnings) are indicated with a red error or yellow warning next to the resource with the error, as well as the parent containers up to the project.

Related concepts:
Tools for Java EE development
Java EE perspective

Related tasks:
Select working sets

Select working sets

You can select projects or components to group in working sets in the workspace.

Before you begin

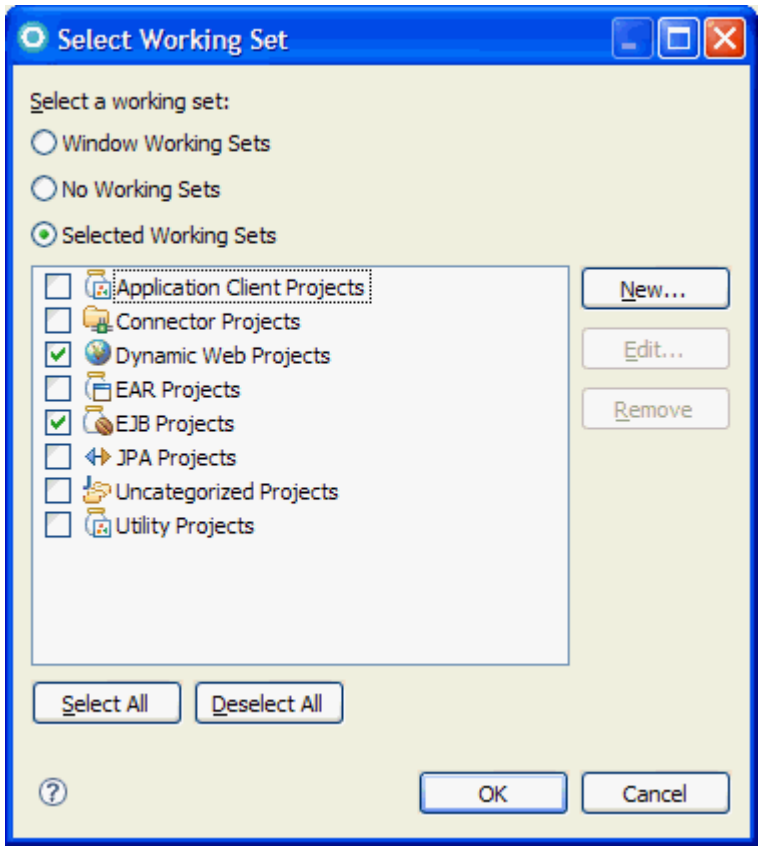
You need to have a project or several projects created within your workspace.

About this task

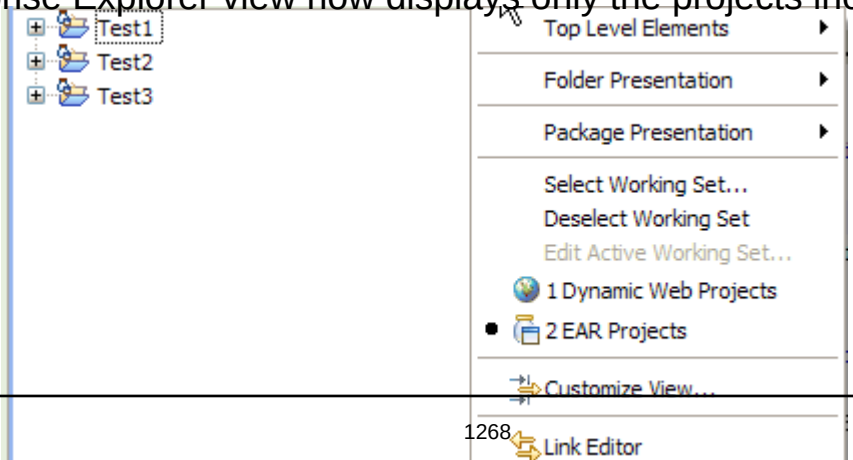
Working sets group elements for display in views or for operations on a set of elements. The navigation views use working sets to restrict the set of resources that are displayed. If a working set is selected in the navigator, only resources, children of resources, and parents of resources contained in the working set are shown.

Procedure

1. In the Enterprise Explorer view of the Java™ EE perspective, click the View Menu icon, , and select **Select working Set**.
2. Select the type of project or projects that you want to group in your working set:



3. Your Enterprise Explorer view now displays only the projects included in your working set:



4. You can create more than one working set, and you can toggle back and forth between a number of working sets in your workspace.

What to do next

For more information about working sets, see Working sets

Related concepts:
Tools for Java EE development
Enterprise explorer view in the Java EE perspective
Java EE perspective

Setting Java EE preferences

Before you begin developing Java™ EE applications, you can optimize the workbench for Java enterprise development by setting a variety of preferences.

Procedure

1. Click **Window** > **Preferences** to open the Preferences page.
2. Expand Java EE, and click the preference category that you want to set.
3. Select the elements that you want to set as the default when creating your Java EE application. These Java EE settings include: JET templates:
   - Use dynamic translation of JET templates (default: cleared)

   Deployment:
   - Perform Incremental Deployment (default: checked)

   Classpath containers:
   - Use Ear Libraries classpath container (default: checked)
     - Use Java Build Path to control EAR Libraries export (default: Cleared)
   - Use Web App Libraries classpath container (default: checked)

   Classpath checking:
   - Duplicate Java EE Module dependency classpath entries:
     - Error
     - Warning (default)
     - Ignore
   - Duplicate Web Library dependency classpath entries:
     - Error
     - Warning (default)
     - Ignore
   - Missing Java EE Modules dependency classpath entries:
     - Error
     - Warning (default)
     - Ignore
   - Missing Web Library dependency classpath entries:
     - Error
     - Warning (default)
     - Ignore

   Note: For information about Java EE classpath Container Preferences, seeJ2EE classpath Container Preferences with Java Build Path and J2EE Module Dependencies Properties

4. Click **Apply** to apply the changes and click **OK** to close the Preferences page.

What to do next

For information about other Java EE preferences, refer to the these topics:
- Selecting default project structures

- Setting Java compiler compliance
- Selecting default security settings

Related concepts:
Developing Java EE Applications
Java EE: Overview
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications
Securing enterprise applications

Selecting default project structures

You can use the Preferences page to set your default project structures.

Procedure

1. Click **Window** > **Preferences** to open the Preferences page.
2. Expand Java™ EE, and select **Project**
3. Select the elements that you want to set as the default when creating your Java EE application. These project settings include: Enterprise Application Membership:
   - Add project to an EAR (default: checked)

   Content directory (indicates the default directory in your workspace where the indicated project is stored. You can change the name of the default directory here):
   - Enterprise Application Project:
     - Content Directory: (default: unspecified)
   - Web Project:
     - Default Source Folder: (default: src)
     - Output Folder: (default: WebContent/WEB-INF/classes)
     - Content Directory: (default: WebContent)
   - EJB Project:
     - Default Source Folder: (default: ejbModule)
     - Output Folder: (default: build\classes)
   - Application Client Project:
     - Default Source Folder: (default: appClientModule)
     - Output Folder: (default: build\classes)
   - Connector Application Project:
     - Default Source Folder: (default: connectorModule)
     - Output Folder: (default: build\classes)
   - Utility/JPA Project:
     - Default source folder: (default: src)
     - Output Folder: (default: src)

   Generate Deployment Descriptor for Java EE 5.0 or above Projects (Select this option if you want a deployment descriptor to be automatically created when you create the project type indicated):
   - Enterprise Application Project (default: Cleared)
   - EJB Project (default: Cleared)
   - Web 2.5 Project (default: checked)
   - Web 3.0 Project (default: cleared)
   - Application Client Project (default: cleared)
   - Connector Project (applies for 1.6 only) (default: checked)

4. Click **Apply** to apply the changes and click **OK** to close the Preferences page.

Related tasks:

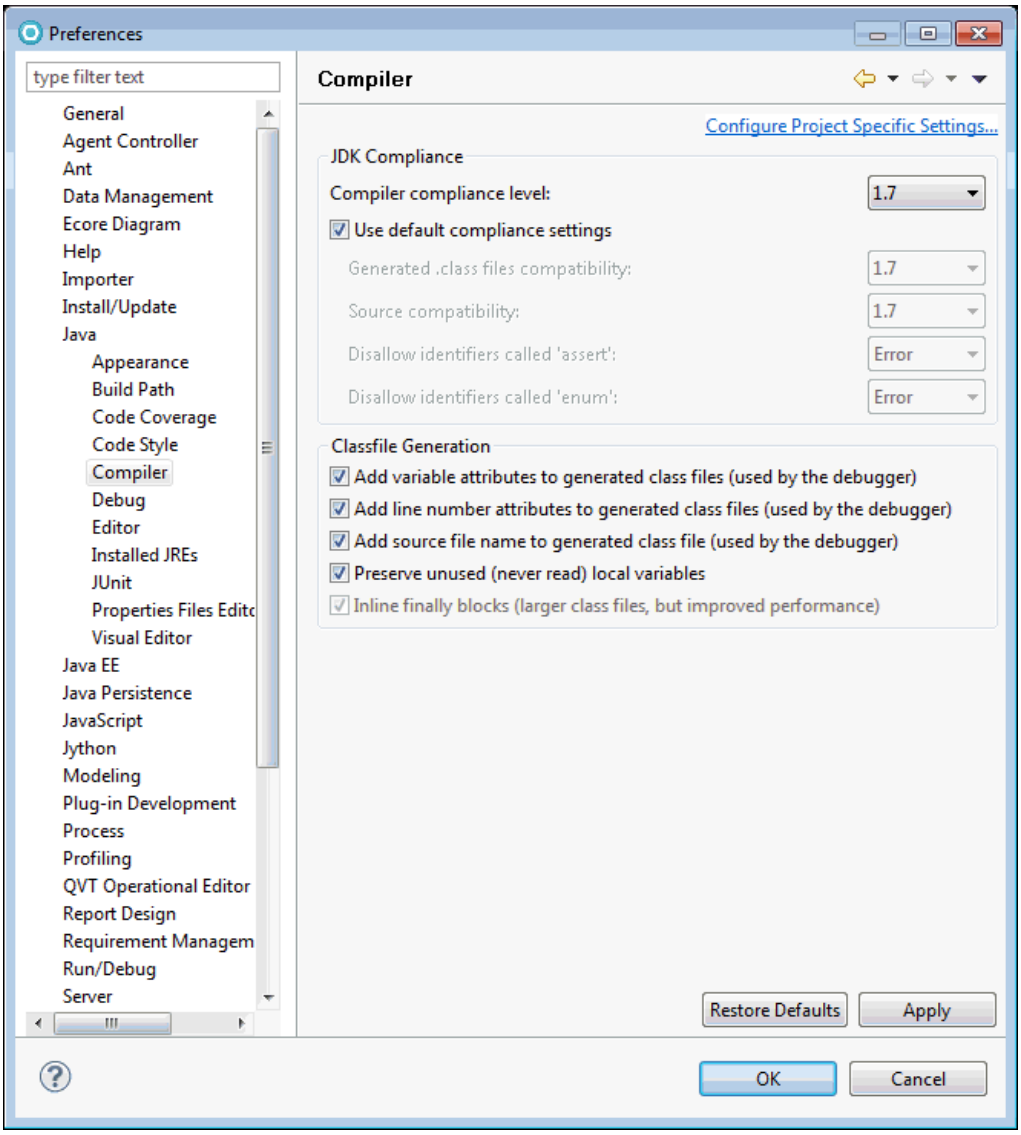
Setting Java compiler compliance

Setting Java compiler compliance

Use the Workspace Preferences page to set the Java™ compiler compliance.

Procedure

1. To open the workspace preferences page, select **Window** > **Preferences** > **Java** > **Compiler**. The Compiler page opens:



2. In the Compiler compliance field, select the compiler level that you want to use for your application. You must select the compiler level that is supported by the version of the WebSphere® Application Server that you are deploying to.
   A. If you are using WebSphere Application Server version 8.5, select Java 1.6 or Java 1.7 compiler compliance.
   B. If you are using WebSphere Application Server version 7.0 or 8.0, select Java 1.6 compiler compliance.
   C. If you are using WebSphere Application Server version 7.0, select Java 1.6 compiler compliance.
   D. If you are using WebSphere Application Server version 6.1 with the fix pack for EJB 3.0, select Java 1.5 compiler compliance. For more information about Compiler versions supported by WebSphere Application Server, see Testing

and publishing on your server

Related tasks:
Selecting default project structures

Selecting default security settings

You can set your default security settings in the Preferences page.

Procedure

1. Click **Window** > **Preferences** to open the Preferences page.
2. Expand Java™ EE, and select **Security**.
3. Select the elements that you want to set as the default when creating your Java EE application: These security settings include: Default HTTP method access:
   - GET (default: checked)
   - PUT (default: checked)
   - HEAD (default: checked)
   - TRACE (default: checked)
   - POST (default: checked)
   - DELETE (default: checked)
   - OPTIONS (default: checked)

   Default wildcard pattern:
   - /*
   - Apply the registered Faces servlet mapping to all generated wildcard patterns if the project is a Faces project. (default: checked)

   Prompts:
   - Prompt to create the Web Modules first Security Role. (default: checked)
   - Show annotation warning dialog for Web 2.5 and EJB 3.1 projects. (default: checked)

4. Click **Apply** to apply the changes and click **OK** to close the Preferences page.

Project facets

Facets define characteristics and requirements for Java™ EE projects and are used as part of the runtime configuration.

When you add a facet to a project, that project is configured to perform a certain task, fulfill certain requirements, or have certain characteristics. For example, the EAR facet sets up a project to function as an enterprise application by adding a deployment descriptor and setting up the project's classpath.
You can add facets to Java EE projects and other types of projects that are based on Java EE projects, such as enterprise application projects, web projects, and EJB projects. You can add facets to a Java project, by selecting the properties page of the project and selecting **Project Facets** > **Convert to faceted form...**. Typically, a facet-enabled project has at least one facet when it is created, allowing you to add more facets if necessary. For example, a new EJB project has the EJB Module facet. You can then add other facets to this project like the EJBDoclet (XDoclet) facet. To add a facet to a project, see Adding a facet to a Java EE project.
Some facets require other facets as prerequisites. Other facets cannot be in the same project together. For example, you cannot add the Web Module facet to an EJB project because the EJB project already has the EJB Module facet. Some facets can be removed from a project and others cannot.
Facets also have version numbers. You can change the version numbers of facets as long as you stay within the requirements for the facets. To change the version number of a facet, see Changing the version of a facet.
For information about Project Facets, .settings folder and Builders, see What is new in version 7.0 (Eclipse v3.2/WTP v1.5+) J2EE Projects: Project Facets, .settings folder and Builders.

Adding a facet to a Java EE project

This topic explains how to add a facet to an existing project in your workspace.

When you add a facet to a project, that project is configured to perform a certain task, fulfill certain requirements, or have certain characteristics. For example, the EAR facet sets up a project to function as an enterprise application by adding a deployment descriptor and setting up the project's classpath.
New projects generally have facets added to them when they are created. To add another facet to a project that already exists, complete the following steps:

1. In the Project Explorer view of the Java™ EE perspective, right-click the project and then select **Properties**.
2. Select the **Project Facets** page in the in the Properties window. This page lists the facets in the project and their versions.
3. Click **Modify Project** and select the check boxes next to the facets you want the project to have.Only the facets that are valid for the project are listed:
   - The list of runtimes selected for the project limits the facets shown in the list. Only the facets compatible with all selected target runtimes are shown.
   - The currently selected facets and their version numbers limit the other facets shown in the list. For example, if the project contains the Dynamic Web Module facet, the EJB Module facet is not listed because these two facets cannot be in the same project.

   You can find out more about the requirements and limitations for each facet by right-clicking the facet name and then clicking **Show Constraints**.
   You can also choose a preset combination of facets from the **Presets** list.
4. Choose a version number for the facet by clicking the current version number and selecting the version number you want from the drop-down list.
5. **Optional:** To remove a facet, clear its check box. Not all facets can be removed.
6. **Optional:** If you want to limit the project so it will be compatible with one or more runtimes, click on the Runtimes tab and select the runtimes that you want the project to be compatible with. For more information on runtimes, see Specifying target servers for J2EE projects.
7. Click **Finish** to exit the Modify Faceted Project dialog and then click **OK**.

Related concepts

Project facets

Related tasks

Changing the Java compiler version for a Java EE project

Changing the version of a facet

Changing the version of a facet

You can change the version of a facet in a Java EE project by editing the facets for the project.

Changing the Java™ compiler version of a Java EE project involves changing the version of the **Java** facet. For more information, see Changing the Java compiler version for a Java EE project.

To change the version of a facet in your project, complete the following steps:

1. In the Project Explorer view of the Java EE perspective, right-click the project and then select **Properties**.
2. Select the **Project Facets** page in the in the Properties window. This page lists the facets in the project and their versions.
3. Click **Modify Project** and click the facet you want to change.
4. Select the version of the facet from the drop-down box next to the facet's name.
5. Click **Finish** to close the Modify Faceted Project window and then click **OK**.

Related concepts

Project facets

Related tasks

Adding a facet to a Java EE project

Changing the Java compiler version for a Java EE project

Changing the Java compiler version for a Java EE project

You can change the version of Java™ used in a Java EE project by changing the value of the **Java** facet.

The **Java** facet applies only to J2EE projects. To set the Java compiler level of a non-Java EE project, such as a Java project, see Java Compiler.

To change the Java compiler version, complete the following steps:

1. In the Project Explorer view of the Java EE perspective, right-click the project and then select **Properties**.
2. Select the **Project Facets** page in the in the Properties window. This page lists the facets in the project and their versions.
3. Click **Modify Project**.
4. Double click the version number next to the**Java** facet to select a different level of Java compiler.
5. Click **Finish** to close the Modify Faceted Project window and then click **OK**.

Related concepts

Project facets

Related tasks

Adding a facet to a Java EE project

Changing the version of a facet

Creating and configuring Java EE projects using wizards

You can use wizards to create Java™ EE projects in your workspace.

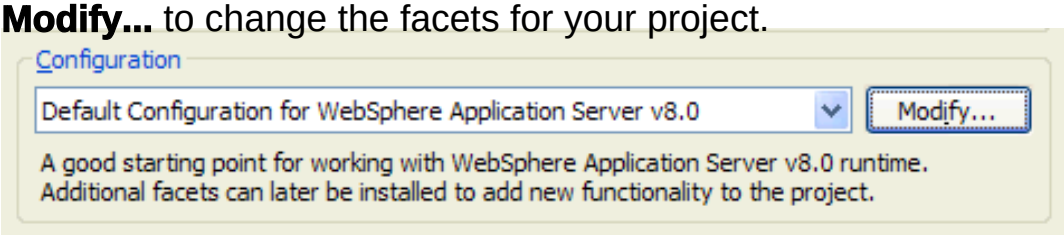
Before you begin

If you do not see the Java EE icon, , in the top right tab of the workspace, you need to switch to the Java EE perspective.

Procedure

1. To use the Java EE wizard from the menu bar, select **File** > **New**, and select the type of enterprise application you want to create:
   - Enterprise Application Project
   - Web Project
   - EJB Project
   - Connector Project
   - Application Client Project

   To create Utility projects, select **File** > **New** > **Project ...** > **Java EE** > **Utility Project**. To create Web fragment projects, select **File** > **New** > **Project ...** > **Web** > **Web Fragment Project** . Click **Next**.
2. In the **Project Name** field, type a name for your project.
3. In the **Target runtime** field, select your target runtime, or click **New** to create a new runtime.
4. In the **<module-type> Module version** field, you can accept the default value or change the module version number. This field controls the primary facet version for your project. The available versions are filtered based on the target runtime that you selected in the **Target runtime** field. For example, if the target runtime is WebSphere® Application Server version 6.1, then the web project versions available are versions 2.2, 2.3, and 2.4. If the target runtime is WebSphere Application Server version 7.0, then the web project versions available are versions 2.2, 2.3, 2.4, and 2.5. If the target runtime is WebSphere Application Server version 8., then the web project versions available are versions 2.2, 2.3, 2.4, 2.5, and 3.0. The module version here is like clicking **Modify...** beside the **Configuration** field.
5. In the **Configuration** field, you can accept the default configuration, select a pre-defined project configuration from the **Configurations** drop-down list, or click **Modify...** to change the facets for your project.

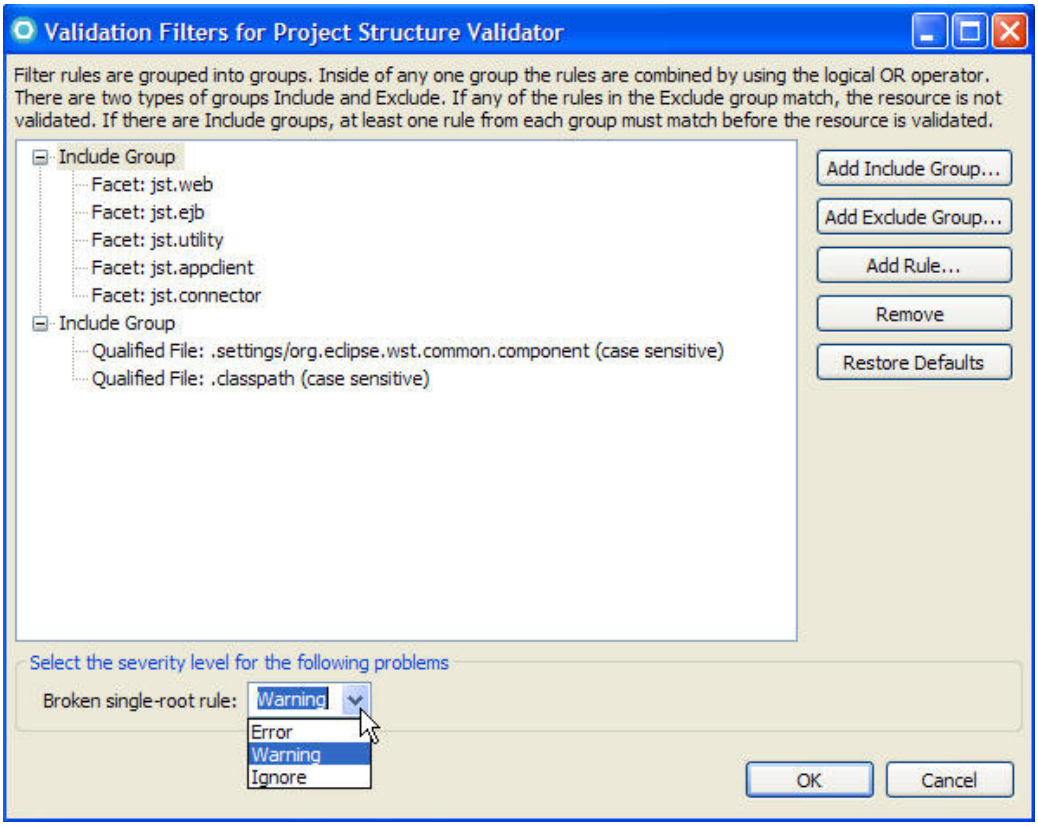


If you plan to create JPA entities in the EJB 3.1 project, for example, select **Java Persistence**. Save this configuration with a meaningful name, for example, `EJBDevelopmentWithJPA` so that you can reference this configuration in any EJB 3.1 projects that are subsequently created.

A. Select one or more project facets from the **Project Facets** list.
B. To specify server runtime environments, click **Runtimes** and select one or more runtimes.
C. After making your selections, you can save your custom configuration by clicking **Save**.

6. Optional: Select **Add project to an EAR module** in the **EAR Membership** field to add the new module to an enterprise module (EAR) project. Type a new project name or select an existing enterprise module project from the drop-down list in the **EAR Project Name** combination box. Or, click **New** to launch the New EAR module Project wizard.
7. In the **Content Directory:** field, specify a folder for your project files. The project validator checks to ensure that project conforms to the following guidelines:
   - The root folders of the project may not have linked resources.
   - Must have only one output folder.
   - If the output folder is a Java source folder, then it must map to the root folders

   The default for single rootedness problems is set to warning. To change this setting, select **Window** > **Preferences** > **Validation** > **Project Structure Validation**. Click the **...** settings field, and select
   A. **Error**
   B. **Warning**
   C. **Ignore**



8. Click **Next** to continue defining your enterprise project, or click **Finish**.

Related concepts:
Developing Java EE Applications

Creating enterprise application projects

You can use wizards to create an enterprise application project in your Java™ EE project.

Before you begin

If you do not see the Java EE icon,  , in the top right tab of the workspace, you need to switch to the Java EE perspective.

Procedure

1. To use the Java EE wizard from the menu bar, select **File** > **New** > **Project...** > **Enterprise Application Project**. Alternatively, right-click the Enterprise Explorer view, and select **New** > **Enterprise Application Project**.
2. On the Enterprise Application project page, provide the required information:
   - In the **Project name** field, type a name for your Enterprise Application project.
   - In the **Project location** field, accept the default, or clear **Use default location** and type a directory name to contain your project contents.
   - In the **Target Runtime** field, select a target runtime from the drop-down list or click **New** to create an application server runtime.
   - In the **EAR version** field, accept the default or select a different version from the drop-down list.
   - In the **Configuration** field, select the desired application server configuration, or click **Modify** to modify the configuration.
3. Click **Next**.
4. On the Configure enterprise application settings page, provide the required information:
   - In the **Java EE Module dependencies** field, select a module listed or click **New Module** to create a module.
   - Type a directory location name in the **Content Directory:** field.
   - Accept the default value in the **Generate deployment descriptor** field (cleared), or select if you want to generate a deployment descriptor.
5. Click **Finish**.

Creating application client modules

You can use wizards to create an application client module in your Java™ EE project.

Before you begin

If you do not see the Java EE icon,  , in the top right tab of the workspace, you need to switch to the Java EE perspective.

Procedure

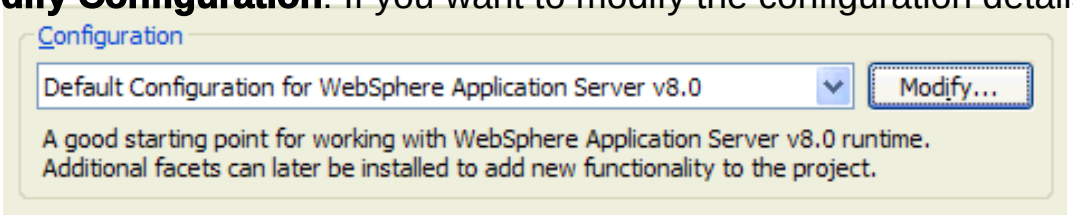
1. To use the Java EE wizards from the menu bar, select **File** > **New** > **Application Client Project**.
2. Alternatively, right-click the Enterprise Explorer view, and select **New** > **Other** > **Java EE** > **Application Client Project**
3. On the Application Client module page, provide the required information:
   - In the **Project name** field, type a name for your Application Client project.
   - In the **Project location** field, accept the default, or clear **Use default location** and type a directory name to contain your project contents.
   - In the **Target Runtime** field, select a target runtime from the drop-down list or click **New** to create an application server runtime.
   - In the **Application Client module version** field, accept the default or select a different version from the drop-down list.
   - In the **Configuration** field, select the desired application server configuration, or click **Modify** to modify the configuration.
   - In the **Ear Membership** field, accept the default **Add project to an EAR** and accept the default EAR project name, or type an alternative in the **EAR Project Name** field. To create an EAR project, click **NEW**.
4. Click **Next**.
5. On the Configure Application Client module settings page, provide the required information:
   - In the **Source Folder** field, specify a folder for your source files or accept the default value (appClientModule).
   - In the **Output Folder:** field, specify a folder for your output files or accept the default value (build\classes).
6. Click **Next**.
7. On the Application Client module page, provide the required information:
   - In the **Create a default Main class** field, accept the default (selected) or clear if you do not want to create a main class.
   - In the **Generate deployment descriptor** field, accept the default (cleared), select if you want to generate a deployment descriptor.
8. Click **Finish**.

Creating EJB modules using wizards

You can use wizards to create an EJB module in your Java™ EE project.

Procedure

1. In the Java EE perspective, right-click your enterprise application project and select **New** > **EJB Project**. The New EJB Project wizard opens.
2. In the **Name** field, type a name for the EJB project. To change the default **Project location**, click the **Browse** button to select a new location.If you specify a non-default project location that is already being used by another project, the project creation will fail.
3. In the **Target runtime** drop-down list, select the server that you want to target for your development. Or, create a target runtime environment by clicking **New**. The target runtime selection affects the compilation and runtime settings by modifying the class path entries for the project. To create an EJB 3.1 project, select WebSphere® application server V8.0, WebSphere application server V7.0, or WebSphere application server V6.1 with the feature Pack for EJB 3.1 installed.
4. Optional: Select a pre-defined project configuration from the **Configurations** drop-down list.
5. Optional: **Modify Configuration**: If you want to modify the configuration details, click **modify**:



   If you are creating a entity, for example, select **Java Persistence**. Save this configuration with a meaningful name, for example, `EJBDevelopmentWithEntityBeans` so that you can reference this configuration in any EJB 3.1 projects that are subsequently created.
6. Optional: Select **Add project to an EAR module** to add the new module to an enterprise module (EAR) project. Type a new project name or select an existing enterprise module project from the drop-down list in the **EAR Project Name** combination box. Or, click **New** to launch the New EAR module Project wizard.
7. Click **Next**.
8. In the **Source Folder** field, specify a folder for your source files or accept the default value (ejbModule).
9. In the **Output Folder** field, specify a folder for your output files or accept the default value (build\classes).
10. Optional: If in the previous page, you selected Add project to an EAR module, then you are able to create an EJB Client Jar. Select **Create an EJB Client JAR module** to hold the client interfaces and classes if you want the client interface and classes for your enterprise beans to be kept in a separate EJB client JAR file. This EJB client JAR file is added to the enterprise module as a project utility JAR file. Specify values for the **Name** and **Client JAR URI** fields, or accept the defaults. If you select this option, the Deployment Descriptor will be generated by default.

11. Optional: Select **Generate Deployment Descriptor** if you want to create a deployment descriptor, although the deployment descriptor is optional in EJB 3.0 and above. The deployment descriptor stores information relating to the EJB project in an Extensible Markup Language (XML) file, serving three functions:
    - Declaring the contents of the module
    - Defining the structure and external dependencies of the beans in the module
    - Describing how the enterprise beans are to be used at run time

    You can also add a deployment descriptor to your EJB module later. See Generating deployment descriptors.
12. Click **Finish**.

Creating web modules

You can use wizards to create web modules in your Java™ EE project.

Procedure

1. In the Java EE perspective, right-click your enterprise application project and select **New** > **Web Project** to open the web project wizard.
2. In the **Name** field, type a name for your new web project.
3. In the Project Templates section, select the type of web template you want to use:

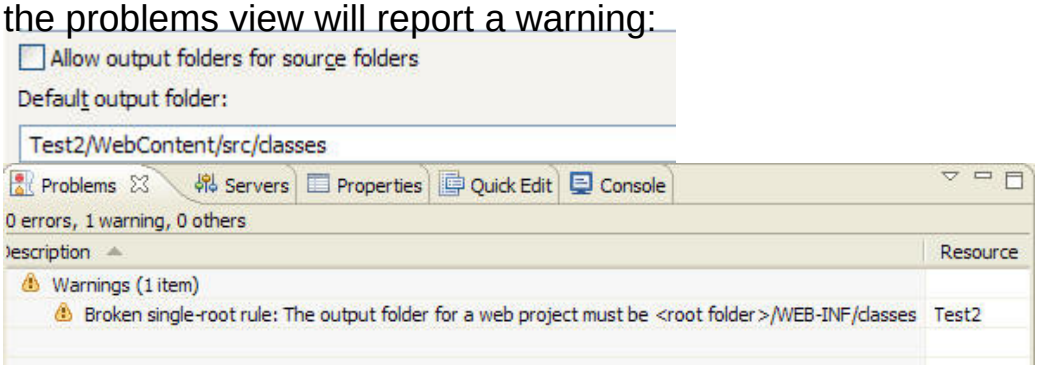
Option	Description
Dojo Toolkit	Configures the project to have Dojo capabilities. The Dojo resources can be in the project itself, a separate project, or a remote location accessible via HTTP.
JavaServer Faces	Enables the project to be deployed with JSF capabilities. Configuration is provided for either JSP or Facelets.
REST Services	A project configured for REST Services based on JAX-RS
Simple	This creates a basic web project.

4. In the Programming Model section, select the programming model you want to use:
   - Client-side only (HTML, JavaScript,...)
   - Java EE
   - OSGi

   Click **Next** to configure your new web project.
5. On the deployment page, from the list of available configuration options, click **Deployment** to open the Deployment configuration page.
   - The **Target runtime** field is pre-populated with the selection from the enterprise project.You can change **Target runtime** by selecting another one from the drop-down box. Click **Change Features** to open the Project Facets window.
   - Click **Add support for WebSphere bindings and extensions** or clear this field.
   - In the **Web module version** field, select the web module version you want to use.
   - In the **EAR membership** field, click **Add project to an EAR**, if you want to include EAR membership; clear this field if you do not want to add the web project to an EAR file.
   - In the **EAR project name** field, the name of your existent EAR file appears. You can click **Browse** to select a different EAR file.

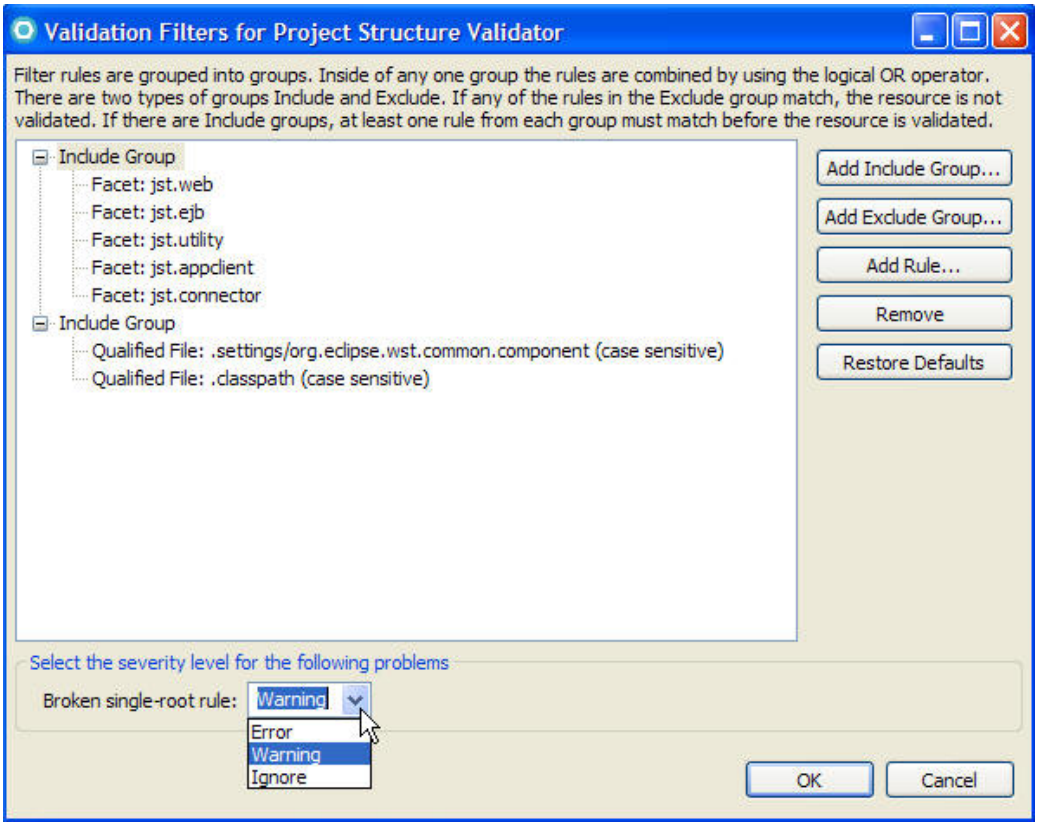
   Note: The deployment option is available only if you selected to use the Java EE programming model for your new web project.
6. From the list of available configuration options, click **Java** to open the Java configuration page.

- In the **Source folders on build path** field, accept the default **src** directory, or click **Add Folder**, **Edit...** or **Remove** to specify a folder for your source files.
- In the **Default output folder:** field, specify a folder for your output files or accept the default value (WebContent\WEB-INF\classes). **Important:** If you choose a folder other than WebContent\WEB-INF\classes for your default output folder, the problems view will report a warning:



The default for single rootedness problems is set to warning. To change this setting, select **Window** > **Preferences** > **Validation** > **Project Structure Validation**. Click the **...** settings field, and select

A. **Error**
B. **Warning**
C. **Ignore**



7. From the list of available configuration options, click **Web Module**.. On the Web Module configuration page,
   - In the **Context root** field, type the name of your web project root, or accept the default (which is the name of your web project).

- In the **Content directory** field, type the name of your content directory, or accept the default (WebContent).
- Select **Generate web.xml deployment descriptor** if you want to create a deployment descriptor. You can also add a deployment descriptor to your web module later.

8. Click **Finish** to create your web project.

Loose classpath web libraries support

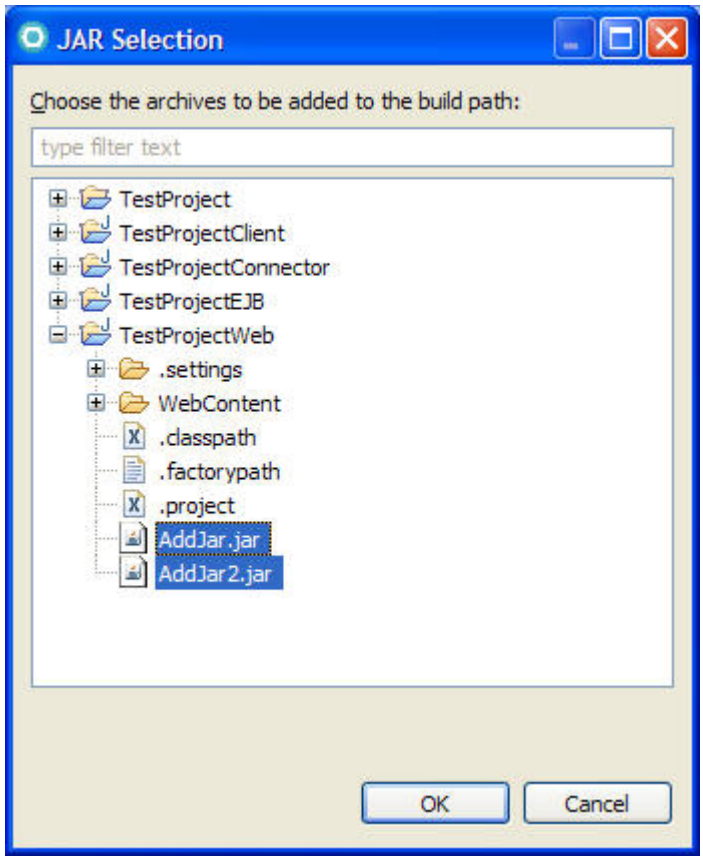
The workbench now provide loose classpath web libraries support for your web projects.

About this task

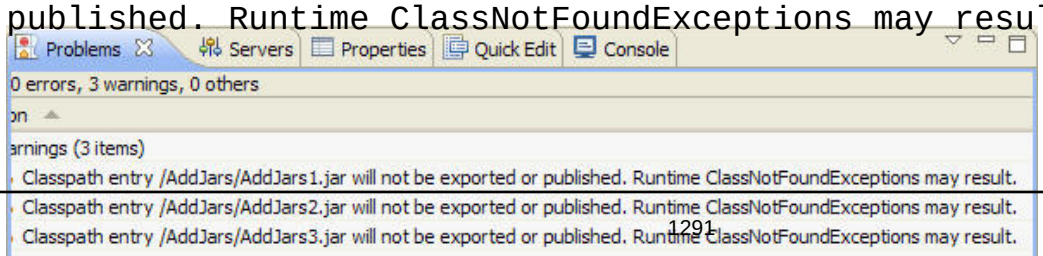
Loose classpath support is an optional mechanism that allows for the inclusion of Eclipse Java™ development tools classpath artifacts in a web project's WEB-INF/lib folder. You can add dependences using the Deployment Assembly page, but you can also add the web libraries from the Libraries page under the Java Build Page. You can include JAR files, external JAR files, libraries, and variables. The project validator now detects loose classpath issues and reports them in the problems view.

Procedure

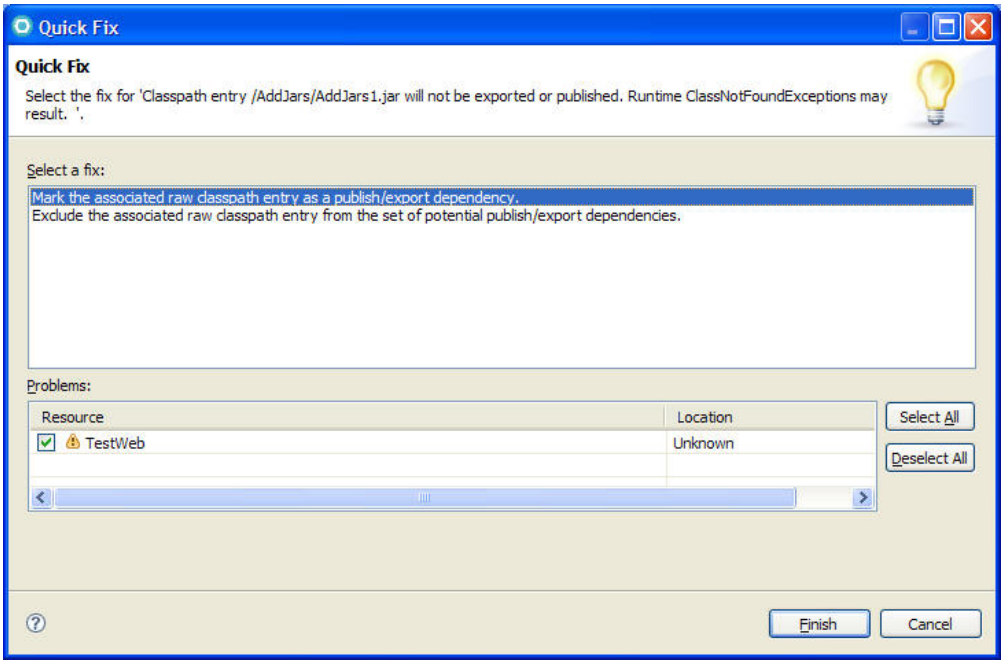
1. Right-click your web project and select **Properties** > **Java Build Path** > **Libraries** .
2. Select **Add Jars**, and select the jars that you want to add to your web project. Click **Okay**.



3. Because these JAR files are added only to the build path and are linked as components, if you deploy this EAR/WAR or export this EAR/WAR none of these JAR files are included. A warning message appears in the Problems view: `Classpath entry /AddJars/AddJars1.jar will not be exported or published. Runtime ClassNotFoundExceptions may result.`



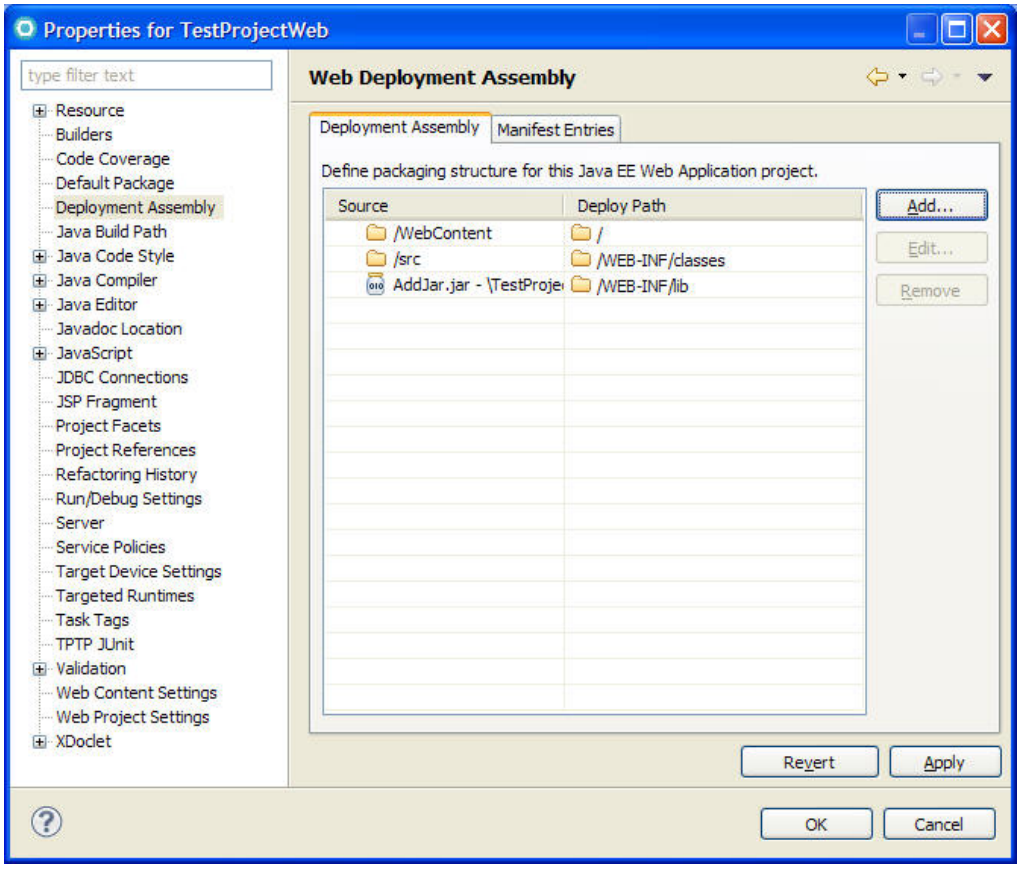
4. Right-click the warning message and select **Quick Fix**. You can either:
   - Mark the associated raw classpath entry as a publish/export dependency.
   - Exclude the associated raw classpath entry from the set of potential publish/export dependencies.



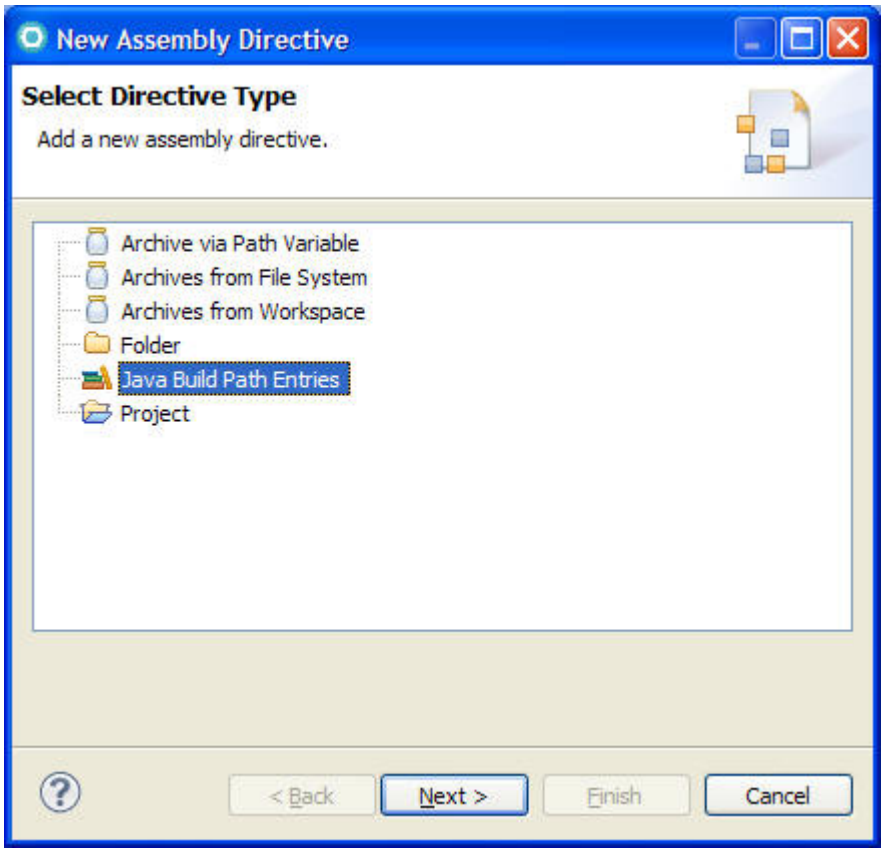
5. If you look at the .classpath file, the JAR files are now either included or excluded from the dependencies:

```
<classpathentry kind="lib" path="/AddJars/AddJars1.jar">
    <attributes>
        <attribute name="org.eclipse.jst.component.dependency" value="/WEB-INF/lib"/>
    </attributes>
</classpathentry>
<classpathentry kind="lib" path="/AddJars/AddJars2.jar">
    <attributes>
        <attribute name="org.eclipse.jst.component.nondependency" value=""/>
    </attributes>
```

6. Right-click web project, and select**Properties** > **Deployment Assembly** to see what jars are included as a web library dependency:



Also, you can add JAR files as a web library dependency by clicking **Add**:



Creating and configuring resource references for Web 2.5 and Web 3.0

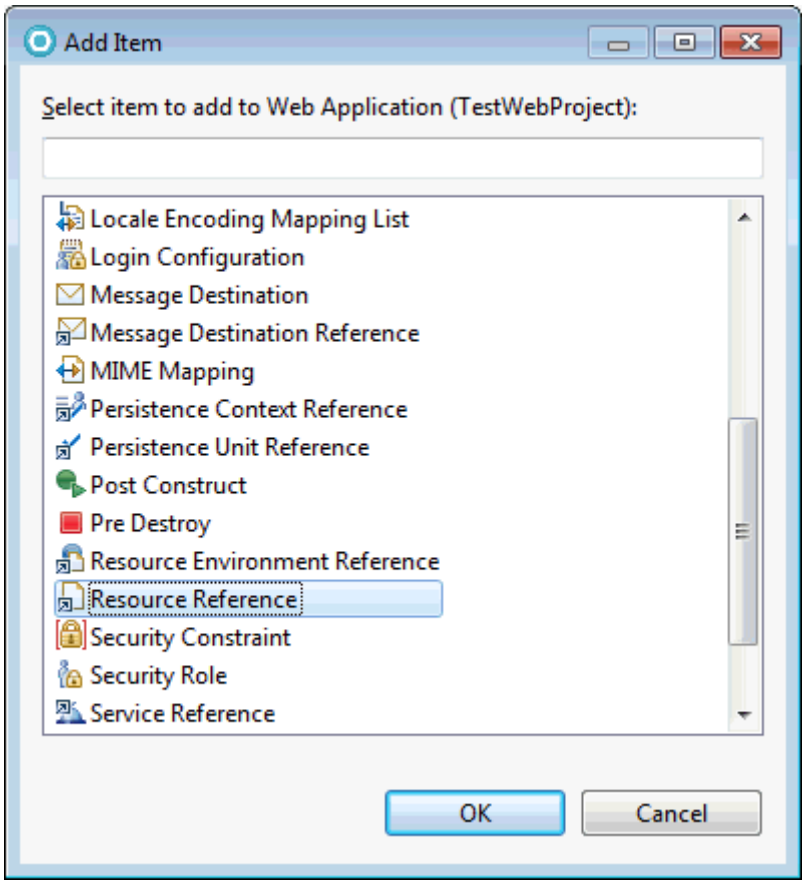
You can create and configure resource references for Web 2.5 and Web 3.0 projects using the deployment descriptor.

About this task

You need to create a web project using Web 2.5 and Web 3.0 before you can create and configure resources references. Select Generate deployment descriptor in the project creation wizard.

Procedure

1. Expand your web project, and select **WebContent** > **WEB-INF** > **web.xml**.
2. Right click web.xml and select **Open with** > **Web Application Deployment Descriptor Editor**.
3. In the Web application field, select **Add** and select **Resource Reference** and click **OK**:



4. In the **Details** section, provide the details for your resource reference:
   A. In the **Name** field, provide a name for your resource reference.
   B. In the **Type** field, provide the type of resource reference.
   C. In the **Authentication** field, select either **Application** or **Container** for the authentication of your resource reference.
   D. In the **Sharing Scope** field, select either **Shareable** or **Unshareable** for the sharing scope of you resource reference.

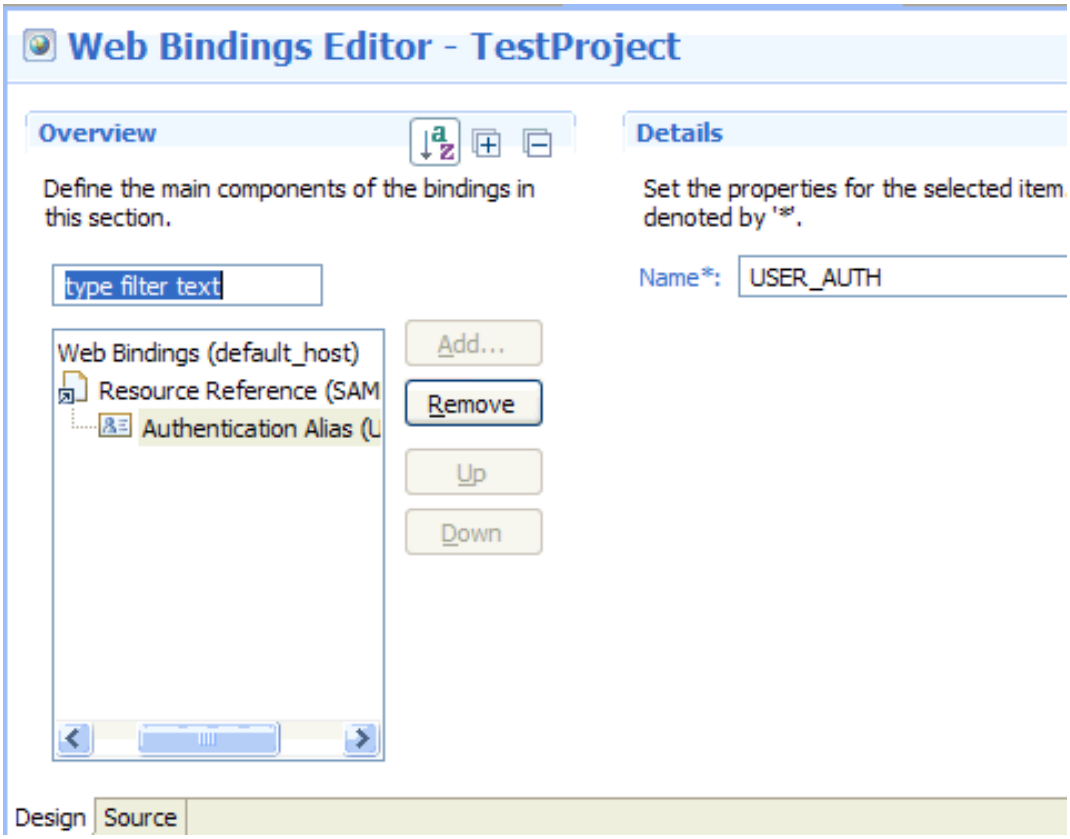
E. In the **Description** field, type a description of this resource reference.

5. To view the web.xml source code, select **Source**:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"> <servlet>
  <description>
  </description>
  <display-name>
  TestServlet</display-name>
  <servlet-name>TestServlet</servlet-name>
  <servlet-class>test.TestServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>TestServlet</servlet-name>
  <url-pattern>
  /TestServlet</url-pattern>
 </servlet-mapping>
 <welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
 </welcome-file-list>
 <resource-ref>
  <res-ref-name>SAMPLE</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
 </resource-ref>
</web-app>
```

6. To edit your resource reference, you can make changes in either the Design or the Source view of the deployment descriptor page.
7. To bind this resource reference to a data source on the server: with the JNDI name jdbc/SAMPLE using a JAAS authentication alias called USER_AUTH, then you edit the ibm-web-bnd.xml file and add the following definitions
   A. Right click the ibm-web-bnd.xml file and select **Open with** > **Web Bindings Editor**.
   B. Add a resource reference:
      1. In the Design view, click **Add**.
      2. In the Add Item window, select Resource Reference. Click **OK**.
      3. In the **Name** field, type the name of your resource reference, for example: SAMPLE.
      4. In the **Binding Name** field, type the of your resource reference, for example: jdbc/SAMPLE.
   C. In the Design view, highlight your Resource Reference, and click **Add** > **Authentication Alias**.

D. In the Details section, in the **Name** field, provide a name for your Authentication Alias (for example, use the JAAS authentication alias called USER_AUTH).



The resulting

E. To view the web.xml source code, select **Source**:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-bnd
 xmlns="http://websphere.ibm.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-web-bnd_1_0.xsd"
 version="1.0">

 <virtual-host name="default_host" />

 <resource-ref name="SAMPLE" binding-name="jdbc/SAMPLE">
  <authentication-alias name="USER_AUTH" />
 </resource-ref>
</web-bnd>
```

Creating web fragment projects

You can use the web fragment project wizard to create web fragment projects in your workspace.

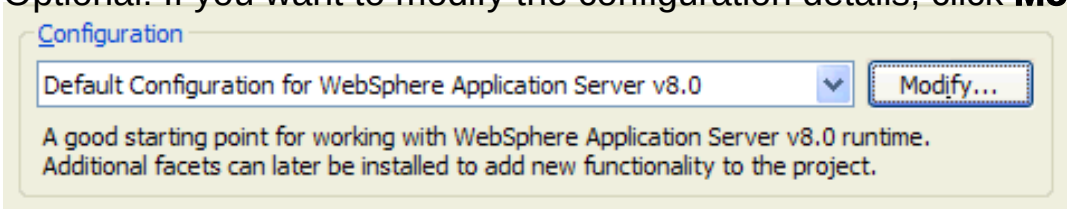
Before you begin

A web fragment is a logical partitioning of the web application in such a way that the frameworks being used within the web application can define all the artifacts without requiring you to edit or add information in the web.xml. It can include almost all the same elements that the web.xml descriptor uses, with these requirements:
- The top level element for the descriptor must be web-fragment
- The corresponding descriptor file must be called web-fragment.xml

If a framework is packaged as a JAR file and has metadata information in the form of deployment descriptor, then the web-fragment.xml descriptor must be in the META-INF/ directory of the JAR file.
A web fragment is a mechanism for either defining or extending the deployment descriptor of a web application by means of pluggable library JAR files that contain both the incremental deployment information (in the web-fragment.xml) and potentially any related or relevant classes. The web fragment is also packaged as a library (JAR), with the web-fragment.xml in the META-INF directory. Consequently, the web fragment project is essentially a utility project, with the addition of a web fragment facet to it. The web fragment facet enables you to add relevant context-sensitive functionality to the fragment project.

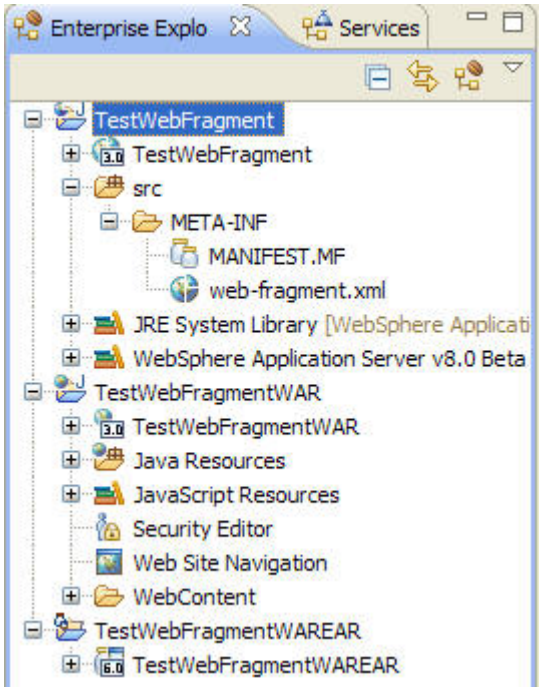
Procedure

1. In the Java™ EE perspective, select **File** > **New** > **Project...** > **Web Fragment Project**. Alternatively, right-click the Enterprise Explorer context menu, and select **New** > **Web Fragment Project**. The Web fragment wizard opens.
2. In the **Project name** field, type a name for the web fragment project. To change the default **Project location**, click the **Browse** button to select a new location.If you specify a non-default project location that is already being used by another project, the project creation fails.
3. The **Target runtime** field is pre-populated with the selection from the enterprise project.
4. Optional: Select a pre-defined project configuration from the **Configuration** drop-down list.
5. Optional: If you want to modify the configuration details, click **Modify**:



6. Optional: Select one or more project facets from the **Project Facets** list. To specify server runtime environments, click **Runtimes** and select one or more runtimes. After making your selections, you can save your custom configuration by clicking **Save**.

7. Optional: Select the **Add project to Dynamic Web project** check box to add the new module to an enterprise module (WAR) project. Type a new project name or select an existing enterprise module project from the drop-down list in the **Dynamic Web project name** combination box. Or, click **New** to launch the New EAR module Project wizard.
8. Select **Add project to working sets** to add the web fragment project to an existing working set, or click **Select** to locate a working set, and click **Next**.
9. On the Configure project for building a Java application page, on the **Source folders on build path** field, click **Add Folder...** to add folders for your source on the build path, or accept the default value (src).
10. In the **Default output Folder:** field, specify a folder for your output files or accept the default value (bin), and click **Finish**.
11. In Enterprise Explorer view, you see the resulting web fragment project folders:

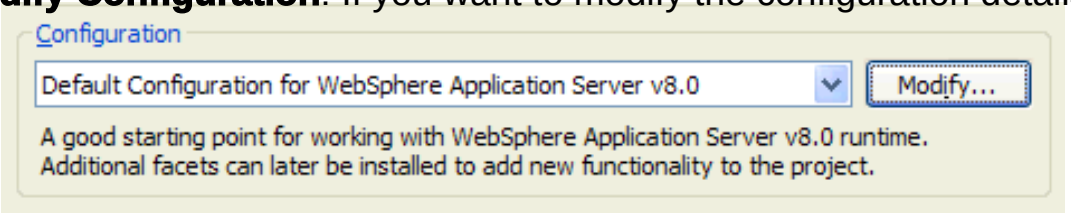


Creating connector modules

You can use wizards to create a connector module in your Java™ EE project.

Procedure

1. In the Java EE perspective, right-click your enterprise application project and select **New** > **Connector project**. The Connector project wizard opens.
2. In the **Name** field, type a name for the connector project. To change the default **Project location**, click **Browse** to select a new location.If you specify a non-default project location that is already being used by another project, the project creation fails.
3. The **Target runtime** field is pre-populated with the selection from the enterprise project.
4. In the **Connector Module version** field, accept the default value.
5. Optional: Select a pre-defined project configuration from the **Configurations** drop-down list.
6. Optional: **Modify Configuration**: If you want to modify the configuration details, click **modify**:



7. Optional: Select one or more project facets from the **Project Facets** list. To specify server runtime environments, click **Runtimes** and select one or more runtimes. After making your selections, you can save your custom configuration by clicking **Save**.
8. Optional: Select the **Add project to an EAR module** check box to add the new module to an enterprise module (EAR) project. Type a new project name or select an existing enterprise module project from the drop-down list in the **EAR Project Name** combination box. Or, click **New** to launch the New EAR module Project wizard.
9. Click **Next**.
10. On the Connector Project page, in the **Source Folder:** field, specify a folder for your source files or accept the default value (connectorModule).
11. In the **Output Folder:** field, specify a folder for your output files or accept the default value (build\classes).
12. Click **Next**. Select **Generate ra.xml deployment descriptor** if you want to create a deployment descriptor.
13. Click **Finish**.

Context and dependency injection (CDI) Overview (JSR 299)

Contexts and dependency injection (CDI) for the Java™ EE platform is an implementation based on the JSR 299 specification. You can create applications that implement CDI in your Java EE projects.

CDI applications are activated by the presence of a beans.xml file that exists in the WEB-INF directory of a web archive (WAR), or in the META-INF directory of other types of archives, as defined by the JSR 299 specification. When activated, the container provides services such as:

- Context management
- Type-safe dependency injection: A CDI-managed bean is instantiated and injected as needed.
- Decorators, which implement one or more bean interfaces and can contain business logic. Decorators are disabled by default. You can have multiple decorators per bean and order is defined by the beans.
- Interceptor bindings. Interceptors, which are enabled manually in the beans.xml file, are bound using an interceptor binding type.
- Event model
- Integration into JavaServer Faces (JSF) and JavaServer Pages (JSP) files using the Expression Language (EL)

For more information about the CDI annotations, see Package javax.inject and Package javax.enterprise.context .

For more information about using CDI in WebSphere® Application Server, see Contexts and Dependency Injection (CDI).

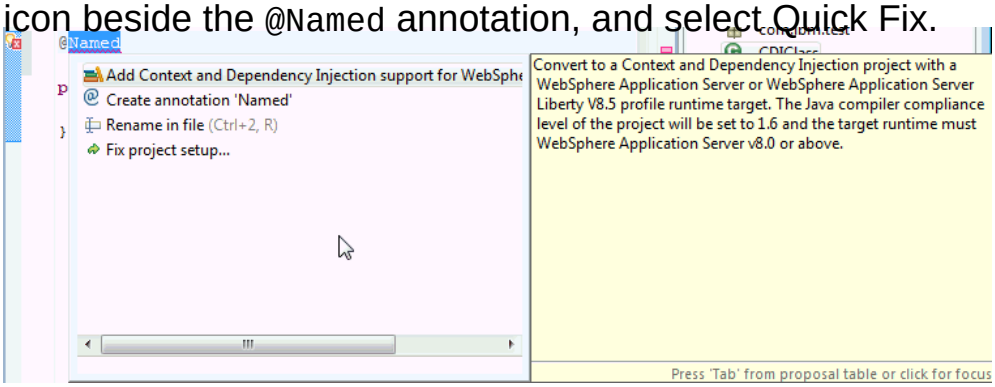
Creating applications that use Contexts and Dependency Injection (CDI) from plain old Java projects (POJOs)

You can use wizards to create applications that use Contexts and Dependency Injection (CDI) from plain old Java™ projects (POJOs).

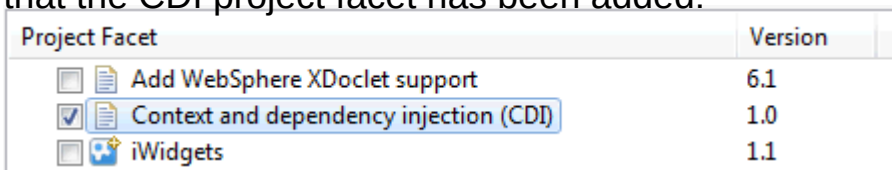
About this task

Procedure

1. In the Java EE perspective, select **File** > **New** > **Project...** > **Java Project**.
2. In the **Project Name** field, click **Use an execution environment**, and select **JavaSE-1.6** or above.
3. Click **Finish**. If a dialogue appears asking if you want to open the Java Perspective, click **No**.
4. Right-click your project and select **New** > **Class**. In the **Package** field, type a name for your package; in the **Name** field, type a name for your class.
5. In the Java class editor, below the package statement, and above the class declaration, type a CDI annotation, for example, `@Named`. Right-click the Quick Fix icon beside the `@Named` annotation, and select Quick Fix.



6. Select **Add Context and Dependency Injection support for Websphere Application Server**. A dialogue appears stating that **The Context and Dependency Injection facet was added successfully to the project <yourproject>**. In the Java editor, the CDI dependency was added to the Java code: `import javax.inject.Named;`.
7. Right-click your project and select **Properties** > **Project Facets**, and you will see that the CDI project facet has been added:



When you expand **<your project>** > **src** > **META-INF**f folder, you see the beans.xml file. At this point, it is virtually empty:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

8. You now have a CDI enabled project.

Creating applications that use Contexts and Dependency Injection (CDI) from Java EE-faceted projects

You can use wizards to create applications that use Contexts and Dependency Injection (CDI) from Java EE-faceted projects

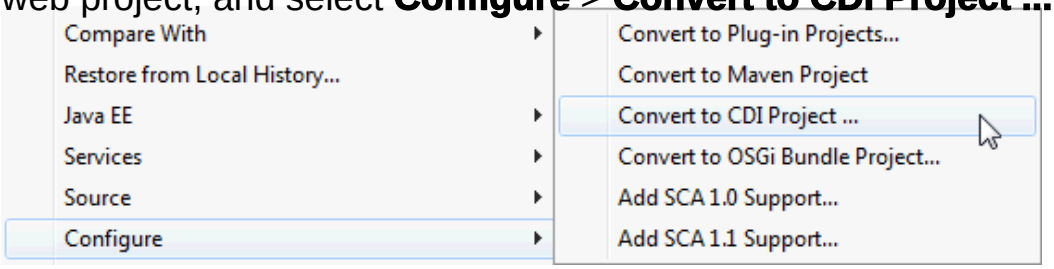
Before you begin

Create a Java EE-faceted project (that is, a utility project, or an EJB project, or a Web project) in your workspace. For information about creating one of these projects see: Creating and configuring Java EE projects using wizards.

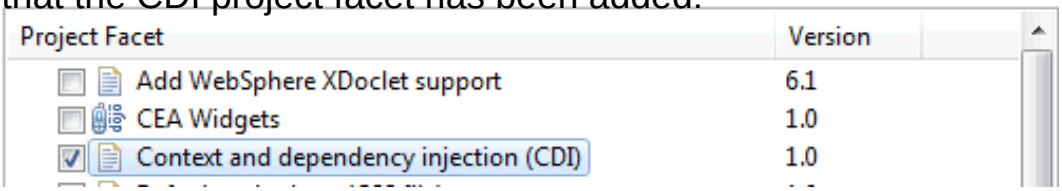
About this task

Procedure

1. In the Java EE perspective, right-click your Java EE project, for example, your web project, and select **Configure** > **Convert to CDI Project ...**.



   A dialogue appears stating that **The Context and Dependency Injection facet was added successfully to the project <yourproject>**
2. Right-click your project and select**Properties** > **Project Facets**, and you will see that the CDI project facet has been added:



   When you expand **<your project>** > **src** > **META-INF**f folder, you see the beans.xml file. At this point, it is virtually empty:
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```
3. You are able to convert any Java EE-faceted project to a CDI project, including a utility project, an EJB project, or a Web project

Creating applications that use Contexts and Dependency Injection (CDI)

You can use wizards to create applications that use Contexts and Dependency Injection (CDI).

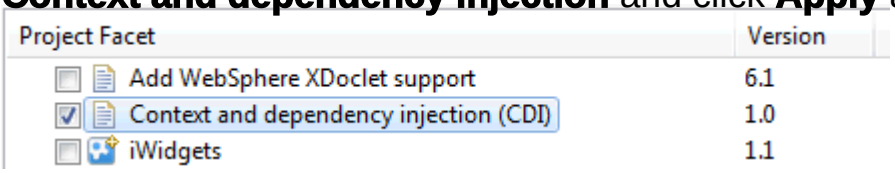
Before you begin

Create a Java EE-faceted project (that is, a utility project, or an EJB project, or a Web project) in your workspace. For information about creating one of these projects see: Creating and configuring Java EE projects using wizards.

About this task

Procedure

1. In the Java EE perspective, right-click your Java EE-faceted project, and select **Properties** > **Project Facets**.
2. Right-click your project and select**Properties** > **Project Facets**, and select **Context and dependency injection** and click **Apply** and **OK**:



   When you expand **<your project>** > **src (for utility projects / ejbModule (for EJBprojects) /WebContent-> WEB-INF (for web projects)** > > **META-INF** folder, you see the beans.xml file. At this point, it is virtually empty: `<?xml version="1.0" encoding="UTF-8"?>`

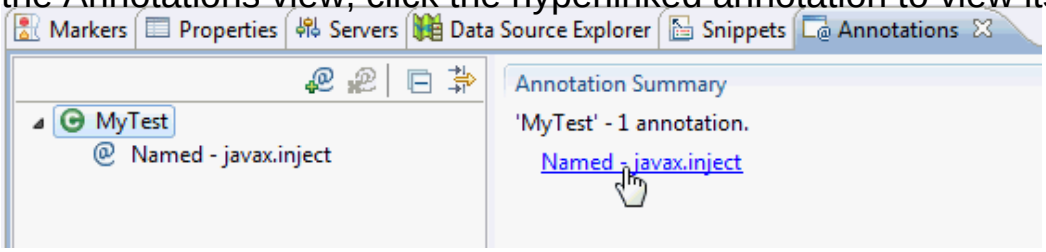
```
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

3. You now have a CDI enabled project.

Implied values in contexts and dependency injection annotations

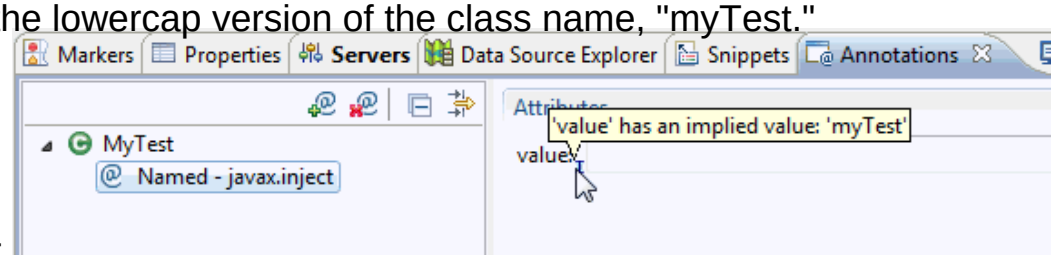
The workspace supports implied value support for the `@Named` and `@New` annotations.

When you specify an injection point (for example, `@Named`) and you do not specify a value, the implied value is derived from the declared type of the injection point. In the Annotations view, click the hyperlinked annotation to view its attributes:



Implied value for the `@Named` annotation

For `@Named`, the implied value is derived from the declared type of the injection point, the lowercap version of the class name, "myTest."

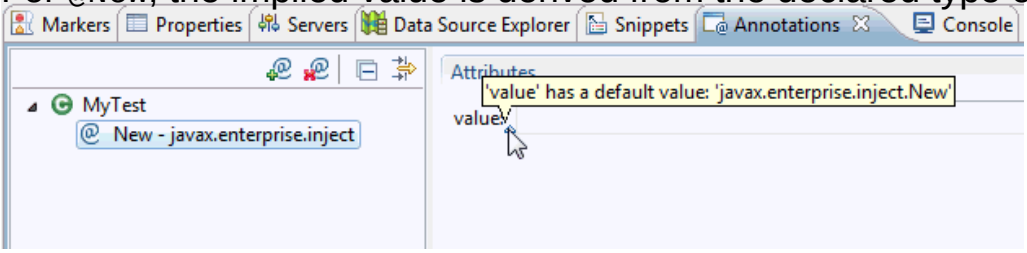


-

- For `Type`, it is the lowercase element name
- For Producer methods, it is the lowercase producer method name.)
- For Producer fields, it is the field name.
- There is no implied value for all other cases.

Implied value for the `@New` annotation

For `@New`, the implied value is derived from the declared type of the injection point:



Validating applications that use Contexts and Dependency Injection (CDI)

Your workspace provides in-line and quick-fix validations for contexts and dependency injection applications.

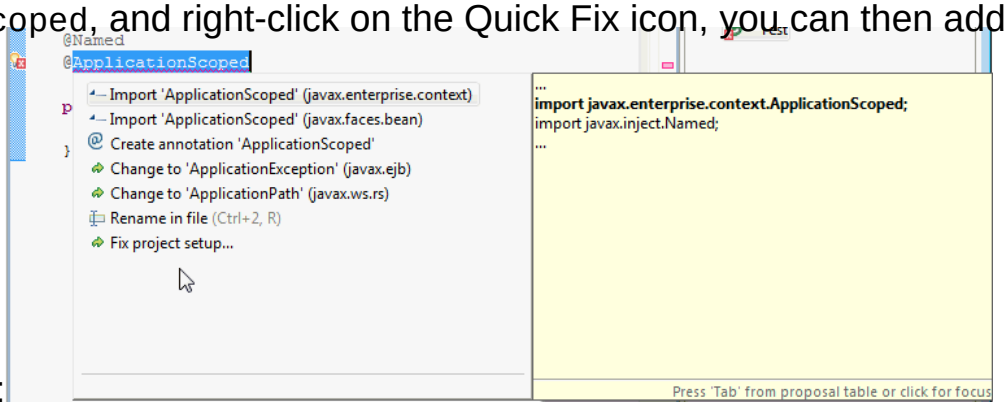
Before you begin

Create a Java™ EE-faceted project (that is, a utility project, or an EJB project or a Web project) in your workspace. For information about creating one of these projects see: Creating and configuring Java EE projects using wizards.

About this task

As-you-type validation is supported in CDI-faceted projects.

Procedure

1. In the Java EE perspective, open your class that uses context and dependency injection annotations. If you add a new annotation, for example, `@ApplicationScoped`, and right-click on the Quick Fix icon, you can then add the required imports: 

   The `import javax.enterprise.context.ApplicationScoped;` import statement is added to your class.
2. **Scoped validation**
   A. Validation is limited to the following built in CDI scope types:
      - `@RequestScoped`
      - `@ApplicationScoped`
      - `@SessionScoped`
      - `@ConversationScoped`
      - `@Dependent`
   B. Multiple scope declarations are not allowed on the bean class, producer methods, or producer fields. Validation flags this as an error and provides a Quick Fix to correct the problem.
   C. A subset of validators support the following passivating scopes:
      - `@SessionScoped`
      - `@ConversationScoped`

      These annotations can be declared on session beans and managed beans. In order for these beans to be passivating-enabled, they must conform to the rules:
      - Only Stateful Session Beans are passivating capable. Validation errors are issued if stateless or singleton annotated session beans are declared with one of these passivating scopes. Quick Fix support allows you to replace

with the `@Stateful` annotation.
   - Managed Beans are passivating capable if the bean class, and all Interceptors and decorators, are serializable. **Note:** Validation flags an error if the bean class is not serializable and offers a Quick Fix to make it serializable. No validation is provided for Interceptors and Decorators.
3. **Type Restriction Validation** A managed Bean has a set of legal bean types which include the `java.lang.Object`, the bean class, the superclass, and all interfaces it implements directly or indirectly. You can restrict the set of bean types by using the `@Typed` annotation but the restricted set must be a subset of the legal bean types set, otherwise a validation error will be issued. No Quick Fix support is provided for this.

Deleting Java EE modules

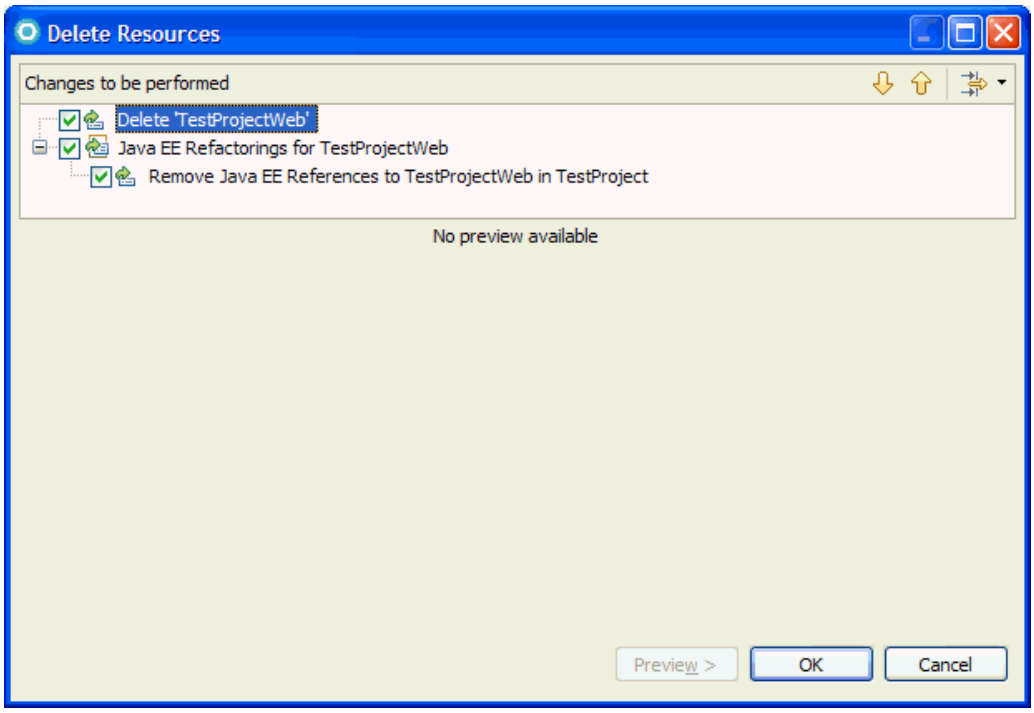
Once you have created a Java™ EE application, you might want to delete modules within it.

About this task

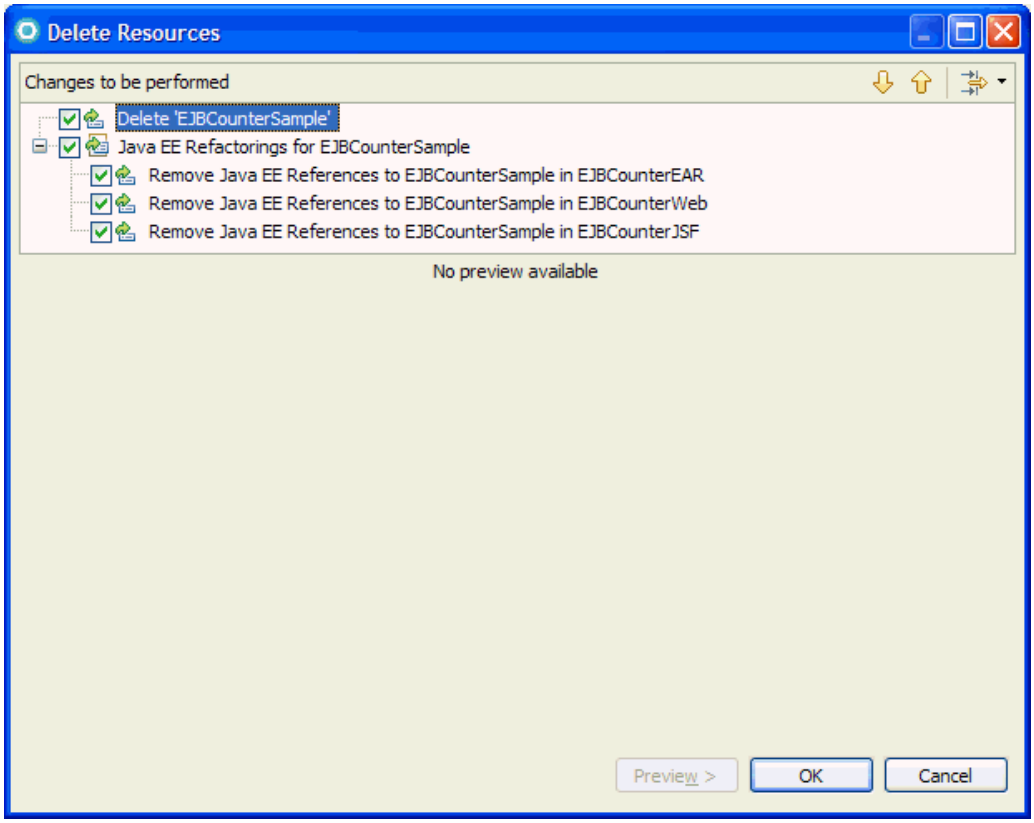
When you create a Java EE project, code is generated in files as part of the process of creating the project. After you delete a module or a bean in a Java EE project, the affected files are also cleaned up. To delete a web module, EJB module, Application Client module, Connector module, or EJB client, or utility module:

Procedure

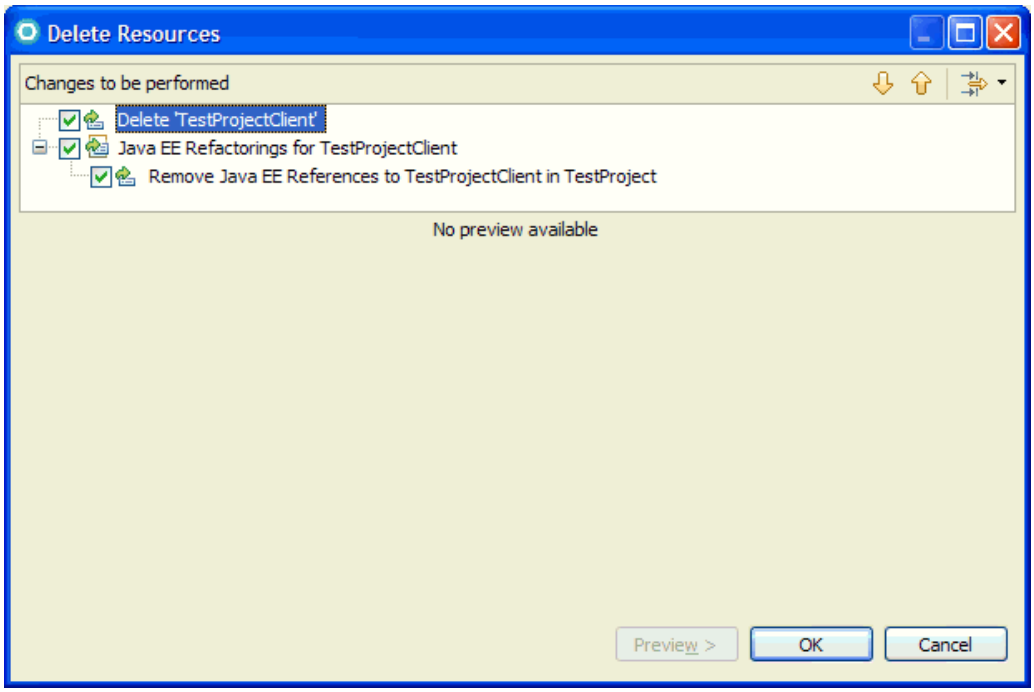
1. Right-click the module that you want to delete, and select **Delete**. Alternatively, you can select the module in the project view, and press `Delete`. A dialog box appears, confirming the delete.
2. Select **Preview**. A refactoring page appears listing the items affected by your delete action. Clear the items you do not want to be removed with the delete process, and click **OK**.
   - Deleting a web module:



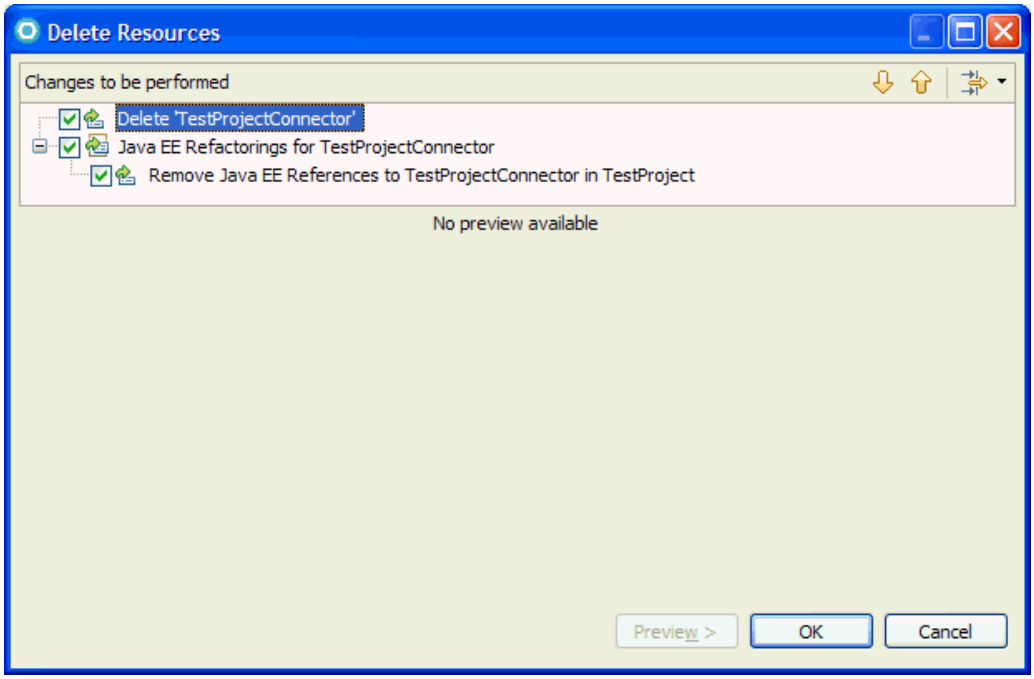
   - Deleting an EJB module:



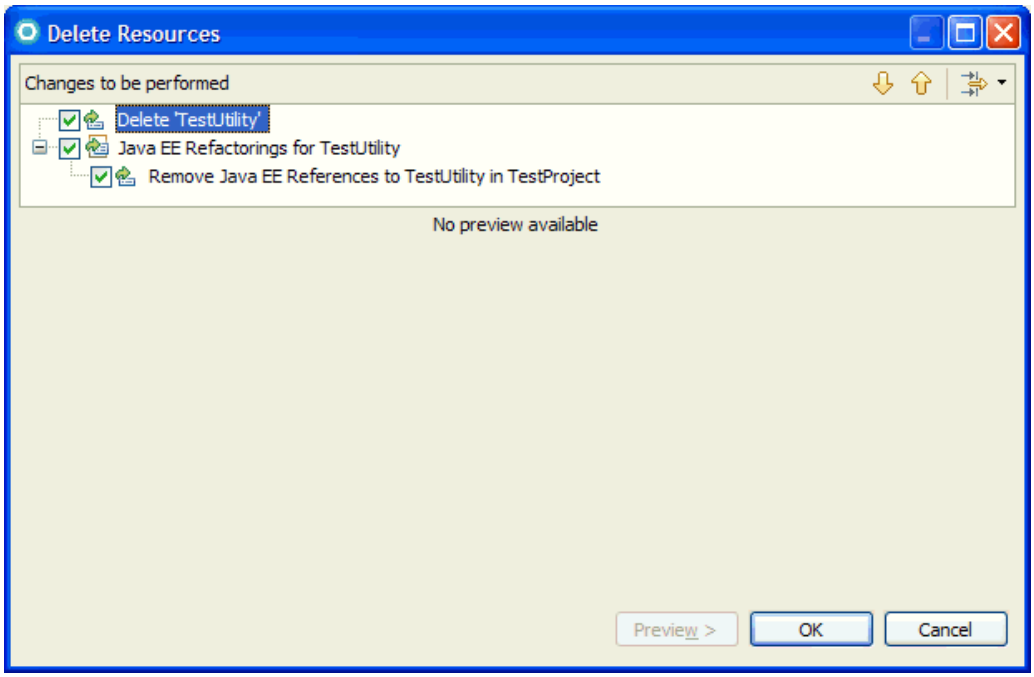
- Deleting an Application Client module:



- Deleting a Connector module:



- Deleting a utility module:



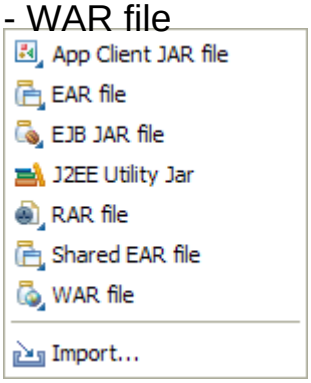
Importing JAR, WAR, EJB JAR, client application JAR, and RAR files

About this task

You can use the Import wizard to add JAR, WAR, EJB JAR, client application JAR, and RAR files to your enterprise project.

Procedure

1. Right-click your project and select **Import**. Select the type of file you want to import:
   - Application Client JAR file
   - EAR file
   - EJB JAR file
   - J2EE Utility Jar
   - RAR file
   - Shared EAR file
   - WAR file



2. Follow the steps to import your selected type of file:
   - **Application Client JAR file**:
     A. In the Application Client Import page, beside the **Application Client file** field, click **Browse** to locate the file you want to import. When the file is added, the name of the imported project appears in the **Application Client project** field. You can change the name of the project by typing in a different name. Select **Add project to an EAR** and provide a name for the EAR file (the default is derived from your <project_name> + EAR).
     B. Click **Finish**.
   - **EAR file**:
     A. In the Enterprise Application Import page, beside the **EAR file** field, click **Browse** to locate the file you want to import. When the file is added, the name of the imported project appears in the **EAR project** field. You can change the name of the project by typing in a different name. Select your preferred target runtime in the **Target Runtime** field, and click **Next**.
     B. In the Enterprise Application Import page, in the **Utility jars and web libraries** field, select the additional jars you want to include in your project, and click **Finish**.
   - **EJB JAR file**:
     A. In the EJB JAR Import page, beside the **EJB JAR file** field, click **Browse** to

locate the file you want to import. When the file is added, the name of the imported project appears in the **EJB project** field. You can change the name of the project by typing in a different name. Select **Add project to an EAR** and provide a name for the EAR file (the default is derived from your <project_name> + EAR).

B. Click **Finish**.

- **Java™ EE Utility Jar**:

A. On the Utility Jar Import page, in the **Select EAR Project** field, ensure that your project appears, or type a new project name. In the **Select import type** field, select one of the following options:
- Create Java Projects from Utility Jars
- Create linked Java Projects from Utility Jars
- Copy Utility Jars into an existing EAR from an external location
- Create Linked Utility Jars in an existing EAR from an external location

In the **Project import options** field, you can select **Override Project Root (Specify location below)**, and type an alternative project location, or click **Browse** to locate a directory in your file system. Click **Next**.

B. On the Utility Jar Import page, click **Browse** to locate the director that contains the utility jar files that you want to import. In the **Utility Jars and Web libraries** field, select the files that you want to import. Click **Finish**.

- **RAR file**:

A. In the Connector Import page, beside the **Connector file** field, click **Browse** to locate the file you want to import. When the file is added, the name of the imported project appears in the **Connector module** field. You can change the name of the project by typing in a different name. Select **Add project to an EAR** and provide a name for the EAR file (the default is derived from your <project_name> + EAR).

B. Click **Finish**.

- **Shared EAR file**:

A. In the Shared EAR Import page, beside the **EAR file** field, click **Browse** to locate the file you want to import. When the file is added, the name of the imported project appears in the **EAR project** field. You can change the name of the project by typing in a different name. Select your preferred target runtime in the **Target Runtime** field.

B. Click **Finish**.

- **WAR file**:

A. In the WAR Import page, beside the **WAR file** field, click **Browse** to locate the file you want to import. When the file is added, the name of the imported project appears in the **Web project** field. You can change the name of the project by typing in a different name. Select **Add project to an EAR** and provide a name for the EAR file (the default is derived from your <project_name> + EAR). Click **Next**.

B. On the WAR Import: Web Libraries page, select any Web libraries you want to add to your project. Click **Finish**.

Exporting JAR, WAR, EJB JAR, client application JAR, and RAR files

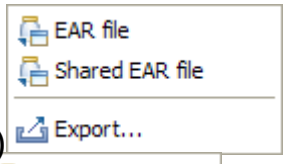
About this task

You can use the Export wizard to export your enterprise applications as JAR, WAR, EJB JAR, client application JAR, and RAR files.

Procedure

1. Right-click your enterprise project and select **Export**. Depending on the type of your project, select:

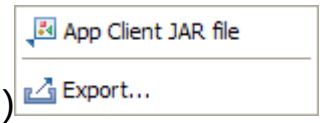
   - EAR file (enterprise application)

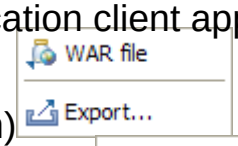
   - EJB JAR file (EJB application)

   - App Client JAR file (application client application)

   - WAR file (Web application)

   - RAR file (connector application)

2. In the file Export page, your selected project appears in the **project** field, or click **Browse** to locate the project in your workspace. In the **Destination** field, type a folder destination, or click **Browse** to locate a destination directory for your exported file. In the **Target Runtime** field, select your preferred target runtime. Select **Export source files**, if you want to include your source files. Select **Overwrite existing file**, if you want to overwrite an existing file. Click **Finish**.

Creating and configuring Java EE modules using annotations

The goal of Java™ EE 6 platform development is to minimize the number of artifacts that you have to create and maintain, thereby simplifying the development process. Java EE supports the injection of annotations into your source code, so that you can embed resources, dependencies, services, and lifecycle notifications in your source code, without having to maintain these artifacts elsewhere.
An annotation is a modifier or metadata tag that provides additional data to Java classes, interfaces, constructors, methods, fields, parameters, and local variables. Annotations replace boilerplate code, common code that is required by certain applications. For example, an annotation can replace the paired interface and implementation required for a web service. Annotations can also replace additional files that programs require, which are maintained separately. For example, annotations can replace the need for a separately maintained deployment descriptor for enterprise Java beans.
Annotations
- Replace descriptors for most purposes
- Remove the need for marker interfaces (like java.rmi.Remote)
- Allow application settings to be visible in the component they affect

Java EE provides annotations for the following tasks, among others:
- Developing Enterprise Java bean applications
- Defining and using web services
- Mapping Java technology classes to XML
- Mapping Java technology classes to databases
- Mapping methods to operations
- Specifying external dependencies
- Specifying deployment information, including security attributes

Java EE defines a number of annotations that can be injected into your source code. To declare an annotation, you simply precede the keyword with an "at" sign (@).

```
package com.ibm.counter;


import javax.ejb.Stateless;


@Stateless


public class CounterBean {


}
```

For more information about the categories of annotations that Java EE supports, see Types of annotations.

Related concepts:
Developing Java EE Applications

Java EE: Overview
Tools for Java EE development
Project facets
Defining Java EE applications
Securing enterprise applications

Related tasks:
Setting Java EE preferences
Creating and configuring Java EE projects using wizards
Validating code in enterprise applications
Deploying Java EE applications
Migrating the specification level of Java EE projects
Defining your own Annotations
Adding annotations
Editing annotations
Removing annotations
Overridden annotations
Excluding files from annotation scanning

Scope and placement of annotations

You can add annotations to your source code at the class, method, and field level.

Using Annotations

EJB 3.1 and the Java™ persistence API make use of metadata annotations, a feature that was introduced in J2SE 5.0. An annotation consists of the @ sign preceding an annotation type, sometimes followed by a parenthetical list of element-value pairs. The EJB 3.1 specification defines various annotation types, for example:

- Component-defining annotation, such as `@Stateless`, which specifies the bean type
- `@Remote` and `@Local` specify whether a bean is remotely or locally accessible
- `@TransactionAttribute` specifies transaction attributes
- `@MethodPermissions`, `@Unchecked`, and `@SecurityRoles` specify security and method permissions

The Java Persistence API adds annotations specific to the creation of entities, for example:

- `@Entity` is a component-defining annotation that specifies that a class is an entity
- `@Table` specifies the data source to be used in the class

Note: JPA mapping annotations (`@Id`, `@Column`, for example) can be applied to both fields and methods, but for any one entity class you can only apply them to one or the other; that is, either all annotations must be applied to fields, or they all must be applied to methods. Fields can only have private, protected or package visibility, and clients of an entity are not allowed to access the fields directly, so you need to define public getters and setters.

- `@MethodPermissions`, `@Unchecked`, and `@SecurityRoles` specify security and method permissions

Scope and placement of annotations

Annotations operate at a class, interface, method, or field level. For example, component-defining annotations (like `@Stateless` or `@Entity`) are class-level annotations and they are inserted in the comments section before the class declaration:

```
package com.ibm.websphere.ejb3sample.counter;


import javax.ejb.Stateless;

import javax.interceptor.Interceptors;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;


@Stateless

@Interceptors ( Audit.class )


public class StatelessCounterBean implements LocalCounter, RemoteCounter {
```

The order of these annotations is not significant; typically, the component-defining annotation is generally placed before other annotations, but this placement is not required. Method-level and field-level annotations appear within the class or method:

```
public class JPACounterEntity {
```

```
@Id
private String primarykey = "PRIMARYKEY";


private int value = 0;
```

Related tasks:
Defining your own Annotations
Adding annotations
Editing annotations
Removing annotations
Overridden annotations
Excluding files from annotation scanning

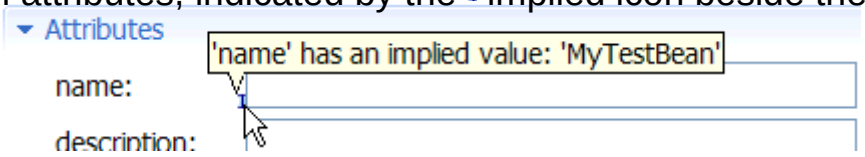
Using the Annotations view

The Annotations view detects the annotation types from the metadata of the annotation tag implementation class, shows you any default values, and provides a place to add, edit and remove annotations.
From the Java™ EE perspective, if you do not see the Annotations view, open by selecting **Window** > **Show View** > **Other** > **Java** > **Annotations**

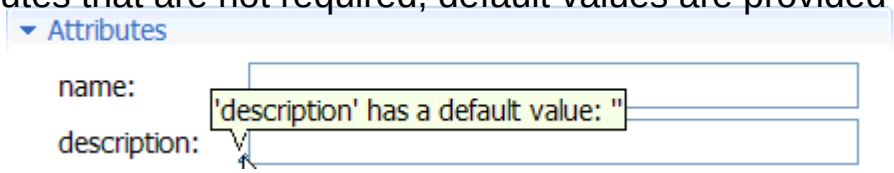
Annotations view

The Annotations view provides a way for you to create, edit, browse, and generally keep track of the annotations that you use in your applications. The Annotations view performs the following functions:

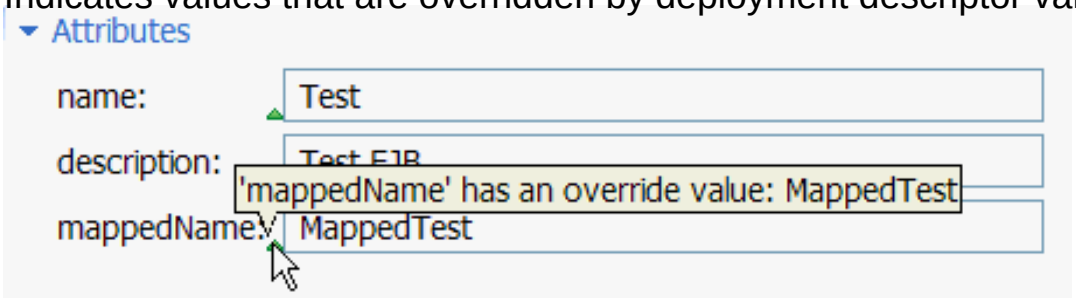
- Displays in an easy-to-navigate tree structure all of your annotations in your Java classes. You can add and remove annotations using the toolbar icons above the tree. You can filter the tree by typing a filter value in the **type filter text field**.
- Detects annotation types from the metadata in the annotation tag implementation class to provide rich editing capability.
  - Indicates what attributes can be defined for an annotation, and indicates if an annotation does not define attributes.
  - Provides default validation and user assistance for each annotation.
  - Indicates which attributes are required.
- Displays implied annotation attributes, indicated by the implied icon beside the attribute name, in this way:



  By hovering over the , you can see the implied value for the attribute.
- For attributes that are not required, default values are provided (which you can change):



  By hovering over the , you can see the default value for the attribute.
- Indicates values that are overridden by deployment descriptor values:



  By hovering over the , you can see the overridden value for the attribute.

Related tasks:
Defining your own Annotations
Adding annotations

Defining your own Annotations

You can use the `@Interface` annotation to create your own annotation definition.

Procedure

Use the `@Interface` annotation to define your own annotation definition:
- Annotation definitions resemble interface definitions
- Annotation method declarations have neither parameters nor `throws` clauses, and return one of the following elements:
  - primitives
  - String
  - Class
  - enum
  - array of the above types
- Methods can have default values

```
public @interface CreatedBy{
    String name();
    String date();
    boolean contractor() default false;
}
@CreatedBy(name = "Mary Smith",date="02/02/2008");
public class MyClass{....}
```

Results

Meta-annotations: Meta-annotations (annotations of annotations) provide additional information about how an annotation can be used:
- @Target
  - Restricts the use of an annotation
  - Single argument must be from Enum ElementType
    - {TYPE, FIELD,METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE}
- @Retention
  - Indicates where the annotation information is retained
  - Single argument must be from Enum RetentionPolicy
    - {SOURCE, CLASS, RUNTIME}
- @Documented
  - Marker for annotations that are be included in Javadoc
- @Inherited
  - Marker for Type annotations that are to be inherited by subtypes

Other built-in annotations:
- @Overrides
  - Applied to a method
  - Indicates that the compiler generates an error if the method does not actually override a superclass method.

- @Deprecated
  - Applied to a method
  - Indicates that the compiler generates a warning when the method is used externally
- @SuppressWarnings
  - Applies to a type or a method
  - Indicates that the compiler suppresses warnings for that element and all subelements `@Deprecated`

```
public void oldMethod() {...}


@ SupressWarnings
public void yesIknowIuseDeprecatedMethods() {...}
```

Related concepts:
Creating and configuring Java EE modules using annotations
Scope and placement of annotations
Using the Annotations view

Related tasks:
Adding annotations
Editing annotations
Removing annotations
Overridden annotations
Excluding files from annotation scanning

Adding annotations

You can add annotations in your source code by directly adding the annotation in the Java™ editor or by using the Annotations view.

Before you begin

In the Java EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Related concepts:
Creating and configuring Java EE modules using annotations
Scope and placement of annotations
Using the Annotations view

Related tasks:
Defining your own Annotations
Editing annotations
Removing annotations
Overridden annotations
Excluding files from annotation scanning

Adding annotations using the Annotations view.

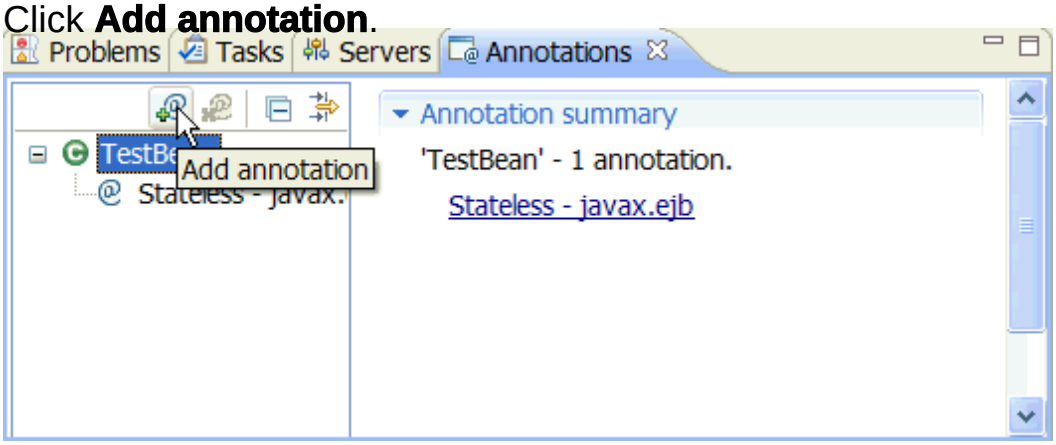
You can add annotations in your source code by using the Annotations view.

Before you begin

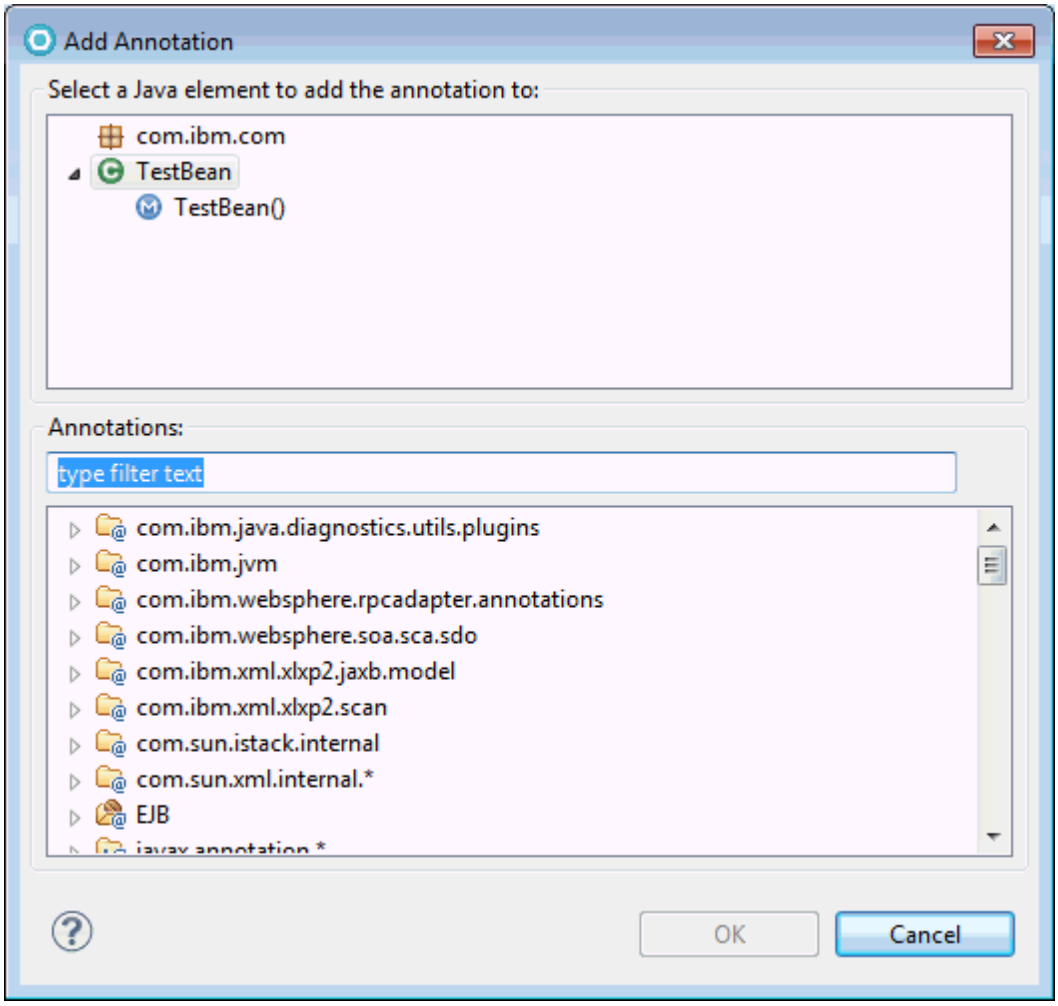
In the Java™ EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Procedure

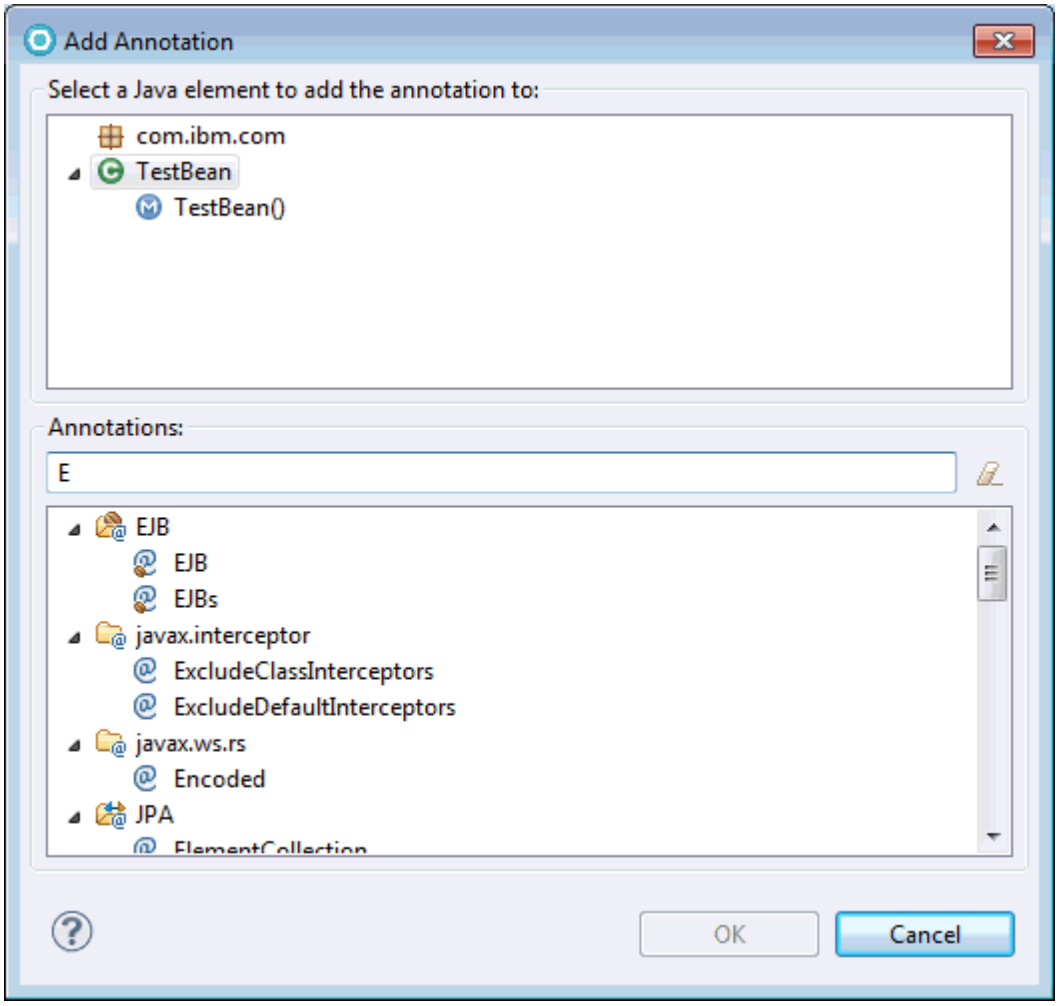
1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. Click the Annotations view tab.
3. Select the Java element that you want to add an annotation to.
4. Click **Add annotation**.



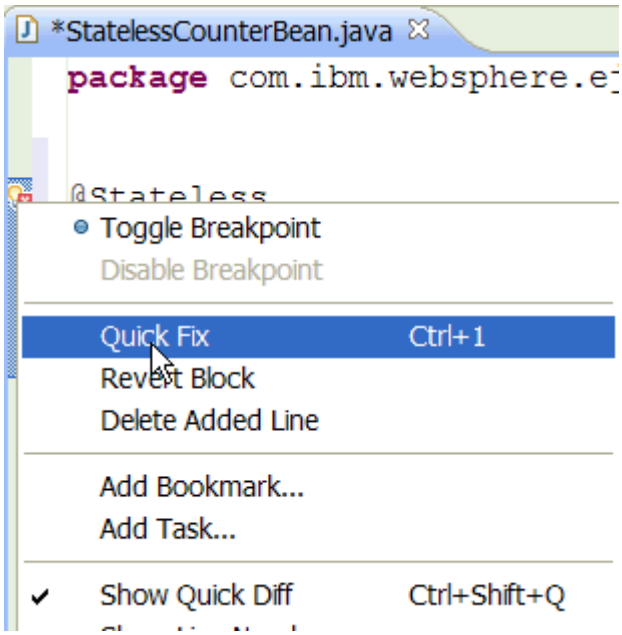
5. In the Add annotation page, select the annotation that you want to add:



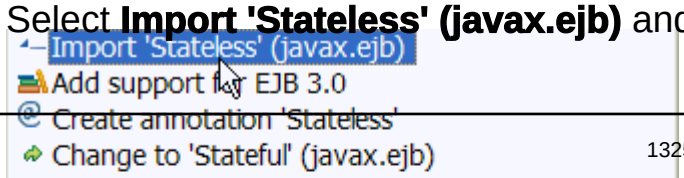
6. To filter the annotations, type a letter or term into the **type filter text** field. You can use "*", "?", or camel case:



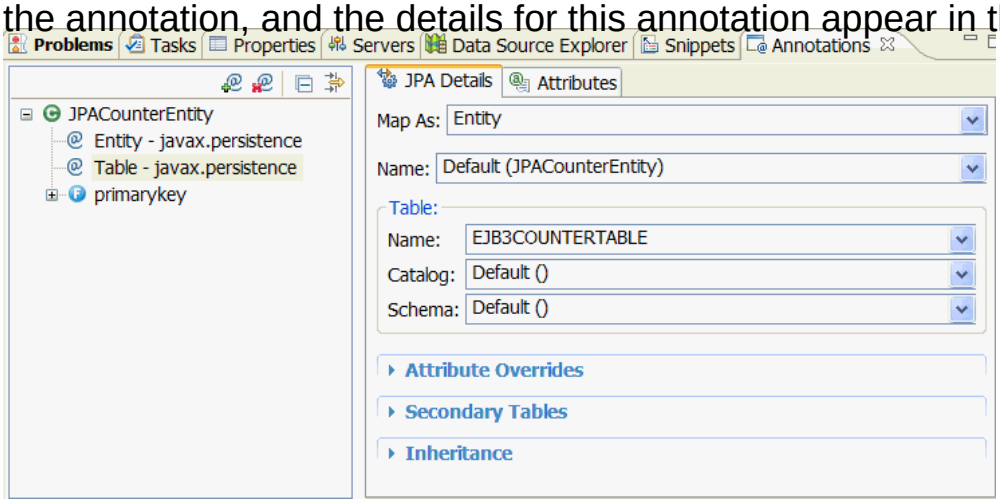
7. Select the annotation you want to insert into your Java class, and click **OK**
8. You see in the Java class the @Stateless annotation is added to your code.
9. When you press CTRL+S to save, you can see a quick fix icon beside the @Stateless line
10. Right-click the quick fix icon and select **Quick Fix**:



11. Select **Import 'Stateless' (javax.ejb)** and press CTRL+S to save:



12. You see in the Java class that your annotation is added to the code. Highlight the annotation, and the details for this annotation appear in the Annotation view.



13. The attributes pane shows both the default values (which you can change) and assigned values.

Adding annotations manually

You can add annotations in your source code by directly adding the annotation in the Java™ editor.

Before you begin

In the Java EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Procedure

1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. Place your cursor in the section of your class (class level, method level) where you want to add an annotation.
3. Type the symbol @ followed by the name of your annotation:`@Session`

   `@Interceptors`

   Note: The order in which you place the annotations is not significant, though generally the component-defining annotation appears before other types of annotations. What is important is that annotations are placed in the appropriate place in the source file; class-level annotations must appear before the class declaration; method-level annotations are introduced before the method declaration, and field-level annotations are applied to the field itself.
4. If the annotation requires parameters, an error appears in the margin. You can right-click the error and select Quick Fix () to import required jars or to add required parameters.

Editing annotations

You can edit annotations in your source code by directly modifying the annotation in the Java™ editor or by using the Annotations view.

Before you begin

In the Java EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Related concepts:
Creating and configuring Java EE modules using annotations
Scope and placement of annotations
Using the Annotations view

Related tasks:
Defining your own Annotations
Adding annotations
Removing annotations
Overridden annotations
Excluding files from annotation scanning

Editing annotations manually

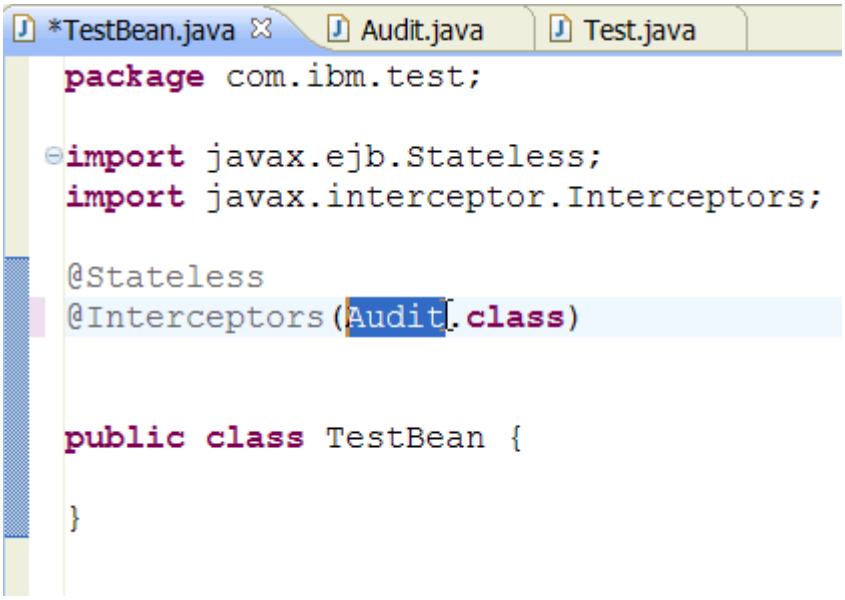
You can edit annotations in your source code by directly modifying the annotation in the Java™ editor or by using the Annotations view.

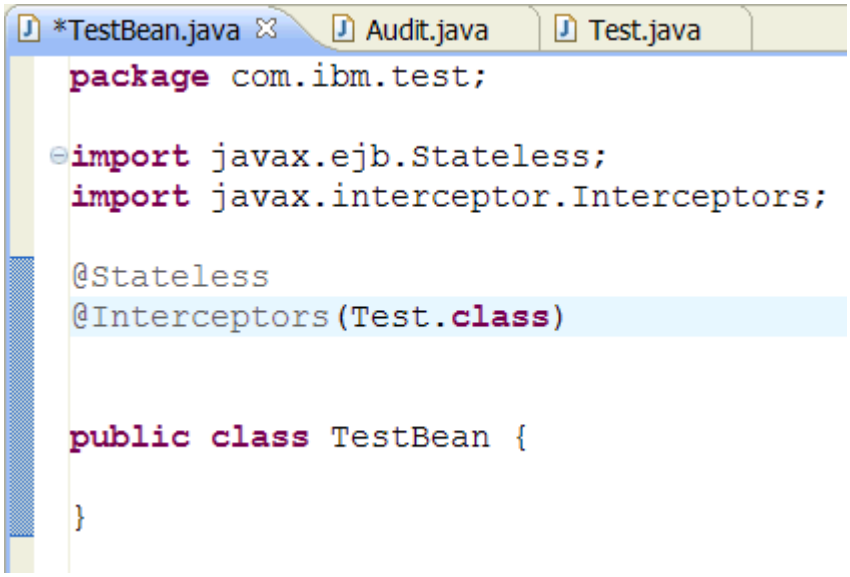
Before you begin

In the Java EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Procedure

1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. In the Java editor, highlight the annotation that you want to edit. You can change the annotation by typing in a different value, for example, you can change `@Stateless` to `@Stateful` to change an EJB from a Stateless Java bean to a Stateful one.
3. You can also edit the values of the parameters associated with the annotation, by highlighting the value and typing the value you want:





In this example, the default value of the attribute was changed from `Audit.class` to `Test.class`.

Editing annotations using the Annotations view

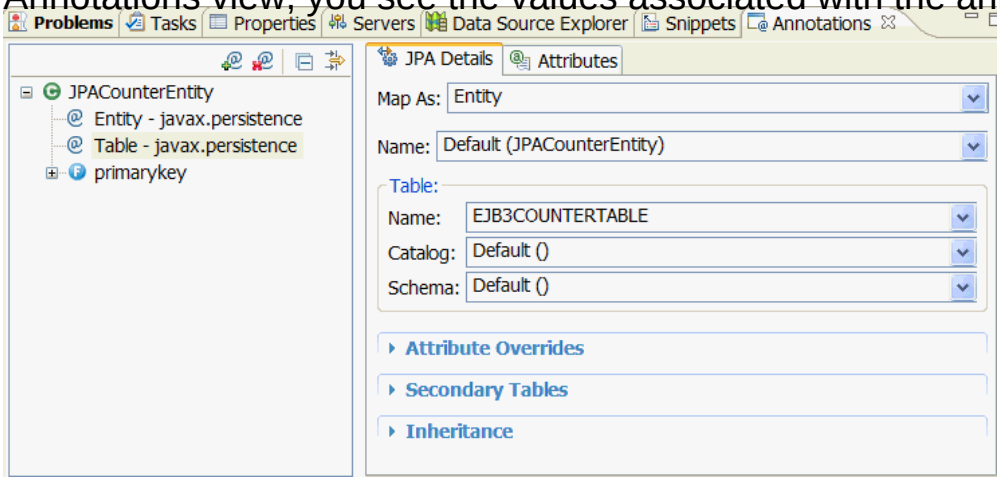
You can edit annotations in your source code by directly modifying the annotation in the Java™ editor or by using the Annotations view.

Before you begin

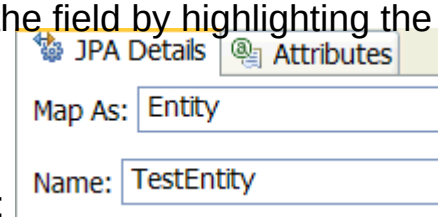
In the Java EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Procedure

1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. Click the Annotations view tab.
3. In the Java editor, highlight the annotation that you want to edit. In the Annotations view, you see the values associated with the annotation:



4. Edit the field by highlighting the value in the text box and typing the value you want:



   In this example, the default value of the Name field was changed from `Default (JPACounterEntity)` to `TestEntity>`
5. When you press CTRL+S to save, the change is reflected in the Java class:

```
package com.ibm.websphere.ejb3sample.counter;

import javax.persistence.Entity;


@Entity(name="TestEntity")
@Table(name="EJB3COUNTERTABLE")
```

Removing annotations

You can remove annotations in your source code by directly deleting the annotation in the Java™ editor or by using the Annotations view.

Before you begin

In the Java EE perspective, if the Annotations view does not appear in the the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Related concepts:
Creating and configuring Java EE modules using annotations
Scope and placement of annotations
Using the Annotations view

Related tasks:
Defining your own Annotations
Adding annotations
Editing annotations
Overridden annotations
Excluding files from annotation scanning

Removing annotations using the Annotations view

You can remove annotations in your source code by using the Annotations view.

Before you begin

In the Java™ EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Procedure

1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. Click the Annotations view tab.
3. In the navigation tree, navigate to the annotation that you want to delete.
4. Click the delete annotation icon from the navigation tree.
5. Save your file by pressing CTRL+S.

Removing annotations manually

You can remove annotations in your source code by directly deleting the annotation in the Java™ editor or by using the Annotations view.

Procedure

1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. Highlight the annotation that you want to remove, and right click and select **Cut** . You might need to remove unused imports following the delete operation. If a warning icon appears beside an import statement, right click the icon, and select **Quick Fix**. Select **Remove unused imports**.
3. Save your file by pressing CTRL+S.

Overridden annotations

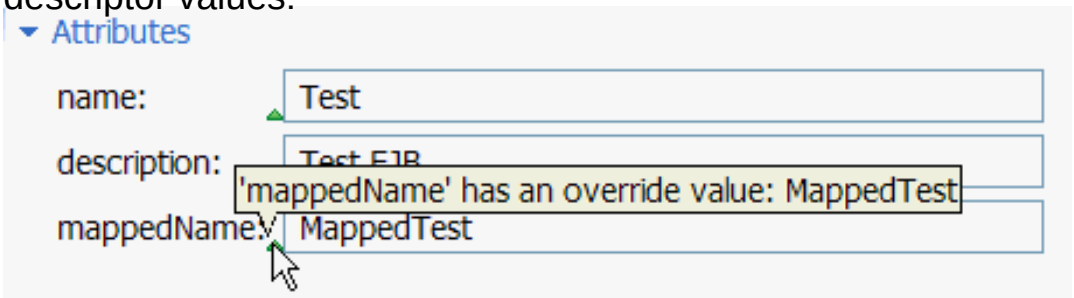
The Annotations view detects if the value of an annotation or a parameter has been overridden by a deployment descriptor value.

Before you begin

In the Java™ EE perspective, if the Annotations view does not appear in the bottom of the workspace, you need to add the Annotations view. For more information, see Using the Annotations view.

Procedure

1. In the Enterprise Explorer view, double-click your Java class to open the file in the Java editor.
2. Click the Annotations view tab.
3. Highlight the annotation that you want to view.
4. The annotation view indicates values that are overridden by deployment descriptor values:



   By hovering over the Override icon (▲) , you can see the default value for the attribute.

Related concepts:
Creating and configuring Java EE modules using annotations
Scope and placement of annotations
Using the Annotations view

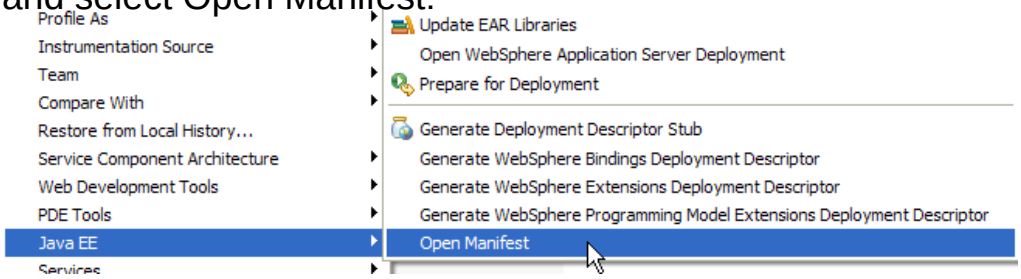
Related tasks:
Defining your own Annotations
Adding annotations
Editing annotations
Removing annotations
Excluding files from annotation scanning

Excluding files from annotation scanning

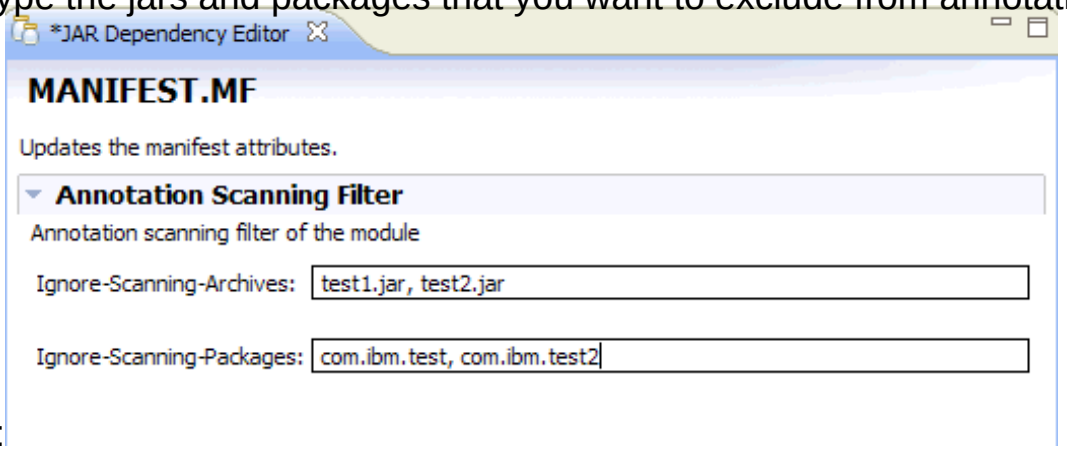
Using the annotation exclusion tool, you can indicate files that you want to exclude from annotation scanning.

Procedure

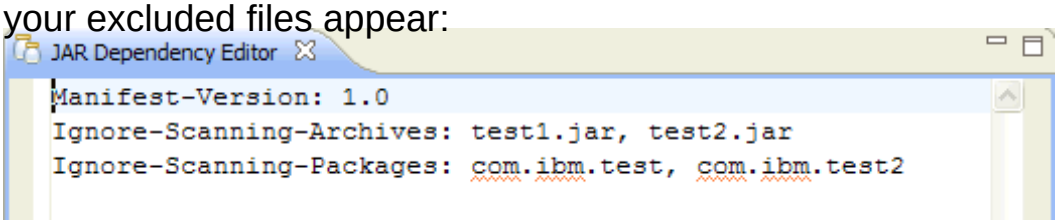
1. **To use the annotation exclusion tool**:
   A. **In an EAR project**: In the Enterprise Explorer view, right-click your EAR file, and select Open Manifest:



   B. **In an EJB or web project**: In the Enterprise Explorer view, right-click your MANIFEST.MF file (located in the META-INF folder under the ejbModule folder in EJB projects and under the WebContent folder of web projects), and select **Open With** > **Manifest Editor**.
2. In the Design view of the Manifest editor, in the **Annotation Scanning Filter** section, type the jars and packages that you want to exclude from annotation scanning:



3. Press CTRL+S to save. When you open the Source view of the Manifest editor, your excluded files appear:



Related concepts:
Creating and configuring Java EE modules using annotations
Scope and placement of annotations
Using the Annotations view

Related tasks:
Defining your own Annotations
Adding annotations
Editing annotations

Types of annotations

Java™ EE 5 and Java EE 6 define a number of types or groups of annotations, defined in a number of Java Specification Requests (JSRs).
Java EE 6 API
Java EE 5 API

For additional information on Java EE 6.0, see the official specification: JSR 316: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification
For additional information on Java EE 5.0, see the official specification: JSR 244: Java Platform, Enterprise Edition 5 (Java EE 5) Specification

JavaTM Platform, Enterprise Edition, v 5.0 API Specifications.

Java EE 5 Platform Packages	
javax.activation	The JavaBeans(TM) Activation Framework is used by the JavaMail(TM) API to manage MIME data.
javax.annotation	This package defines the common annotations.
javax.annotation.security	This package contains the security common annotations.
javax.ejb	The javax.ejb package contains the Enterprise JavaBeans classes and interfaces that define the contracts between the enterprise bean and its clients and between the enterprise bean and the EJB container.
javax.ejb.spi	The javax.ejb.spi package defines interfaces that are implemented by the EJB container.
javax.el	Provides the API for the **Unified Expression Language** shared by the JSP 2.1 and JSF 1.2 technologies.
javax.enterprise.deploy.model	Provides Tool Vendor implementation classes.
javax.enterprise.deploy.model.exceptions	Provides Tool Vendor exception implementation classes.
javax.enterprise.deploy.shared	Provides shared objects for Tool Vendor and Product Vendor implementation classes.
javax.enterprise.deploy.shared.factories	Provides shared factory manager object for Tool Vendor and Product Vendor implementation classes.
javax.enterprise.deploy.spi	Provides J2EE Product Vendor implementation classes.
javax.enterprise.deploy.spi.exceptions	Provides J2EE Product Vendor deployment exception implementation classes.
javax.enterprise.deploy.spi.factories	Provides J2EE Product Vendor deployment factory implementation classes.

javax.enterprise.deploy.spi.status	Provides J2EE Product Vendor deployment status implementation classes.
javax.faces	Top level classes for the JavaServer(tm) Faces API.
javax.faces.application	APIs that are used to link an application's business logic objects to JavaServer Faces, as well as convenient pluggable mechanisms to manage the execution of an application that is based on JavaServer Faces.
javax.faces.component	Fundamental APIs for user interface components.
javax.faces.component.html	Specialized user interface component classes for HTML.
javax.faces.context	Classes and interfaces defining per-request state information.
javax.faces.convert	Contains classes and interfaces defining converters.
javax.faces.el	**DEPRECATED** Classes and interfaces for evaluating and processing reference expressions.
javax.faces.event	Interfaces describing events and event listeners, and concrete event implementation classes.
javax.faces.lifecycle	Classes and interfaces defining lifecycle management for the JavaServer Faces implementation.
javax.faces.model	Standard model data beans for JavaServer Faces.
javax.faces.render	Classes and interfaces defining the rendering model.
javax.faces.validator	Interface defining the validator model, and concrete validator implementation classes.
javax.faces.webapp	Classes required for integration of JavaServer Faces into web applications, including a standard servlet, base classes for JSP custom component tags, and concrete tag implementations for core tags.
javax.interceptor	The javax.interceptor package contains classes and interfaces for use with EJB interceptors.

javax.jms	The Java Message Service (JMS) API provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages.
javax.jws	
javax.jws.soap	
javax.mail	Classes modeling a mail system.
javax.mail.event	Listeners and events for the JavaMail API.
javax.mail.internet	Classes specific to Internet mail systems.
javax.mail.search	Message search terms for the JavaMail API.
javax.mail.util	Utility classes.
javax.management.j2ee	Provides the J2EE Management Enterprise Bean component (MEJB) interfaces.
javax.management.j2ee.statistics	Provides the standard interfaces for accessing performance data from J2EE managed objects Package Specification JSR 77, J2EE Management Related Documentation For overviews, tutorials, examples, guides, and tool documentation, please see: J2EE Tools
javax.persistence	The javax.persistence package contains the classes and interfaces that define the contracts between a persistence provider and the managed classes and the clients of the Java Persistence API.
javax.persistence.spi	The javax.persistence.spi package defines the classes and interfaces that are implemented by the persistence provider and the Java EE container for use by the container, provider, and/or Persistence bootstrap class in deployment and bootstrapping.
javax.resource	The javax.resource package is the top-level package for the J2EE Connector API specification.
javax.resource.cci	The javax.resource.cci package contains API specification for the Common Client Interface (CCI).

javax.resource.spi	The javax.resource.spi package contains APIs for the system contracts defined in the J2EE Connector Architecture specification.
javax.resource.spi.endpoint	This package contains system contracts for service endpoint interactions.
javax.resource.spi.security	The javax.resource.spi.security package contains APIs for the security management contract.
javax.resource.spi.work	This package contains APIs for the work management contract.
javax.security.jacc	This package contains the Java Authorization Contract for Containers API
javax.servlet	The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.
javax.servlet.http	The javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container.
javax.servlet.jsp	Classes and interfaces for the Core JSP 2.1 API.
javax.servlet.jsp.el	Provides the `ELResolver` classes that define the object resolution rules that must be supported by a JSP container with the new unified Expression Language.
javax.servlet.jsp.tagext	Classes and interfaces for the definition of JavaServer Pages Tag Libraries.
javax.transaction	Provides the API that defines the contract between the transaction manager and the various parties involved in a distributed transaction namely : resource manager, application, and application server.

javax.transaction.xa	Provides the API that defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects (supplied by the resource manager driver) in JTA transactions.
javax.xml.bind	Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities.
javax.xml.bind.annotation	Defines annotations for customizing Java program elements to XML Schema mapping.
javax.xml.bind.annotation.adapters	`XmlAdapter` and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB.
javax.xml.bind.attachment	This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format.
javax.xml.bind.helpers	**JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces.**
javax.xml.bind.util	Useful client utility classes.
javax.xml.registry	This package and its sub-packages describe the API classes and interfaces for the JAXR API.
javax.xml.registry.infomodel	This package describes the information model for the JAXR API.
javax.xml.rpc	This package contains the core JAX-RPC APIs for the client programming model.
javax.xml.rpc.encoding	This package defines APIs for the extensible type mapping framework.
javax.xml.rpc.handler	This package defines APIs for SOAP Message Handlers

javax.xml.rpc.handler.soap	This package defines APIs for SOAP Message Handlers
javax.xml.rpc.holders	This package contains the standard Java Holder classes.
javax.xml.rpc.server	This package defines APIs for the servlet based JAX-RPC endpoint model.
javax.xml.rpc.soap	This package defines APIs specific to the SOAP binding.
javax.xml.soap	Provides the API for creating and building SOAP messages.
javax.xml.stream	
javax.xml.stream.events	
javax.xml.stream.util	
javax.xml.ws	This package contains the core JAX-WS APIs.
javax.xml.ws.handler	This package defines APIs for message handlers.
javax.xml.ws.handler.soap	This package defines APIs for SOAP message handlers.
javax.xml.ws.http	This package defines APIs specific to the HTTP binding.
javax.xml.ws.soap	This package defines APIs specific to the SOAP binding.
javax.xml.ws.spi	This package defines SPIs for JAX-WS 2.0.
	Java EE 5 SDK

Package javax.annotation

This package defines the common annotations. **See:**
Description

Enum Summary	
Resource.AuthenticationType	The two possible authentication types for a resource.
Annotation Types Summary	
Generated	The Generated annoation is used to mark source code that has been generated.
PostConstruct	The PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization.
PreDestroy	The PreDestroy annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container.
Resource	The Resource annotation marks a resource that is needed by the application.
Resources	This class is used to allow multiple resources declarations.

Java EE 5 SDK

Package javax.annotation Description

This package defines the common annotations.

Package javax.annotation.security

This package contains the security common annotations. **See:**
Description

Annotation Types Summary

DeclareRoles	Used by application to declare roles.
DenyAll	Specifies that no security roles are allowed to invoke the specified method(s) - i.e that the methods are to be excluded from execution in the J2EE container.
PermitAll	Specifies that all security roles are allowed to invoke the specified method(s) i.e that the specified method(s) are "unchecked".
RolesAllowed	Specifies the list of roles permitted to access method(s) in an application.
RunAs	Defines the identity of the application during execution in a J2EE container.

Java EE 5 SDK

Package javax.annotation.security Description

This package contains the security common annotations.

Package javax.ejb

The javax.ejb package contains the Enterprise JavaBeans classes and interfaces that define the contracts between the enterprise bean and its clients and between the enterprise bean and the EJB container.

See:
Description

Interface Summary	
EJBContext	The EJBContext interface provides an instance with access to the container-provided runtime context of an enterprise Bean instance.
EJBHome	The EJBHome interface must be extended by all enterprise Beans' remote home interfaces.
EJBLocalHome	The EJBLocalHome interface must be extended by all enterprise Beans' local home interfaces.
EJBLocalObject	The EJBLocalObject interface must be extended by all enterprise Beans' local interfaces.
EJBMetaData	The EJBMetaData interface allows a client to obtain the enterprise Bean's meta-data information.
EJBObject	The EJBObject interface is extended by all enterprise Beans' remote interfaces.
EnterpriseBean	The EnterpriseBean interface must be implemented by every enterprise Bean class.
EntityBean	The EntityBean interface is implemented by every entity enterprise Bean class.
EntityContext	The EntityContext interface provides an instance with access to the container-provided runtime context of an entity enterprise Bean instance.
Handle	The Handle interface is implemented by all EJB object handles.
HomeHandle	The HomeHandle interface is implemented by all home object handles.

MessageDrivenBean	The MessageDrivenBean interface is implemented by every message-driven enterprise Bean class.
MessageDrivenContext	The MessageDrivenContext interface provides access to the runtime message-driven context that the container provides for a message-driven enterprise Bean instance.
SessionBean	The SessionBean interface is implemented by every session enterprise Bean class.
SessionContext	The SessionContext interface provides access to the runtime session context that the container provides for a session enterprise Bean instance.
SessionSynchronization	The SessionSynchronization interface allows a session Bean instance to be notified by its container of transaction boundaries.
TimedObject	The TimedObject interface contains the callback method that is used to deliver timer expiration notifications.
Timer	The Timer interface contains information about a timer that was created through the EJB Timer Service.
TimerHandle	The TimerHandle interface is implemented by all EJB timer handles.
TimerService	The TimerService interface provides enterprise bean components with access to the container-provided Timer Service.
Enum Summary	
TransactionAttributeType	
TransactionManagementType	
Exception Summary	
AccessLocalException	An AccessLocalException is thrown to indicate that the caller does not have permission to call the method.

ConcurrentAccessException	A ConcurrentAccessException indicates that the client has attempted an invocation on a stateful session bean while another invocation is in progress.
CreateException	The CreateException exception must be included in the throws clauses of all create methods defined in an enterprise Bean's home interface.
DuplicateKeyException	The DuplicateKeyException exception is thrown if an entity EJB object cannot be created because an object with the same key already exists.
EJBAccessException	This exception indicates that client access to a business method was denied.
EJBException	The EJBException exception is thrown by an enterprise Bean instance to its container to report that the invoked business method or callback method could not be completed because of an unexpected error (e.g. the instance failed to open a database connection).
EJBTransactionRequiredException	This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.
EJBTransactionRolledbackException	This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back.
FinderException	The FinderException exception must be included in the throws clause of every findMETHOD(...) method of an entity Bean's home interface.
NoSuchEJBException	A NoSuchEJBException is thrown if an attempt is made to invoke a method on an object that no longer exists.

NoSuchEntityException	The NoSuchEntityException exception is thrown by an Entity Bean instance to its container to report that the invoked business method or callback method could not be completed because of the underlying entity was removed from the database.
NoSuchObjectLocalException	A NoSuchObjectLocalException is thrown if an attempt is made to invoke a method on an object that no longer exists.
ObjectNotFoundException	The ObjectNotFoundException exception is thrown by a finder method to indicate that the specified EJB object does not exist.
RemoveException	The RemoveException exception is thrown at an attempt to remove an EJB object when the enterprise Bean or the container does not allow the EJB object to be removed.
TransactionRequiredLocalException	This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.
TransactionRolledbackLocalException	This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back.
Annotation Types Summary	
ActivationConfigProperty	
ApplicationException	Applied to an exception to denote that it is an application exception and should be reported to the client directly(i.e., unwrapped).
EJB	Indicates a dependency on the local or remote view of an Enterprise Java Bean.
EJBs	Declares multiple TYPE-level @EJB annotations.
Init	Designates a method of a session bean that corresponds to the create method of an adapted Home interface or an adapted Local Home interface.

Local	When used on the bean class, declares the local business interface(s) for a session bean.
LocalHome	Declares the Local Home or adapted Local Home interface for a session bean.
MessageDriven	Component-defining annotation for a message driven bean.
PostActivate	Designates a method to receive a callback after a stateful session bean has been activated.
PrePassivate	Designates a method to receive a callback before a stateful session bean is passivated.
Remote	Declares the remote business interface(s) for a session bean.
RemoteHome	Declares the Remote Home or adapted Remote Home interface for a session bean.
Remove	Applied to a business method of a stateful session bean class.
Stateful	Component-defining annotation for a stateful session bean.
Stateless	Component-defining annotation for a stateless session bean.
Timeout	Designates a method on a stateless session bean class or message driven bean class that should receive EJB timer expirations for that bean.
TransactionAttribute	When applied at the TYPE-level, designates the default transaction attribute for all business methods of the session or message driven bean.
TransactionManagement	Declares whether a session bean or message driven bean has container managed transactions or bean managed transactions.

Package javax.ejb Description

The javax.ejb package contains the Enterprise JavaBeans classes and interfaces that define the contracts between the enterprise bean and its clients and between the enterprise bean and the EJB container.

Package javax.ejb.spi

The javax.ejb.spi package defines interfaces that are implemented by the EJB container. See Description

Interface Summary	
HandleDelegate	The HandleDelegate interface is implemented by the EJB container.

Java EE 5 SDK

Package javax.ejb.spi Description

The javax.ejb.spi package defines interfaces that are implemented by the EJB container. These interfaces are not used by application components.

Package javax.persistence

The javax.persistence package contains the classes and interfaces that define the contracts between a persistence provider and the managed classes and the clients of the Java Persistence API.

See:

Description

Interface Summary	
EntityManager	Interface used to interact with the persistence context.
EntityManagerFactory	The `EntityManagerFactory` interface is used by the application to obtain an application-managed entity manager.
EntityTransaction	The `EntityTransaction` interface is used to control resource transactions on resource-local entity managers.
Query	Interface used to control query execution.
Class Summary	
Persistence	Bootstrap class that is used to obtain an `EntityManagerFactory`.
Enum Summary	
CascadeType	Defines the set of cascadable operations that are propagated to the associated entity.
DiscriminatorType	Defines supported types of the discriminator column.
EnumType	Defines mapping for the enumerated types.
FetchType	Defines strategies for fetching data from the database.
FlushModeType	Flush mode setting.
GenerationType	Defines the types of primary key generation.
InheritanceType	Defines inheritance strategy options.
LockModeType	Lock modes that can be specified by means of the `EntityManager.lock()` method.
PersistenceContextType	Specifies whether a transaction-scoped or extended persistence context is to be used in `PersistenceContext`.

TemporalType	Type used to indicate a specific mapping of `Date` or `Calendar`.
Exception Summary	
EntityExistsException	Thrown by the persistence provider when `EntityManager.persist(Object)` is called and the entity already exists.
EntityNotFoundException	Thrown by the persistence provider when an entity reference obtained by `EntityManager.getReference(Class,Object)` is accessed but the entity does not exist.
NonUniqueResultException	Thrown by the persistence provider when `getSingleResult()` is executed on a query and there is more than one result from the query.
NoResultException	Thrown by the persistence provider when `getSingleResult()` is executed on a query and there is no result to return.
OptimisticLockException	Thrown by the persistence provider when an optimistic locking conflict occurs.
PersistenceException	Thrown by the persistence provider when a problem occurs.
RollbackException	Thrown by the persistence provider when the `EntityTransaction.commit()` fails.
TransactionRequiredException	Thrown by the persistence provider when a transaction is required but is not active.
Annotation Types Summary	
AssociationOverride	This annotation is used to override a many-to-one or one-to-one mapping of property or field for an entity relationship.
AssociationOverrides	This annotation is used to override mappings of multiple many-to-one or one-to-one relationship properties or fields.

AttributeOverride	The `AttributeOverride` annotation is used to override the mapping of a `Basic` (whether explicit or default) property or field or Id property or field.
AttributeOverrides	Is used to override mappings of multiple properties or fields.
Basic	The `Basic` annotation is the simplest type of mapping to a database column.
Column	Is used to specify a mapped column for a persistent property or field.
ColumnResult	References name of a column in the SELECT clause of a SQL query - i.e., column alias, if applicable.
DiscriminatorColumn	Is used to define the discriminator column for the `SINGLE_TABLE` and `JOINED` inheritance mapping strategies.
DiscriminatorValue	Is used to specify the value of the discriminator column for entities of the given type.
Embeddable	Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity.
Embedded	Defines a persistent field or property of an entity whose value is an instance of an embeddable class.
EmbeddedId	Is applied to a persistent field or property of an entity class or mapped superclass to denote a composite primary key that is an embeddable class.
Entity	Specifies that the class is an entity.
EntityListeners	Specifies the callback listener classes to be used for an entity or mapped superclass.
EntityResult	References an entity in the SELECT clause of a SQL query.
Enumerated	Specifies that a persistent property or field should be persisted as a enumerated type.

ExcludeDefaultListeners	Specifies that the invocation of default listeners is to be excluded for the entity class (or mapped superclass) and its subclasses.
ExcludeSuperclassListeners	Specifies that the invocation of superclass listeners is to be excluded for the entity class (or mapped superclass) and its subclasses.
FieldResult	Is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.
GeneratedValue	Provides for the specification of generation strategies for the values of primary keys.
Id	Specifies the primary key property or field of an entity.
IdClass	Specifies a composite primary key class that is mapped to multiple fields or properties of the entity.
Inheritance	Defines the inheritance strategy to be used for an entity class hierarchy.
JoinColumn	Is used to specify a mapped column for joining an entity association.
JoinColumns	Defines mapping for the composite foreign keys.
JoinTable	This annotation is used in the mapping of associations.
Lob	Specifies that a persistent property or field should be persisted as a large object to a database-supported large object type.
ManyToMany	Defines a many-valued association with many-to-many multiplicity.
ManyToOne	This annotation defines a single-valued association to another entity class that has many-to-one multiplicity.
MapKey	Is used to specify the map key for associations of type `Map`.
MappedSuperclass	Designates a class whose mapping information is applied to the entities that inherit from it.

NamedNativeQueries	Is used to specify an array of native SQL named queries.
NamedNativeQuery	Is used to specify a native SQL named query.
NamedQueries	Specifies an array of named Java Persistence query language queries.
NamedQuery	Is used to specify a named query in the Java Persistence query language, which is a static query expressed in metadata.
OneToMany	Defines a many-valued association with one-to-many multiplicity.
OneToOne	This annotation defines a single-valued association to another entity that has one-to-one multiplicity.
OrderBy	This annotation specifies the ordering of the elements of a collection valued association at the point when the association is retrieved.
PersistenceContext	Expresses a dependency on an `EntityManager` persistence context.
PersistenceContexts	Declares one or more `PersistenceContext` annotations.
PersistenceProperty	Describes a single container or persistence provider property.
PersistenceUnit	Expresses a dependency on an `EntityManagerFactory`.
PersistenceUnits	Declares one or more `PersistenceUnit` annotations.
PostLoad	Is used to specify callback methods for the corresponding lifecycle event.
PostPersist	Is used to specify callback methods for the corresponding lifecycle event.
PostRemove	Is used to specify callback methods for the corresponding lifecycle event.
PostUpdate	Is used to specify callback methods for the corresponding lifecycle event.

PrePersist	Is used to specify callback methods for the corresponding lifecycle event.
PreRemove	Is used to specify callback methods for the corresponding lifecycle event.
PreUpdate	Is used to specify callback methods for the corresponding lifecycle event.
PrimaryKeyJoinColumn	This annotation specifies a primary key column that is used as a foreign key to join to another table.
PrimaryKeyJoinColumns	This annotation groups `PrimaryKeyJoinColumn` annotations.
QueryHint	An implementation-specific `Query` hint.
SecondaryTable	This annotation is used to specify a secondary table for the annotated entity class.
SecondaryTables	This annotation is used to specify multiple secondary tables for an entity.
SequenceGenerator	This annotation defines a primary key generator that may be referenced by name when a generator element is specified for the `GeneratedValue` annotation.
SqlResultSetMapping	This annotation is used to specify the mapping of the result of a native SQL query.
SqlResultSetMappings	This annotation is used to define one or more `SqlResultSetMapping`.
Table	This annotation specifies the primary table for the annotated entity.
TableGenerator	This annotation defines a primary key generator that may be referenced by name when a generator element is specified for the `GeneratedValue` annotation.
Temporal	This annotation must be specified for persistent fields or properties of type `Date` and `Calendar`.

Transient	This annotation specifies that the property or field is not persistent.
UniqueConstraint	This annotation is used to specify that a unique constraint is to be included in the generated DDL for a primary or secondary table.
Version	This annotation specifies the version field or property of an entity class that serves as its optimistic lock value.

Package javax.persistence Description

The javax.persistence package contains the classes and interfaces that define the contracts between a persistence provider and the managed classes and the clients of the Java Persistence API.

Package javax.persistence.spi

The javax.persistence.spi package defines the classes and interfaces that are implemented by the persistence provider and the Java EE container for use by the container, provider, and/or Persistence bootstrap class in deployment and bootstrapping.

Interface Summary	
ClassTransformer	A persistence provider supplies an instance of this interface to the `PersistenceUnitInfo.addTransformer` method.
PersistenceProvider	Interface implemented by a persistence provider.
PersistenceUnitInfo	Interface implemented by the container and used by the persistence provider when creating an `EntityManagerFactory`.
Enum Summary	
PersistenceUnitTransactionType	This enum class defines whether the entity managers created by the `EntityManagerFactory` will be JTA or resource-local entity managers.

Package javax.persistence.spi Description

Java EE 5 SDK

The javax.persistence.spi package defines the classes and interfaces that are implemented by the persistence provider and the Java EE container for use by the container, provider, and/or Persistence bootstrap class in deployment and bootstrapping.

Package javax.resource

The javax.resource package is the top-level package for the J2EE Connector API specification. See:

Description

Interface Summary	
Referenceable	The Referenceable interface extends the javax.naming.Referenceable interface.
Exception Summary	
NotSupportedException	A `NotSupportedException` is thrown to indicate that callee (resource adapter or application server for system contracts) cannot execute an operation because the operation is not a supported feature.
ResourceException	This is the root interface of the exception hierarchy defined for the Connector architecture.

Java EE 5 SDK

Package javax.resource Description

The javax.resource package is the top-level package for the J2EE Connector API specification.

Package javax.resource.cci

The javax.resource.cci package contains API specification for the Common Client Interface (CCI). See:

Description

Interface Summary	
Connection	A Connection represents an application-level handle that is used by a client to access the underlying physical connection.
ConnectionFactory	`ConnectionFactory` provides an interface for getting connection to an EIS instance.
ConnectionMetaData	The interface `ConnectionMetaData` provides information about an EIS instance connected through a Connection instance.
ConnectionSpec	ConnectionSpec is used by an application component to pass connection request-specific properties to the ConnectionFactory.
IndexedRecord	IndexedRecord represents an ordered collection of record elements based on the `java.util.List` interface.
Interaction	The `javax.resource.cci.Interaction` enables a component to execute EIS functions.
InteractionSpec	An InteractionSpec holds properties for driving an Interaction with an EIS instance.
LocalTransaction	The LocalTransaction defines a transaction demarcation interface for resource manager local transactions.
MappedRecord	The interface `javax.resource.cci.MappedRecord` is used for key-value map based representation of record elements.
MessageListener	This serves as a request-response message listener type that message endpoints (message-driven beans) may implement.

Record	The `javax.resource.cci.Record` interface is the base interface for the representation of an input or output to the execute methods defined on an Interaction.
RecordFactory	The RecordFactory interface is used for creating MappedRecord and IndexedRecord instances.
ResourceAdapterMetaData	The interface `javax.resource.cci.ResourceAdapterMetaData` provides information about capabilities of a resource adapter implementation.
ResultSet	A ResultSet represents tabular data that is retrieved from an EIS instance by the execution of an Interaction..
ResultSetInfo	The interface `javax.resource.cci.ResultSetInfo` provides information on the support provided for ResultSet by a connected EIS instance.
Streamable	Streamable interface enables a resource adapter to extract data from an input Record or set data into an output Record as a stream of bytes.
Exception Summary	
ResourceWarning	A `ResourceWarning` provides information on warnings related to execution of an interaction with an EIS.

Package javax.resource.cci Description

The javax.resource.cci package contains API specification for the Common Client Interface (CCI).

Package javax.resource.spi

The javax.resource.spi package contains APIs for the system contracts defined in the J2EE Connector Architecture specification. **See:** **Description**

Interface Summary

ActivationSpec	This interface serves as a marker.
BootstrapContext	This provides a mechanism to pass a bootstrap context to a resource adapter instance when it is bootstrapped.
ConnectionEventListener	The `ConnectionEventListener` interface provides an event callback mechanism to enable an application server to receive notifications from a `ManagedConnection` instance.
ConnectionManager	ConnectionManager interface provides a hook for the resource adapter to pass a connection request to the application server.
ConnectionRequestInfo	The ConnectionRequestInfo interface enables a resource adapter to pass its own request specific data structure across the connection request flow.
DissociatableManagedConnection	This is a mix-in interface that may be optionally implemented by a `ManagedConnection` implementation.
LazyAssociatableConnectionManager	This is a mix-in interface that may be optionally implemented by a `ConnectionManager` implementation.
LazyEnlistableConnectionManager	This is a mix-in interface that may be optionally implemented by a `ConnectionManager` implementation.
LazyEnlistableManagedConnection	This is a mix-in interface that may be optionally implemented by a `ManagedConnection` implementation.
LocalTransaction	LocalTransaction interface provides support for transactions that are managed internal to an EIS resource manager, and do not require an external transaction manager.

ManagedConnection	ManagedConnection instance represents a physical connection to the underlying EIS.
ManagedConnectionFactory	ManagedConnectionFactory instance is a factory of both ManagedConnection and EIS-specific connection factory instances.
ManagedConnectionMetaData	The ManagedConnectionMetaData interface provides information about the underlying EIS instance associated with a ManagedConnection instance.
ResourceAdapter	This represents a resource adapter instance and contains operations for lifecycle management and message endpoint setup.
ResourceAdapterAssociation	This interface specifies the methods to associate a `ResourceAdapter` object with other objects that implement this interface like `ManagedConnectionFactory` and `ActivationSpec`.
ValidatingManagedConnectionFactory	This interface is implemented by a `ManagedConnectionFactory` instance that supports the ability to validate `ManagedConnection` objects.
XATerminator	The XATerminator interface is used for transaction completion and crash recovery flows.
Class Summary	
ConnectionEvent	The ConnectionEvent class provides information about the source of a connection related event.A ConnectionEvent instance contains the following information: Type of the connection event ManagedConnection instance that generated the connection event.
Exception Summary	

ApplicationServerInternalException	An `ApplicationServerInternalException` is thrown by an application server to indicate error conditions specific to an application server.
CommException	This indicates errors related to failed or interrupted communication with an EIS instance.
EISSystemException	An `EISSystemException` is used to indicate any EIS specific system-level error conditions.
IllegalStateException	An `IllegalStateException` is thrown from a method if the callee (resource adapter or application server for system contracts) is in an illegal or inappropriate state for the method invocation.
InvalidPropertyException	This exception is thrown to indicate invalid configuration property settings.
LocalTransactionException	A `LocalTransactionException` represents various error conditions related to the local transaction management contract.
ResourceAdapterInternalException	A `ResourceAdapterInternalException` indicates any system-level error conditions related to a resource adapter.
ResourceAllocationException	A `ResourceAllocationException` can be thrown by an application server or resource adapter to indicate any failure to allocate system resources (example: threads, physical connections).
SecurityException	A `SecurityException` indicates error conditions related to the security contract between an application server and resource adapter.
SharingViolationException	This is thrown to indicate a connection sharing violation.

UnavailableException	This is thrown to indicate that a service is unavailable.

Package javax.resource.spi Description

The javax.resource.spi package contains APIs for the system contracts defined in the J2EE Connector Architecture specification.

Package javax.resource.spi.endpoint

This package contains system contracts for service endpoint interactions. **See:** **Description**

Interface Summary	
MessageEndpoint	This defines a contract for a message endpoint.
MessageEndpointFactory	This serves as a factory for creating message endpoints.

Java EE 5 SDK

Package javax.resource.spi.endpoint Description

This package contains system contracts for service endpoint interactions.

Package javax.resource.spi.security

The javax.resource.spi.security package contains APIs for the security management contract. See:

Description

Interface Summary	
GenericCredential	**Deprecated.** *The preferred way to represent generic credential information is via the `org.ietf.jgss.GSSCredential` interface in J2SE Version 1.4, which provides similar functionality.*
Class Summary	
PasswordCredential	The class PasswordCredential acts as a holder for username and password.

Java EE 5 SDK

Package javax.resource.spi.security Description

The javax.resource.spi.security package contains APIs for the security management contract.

Package javax.resource.spi.work

This package contains APIs for the work management contract. **See:** Description

Interface Summary	
Work	This models a `Work` instance that would be executed by a `WorkManager` upon submission.
WorkListener	This models a `WorkListener` instance which would be notified by the `WorkManager` when the various `Work` processing events (work accepted, work rejected, work started, work completed) occur.
WorkManager	This interface models a `WorkManager` which provides a facility to submit `Work` instances for execution.
Class Summary	
ExecutionContext	This class models an execution context (transaction, security, etc) with which the `Work` instance must be executed.
WorkAdapter	This class is provided as a convenience for easily creating `WorkListener` instances by extending this class and overriding only those methods of interest.
WorkEvent	This class models the various events that occur during the processing of a `Work` instance.
Exception Summary	
WorkCompletedException	This exception is thrown by a `WorkManager` to indicate that a submitted `Work` instance has completed with an exception.
WorkException	A common base class for all `Work` processing related exceptions.
WorkRejectedException	This exception is thrown by a `WorkManager` to indicate that a submitted `Work` instance has been rejected.

Java EE 5 SDK

Package javax.resource.spi.work Description

This package contains APIs for the work management contract.

Package javax.xml.bind.annotation

Defines annotations for customizing Java program elements to XML Schema mapping. See:

Description

Interface Summary	
DomHandler<ElementT,ResultT extends Result>	Converts an element (and its descendants) from/to DOM (or similar) representation.
Class Summary	
W3CDomHandler	`DomHandler` implementation for W3C DOM (`org.w3c.dom` package.)
XmlElement.DEFAULT	Used in `XmlElement.type()` to signal that the type be inferred from the signature of the property.
XmlElementDecl.GLOBAL	Used in `XmlElementDecl.scope()` to signal that the declaration is in the global scope.
XmlElementRef.DEFAULT	Used in `XmlElementRef.type()` to signal that the type be inferred from the signature of the property.
XmlSchemaType.DEFAULT	Used in `XmlSchemaType.type()` to signal that the type be inferred from the signature of the property.
XmlType.DEFAULT	Used in `XmlType.factoryClass()` to signal that either factory mehod is not used or that it's in the class with this `XmlType` itself.
Enum Summary	
XmlAccessOrder	Used by XmlAccessorOrder to control the ordering of properties and fields in a JAXB bound class.
XmlAccessType	Used by XmlAccessorType to control serialization of fields or properties.
XmlNsForm	Enumeration of XML Schema namespace qualifications.
Annotation Types Summary	
XmlAccessorOrder	Controls the ordering of fields and properties in a class.

XmlAccessorType	Controls whether fields or Javabean properties are serialized by default.
XmlAnyAttribute	Maps a JavaBean property to a map of wildcard attributes.
XmlAnyElement	Maps a JavaBean property to XML infoset representation and/or JAXB element.
XmlAttachmentRef	Marks a field/property that its XML form is a uri reference to mime content.
XmlAttribute	Maps a JavaBean property to a XML attribute.
XmlElement	Maps a JavaBean property to a XML element derived from property name.
XmlElementDecl	Maps a factory method to a XML element.
XmlElementRef	Maps a JavaBean property to a XML element derived from property's type.
XmlElementRefs	Marks a property that refers to classes with `XmlElement` or JAXBElement.
XmlElements	A container for multiple @`XmlElement` annotations.
XmlElementWrapper	Generates a wrapper element around XML representation.
XmlEnum	Maps an enum type `Enum` to XML representation.
XmlEnumValue	Maps an enum constant in `Enum` type to XML representation.
XmlID	Maps a JavaBean property to XML ID.
XmlIDREF	Maps a JavaBean property to XML IDREF.
XmlInlineBinaryData	Disable consideration of XOP encoding for datatypes that are bound to base64-encoded binary data in XML.
XmlList	Used to map a property to a list simple type.
XmlMimeType	Associates the MIME type that controls the XML representation of the property.
XmlMixed	Annotate a JavaBean multi-valued property to support mixed content.

XmlNs	Associates a namespace prefix with a XML namespace URI.
XmlRegistry	Marks a class that has `XmlElementDecl`s.
XmlRootElement	Maps a class or an enum type to an XML element.
XmlSchema	Maps a package name to a XML namespace.
XmlSchemaType	Maps a Java type to a simple schema built-in type.
XmlSchemaTypes	A container for multiple `@XmlSchemaType` annotations.
XmlTransient	Prevents the mapping of a JavaBean property to XML representation.
XmlType	Maps a class or an enum type to a XML Schema type.
XmlValue	Enables mapping a class to a XML Schema complex type with a simpleContent or a XML Schema simple type.

Package javax.xml.bind.annotation Description

Defines annotations for customizing Java program elements to XML Schema mapping.

Package Specification

The following table shows the JAXB mapping annotations that can be associated with each program element.

Program Element	JAXB annotation
Package	
Class	
Enum type	
JavaBean Property/field	
Parameter	

Terminology

JavaBean property and field: For the purposes of mapping, there is no semantic difference between a field and a JavaBean property. Thus, an annotation that can be applied to a JavaBean property can always be applied to a field. Hence in the Javadoc documentation, for brevity, the term JavaBean property or property is used to mean either JavaBean property or a field. Where required, both are explicitly mentioned. **top level class:** For the purpose of mapping, there is no semantic difference between a top level class and a static nested class. Thus, an annotation that can be applied to a top level class, can always be applied to a nested static class. Hence in the Javadoc documentation, for brevity, the term "top level class" or just class is used to mean either a top level class or a nested static class. **mapping annotation:**A JAXB 2.0 defined program annotation based on the JSR 175 programming annotation facility.

Common Usage

Constraints

The following usage constraints are defined here since they apply to more than annotation:

- For a property, a given annotation can be applied to either read or write property but not both.
- A property name must be different from any other property name in any of the super classes of the class being mapped.
- A mapped field name or the decapitalized name of a mapped property must be unique within a class.

Notations

Namespace prefixes The following namespace prefixes are used in the XML Schema fragments in this package.

Prefix	Namespace	Notes
xs	http://www.w3.org/2001/XMLSchema	Namespace of XML Schema namespace
ref	http://ws-i.org/profiles/basic/1.1/xsd	Namespace for swaref schema component
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema namespace for instances

- Since:

- JAXB 2.0

	Java EE 5 SDK

Package javax.xml.bind.annotation.adapters

`XmlAdapter` and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB.

See: Description

Class Summary	
CollapsedStringAdapter	Built-in `XmlAdapter` to handle xs:token and its derived types.
HexBinaryAdapter	`XmlAdapter` for xs:hexBinary.
NormalizedStringAdapter	`XmlAdapter` to handle xs:normalizedString.
XmlAdapter<ValueType,BoundType>	Adapts a Java type for custom marshaling.
XmlJavaTypeAdapter.DEFAULT	Used in `XmlJavaTypeAdapter.type()` to signal that the type be inferred from the signature of the field, property, parameter or the class.
Annotation Types Summary	
XmlJavaTypeAdapter	Use an adapter that implements `XmlAdapter` for custom marshaling.
XmlJavaTypeAdapters	A container for multiple `@XmlJavaTypeAdapter` annotations.

Package javax.xml.bind.annotation.adapters Description

`XmlAdapter` and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB.

Package Specification

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

- The JAXB Website

Java EE 5 SDK

Package javax.xml.bind.attachment

This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format. **See:**

Description

Class Summary	
AttachmentMarshaller	Enable JAXB marshalling to optimize storage of binary data.
AttachmentUnmarshaller	Enables JAXB unmarshalling of a root document containing optimized binary data formats.

Package javax.xml.bind.attachment Description

This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format. Soap MTOM[1], XOP([2][3]) and WS-I AP[4] standardize approaches to optimized transmission of binary datatypes as an attachment. To optimally support these standards within a message passing environment, this package enables an integrated solution between a MIME-based package processor and JAXB unmarshall/marshal processes.

Package Specification

- JAXB Specification

Related Standards

- [1]SOAP Message Transmission Optimization Mechanism
- [2]XML-binary Optimized Packaging
- [3]WS-I Attachments Profile Version 1.0.
- [4]Describing Media Content of Binary Data in XML

- Since:

- JAXB 2.0

Java EE 5 SDK

Package javax.xml.bind.helpers

JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces. See: Description

Class Summary	
AbstractMarshallerImpl	Partial default Marshaller implementation.
AbstractUnmarshallerImpl	Partial default Unmarshaller implementation.
DefaultValidationEventHandler	JAXB 1.0 only default validation event handler.
NotIdentifiableEventImpl	Default implementation of the NotIdentifiableEvent interface.
ParseConversionEventImpl	Default implementation of the ParseConversionEvent interface.
PrintConversionEventImpl	Default implementation of the PrintConversionEvent interface.
ValidationEventImpl	Default implementation of the ValidationEvent interface.
ValidationEventLocatorImpl	Default implementation of the ValidationEventLocator interface.

Package javax.xml.bind.helpers Description

JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces. JAXB Providers can extend these classes and implement the abstract methods.

Package Specification

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

Java EE 5 SDK

- The JAXB Website

Package javax.xml.bind.util

Useful client utility classes. **See: Description**

Class Summary	
JAXBResult	JAXP `Result` implementation that unmarshals a JAXB object.
JAXBSource	JAXP `Source` implementation that marshals a JAXB-generated object.
ValidationEventCollector	`ValidationEventHandler` implementation that collects all events.

Package javax.xml.bind.util Description

Useful client utility classes.

Package Specification

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

Java EE 5 SDK

- The JAXB Website

Package javax.xml.ws

This package contains the core JAX-WS APIs. **See:**

Description

Interface Summary	
AsyncHandler<T>	The `AsyncHandler` interface is implemented by clients that wish to receive callback notification of the completion of service endpoint operations invoked asynchronously.
Binding	The `Binding` interface is the base interface for JAX-WS protocol bindings.
BindingProvider	The `BindingProvider` interface provides access to the protocol binding and associated context objects for request and response message processing.
Dispatch<T>	The `Dispatch` interface provides support for the dynamic invocation of a service endpoint operations.
LogicalMessage	The `LogicalMessage` interface represents a protocol agnostic XML message and contains methods that provide access to the payload of the message.
Provider<T>	Service endpoints may implement the `Provider` interface as a dynamic alternative to an SEI.
Response<T>	The `Response` interface provides methods used to obtain the payload and context of a message sent in response to an operation invocation.
WebServiceContext	A `WebServiceContext` makes it possible for a web service endpoint implementation class to access message context and security information relative to a request being served.
Class Summary	
Endpoint	A Web service endpoint.
Holder<T>	Holds a value of type T.
Service	`Service` objects provide the client view of a Web service.
WebServicePermission	This class defines web service permissions.

Enum Summary	
Service.Mode	The orientation of a dynamic client or service.
Exception Summary	
ProtocolException	The `ProtocolException` class is a base class for exceptions related to a specific protocol binding.
WebServiceException	The `WebServiceException` class is the base exception class for all JAX-WS API runtime exceptions.
Annotation Types Summary	
BindingType	The `BindingType` annotation is used to specify the binding to use for a web service endpoint implementation class.
RequestWrapper	Used to annotate methods in the Service Endpoint Interface with the request wrapper bean to be used at runtime.
ResponseWrapper	Used to annotate methods in the Service Endpoint Interface with the response wrapper bean to be used at runtime.
ServiceMode	Used to indicate whether a Provider implementation wishes to work with entire protocol messages or just with protocol message payloads.
WebEndpoint	Used to annotate the `get`*`PortName`*`()` methods of a generated service interface.
WebFault	Used to annotate service specific exception classes to customize to the local and namespace name of the fault element and the name of the fault bean.
WebServiceClient	Used to annotate a generated service interface.
WebServiceProvider	Used to annotate a Provider implementation class.
WebServiceRef	The `WebServiceRef` annotation is used to define a reference to a web service and (optionally) an injection target for it.

WebServiceRefs	The `WebServiceRefs` annotation allows multiple web service references to be declared at the class level.

Package javax.xml.ws Description

This package contains the core JAX-WS APIs.

Java EE 5 SDK

Submit a bug or featureCopyright 2006 Sun Microsystems, Inc. All rights reserved.

Package javax.xml.ws.handler

This package defines APIs for message handlers. **See:**

Description

Interface Summary	
Handler<C extends MessageContext>	The `Handler` interface is the base interface for JAX-WS handlers.
HandlerResolver	`HandlerResolver` is an interface implemented by an application to get control over the handler chain set on proxy/dispatch objects at the time of their creation.
LogicalHandler<C extends LogicalMessageContext>	The `LogicalHandler` extends Handler to provide typesafety for the message context parameter.
LogicalMessageContext	The `LogicalMessageContext` interface extends `MessageContext` to provide access to a the contained message as a protocol neutral LogicalMessage
MessageContext	The interface `MessageContext` abstracts the message context that is processed by a handler in the `handle` method.
PortInfo	The `PortInfo` interface is used by a `HandlerResolver` to query information about the port it is being asked to create a handler chain for.
Enum Summary	
MessageContext.Scope	Property scope.

Java EE 5 SDK

Package javax.xml.ws.handler Description

This package defines APIs for message handlers.

Package javax.xml.ws.handler.soap

This package defines APIs for SOAP message handlers. **See:**
Description

Interface Summary	
SOAPHandler<T extends SOAPMessageContext>	The `SOAPHandler` class extends `Handler` to provide typesafety for the message context parameter and add a method to obtain access to the headers that may be processed by the handler.
SOAPMessageContext	The interface `SOAPMessageContext` provides access to the SOAP message for either RPC request or response.

Java EE 5 SDK

Package javax.xml.ws.handler.soap Description

This package defines APIs for SOAP message handlers.

Submit a bug or feature

Package javax.xml.ws.http

This package defines APIs specific to the HTTP binding. **See:**
Description

Interface Summary	
HTTPBinding	The `HTTPBinding` interface is an abstraction for the XML/HTTP binding.
Exception Summary	
HTTPException	The `HTTPException` exception represents a XML/HTTP fault.

Java EE 5 SDK

Package javax.xml.ws.http Description

This package defines APIs specific to the HTTP binding.

Package javax.xml.ws.soap

This package defines APIs specific to the SOAP binding. **See:**
Description

Interface Summary	
SOAPBinding	The `SOAPBinding` interface is an abstraction for the SOAP binding.
Exception Summary	
SOAPFaultException	The `SOAPFaultException` exception represents a SOAP 1.1 or 1.2 fault.

Java EE 5 SDK

Package javax.xml.ws.soap Description

This package defines APIs specific to the SOAP binding.

Package javax.xml.ws.spi

This package defines SPIs for JAX-WS 2.0. **See:** **Description**

Class Summary	
Provider	Service provider for `ServiceDelegate` and `Endpoint` objects.
ServiceDelegate	Service delegates are used internally by `Service` objects to allow pluggability of JAX-WS implementations.

Java EE 5 SDK

Package javax.xml.ws.spi Description

This package defines SPIs for JAX-WS 2.0.

JavaTM Platform, Enterprise Edition 6 API Specification

This document is the API specification for version 6 of the JavaTM Platform, Enterprise Edition.

Packages	
javax.activation	The JavaBeans(TM) Activation Framework is used by the JavaMail(TM) API to manage MIME data.
javax.annotation	This package defines the common annotations.
javax.annotation.security	This package contains the security common annotations.
javax.annotation.sql	
javax.decorator	Annotations relating to decorators.
javax.ejb	Contains the Enterprise JavaBeans classes and interfaces that define the contracts between the enterprise bean and its clients and between the enterprise bean and the EJB container.
javax.ejb.embeddable	Defines the classes for the EJB 3.1 Embeddable API.
javax.ejb.spi	Defines interfaces that are implemented by the EJB container.
javax.el	Provides the API for the **Unified Expression Language 2.2** used by the JSP 2.2 and JSF 2.0 technologies.
javax.enterprise.context	Annotations and interfaces relating to scopes and contexts.
javax.enterprise.context.spi	The custom context SPI.
javax.enterprise.deploy.model	Provides Tool Vendor implementation classes.
javax.enterprise.deploy.model.exceptions	Provides Tool Vendor exception implementation classes.
javax.enterprise.deploy.shared	Provides shared objects for Tool Vendor and Product Vendor implementation classes.

javax.enterprise.deploy.shared.factories	Provides shared factory manager object for Tool Vendor and Product Vendor implementation classes.
javax.enterprise.deploy.spi	Provides J2EE Product Vendor implementation classes.
javax.enterprise.deploy.spi.exceptions	Provides J2EE Product Vendor deployment exception implementation classes.
javax.enterprise.deploy.spi.factories	Provides J2EE Product Vendor deployment factory implementation classes.
javax.enterprise.deploy.spi.status	Provides J2EE Product Vendor deployment status implementation classes.
javax.enterprise.event	Annotations and interfaces relating to events.
javax.enterprise.inject	Annotations relating to bean and stereotype definition, built-in qualifiers, and interfaces and classes relating to programmatic lookup.
javax.enterprise.inject.spi	The portable extension integration SPI.
javax.enterprise.util	Contains shared, general-purpose helper classes and annotations.
javax.faces	Top level classes for the JavaServer(tm) Faces API.
javax.faces.application	APIs that are used to link an application's business logic objects to JavaServer Faces, as well as convenient pluggable mechanisms to manage the execution of an application that is based on JavaServer Faces.
javax.faces.bean	These javadoc files constitute the “Faces Managed Bean Annotation Specification for Containers Conforming to Servlet 2.5 and Beyond”
javax.faces.component	Fundamental APIs for user interface components.
javax.faces.component.behavior	APIs for attaching additional behavior to user interface components.
javax.faces.component.html	Specialized user interface component classes for HTML.
javax.faces.component.visit	APIs for traversing a user interface component view.

javax.faces.context	Classes and interfaces defining per-request state information.
javax.faces.convert	Contains classes and interfaces defining converters.
javax.faces.el	**DEPRECATED** Classes and interfaces for evaluating and processing reference expressions.
javax.faces.event	Interfaces describing events and event listeners, and concrete event implementation classes.
javax.faces.lifecycle	Classes and interfaces defining lifecycle management for the JavaServer Faces implementation.
javax.faces.model	Standard model data beans for JavaServer Faces.
javax.faces.render	Classes and interfaces defining the rendering model.
javax.faces.validator	Interface defining the validator model, and concrete validator implementation classes.
javax.faces.view	Classes for defining a View Declaration Language (VDL) for authoring JavaServer Faces user interfaces.
javax.faces.view.facelets	This package contains public classes for the Java code API of Facelets.
javax.faces.webapp	Classes required for integration of JavaServer Faces into web applications, including a standard servlet, base classes for JSP custom component tags, and concrete tag implementations for core tags.
javax.inject	This package specifies a means for obtaining objects in such a way as to maximize reusability, testability and maintainability compared to traditional approaches such as constructors, factories, and service locators (e.g., JNDI). This process, known as *dependency injection*, is beneficial to most nontrivial applications.

javax.interceptor	Contains annotations and interfaces for defining interceptor methods, interceptor classes and for binding interceptor classes to target classes.
javax.jms	The Java Message Service (JMS) API provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages.
javax.jws	
javax.jws.soap	
javax.mail	The JavaMailTM API provides classes that model a mail system.
javax.mail.event	Listeners and events for the JavaMail API.
javax.mail.internet	Classes specific to Internet mail systems.
javax.mail.search	Message search terms for the JavaMail API.
javax.mail.util	JavaMail API utility classes.
javax.management.j2ee	Provides the J2EE Management Enterprise Bean component (MEJB) interfaces.
javax.management.j2ee.statistics	Provides the standard interfaces for accessing performance data from J2EE managed objects
javax.persistence	
javax.persistence.criteria	
javax.persistence.metamodel	
javax.persistence.spi	
javax.resource	The javax.resource package is the top-level package for the Java EE Connector API specification.
javax.resource.cci	The javax.resource.cci package contains API specification for the Common Client Interface (CCI).
javax.resource.spi	The javax.resource.spi package contains APIs for the system contracts defined in the Java EE Connector Architecture specification.
javax.resource.spi.endpoint	This package contains system contracts for service endpoint interactions.

javax.resource.spi.security	The javax.resource.spi.security package contains APIs for the security management contract.
javax.resource.spi.work	This package contains APIs for the Work Management, Generic and Security Work Context contracts.
javax.security.auth.message	This package defines the core interfaces of the JSR 196 message authentication SPI.
javax.security.auth.message.callback	This package defines callback interfaces that may be used by a pluggable message authentication module to interact with the message processing runtime that invoked the module.
javax.security.auth.message.config	This package defines the interfaces implemented by JSR 196 compatible configuration systems.
javax.security.auth.message.module	This package defines the interfaces implemented by JSR 196 compatible authentication modules.
javax.security.jacc	This package contains the Java Authorization Contract for Containers API
javax.servlet	The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.
javax.servlet.annotation	The javax.servlet.annotation package contains a number of annotations that allow users to use annotations to declare servlets, filters, listeners and specify the metadata for the declared component.
javax.servlet.descriptor	Provides programmatic access to a web application's configuration information that was aggregated from the `web.xml` and `web-fragment.xml` descriptors.

javax.servlet.http	The javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container.
javax.servlet.jsp	Classes and interfaces for the Core JSP 2.1 API.
javax.servlet.jsp.el	Provides the `ELResolver` classes that define the object resolution rules that must be supported by a JSP container with the new unified Expression Language.
javax.servlet.jsp.tagext	Classes and interfaces for the definition of JavaServer Pages Tag Libraries.
javax.transaction	Provides the API that defines the contract between the transaction manager and the various parties involved in a distributed transaction namely : resource manager, application, and application server.
javax.transaction.xa	Provides the API that defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects (supplied by the resource manager driver) in JTA transactions.
javax.ws.rs	High-level interfaces and annotations used to create RESTful service resources.
javax.ws.rs.core	Low-level interfaces and annotations used to create RESTful service resources.
javax.ws.rs.ext	APIs that provide extensions to the types supported by the JAX-RS API.
javax.xml.bind	Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities.

javax.xml.bind.annotation	Defines annotations for customizing Java program elements to XML Schema mapping.
javax.xml.bind.annotation.adapters	`XmlAdapter` and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB.
javax.xml.bind.attachment	This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format.
javax.xml.bind.helpers	**JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces.**
javax.xml.bind.util	Useful client utility classes.
javax.xml.registry	This package and its sub-packages describe the API classes and interfaces for the JAXR API.
javax.xml.registry.infomodel	This package describes the information model for JAXR API.
javax.xml.rpc	This package contains the core JAX-RPC APIs for the client programming model.
javax.xml.rpc.encoding	This package defines APIs for the extensible type mapping framework.
javax.xml.rpc.handler	This package defines APIs for SOAP Message Handlers
javax.xml.rpc.handler.soap	This package defines APIs for SOAP Message Handlers
javax.xml.rpc.holders	This package contains the standard Java Holder classes.
javax.xml.rpc.server	This package defines APIs for the servlet based JAX-RPC endpoint model.
javax.xml.rpc.soap	This package defines APIs specific to the SOAP binding.
javax.xml.soap	Provides the API for creating and building SOAP messages.
javax.xml.ws	This package contains the core JAX-WS APIs.
javax.xml.ws.handler	This package defines APIs for message handlers.

javax.xml.ws.handler.soap	This package defines APIs for SOAP message handlers.
javax.xml.ws.http	This package defines APIs specific to the HTTP binding.
javax.xml.ws.soap	This package defines APIs specific to the SOAP binding.
javax.xml.ws.spi	This package defines SPIs for JAX-WS.
javax.xml.ws.spi.http	Provides HTTP SPI that is used for portable deployment of JAX-WS web services in containers(for e.g.
javax.xml.ws.wsaddressing	This package defines APIs related to WS-Addressing.

Package javax.annotation

This package defines the common annotations. **See:**
Description

Enum Summary	
Resource.AuthenticationType	The two possible authentication types for a resource.
Annotation Types Summary	
Generated	The Generated annoation is used to mark source code that has been generated.
ManagedBean	The ManagedBean annotation marks a POJO (Plain Old Java Object) as a ManagedBean.A ManagedBean supports a small set of basic services such as resource injection, lifecycle callbacks and interceptors.
PostConstruct	The PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization.
PreDestroy	The PreDestroy annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container.
Resource	The Resource annotation marks a resource that is needed by the application.
Resources	This class is used to allow multiple resources declarations.

Package javax.annotation Description

This package defines the common annotations.

Package javax.annotation.security

This package contains the security common annotations. **See:**
Description

Annotation Types Summary	
DeclareRoles	Used by application to declare roles.
DenyAll	Specifies that no security roles are allowed to invoke the specified method(s) - i.e that the methods are to be excluded from execution in the J2EE container.
PermitAll	Specifies that all security roles are allowed to invoke the specified method(s) i.e that the specified method(s) are "unchecked".
RolesAllowed	Specifies the list of roles permitted to access method(s) in an application.
RunAs	Defines the identity of the application during execution in a J2EE container.

Package javax.annotation.security Description

This package contains the security common annotations.

Package javax.ejb

Contains the Enterprise JavaBeans classes and interfaces that define the contracts between the enterprise bean and its clients and between the enterprise bean and the EJB container. See:

Description

Interface Summary	
EJBContext	The EJBContext interface provides an instance with access to the container-provided runtime context of an enterprise Bean instance.
EJBHome	The EJBHome interface must be extended by all enterprise Beans' remote home interfaces.
EJBLocalHome	The EJBLocalHome interface must be extended by all enterprise Beans' local home interfaces.
EJBLocalObject	The EJBLocalObject interface must be extended by all enterprise Beans' local interfaces.
EJBMetaData	The EJBMetaData interface allows a client to obtain the enterprise Bean's meta-data information.
EJBObject	The EJBObject interface is extended by all enterprise Beans' remote interfaces.
EnterpriseBean	The EnterpriseBean interface must be implemented by every enterprise Bean class.
EntityBean	The EntityBean interface is implemented by every entity enterprise Bean class.
EntityContext	The EntityContext interface provides an instance with access to the container-provided runtime context of an entity enterprise Bean instance.
Handle	The Handle interface is implemented by all EJB object handles.
HomeHandle	The HomeHandle interface is implemented by all home object handles.

MessageDrivenBean	The MessageDrivenBean interface is implemented by every message-driven enterprise Bean class.
MessageDrivenContext	The MessageDrivenContext interface provides access to the runtime message-driven context that the container provides for a message-driven enterprise Bean instance.
SessionBean	The SessionBean interface is implemented by every session enterprise Bean class.
SessionContext	The SessionContext interface provides access to the runtime session context that the container provides for a session enterprise Bean instance.
SessionSynchronization	The SessionSynchronization interface allows a session Bean instance to be notified by its container of transaction boundaries.
TimedObject	The TimedObject interface contains the callback method that is used to deliver timer expiration notifications.
Timer	The Timer interface contains information about a timer that was created through the EJB Timer Service.
TimerHandle	The TimerHandle interface is implemented by all EJB timer handles.
TimerService	The TimerService interface provides enterprise bean components with access to the container-provided Timer Service.
Class Summary	
AsyncResult<V>	Wraps the result of an asynchronous method call as a Future object, preserving compatability with the business interface signature.
ScheduleExpression	A calendar-based timeout expression for an enterprise bean timer.
TimerConfig	TimerConfig is used to specify additional timer configuration settings during timer creation.

Enum Summary	
ConcurrencyManagementType	Concurrency management type for a singleton or stateful session bean.
LockType	Concurrency lock type for Singletons with container-managed concurrency.
TransactionAttributeType	
TransactionManagementType	
Exception Summary	
AccessLocalException	An AccessLocalException is thrown to indicate that the caller does not have permission to call the method.
ConcurrentAccessException	A ConcurrentAccessException indicates that the client has attempted an invocation on a stateful session bean while another invocation is in progress.
ConcurrentAccessTimeoutException	This exception indicates that an attempt to concurrently access a bean method resulted in a timeout.
CreateException	The CreateException exception must be included in the throws clauses of all create methods defined in an enterprise Bean's home interface.
DuplicateKeyException	The DuplicateKeyException exception is thrown if an entity EJB object cannot be created because an object with the same key already exists.
EJBAccessException	This exception indicates that client access to a business method was denied.
EJBException	The EJBException exception is thrown by an enterprise Bean instance to its container to report that the invoked business method or callback method could not be completed because of an unexpected error (e.g.
EJBTransactionRequiredException	This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.

EJBTransactionRolledbackException	This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back.
FinderException	The FinderException exception must be included in the throws clause of every findMETHOD(...) method of an entity Bean's home interface.
IllegalLoopbackException	This exception indicates that an attempt was made to perform an illegal loopback invocation.
NoMoreTimeoutsException	This exception indicates that a calendar-based timer will not result in any more timeouts.
NoSuchEJBException	A NoSuchEJBException is thrown if an attempt is made to invoke a method on an object that no longer exists.
NoSuchEntityException	The NoSuchEntityException exception is thrown by an Entity Bean instance to its container to report that the invoked business method or callback method could not be completed because of the underlying entity was removed from the database.
NoSuchObjectLocalException	A NoSuchObjectLocalException is thrown if an attempt is made to invoke a method on an object that no longer exists.
ObjectNotFoundException	The ObjectNotFoundException exception is thrown by a finder method to indicate that the specified EJB object does not exist.
RemoveException	The RemoveException exception is thrown at an attempt to remove an EJB object when the enterprise Bean or the container does not allow the EJB object to be removed.
TransactionRequiredLocalException	This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.

TransactionRolledbackLocalException	This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back.
Annotation Types Summary	
AccessTimeout	Specifies the amount of time in a given time unit that a concurrent access attempt should block before timing out.
ActivationConfigProperty	
AfterBegin	Designate a stateful session bean method to receive the AfterBegin Session Synchronization callback.
AfterCompletion	Designate a stateful session bean method to receive the AfterCompletion Session Synchronization callback.
ApplicationException	Applied to an exception to denote that it is an application exception and should be reported to the client directly(i.e., unwrapped).
Asynchronous	Used to mark a method as an asynchronous method or to designate all business methods of a class or interface as asynchronous.
BeforeCompletion	Designate a stateful session bean method to receive the BeforeCompletion Session Synchronization callback.
ConcurrencyManagement	Declares a Singleton or Stateful session bean's concurrency management type.
DependsOn	Used to express an initialization dependency between Singleton components.
EJB	Indicates a dependency on the local, no-interface, or remote view of an Enterprise Java Bean.
EJBs	Declares multiple TYPE-level @EJB annotations.
Init	Designates a method of a session bean that corresponds to the create method of an adapted Home interface or an adapted Local Home interface.

Local	When used on the bean class, declares the local business interface(s) for a session bean.
LocalBean	Designates that a session bean exposes a no-interface view.
LocalHome	Declares the Local Home or adapted Local Home interface for a session bean.
Lock	Declares a concurrency lock for a singleton method.
MessageDriven	Component-defining annotation for a message driven bean.
PostActivate	Designates a method to receive a callback after a stateful session bean has been activated.
PrePassivate	Designates a method to receive a callback before a stateful session bean is passivated.
Remote	Declares the remote business interface(s) for a session bean.
RemoteHome	Declares the Remote Home interface or adapted Remote Home interface for a session bean.
Remove	Applied to a business method of a stateful session bean class.
Schedule	Schedule a timer for automatic creation with a timeout schedule based on a cron-like time expression.
Schedules	Schedules multiple timers that use the same method as the timeout callback method.
Singleton	Component-defining annotation for a singleton session bean.
Startup	Mark a Singleton for eager loading during application initialization.
Stateful	Component-defining annotation for a stateful session bean.
StatefulTimeout	Specifies the amount of time a stateful session bean can be idle (not receive any client invocations) before it is eligible for removal by the container.
Stateless	Component-defining annotation for a stateless session bean.

Timeout	Designates a method on a stateless session bean class or message driven bean class that should receive EJB timer expirations for that bean.
TransactionAttribute	When applied at the TYPE-level, designates the default transaction attribute for all business methods of the session or message driven bean.
TransactionManagement	Declares whether a session bean or message driven bean has container managed transactions or bean managed transactions.

Package javax.ejb Description

Contains the Enterprise JavaBeans classes and interfaces that define the contracts between the enterprise bean and its clients and between the enterprise bean and the EJB container.

Package javax.ejb.spi

Defines interfaces that are implemented by the EJB container. **See: Description**

Interface Summary	
EJBContainerProvider	The EJBContainerProvider SPI is used by the embeddable container bootstrap class to initialize a suitable embeddable container.
HandleDelegate	The HandleDelegate interface is implemented by the EJB container.

Package javax.ejb.spi Description

Defines interfaces that are implemented by the EJB container. These interfaces are not used by application components.

Package javax.ejb.embeddable

Defines the classes for the EJB 3.1 Embeddable API. **See:** **Description**

Class Summary	
EJBContainer	Used to execute an EJB application in an embeddable container.

Package javax.ejb.embeddable Description

Defines the classes for the EJB 3.1 Embeddable API.

 Generated on 1-December-2009 05:21

Package javax.persistence

Interface Summary	
Cache	Interface used to interact with the second-level cache.
EntityManager	Interface used to interact with the persistence context.
EntityManagerFactory	Interface used to interact with the entity manager factory for the persistence unit.
EntityTransaction	Interface used to control transactions on resource-local entity managers.
Parameter<T>	Type for query parameter objects.
PersistenceUnitUtil	Utility interface between the application and the persistence provider managing the persistence unit.
PersistenceUtil	Utility interface between the application and the persistence provider(s).
Query	Interface used to control query execution.
Tuple	Interface for extracting the elements of a query result tuple.
TupleElement<X>	The `TupleElement` interface defines an element that is returned in a query result tuple.
TypedQuery<X>	Interface used to control the execution of typed queries.
Class Summary	
Persistence	Bootstrap class that is used to obtain an `EntityManagerFactory` in Java SE environments.
Enum Summary	
AccessType	Is used with the `Access` annotation to specify an access type to be applied to an entity class, mapped superclass, or embeddable class, or to a specific attribute of such a class.

CacheRetrieveMode	Used as the value of the `javax.persistence.cache.retrieveMode` property to specify the behavior when data is retrieved by the `find` methods and by queries.
CacheStoreMode	Used as the value of the `javax.persistence.cache.storeMode` property to specify the behavior when data is read from the database and when data is committed into the database.
CascadeType	Defines the set of cascadable operations that are propagated to the associated entity.
DiscriminatorType	Defines supported types of the discriminator column.
EnumType	Defines mapping for enumerated types.
FetchType	Defines strategies for fetching data from the database.
FlushModeType	Flush mode setting.
GenerationType	Defines the types of primary key generation strategies.
InheritanceType	Defines inheritance strategy options.
LockModeType	Lock modes can be specified by means of passing a `LockModeType` argument to one of the `EntityManager` methods that take locks (`lock`, `find`, or `refresh`) or to the `Query.setLockMode()` or `TypedQuery.setLockMode()` method.
PersistenceContextType	Specifies whether a transaction-scoped or extended persistence context is to be used in `PersistenceContext`.
PessimisticLockScope	Defines the values of the `javax.persistence.lock.scope` property for pessimistic locking.
SharedCacheMode	Specifies how the provider must use a second-level cache for the persistence unit.
TemporalType	Type used to indicate a specific mapping of `java.util.Date` or `java.util.Calendar`.

ValidationMode	The validation mode to be used by the provider for the persistence unit.
Exception Summary	
EntityExistsException	Thrown by the persistence provider when `EntityManager.persist(Object)` is called and the entity already exists.
EntityNotFoundException	Thrown by the persistence provider when an entity reference obtained by `EntityManager.getReference` is accessed but the entity does not exist.
LockTimeoutException	Thrown by the persistence provider when an pessimistic locking conflict occurs that does not result in transaction rollback.
NonUniqueResultException	Thrown by the persistence provider when `Query.getSingleResult()` or `TypedQuery.getSingleResult()` is executed on a query and there is more than one result from the query.
NoResultException	Thrown by the persistence provider when `Query.getSingleResult()` or `TypedQuery.getSingleResult()`is executed on a query and there is no result to return.
OptimisticLockException	Thrown by the persistence provider when an optimistic locking conflict occurs.
PersistenceException	Thrown by the persistence provider when a problem occurs.
PessimisticLockException	Thrown by the persistence provider when an pessimistic locking conflict occurs.
QueryTimeoutException	Thrown by the persistence provider when a query times out and only the statement is rolled back.
RollbackException	Thrown by the persistence provider when `EntityTransaction.commit()` fails.

TransactionRequiredException	Thrown by the persistence provider when a transaction is required but is not active.
Annotation Types Summary	
Access	Used to specify an access type to be applied to an entity class, mapped superclass, or embeddable class, or to a specific attribute of such a class.
AssociationOverride	Used to override a mapping for an entity relationship.
AssociationOverrides	Used to override mappings of multiple relationship properties or fields.
AttributeOverride	Used to override the mapping of a `Basic` (whether explicit or default) property or field or `Id` property or field.
AttributeOverrides	Used to override mappings of multiple properties or fields.
Basic	The simplest type of mapping to a database column.
Cacheable	Specifies whether an entity should be cached if caching is enabled when the value of the `persistence.xml` caching element is `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE`.
CollectionTable	Specifies the table that is used for the mapping of collections of basic or embeddable types.
Column	Is used to specify the mapped column for a persistent property or field.
ColumnResult	References name of a column in the SELECT clause of a SQL query - i.e., column alias, if applicable.
DiscriminatorColumn	Specifies the discriminator column for the `SINGLE_TABLE` and `JOINEDInheritance` mapping strategies.
DiscriminatorValue	Specifies the value of the discriminator column for entities of the given type.
ElementCollection	Defines a collection of instances of a basic type or embeddable class.

Embeddable	Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity.
Embedded	Specifies a persistent field or property of an entity whose value is an instance of an embeddable class.
EmbeddedId	Applied to a persistent field or property of an entity class or mapped superclass to denote a composite primary key that is an embeddable class.
Entity	Specifies that the class is an entity.
EntityListeners	Specifies the callback listener classes to be used for an entity or mapped superclass.
EntityResult	Used to map the SELECT clause of a SQL query to an entity result.
Enumerated	Specifies that a persistent property or field should be persisted as a enumerated type.
ExcludeDefaultListeners	Specifies that the invocation of default listeners is to be excluded for the entity class (or mapped superclass) and its subclasses.
ExcludeSuperclassListeners	Specifies that the invocation of superclass listeners is to be excluded for the entity class (or mapped superclass) and its subclasses.
FieldResult	Is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.
GeneratedValue	Provides for the specification of generation strategies for the values of primary keys.
Id	Specifies the primary key of an entity.
IdClass	Specifies a composite primary key class that is mapped to multiple fields or properties of the entity.
Inheritance	Defines the inheritance strategy to be used for an entity class hierarchy.

JoinColumn	Specifies a column for joining an entity association or element collection.
JoinColumns	Defines mapping for composite foreign keys.
JoinTable	Used in the mapping of associations.
Lob	Specifies that a persistent property or field should be persisted as a large object to a database-supported large object type.
ManyToMany	Defines a many-valued association with many-to-many multiplicity.
ManyToOne	Defines a single-valued association to another entity class that has many-to-one multiplicity.
MapKey	Specifies the map key for associations of type `java.util.Map` when the map key is itself the primary key or a persistent field or property of the entity that is the value of the map.
MapKeyClass	Specifies the type of the map key for associations of type `java.util.Map`.
MapKeyColumn	Specifies the mapping for the key column of a map whose map key is a basic type.
MapKeyEnumerated	Specifies the enum type for a map key whose basic type is an enumerated type.
MapKeyJoinColumn	Specifies a mapping to an entity that is a map key.
MapKeyJoinColumns	Supports composite map keys that reference entities.
MapKeyTemporal	This annotation must be specified for persistent map keys of type `Date` and `Calendar`.
MappedSuperclass	Designates a class whose mapping information is applied to the entities that inherit from it.

MapsId	Designates a `ManyToOne` or `OneToOne` relationship attribute that provides the mapping for an `EmbeddedId` primary key, an attribute within an `EmbeddedId` primary key, or a simple primary key of the parent entity.
NamedNativeQueries	Used to specify multiple native SQL named queries.
NamedNativeQuery	Specifies a named native SQL query.
NamedQueries	Specifies multiple named Java Persistence query language queries.
NamedQuery	Specifies a static, named query in the Java Persistence query language.
OneToMany	Defines a many-valued association with one-to-many multiplicity.
OneToOne	Defines a single-valued association to another entity that has one-to-one multiplicity.
OrderBy	Specifies the ordering of the elements of a collection valued association or element collection at the point when the association or collection is retrieved.
OrderColumn	Specifies a column that is used to maintain the persistent order of a list.
PersistenceContext	Expresses a dependency on a container-managed `EntityManager` and its associated persistence context.
PersistenceContexts	Declares one or more `PersistenceContext` annotations.
PersistenceProperty	Describes a single container or persistence provider property.
PersistenceUnit	Expresses a dependency on an `EntityManagerFactory` and its associated persistence unit.
PersistenceUnits	Declares one or more `PersistenceUnit` annotations.
PostLoad	Is used to specify callback methods for the corresponding lifecycle event.

PostPersist	Is used to specify callback methods for the corresponding lifecycle event.
PostRemove	Is used to specify callback methods for the corresponding lifecycle event.
PostUpdate	Is used to specify callback methods for the corresponding lifecycle event.
PrePersist	Is used to specify callback methods for the corresponding lifecycle event.
PreRemove	Is used to specify callback methods for the corresponding lifecycle event.
PreUpdate	Is used to specify callback methods for the corresponding lifecycle event.
PrimaryKeyJoinColumn	Specifies a primary key column that is used as a foreign key to join to another table.
PrimaryKeyJoinColumns	Groups `PrimaryKeyJoinColumn` annotations.
QueryHint	Used to supply a query property or hint to the `NamedQuery` or `NamedNativeQuery` annotation.
SecondaryTable	Specifies a secondary table for the annotated entity class.
SecondaryTables	Specifies multiple secondary tables for an entity.
SequenceGenerator	Defines a primary key generator that may be referenced by name when a generator element is specified for the `GeneratedValue` annotation.
SqlResultSetMapping	Specifies the mapping of the result of a native SQL query.
SqlResultSetMappings	Is used to define one or more `SqlResultSetMapping` annotations.
Table	Specifies the primary table for the annotated entity.
TableGenerator	Defines a primary key generator that may be referenced by name when a generator element is specified for the `GeneratedValue` annotation.

Temporal	This annotation must be specified for persistent fields or properties of type `java.util.Date` and `java.util.Calendar`.
Transient	Specifies that the property or field is not persistent.
UniqueConstraint	Specifies that a unique constraint is to be included in the generated DDL for a primary or secondary table.
Version	Specifies the version field or property of an entity class that serves as its optimistic lock value.

Package javax.persistence.spi

Interface Summary	
ClassTransformer	A persistence provider supplies an instance of this interface to the `PersistenceUnitInfo.addTransformer` method.
PersistenceProvider	Interface implemented by the persistence provider.
PersistenceProviderResolver	Determine the list of persistence providers available in the runtime environment.
PersistenceUnitInfo	Interface implemented by the container and used by the persistence provider when creating an `EntityManagerFactory`.
ProviderUtil	Utility interface implemented by the persistence provider.
Class Summary	
PersistenceProviderResolverHolder	Holds the global `PersistenceProviderResolver` instance.
Enum Summary	
LoadState	Load states returned by the `ProviderUtil` SPI methods.
PersistenceUnitTransactionType	Specifies whether entity managers created by the `EntityManagerFactory` will be JTA or resource-local entity managers.

Package javax.persistence.metamodel

Interface Summary	
Attribute<X,Y>	Represents an attribute of a Java type.
BasicType<X>	Instances of the type `BasicType` represent basic types (including temporal and enumerated types).
Bindable<T>	Instances of the type `Bindable` represent object or attribute types that can be bound into a `Path`.
CollectionAttribute<X,E>	Instances of the type `CollectionAttribute` represent persistent `java.util.Collection`-valued attributes.
EmbeddableType<X>	Instances of the type `EmbeddableType` represent embeddable types.
EntityType<X>	Instances of the type `EntityType` represent entity types.
IdentifiableType<X>	Instances of the type `IdentifiableType` represent entity or mapped superclass types.
ListAttribute<X,E>	Instances of the type `ListAttribute` represent persistent `javax.util.List`-valued attributes.
ManagedType<X>	Instances of the type `ManagedType` represent entity, mapped superclass, and embeddable types.
MapAttribute<X,K,V>	Instances of the type `MapAttribute` represent persistent `java.util.Map`-valued attributes.
MappedSuperclassType<X>	Instances of the type `MappedSuperclassType` represent mapped superclass types.
Metamodel	Provides access to the metamodel of persistent entities in the persistence unit.

PluralAttribute<X,C,E>	Instances of the type `PluralAttribute` represent persistent collection-valued attributes.
SetAttribute<X,E>	Instances of the type `SetAttribute` represent persistent `java.util.Set`-valued attributes.
SingularAttribute<X,T>	Instances of the type `SingularAttribute` represents persistent single-valued properties or fields.
Type<X>	Instances of the type `Type` represent persistent object or attribute types.
Enum Summary	
Attribute.PersistentAttributeType	
Bindable.BindableType	
PluralAttribute.CollectionType	
Type.PersistenceType	
Annotation Types Summary	
StaticMetamodel	The `StaticMetamodel` annotation specifies that the class is a metamodel class that represents the entity, mapped superclass, or embeddable class designated by the value element.

Package javax.resource

The javax.resource package is the top-level package for the Java EE Connector API specification. See Description

Interface Summary	
Referenceable	The Referenceable interface extends the javax.naming.Referenceable interface.
Exception Summary	
NotSupportedException	A `NotSupportedException` is thrown to indicate that callee (resource adapter or application server for system contracts) cannot execute an operation because the operation is not a supported feature.
ResourceException	This is the root interface of the exception hierarchy defined for the Connector architecture.

Package javax.resource Description

The javax.resource package is the top-level package for the Java EE Connector API specification.

Package javax.resource.cci

The javax.resource.cci package contains API specification for the Common Client Interface (CCI). See:

Description

Interface Summary	
Connection	A Connection represents an application-level handle that is used by a client to access the underlying physical connection.
ConnectionFactory	`ConnectionFactory` provides an interface for getting connection to an EIS instance.
ConnectionMetaData	The interface `ConnectionMetaData` provides information about an EIS instance connected through a Connection instance.
ConnectionSpec	ConnectionSpec is used by an application component to pass connection request-specific properties to the ConnectionFactory.
IndexedRecord	IndexedRecord represents an ordered collection of record elements based on the `java.util.List` interface.
Interaction	The `javax.resource.cci.Interaction` enables a component to execute EIS functions.
InteractionSpec	An InteractionSpec holds properties for driving an Interaction with an EIS instance.
LocalTransaction	The LocalTransaction defines a transaction demarcation interface for resource manager local transactions.
MappedRecord	The interface `javax.resource.cci.MappedRecord` is used for key-value map based representation of record elements.
MessageListener	This serves as a request-response message listener type that message endpoints (message-driven beans) may implement.

Record	The `javax.resource.cci.Record` interface is the base interface for the representation of an input or output to the execute methods defined on an Interaction.
RecordFactory	The RecordFactory interface is used for creating MappedRecord and IndexedRecord instances.
ResourceAdapterMetaData	The interface `javax.resource.cci.ResourceAdapterMetaData` provides information about capabilities of a resource adapter implementation.
ResultSet	A ResultSet represents tabular data that is retrieved from an EIS instance by the execution of an Interaction..
ResultSetInfo	The interface `javax.resource.cci.ResultSetInfo` provides information on the support provided for ResultSet by a connected EIS instance.
Streamable	Streamable interface enables a resource adapter to extract data from an input Record or set data into an output Record as a stream of bytes.
Exception Summary	
ResourceWarning	A `ResourceWarning` provides information on warnings related to execution of an interaction with an EIS.

Package javax.resource.cci Description

The javax.resource.cci package contains API specification for the Common Client Interface (CCI).

Package javax.resource.spi

The javax.resource.spi package contains APIs for the system contracts defined in the Java EE Connector Architecture specification. **See:**

Description

Interface Summary

ActivationSpec	This interface serves as a marker.
BootstrapContext	This provides a mechanism to pass a bootstrap context to a resource adapter instance when it is bootstrapped.
ConnectionEventListener	The `ConnectionEventListener` interface provides an event callback mechanism to enable an application server to receive notifications from a `ManagedConnection` instance.
ConnectionManager	ConnectionManager interface provides a hook for the resource adapter to pass a connection request to the application server.
ConnectionRequestInfo	The ConnectionRequestInfo interface enables a resource adapter to pass its own request specific data structure across the connection request flow.
DissociatableManagedConnection	This is a mix-in interface that may be optionally implemented by a `ManagedConnection` implementation.
LazyAssociatableConnectionManager	This is a mix-in interface that may be optionally implemented by a `ConnectionManager` implementation.
LazyEnlistableConnectionManager	This is a mix-in interface that may be optionally implemented by a `ConnectionManager` implementation.
LazyEnlistableManagedConnection	This is a mix-in interface that may be optionally implemented by a `ManagedConnection` implementation.
LocalTransaction	LocalTransaction interface provides support for transactions that are managed internal to an EIS resource manager, and do not require an external transaction manager.

ManagedConnection	ManagedConnection instance represents a physical connection to the underlying EIS.
ManagedConnectionFactory	ManagedConnectionFactory instance is a factory of both ManagedConnection and EIS-specific connection factory instances.
ManagedConnectionMetaData	The ManagedConnectionMetaData interface provides information about the underlying EIS instance associated with a ManagedConnection instance.
ResourceAdapter	This represents a resource adapter instance and contains operations for lifecycle management and message endpoint setup.
ResourceAdapterAssociation	This interface specifies the methods to associate a `ResourceAdapter` object with other objects that implement this interface like `ManagedConnectionFactory` and `ActivationSpec`.
RetryableException	A marker interface indicating that the Exception is transient.
TransactionSupport	This interface may be optionally implemented by a `ManagedConnectionFactory` to provide its level of transaction support at runtime.
ValidatingManagedConnectionFactory	This interface is implemented by a `ManagedConnectionFactory` instance that supports the ability to validate `ManagedConnection` objects.
XATerminator	The XATerminator interface is used for transaction completion and crash recovery flows.
Class Summary	

ConnectionEvent	The ConnectionEvent class provides information about the source of a connection related event.A ConnectionEvent instance contains the following information: Type of the connection event ManagedConnection instance that generated the connection event.
Enum Summary	
AuthenticationMechanism.CredentialInterface	An enumerated type that represents the various interfaces that a resource adapter may support for the representation of the credentials.
TransactionSupport.TransactionSupportLevel	An enumerated type that represents the levels of transaction support a resource adapter may support.
Exception Summary	
ApplicationServerInternalException	An `ApplicationServerInternalException` is thrown by an application server to indicate error conditions specific to an application server.
CommException	This indicates errors related to failed or interrupted communication with an EIS instance.
EISSystemException	An `EISSystemException` is used to indicate any EIS specific system-level error conditions.
IllegalStateException	An `IllegalStateException` is thrown from a method if the callee (resource adapter or application server for system contracts) is in an illegal or inappropriate state for the method invocation.
InvalidPropertyException	This exception is thrown to indicate invalid configuration property settings.
LocalTransactionException	A `LocalTransactionException` represents various error conditions related to the local transaction management contract.

ResourceAdapterInternalException	A `ResourceAdapterInternalException` indicates any system-level error conditions related to a resource adapter.
ResourceAllocationException	A `ResourceAllocationException` can be thrown by an application server or resource adapter to indicate any failure to allocate system resources (example: threads, physical connections).
RetryableUnavailableException	A subclass of the `UnavailableException` that indicates that the rejection of the work submission is transient.
SecurityException	A `SecurityException` indicates error conditions related to the security contract between an application server and resource adapter.
SharingViolationException	This is thrown to indicate a connection sharing violation.
UnavailableException	This is thrown to indicate that a service is unavailable.
Annotation Types Summary	
Activation	Designates a JavaBean as an `ActivationSpec`.
AdministeredObject	Designates a JavaBean as an administered object.Administered objects are specific to a messaging style or message provider.
AuthenticationMechanism	
ConfigProperty	Designates a JavaBean property as a configuration property
ConnectionDefinition	Defines a set of connection interfaces and classes pertaining to a particular connection type.
ConnectionDefinitions	Defines a set of connection definitions that the JavaBean, that has been annotated with this annotation, is a part of.
Connector	The `Connector` annotation is a component-defining annotation and it can be used by the resource adapter developer to specify that the JavaBean is a resource adapter JavaBean.

SecurityPermission	The SecurityPermission annotation can be used by the developer, as part of the Connector annotation, to specify the extended security permissions required by the resource adapter

Package javax.resource.spi Description

The javax.resource.spi package contains APIs for the system contracts defined in the Java EE Connector Architecture specification.

Package javax.resource.spi.endpoint

This package contains system contracts for service endpoint interactions. **See: Description**

Interface Summary	
MessageEndpoint	This defines a contract for a message endpoint.
MessageEndpointFactory	This serves as a factory for creating message endpoints.

Package javax.resource.spi.endpoint Description

This package contains system contracts for service endpoint interactions.

 Generated on 1-December-2009 05:21

Package javax.resource.spi.security

The javax.resource.spi.security package contains APIs for the security management contract. See:

Description

Interface Summary	
GenericCredential	**Deprecated.** *The preferred way to represent generic credential information is via the `org.ietf.jgss.GSSCredential` interface in J2SE Version 1.4, which provides similar functionality.*
Class Summary	
PasswordCredential	The class PasswordCredential acts as a holder for username and password.

Package javax.resource.spi.security Description

The javax.resource.spi.security package contains APIs for the security management contract.

Package javax.resource.spi.work

This package contains APIs for the Work Management, Generic and Security Work Context contracts. See:

Description

Interface Summary	
DistributableWork	This models a `Work` instance that would be distributed by a `DistributableWorkManager` for execution in a remote `DistributableWorkManager`
DistributableWorkManager	This interface models a `WorkManager` that supports distributed execution of Work instances.
Work	This models a `Work` instance that would be executed by a `WorkManager` upon submission.
WorkContext	This class serves as a standard mechanism for a resource adapter to propagate an imported context from an enterprise information system to an application server.
WorkContextLifecycleListener	This class models the various events that occur during the processing of the `WorkContexts` associated with a `Work` instance.
WorkContextProvider	This interface specifies the methods a `Work` instance uses to associate a `List` of `WorkContext` instances to be set when the `Work` instance gets executed by a `WorkManager`.
WorkListener	This models a `WorkListener` instance which would be notified by the `WorkManager` when the various `Work` processing events (work accepted, work rejected, work started, work completed) occur.
WorkManager	This interface models a `WorkManager` which provides a facility to submit `Work` instances for execution.
Class Summary	

ExecutionContext	This class models an execution context (transaction, security, etc) with which the `Work` instance must be executed.
HintsContext	A standard `WorkContext` that allows a `Work` instance to propagate quality-of-service (QoS) hints about the `Work` to the `WorkManager`.
SecurityContext	A standard `WorkContext` that allows a `Work` instance to propagate security related context information from an EIS to an application server.
TransactionContext	A standard `WorkContext` that allows a `Work` instance to propagate transaction related context information from an EIS to an application server.
WorkAdapter	This class is provided as a convenience for easily creating `WorkListener` instances by extending this class and overriding only those methods of interest.
WorkContextErrorCodes	This class models the possible error conditions that might occur during associating an `WorkContext` with a `Work` instance.
WorkEvent	This class models the various events that occur during the processing of a `Work` instance.
Exception Summary	
RetryableWorkRejectedException	A subclass of the `WorkRejectedException` that indicates that the the service unavailability is transient.
WorkCompletedException	This exception is thrown by a `WorkManager` to indicate that a submitted `Work` instance has completed with an exception.
WorkException	A common base class for all `Work` processing related exceptions.
WorkRejectedException	This exception is thrown by a `WorkManager` to indicate that a submitted `Work` instance has been rejected.

Package javax.resource.spi.work Description

This package contains APIs for the Work Management, Generic and Security Work

Context contracts.

 Generated on 1-December-2009 05:21

Package javax.xml

Defines core XML constants and functionality from the XML specifications. **See:** **Description**

Class Summary	
XMLConstants	Utility class to contain basic XML values as constants.

Package javax.xml Description

Defines core XML constants and functionality from the XML specifications.
The following core XML standards apply:
- Extensible Markup Language (XML) 1.1
- Extensible Markup Language (XML) 1.0 (Second Edition)
- XML 1.0 Second Edition Specification Errata
- Namespaces in XML 1.1
- Namespaces in XML
- Namespaces in XML Errata

Java™ Platform Standard Ed. 6

Package javax.xml.bind

Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities. **See:**

Description

Interface Summary

DatatypeConverterInterface	The DatatypeConverterInterface is for JAXB provider use only.
Element	This is an element marker interface.
Marshaller	The Marshaller class is responsible for governing the process of serializing Java content trees back into XML data.
NotIdentifiableEvent	This event indicates that a problem was encountered resolving an ID/IDREF.
ParseConversionEvent	This event indicates that a problem was encountered while converting a string from the XML data into a value of the target Java data type.
PrintConversionEvent	This event indicates that a problem was encountered while converting data from the Java content tree into its lexical representation.
Unmarshaller	The Unmarshaller class governs the process of deserializing XML data into newly created Java content trees, optionally validating the XML data as it is unmarshalled.
UnmarshallerHandler	Unmarshaller implemented as SAX ContentHandler.
ValidationEvent	This event indicates that a problem was encountered while validating the incoming XML data during an unmarshal operation, while performing on-demand validation of the Java content tree, or while marshalling the Java content tree back to XML data.
ValidationEventHandler	A basic event handler interface for validation errors.
ValidationEventLocator	Encapsulate the location of a ValidationEvent.

Validator	**Deprecated.** *since JAXB 2.0*
Class Summary	
Binder<XmlNode>	Enable synchronization between XML infoset nodes and JAXB objects representing same XML document.
DatatypeConverter	The javaType binding declaration can be used to customize the binding of an XML schema datatype to a Java datatype.
JAXB	Class that defines convenience methods for common, simple use of JAXB.
JAXBContext	The JAXBContext class provides the client's entry point to the JAXB API.
JAXBElement<T>	JAXB representation of an Xml Element.
JAXBElement.GlobalScope	Designates global scope for an xml element.
JAXBIntrospector	Provide access to JAXB xml binding data for a JAXB object.
JAXBPermission	This class is for JAXB permissions.
Marshaller.Listener	Register an instance of an implementation of this class with a `Marshaller` to externally listen for marshal events.
SchemaOutputResolver	Controls where a JAXB implementation puts the generates schema files.
Unmarshaller.Listener	Register an instance of an implementation of this class with `Unmarshaller` to externally listen for unmarshal events.
Exception Summary	
DataBindingException	Exception that represents a failure in a JAXB operation.
JAXBException	This is the root exception class for all JAXB exceptions.
MarshalException	This exception indicates that an error has occurred while performing a marshal operation that the provider is unable to recover from.

PropertyException	This exception indicates that an error was encountered while getting or setting a property.
TypeConstraintException	This exception indicates that a violation of a dynamically checked type constraint was detected.
UnmarshalException	This exception indicates that an error has occurred while performing an unmarshal operation that prevents the JAXB Provider from completing the operation.
ValidationException	This exception indicates that an error has occurred while performing a validate operation.

Package javax.xml.bind Description

Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities. JAXBContext is the client-entry point to the runtime binding framework. **Package Specification**

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

- The JAXB Website

Package javax.xml.bind.annotation

Defines annotations for customizing Java program elements to XML Schema mapping. See:

Description

Interface Summary	
DomHandler<ElementT,ResultT extends Result>	Converts an element (and its descendants) from/to DOM (or similar) representation.
Class Summary	
W3CDomHandler	`DomHandler` implementation for W3C DOM (`org.w3c.dom` package.)
XmlElement.DEFAULT	Used in `XmlElement#type()` to signal that the type be inferred from the signature of the property.
XmlElementDecl.GLOBAL	Used in `XmlElementDecl#scope()` to signal that the declaration is in the global scope.
XmlElementRef.DEFAULT	Used in `XmlElementRef#type()` to signal that the type be inferred from the signature of the property.
XmlSchemaType.DEFAULT	Used in `XmlSchemaType#type()` to signal that the type be inferred from the signature of the property.
XmlType.DEFAULT	Used in `XmlType#factoryClass()` to signal that either factory mehod is not used or that it's in the class with this `XmlType` itself.
Enum Summary	
XmlAccessOrder	Used by XmlAccessorOrder to control the ordering of properties and fields in a JAXB bound class.
XmlAccessType	Used by XmlAccessorType to control serialization of fields or properties.
XmlNsForm	Enumeration of XML Schema namespace qualifications.
Annotation Types Summary	
XmlAccessorOrder	Controls the ordering of fields and properties in a class.

XmlAccessorType	Controls whether fields or Javabean properties are serialized by default.
XmlAnyAttribute	Maps a JavaBean property to a map of wildcard attributes.
XmlAnyElement	Maps a JavaBean property to XML infoset representation and/or JAXB element.
XmlAttachmentRef	Marks a field/property that its XML form is a uri reference to mime content.
XmlAttribute	Maps a JavaBean property to a XML attribute.
XmlElement	Maps a JavaBean property to a XML element derived from property name.
XmlElementDecl	Maps a factory method to a XML element.
XmlElementRef	Maps a JavaBean property to a XML element derived from property's type.
XmlElementRefs	Marks a property that refers to classes with `XmlElement` or JAXBElement.
XmlElements	A container for multiple `@XmlElement` annotations.
XmlElementWrapper	Generates a wrapper element around XML representation.
XmlEnum	Maps an enum type `Enum` to XML representation.
XmlEnumValue	Maps an enum constant in `Enum` type to XML representation.
XmlID	Maps a JavaBean property to XML ID.
XmlIDREF	Maps a JavaBean property to XML IDREF.
XmlInlineBinaryData	Disable consideration of XOP encoding for datatypes that are bound to base64-encoded binary data in XML.
XmlList	Used to map a property to a list simple type.
XmlMimeType	Associates the MIME type that controls the XML representation of the property.
XmlMixed	Annotate a JavaBean multi-valued property to support mixed content.

XmlNs	Associates a namespace prefix with a XML namespace URI.
XmlRegistry	Marks a class that has `XmlElementDecls`.
XmlRootElement	Maps a class or an enum type to an XML element.
XmlSchema	Maps a package name to a XML namespace.
XmlSchemaType	Maps a Java type to a simple schema built-in type.
XmlSchemaTypes	A container for multiple `@XmlSchemaType` annotations.
XmlSeeAlso	Instructs JAXB to also bind other classes when binding this class.
XmlTransient	Prevents the mapping of a JavaBean property/type to XML representation.
XmlType	Maps a class or an enum type to a XML Schema type.
XmlValue	Enables mapping a class to a XML Schema complex type with a simpleContent or a XML Schema simple type.

Package javax.xml.bind.annotation Description

Defines annotations for customizing Java program elements to XML Schema mapping.

Package Specification

The following table shows the JAXB mapping annotations that can be associated with each program element.

Program Element	JAXB annotation
Package	
Class	
Enum type	
JavaBean Property/field	
Parameter	

Terminology

JavaBean property and field: For the purposes of mapping, there is no semantic difference between a field and a JavaBean property. Thus, an annotation that can be applied to a JavaBean property can always be applied to a field. Hence in the Javadoc documentation, for brevity, the term JavaBean property or property is used to mean either JavaBean property or a field. Where required, both are explicitly mentioned. **top level class:** For the purpose of mapping, there is no semantic difference between a top level class and a static nested class. Thus, an annotation that can be applied to a top level class, can always be applied to a nested static class. Hence in the Javadoc documentation, for brevity, the term "top level class" or just class is used to mean either a top level class or a nested static class.

mapping annotation: A JAXB 2.0 defined program annotation based on the JSR 175 programming annotation facility.

Common Usage Constraints

The following usage constraints are defined here since they apply to more than annotation:

- For a property, a given annotation can be applied to either read or write property but not both.
- A property name must be different from any other property name in any of the super classes of the class being mapped.
- A mapped field name or the decapitalized name of a mapped property must be unique within a class.

Notations

Namespace prefixes The following namespace prefixes are used in the XML Schema fragments in this package.

Prefix	Namespace	Notes
xs	http://www.w3.org/2001/XMLSchema	Namespace of XML Schema namespace
ref	http://ws-i.org/profiles/basic/1.1/xsd	Namespace for swaref schema component
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema namespace for instances

- Since:

- JAXB 2.0

Package javax.xml.bind.annotation.adapters

`XmlAdapter` and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB. See:

Description

Class Summary	
CollapsedStringAdapter	Built-in `XmlAdapter` to handle xs:token and its derived types.
HexBinaryAdapter	`XmlAdapter` for xs:hexBinary.
NormalizedStringAdapter	`XmlAdapter` to handle xs:normalizedString.
XmlAdapter<ValueType,BoundType>	Adapts a Java type for custom marshaling.
XmlJavaTypeAdapter.DEFAULT	Used in `XmlJavaTypeAdapter#type()` to signal that the type be inferred from the signature of the field, property, parameter or the class.
Annotation Types Summary	
XmlJavaTypeAdapter	Use an adapter that implements `XmlAdapter` for custom marshaling.
XmlJavaTypeAdapters	A container for multiple `@XmlJavaTypeAdapter` annotations.

Package javax.xml.bind.annotation.adapters Description

`XmlAdapter` and its spec-defined sub-classes to allow arbitrary Java classes to be used with JAXB.

Package Specification

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

- The JAXB Website

Package javax.xml.bind.attachment

This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format. **See:**

Description

Class Summary	
AttachmentMarshaller	Enable JAXB marshalling to optimize storage of binary data.
AttachmentUnmarshaller	Enables JAXB unmarshalling of a root document containing optimized binary data formats.

Package javax.xml.bind.attachment Description

This package is implemented by a MIME-based package processor that enables the interpretation and creation of optimized binary data within an MIME-based package format. Soap MTOM[1], XOP([2][3]) and WS-I AP[4] standardize approaches to optimized transmission of binary datatypes as an attachment. To optimally support these standards within a message passing environment, this package enables an integrated solution between a MIME-based package processor and JAXB unmarshall/marshal processes.

Package Specification

- JAXB Specification

Related Standards

- [1]SOAP Message Transmission Optimization Mechanism
- [2]XML-binary Optimized Packaging
- [3]WS-I Attachments Profile Version 1.0.
- [4]Describing Media Content of Binary Data in XML

- Since:

- JAXB 2.0

Package javax.xml.bind.helpers

JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces. See: Description

Class Summary	
AbstractMarshallerImpl	Partial default Marshaller implementation.
AbstractUnmarshallerImpl	Partial default Unmarshaller implementation.
DefaultValidationEventHandler	JAXB 1.0 only default validation event handler.
NotIdentifiableEventImpl	Default implementation of the NotIdentifiableEvent interface.
ParseConversionEventImpl	Default implementation of the ParseConversionEvent interface.
PrintConversionEventImpl	Default implementation of the PrintConversionEvent interface.
ValidationEventImpl	Default implementation of the ValidationEvent interface.
ValidationEventLocatorImpl	Default implementation of the ValidationEventLocator interface.

Package javax.xml.bind.helpers Description

JAXB Provider Use Only: Provides partial default implementations for some of the javax.xml.bind interfaces. JAXB Providers can extend these classes and implement the abstract methods.

Package Specification

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

- The JAXB Website

Package javax.xml.bind.util

Useful client utility classes. **See: Description**

Class Summary	
JAXBResult	JAXP `Result` implementation that unmarshals a JAXB object.
JAXBSource	JAXP `Source` implementation that marshals a JAXB-generated object.
ValidationEventCollector	`ValidationEventHandler` implementation that collects all events.

Package javax.xml.bind.util Description

Useful client utility classes.

Package Specification

- JAXB Specification

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

- The JAXB Website

Package javax.xml.ws

This package contains the core JAX-WS APIs. **See:**

Description

Interface Summary	
AsyncHandler<T>	The `AsyncHandler` interface is implemented by clients that wish to receive callback notification of the completion of service endpoint operations invoked asynchronously.
Binding	The `Binding` interface is the base interface for JAX-WS protocol bindings.
BindingProvider	The `BindingProvider` interface provides access to the protocol binding and associated context objects for request and response message processing.
Dispatch<T>	The `Dispatch` interface provides support for the dynamic invocation of a service endpoint operations.
LogicalMessage	The `LogicalMessage` interface represents a protocol agnostic XML message and contains methods that provide access to the payload of the message.
Provider<T>	Service endpoints may implement the `Provider` interface as a dynamic alternative to an SEI.
Response<T>	The `Response` interface provides methods used to obtain the payload and context of a message sent in response to an operation invocation.
WebServiceContext	A `WebServiceContext` makes it possible for a web service endpoint implementation class to access message context and security information relative to a request being served.
Class Summary	
Endpoint	A Web service endpoint.
EndpointContext	`EndpointContext` allows multiple endpoints in an application to share any information.

EndpointReference	This class represents an WS-Addressing EndpointReference which is a remote reference to a web service endpoint.
Holder<T>	Holds a value of type T.
RespectBindingFeature	This feature clarifies the use of the `wsdl:binding` in a JAX-WS runtime.
Service	`Service` objects provide the client view of a Web service.
WebServiceFeature	A WebServiceFeature is used to represent a feature that can be enabled or disabled for a web service.
WebServicePermission	This class defines web service permissions.
Enum Summary	
Service.Mode	The orientation of a dynamic client or service.
Exception Summary	
ProtocolException	The `ProtocolException` class is a base class for exceptions related to a specific protocol binding.
WebServiceException	The `WebServiceException` class is the base exception class for all JAX-WS API runtime exceptions.
Annotation Types Summary	
Action	The `Action` annotation allows explicit association of a WS-Addressing `Action` message addressing property with `input`, `output`, and `fault` messages of the mapped WSDL operation.
BindingType	The `BindingType` annotation is used to specify the binding to use for a web service endpoint implementation class.
FaultAction	The `FaultAction` annotation is used inside an `Action` annotation to allow an explicit association of a WS-Addressing `Action` message addressing property with the `fault` messages of the WSDL operation mapped from the exception class.

RequestWrapper	Used to annotate methods in the Service Endpoint Interface with the request wrapper bean to be used at runtime.
RespectBinding	This feature clarifies the use of the `wsdl:binding` in a JAX-WS runtime.
ResponseWrapper	Used to annotate methods in the Service Endpoint Interface with the response wrapper bean to be used at runtime.
ServiceMode	Used to indicate whether a `Provider` implementation wishes to work with entire protocol messages or just with protocol message payloads.
WebEndpoint	Used to annotate the `get`*`PortName`*`()` methods of a generated service interface.
WebFault	Used to annotate service specific exception classes to customize to the local and namespace name of the fault element and the name of the fault bean.
WebServiceClient	Used to annotate a generated service interface.
WebServiceProvider	Used to annotate a Provider implementation class.
WebServiceRef	The `WebServiceRef` annotation is used to define a reference to a web service and (optionally) an injection target for it.
WebServiceRefs	The `WebServiceRefs` annotation allows multiple web service references to be declared at the class level.

Package javax.xml.ws Description

This package contains the core JAX-WS APIs.

Package javax.xml.ws.handler

This package defines APIs for message handlers. **See:**

Description

Interface Summary	
Handler<C extends MessageContext>	The `Handler` interface is the base interface for JAX-WS handlers.
HandlerResolver	`HandlerResolver` is an interface implemented by an application to get control over the handler chain set on proxy/dispatch objects at the time of their creation.
LogicalHandler<C extends LogicalMessageContext>	The `LogicalHandler` extends Handler to provide typesafety for the message context parameter.
LogicalMessageContext	The `LogicalMessageContext` interface extends `MessageContext` to provide access to a the contained message as a protocol neutral LogicalMessage
MessageContext	The interface `MessageContext` abstracts the message context that is processed by a handler in the `handle` method.
PortInfo	The `PortInfo` interface is used by a `HandlerResolver` to query information about the port it is being asked to create a handler chain for.
Enum Summary	
MessageContext.Scope	Property scope.

Package javax.xml.ws.handler Description

This package defines APIs for message handlers.

Package javax.xml.ws.handler.soap

This package defines APIs for SOAP message handlers. **See:**

Description

Interface Summary	
SOAPHandler<T extends SOAPMessageContext>	The `SOAPHandler` class extends `Handler` to provide typesafety for the message context parameter and add a method to obtain access to the headers that may be processed by the handler.
SOAPMessageContext	The interface `SOAPMessageContext` provides access to the SOAP message for either RPC request or response.

Package javax.xml.ws.handler.soap Description

This package defines APIs for SOAP message handlers.

Package javax.xml.ws.http

This package defines APIs specific to the HTTP binding. **See:**

Description

Interface Summary	
HTTPBinding	The `HTTPBinding` interface is an abstraction for the XML/HTTP binding.
Exception Summary	
HTTPException	The `HTTPException` exception represents a XML/HTTP fault.

Package javax.xml.ws.http Description

This package defines APIs specific to the HTTP binding.

Package javax.xml.ws.soap

This package defines APIs specific to the SOAP binding. **See:**
Description

Interface Summary	
SOAPBinding	The `SOAPBinding` interface is an abstraction for the SOAP binding.
Class Summary	
AddressingFeature	AddressingFeature represents the use of WS-Addressing with either the SOAP 1.1/HTTP or SOAP 1.2/HTTP binding.
MTOMFeature	This feature represents the use of MTOM with a web service.
Enum Summary	
AddressingFeature.Responses	If addressing is enabled, this property determines if endpoint requires the use of only anonymous responses, or only non-anonymous responses, or all.
Exception Summary	
SOAPFaultException	The `SOAPFaultException` exception represents a SOAP 1.1 or 1.2 fault.
Annotation Types Summary	
Addressing	This annotation represents the use of WS-Addressing with either the SOAP 1.1/HTTP or SOAP 1.2/HTTP binding.
MTOM	This feature represents the use of MTOM with a web service.

Package javax.xml.ws.soap Description

This package defines APIs specific to the SOAP binding.

Package javax.xml.ws.spi

This package defines SPIs for JAX-WS. **See:** **Description**

Class Summary	
Invoker	Invoker hides the detail of calling into application endpoint implementation.
Provider	Service provider for `ServiceDelegate` and `Endpoint` objects.
ServiceDelegate	Service delegates are used internally by `Service` objects to allow pluggability of JAX-WS implementations.
Annotation Types Summary	
WebServiceFeatureAnnotation	Annotation used to identify other annotations as a `WebServiceFeature`.

Package javax.xml.ws.spi Description

This package defines SPIs for JAX-WS.

Package javax.xml.ws.spi.http

Provides HTTP SPI that is used for portable deployment of JAX-WS web services in containers(for e.g. servlet containers).

Class Summary	
HttpContext	HttpContext represents a mapping between the root URI path of a web service to a `HttpHandler` which is invoked to handle requests destined for that path on the associated container.
HttpExchange	This class encapsulates a HTTP request received and a response to be generated in one exchange.
HttpHandler	A handler which is invoked to process HTTP requests.

Package javax.xml.ws.spi.http Description

Provides HTTP SPI that is used for portable deployment of JAX-WS web services in containers(for e.g. servlet containers). This SPI is not for end developers but provides a way for the container developers to deploy JAX-WS services portably.

The portable deployment is done as below:

1. Container creates `Endpoint` objects for an application. The necessary information to create Endpoint objects may be got from web service deployment descriptor files.
2. Container needs to create `HttpContext` objects for the deployment. For example, a HttpContext could be created using servlet configuration(for e.g url-pattern) for the web service in servlet container case.
3. Then publishes all the endpoints using `Endpoint.publish(HttpContext)`. During publish(), JAX-WS runtime registers a `HttpHandler` callback to handle incoming requests or `HttpExchange` objects. The HttpExchange object encapsulates a HTTP request and a response.

```
Container                                JAX-WS runtime
---------                                --------------
1. Creates Invoker1, ... InvokerN
2. Provider.createEndpoint(...)     --> 3. creates Endpoint1
   configures Endpoint1
   ...
4. Provider.createEndpoint(...)     --> 5. creates EndpointN
   configures EndpointN
6. Creates ApplicationContext
7. creates HttpContext1, ... HttpContextN
8. Endpoint1.publish(HttpContext1)  --> 9. creates HttpHandler1
                                         HttpContext1.setHandler(HttpHandler1)
   ...
```

10. EndpointN.publish(HttpContextN) --> 11. creates HttpHandlerN
HttpContextN.setHandler(HttpHandlerN)

The request processing is done as below(for every request):

Container JAX-WS runtime

---

1. Creates a HttpExchange
2. Gets handler from HttpContext
3. HttpHandler.handle(HttpExchange) --> 4. reads request from HttpExchange
The portable undeployment is done as below:

Container

---

1. @preDestroy on instances
2. Endpoint1.stop()
...
3. EndpointN.stop()

- Since:

- JAX-WS 2.2

- Author:

- Jitendra Kotamraju

 Generated on 1-December-2009 05:21

Defining Java EE applications

You can use the tools in the workspace to define the module dependencies in the Java™ EE project in your workspace.

Related concepts:
Developing Java EE Applications
Java EE: Overview
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Securing enterprise applications

Related tasks:
Setting Java EE preferences
Creating and configuring Java EE projects using wizards
Validating code in enterprise applications
Deploying Java EE applications
Migrating the specification level of Java EE projects
Generating deployment descriptors
Generating WebSphere extensions and bindings deployment descriptors
Specifying dependent JAR files or modules
Exporting and importing binary modules

Application deployment descriptor editor

While deployment descriptors are not required in Java™ EE as they were in J2EE, you can include deployment descriptors in your enterprise applications, and change to your module dependencies using the deployment descriptor editor.

The application deployment descriptor editor includes scrollable pages and collapsible sections that represent the various properties and settings in the deployment descriptor (`application.xml`) and other metadata written to bindings and extensions files. The editor is dynamic, and sections and pages are created based on the application deployment descriptor version and the workbench capabilities that are enabled. To open the editor, right click the deployment descriptor for your project and select **Open With** > **Deployment Descriptor Editor**.

Deployment descriptors are not automatically generated when you create your enterprise project, unless you select **Generate Deployment Descriptor** in the project creation wizard. For more information on creating deployment descriptors, see Generating deployment descriptors

The deployment descriptor editor contains two pages, the Design page and the Source page. The Design page contains a number of sections, including Overview, General Information and Icons. Collapsing a section hides the content, but leaves the heading information. This is useful in filtering through the data and properties. The editor remembers the sections that you collapse when you close and reopen the editor. The Source page contains the source for your application, the `application.xml`.

Design page

The Design page in the application editor provides a quick summary of the contents in the application deployment descriptor. It includes the following sections: General Information, Modules, Security Roles, Icons, and WebSphere® Extensions.

- **Overview section**
  - The Overview section displays the names of the modules that are defined for the application, and provides a quick link to the Module page of the editor. You can use this section to add, edit, browse, and remove EJB, Web, and Application Client modules from the enterprise application. When you select a module in the list, its attributes are displayed on the fields on the right side of the pane. The list of fields changes dynamically to match the type of module selected.
- **General Information section**
  - Use the General Information section to view the display name and description for the enterprise application, as stored in the `application.xml` file.
- **Icons section**
  - Use the Icons section to choose icons that represent your enterprise application. These icons are used for identification on the server. In order to use an icon, you must first import the graphic file into the enterprise application project (basically, it must be contained inside the EAR file in order for it to be found at deploy time). Once the file has been imported into the project, you are able to select it within the icon dialog on the application deployment descriptor editor. If you do not import the file into the project, you do not see any icons within the dialogs.

- **Actions section**
  - The Actions section provides links for you to perform the following actions:
    - **Manage Utility Jars**: Use this link to add a Java project as a utility JAR file that can be used by modules in the enterprise application. For each Java project, a utility JAR is created when the EAR file is exported.
- **WebSphere Deployment Descriptors**
  - The Actions section provides links for you to perform the following actions:
    - **Open WebSphere Programming Model Extensions Descriptor** : (For applications that target WebSphere Application Server) The WebSphere bindings section provides a place to add WebSphere Programming Model Extensions.
    - **Open WebSphere Bindings**: (For applications that target WebSphere Application Server) The WebSphere bindings section provides a place to add users and groups to the security roles.
    - **Open WebSphere Extensions**: (For applications that target WebSphere Application Server) The WebSphere Extensions section provides a place to add WebSphere Extensions.

    If you do not have WebSphere bindings or extensions files created for your project, when you click the action link, a message appears stating that you do not have these files. For information about how to create WebSphere Extensions and Bindings deployment descriptors, see Generating WebSphere extensions and bindings deployment descriptors

Source page

Use the Source page to view and modify the `application.xml` file directly. The XML on the source page changes dynamically when the deployment descriptor is edited, and the other pages of the application deployment descriptor editor reflect changes that you make on the Source page. Editing the XML source is not the best was to edit the deployment descriptor; use the Design page of the editor to make your changes.

Related tasks:
Generating deployment descriptors
Generating WebSphere extensions and bindings deployment descriptors
Specifying dependent JAR files or modules
Exporting and importing binary modules

Generating deployment descriptors

You can create deployment descriptors to manage the dependencies for your enterprise projects.

About this task

You can create deployment descriptors for your enterprise applications, either by selecting **Generate Deployment Descriptor** in the project or module creation wizard, or you can add it later:

Procedure

1. Right click your enterprise application project.
2. Select **Java EE** > **Generate** > **Generate Deployment Descriptor Stub**
3. The deployment descriptor file for your project or module is created in your project.

Related concepts:
Application deployment descriptor editor
Defining Java EE applications

Generating WebSphere extensions and bindings deployment descriptors

You can create deployment descriptors to manage the WebSphere® extensions and bindings for your enterprise application and web projects.

Procedure

1. Right click your enterprise application or web project. Select **Java EE** > **Generate** , and then select one of the following options:
   - Generate WebSphere Bindings Deployment Descriptor
   - Generate WebSphere Extensions Deployment Descriptor
   - Generate WebSphere Programming Model Extensions Deployment Descriptor
2. Alternatively, open the deployment descriptor for your project, and in the WebSphere Deployment Descriptor section, select one of the links. If the descriptor that you select does not exist, a message appears asking if you want to create one. Click **Yes**.

3. The deployment descriptor file for your project or module is created in your project.

Related concepts:
Application deployment descriptor editor
Defining Java EE applications

Application Client Deployment Descriptor editor

The Application Client deployment descriptor editor is organized with scrollable pages and collapsible sections that represent the various properties and settings that can be defined in the deployment descriptor (application-client.xml) and other extensions and bindings files. The editor is dynamic, and sections and pages are created based on the Application Client deployment descriptor version and the workbench capabilities preferences. To open the editor, right click the deployment descriptor for your project and select **Open With** > **Deployment Descriptor Editor**.

The core function is typically located at the top of the page. The extensions and bindings are typically nested sections at the bottom of the editor pages. Collapsing a section hides the content, but leaves the heading information. This is useful in filtering through the data and properties on each page.

Depending on the properties of the project and the enabled capabilities, the Application Client deployment descriptor editor typically helps you to modify the following resources:

- application-client.xml
- ibm-application-client-bnd.xmi
- ibm-application-client-ext.xmi
- ibm-application-client-ext-pme.xml

The Application Client deployment descriptor editor typically displays the following pages:

Overview page

The Overview page in the Application Client editor provides a quick summary of the contents in the Application Client deployment descriptor. This page includes the following sections:

- **General Information section**
  - Use the General Information section to view and edit the display name and description for the application Application Client.
- **References section**
  - The References section shows the currently defined references and links you to the References page of the editor.
- **Environment Variables**
  - Use the Environment Variables section to define environment variables for the application Application Client module.
- **Icons section**
  - Use the Icons section to choose icons that represent your Application Client application. These icons are used for identification on the server. In order to use an icon, you must first import the graphic file into the enterprise application project (it must be contained inside the EAR file in order for it to be found at deploy time). Once the file has been imported into the project, you can select it within the icon dialog on the Application Client deployment descriptor editor. If you do not import the file into the project, you do not see any icons within the dialogs.
- **Main Class section**
  - Use the Main Class section to edit or refresh the Main Class for the application client. Clicking **Edit** opens the Manifest editor where the Main Class attribute is defined. Clicking **Refresh** forces an update of the field in the application client editor.
- **Callback Handler**
  - For J2EE 1.3 and 1.4 only, use the Callback Handler section to specify the class that the application client uses for handling callback.
- **WebSphere® Extensions**
  - This section shows additional extension properties that you can define specifically for the WebSphere Application Server. For example, you could specify whether you want to allow JTA demarcation.
- **Message Driven Destinations**
  - Use this section to define a message driven destination for the application client. Message driven destinations are elements that specify logical message destinations within an application.

References page

Use the References page to define references for the application client deployment descriptor. See Defining references for J2EE modules for more information.On this page, you can also define a JNDI name for the reference. This is a WebSphere Application Server binding property. For more information, see the WebSphere Application Server documentation.

WS Extension page

For J2EE 1.4 project only, use this page to define Web service client security extensions for WebSphere Application Server.

WS Binding page

For J2EE 1.4 project only, use this page to define Web service client bindings for WebSphere Application Server.

WS Handler page

For J2EE 1.3 and 1.4 project only, use the Handlers page to define Web service handlers for each Web service reference that is defined for the enterprise beans in the EJB module. See Defining Web service handlers for more information.

Extended Services page

Use this page to view, add, edit, or remove application-managed tasks and container-managed tasks associated with the project.

Source page

Use the Source page to view and modify the XML source code directly. The XML on the source page changes dynamically when the deployment descriptor is edited, and the other pages of the application deployment descriptor editor reflect changes that you make on the Source page. Editing the XML source is not the recommended method for editing the deployment descriptor. It is suggested that you make as many changes as possible using the other pages and sections of the editor.

Related concepts:

Manifest editor
J2EE architecture
Application client projects

Related tasks:

Specifying dependent JAR files or modules
Defining the Main Class for J2EE modules
Defining Web service handlers (J2EE 1.x)
Defining icons for Java EE modules (J2EE 1.4)
Creating an application client project
Exporting an application client project
Importing an application client JAR file

Defining references in Java EE modules (J2EE 1.4)

Java™ EE application clients, Web modules, and individual enterprise beans in EJB modules can include references to other resources. Reference types include EJB references, message destination references, resource references, resource environment references, security role references, and Web service references. You can use the deployment descriptor editors to invoke the Add Reference wizard from thedeployment descriptor editor.

About this task

The Java EE and EJB specifications provide for the means for components to define "logical" names that refer to other resources. For example, an EJB reference is a logical name that refers to the home of an enterprise bean. A Web service reference is a logical name that refers to a Web service.You can use the various deployment descriptor editors to define references. A wizard helps you to define new references, and you can quickly remove or modify properties of existing references within the deployment descriptor editor.

Procedure

1. In the Enterprise Explorer of the Java EE perspective, select the deployment descriptor for your project where you want to include a reference. You can define references for EJB projects, application client projects, and web projects.
2. Right-click the deployment descriptor and select **Open With** > **Deployment Descriptor Editor** from the pop-up menu.
3. Click the **References** tab to go to the References page of the deployment descriptor editor.
4. Click **Add** to open the Add Reference wizard.**Note:** For EJB projects, references are defined on a per bean basis. You must select an existing bean before you can click **Add** to define a reference.
   For more information about how to use the Add Reference wizard, see the related tasks links in this topic.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Adding EJB references (EJB 1.x)
Adding message destination references (J2EE 1.4)
Adding resource manager connection factory references (J2EE 1.x)
Adding resource environment references (J2EE 1.4)
Adding security role references (J2EE 1.4)
Adding Web service references (J2EE 1.4)
Adding message destinations

Adding EJB references (EJB 1.x)

An EJB reference is a logical name used by a client (or another bean) to locate the home interface of an enterprise bean. You can define references to enterprise beans in EJB modules, J2EE application client modules, and J2EE web modules.

About this task

It is a best practice to use an EJB reference for any enterprise bean that you need to reference. Using an EJB reference allows you to safely write Java™ code to look up the home interface of the target enterprise bean without the worry of the binding changing for the target enterprise bean. This is necessary if you need to install the same EJB module on the same server with different bindings.

At deployment, the EJB reference is bound to the enterprise bean's home in the target operational environment. The container makes the application's EJB references available in a JNDI naming context.

Note:

- EJB 1.1 enterprise beans and application client modules cannot reference local interfaces of enterprise beans. They must reference the remote interface.
- In order for a module or a bean to reference the local interface of an enterprise bean, the referenced bean must be included in the same enterprise application (EAR) as the module or bean that is referencing it.
- If either local or remote references are possible, you should use local references for runtime performance advantages.

For each EJB reference that you define, an `ejb-ref` element is added to the deployment descriptor.

Procedure

1. Open the Add Reference wizard from the deployment descriptor editor for your J2EE project.
2. Select **EJB reference** and click **Next**.
3. Choose one of the following options for specifying the enterprise bean that you want to reference:
   - **Enterprise beans in the workspace**: Select this option to choose an enterprise bean from the projects currently in your workspace. The tree shows EJB modules in the workspace and lists the available enterprise beans in the EJB modules.
   - **Enterprise beans not in the workspace**: Select this option to choose an enterprise bean outside of your workspace.
4. If you selected **Enterprise beans in the workspace** complete the following steps:
   A. Expand the project tree and select the enterprise bean that you want to reference.
   B. In the **Name** field, enter a name for the reference.
   C. If you are referencing an EJB 2.1 bean, in the **Ref type** field specify whether you are referencing the local or remote interface of the enterprise bean.**Important:** If you reference an enterprise bean in another EAR, and that bean's project does not include an EJB client JAR, the workbench automatically creates an EJB client JAR for that referenced EJB project. This improves runtime performance and allows for visibility across different EAR files.
5. If you selected **Enterprise beans not in the workspace**, complete the following steps:
   A. In the **Name** field, enter a name for the reference.
   B. Select **Remote** from the **Ref type** drop-down.
   C. In the **Type** field, specify whether the referenced bean is a session or entity bean.
   D. In the **Home** field, enter the qualified path to the remote home of the enterprise bean. Click **Browse** to locate the remote home using the **Type Selection** window.
   E. In the **Remote** field, enter the qualified path to the remote interface for the enterprise bean. Click **Browse** to locate the remote interface using the **Type Selection** window.
6. Click **Next** to review your selections and enter a description for the reference. Click **Finish**.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Defining references in Java EE modules (J2EE 1.4)
Adding message destination references (J2EE 1.4)
Adding resource manager connection factory references (J2EE 1.x)
Adding resource environment references (J2EE 1.4)
Adding security role references (J2EE 1.4)
Adding Web service references (J2EE 1.4)

Adding Web service references (J2EE 1.4)

Starting with the J2EE 1.3 specification, application components (application clients, Web modules, EJB modules) can define references to external Web services by using "logical" names called Web service references. In J2EE 1.3, the references are added to a webservicesclient.xml in a module. Beginning with J2EE 1.4, the reference is included in the deployment descriptor. You can use the deployment descriptor editor todefine a web service reference.

About this task

At deployment, the Web service references are bound to the Web service interfaces in the target operational environment. For each Web service reference that you define, a `service-ref` element is added to that application component. Web service references are scoped to the application component or enterprise bean where they are defined, so they are not accessible to other application components or beans during run time. Other components can define Web service references with the same name without causing a name conflict.

Note: For project levels previous to the J2EE 1.4 specification level, the workbench allows you to define Web service references, but the `service-ref` elements are not added to the deployment descriptors (the application-client.xml, web.xml, or ejb-jar.xml). Rather, the Web service references are declared in a webservicesclient.xml file in the WEB-INF folder of the module. For example, if you add a Web service reference to a J2EE 1.3 application client, the reference is added to webservicesclient.xml. If you add a Web service reference to a J2EE 1.4 application client, the reference is added to application-client.xml.

Tip: The EJB specification recommends that you organize all Web service references in the service sub-context of the bean's environment, the `java:comp/env/service` JNDI context.

Procedure

1. Open the Add Reference wizard from the deployment descriptor editor for your J2EE project.
2. Select **Service reference** and click **Next**.
3. From the list of Web services in your workspace, select the Web service that you want to reference. A default name appears in the **Name** field that you can change. This is used as the name for the reference in the `service-ref-name` entry in the deployment descriptor.Click **Next**.
4. In the **Description** text area, enter a description for the reference.
5. In the **Namespace URI** field, enter a new namespace URI. The value entered in this field is used as the `xmlns:prefix` attribute in the `service-qname` entry in the deployment descriptor.
6. In the **Local part** field, indicate whether the message destination consumes or produces messages. The value entered in this field is used as the value in the `service-qname` entry in the deployment descriptor:
```
<service-qname
xmlns:prefix="http://service.directory">
  prefix:EmployeeDirectoryService
</service-qname>
```
7. Click **Finish**.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Defining references in Java EE modules (J2EE 1.4)

Adding EJB references (EJB 1.x)

Adding message destination references (J2EE 1.4)

Adding resource manager connection factory references (J2EE 1.x)

Adding resource manager connection factory references (J2EE 1.x)

The J2EE specification provides a means for J2EE components to refer to resource manager connection factories by using "logical" names called resource manager connection factory references. You can use the deployment descriptor editors to define resource manager connection factory references.

About this task

A resource manager connection factory is an object that is used to create connections to a resource manager. For example, an object that implements the javax.sql.DataSource interface is a resource manager connection factory for java.sql.Connection objects that implement connections to a database management system.

At deployment, the resource manager connection factory references are bound to the actual resource manager connection factories that exist in the target operational environment.

Resource manager connection factory objects that are accessed through a reference are only valid within the component instance that performed the lookup.

For each resource manager connection factory reference that you define, a `resourece-ref` element is added to the deployment descriptor for that application component. The references are scoped to the application component where they are defined, so they are not accessible to other application components during run time. Other components can define resource manager connection factory references with the same name without causing a name conflict.

Procedure

1. Open the Add Reference wizard from the deployment descriptor editor for your J2EE project.
2. Select **Resource reference** and click **Next**.
3. In the **Name** field, specify a name for the reference, or accept the default name provided by the wizard (recommended). The value entered in the **Name** field is used in the `res-ref-name` entry in the deployment descriptor.**Note:** The name of the reference is relative to the `java:comp/env` context. For example, the name should be `jms/StockHistoryDB` rather than `java:comp/env/jms/StockHistoryDB`.
4. In the **Type** field, select the Java programming language type of the resource manager connection factory that the application component code expects. The value entered in this field is used in the `res-type` entry in the deployment descriptor.
5. In the **Authentication** field, indicate whether your application (or servlet, for JSP 1.2 Web projects) performs resource sign-on programmatically (Application), or whether the container manages all authentication for this resource (Container). The value specified here is used in the `res-auth` entry in the deployment descriptor.
6. For J2EE 1.3 or later projects, in the **Shareable** field, indicate whether the connections to the resource manager that are obtained through the given resource manager connection factory reference can be shared. By default, connections are assumed to be shareable. The value specified here is used in the `res-sharing-scope` entry in the deployment descriptor.
7. In the **Description** text area, enter a description for the reference. Click **Finish**.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Defining references in Java EE modules (J2EE 1.4)
Adding EJB references (EJB 1.x)
Adding message destination references (J2EE 1.4)

Adding message destination references (J2EE 1.4)

The J2EE 1.4 specification provides a means for J2EE components to refer to message destination objects by using "logical" names called message destination references. You can use the deployment descriptor editors to define message destination references.

About this task

A message destination reference points to a message destination that is defined in an EJB module, application client module, or web module.

At deployment, the message destination references are bound to the administered message destinations in the target operational environment.

Restriction: Only the following minimum project levels can include message destination references:

- J2EE 1.4 Application Clients
- EJB 2.1 project
- 2.4 Web Applications

For each message destination reference that you define, a `message-destination-ref` element is added to the deployment descriptor for that application component. Message destination references are scoped to the application component where they are defined, so they are not accessible to other application components during run time. Other components can define message destination references with the same name without causing a name conflict.

The following code shows an example message destination defined in a deployment descriptor:

```
<message-destination-ref>
 <description></description>
 <message-destination-ref-name>MyDest_Ref</message-destination-ref-name>
 <message-destination-type>java.net.URL</message-destination-type>
 <message-destination-usage>Produces</message-destination-usage>
 <message-destination-link>MyDest</message-destination-link>
</message-destination-ref>
```

Message destination links (the `message-destination-link` element) can be defined on message-driven beans and message destination references. The `message-destination-link` element of the `message-destination-ref` element of an enterprise bean produces messages to link to a target destination. The value for the `message-destination-link` element is the name of a message destination.

Procedure

1. Open the Add Reference wizard from the deployment descriptor editor for your J2EE project.
2. Select **Message destination reference** and click **Next**.
3. In the **Name** field, specify a name for the reference. The value entered in the **Name** field is used in the `message-destination-ref-name` entry in the deployment descriptor.**Note:** The name of the message destination reference is relative to the `java:comp/env` context. For example, the name should be `jms/BidQueue` rather than `java:comp/env/jms/BidQueue`.
4. In the project tree, select the message destination that you want to reference. You can click **New Destination** to add a new message destination to the deployment descriptor for your current project.**Note:** The wizard allows you to finish without linking the reference to a message destination, but you need to add the link later in the deployment descriptor.
5. Click **Next**.
6. In the **Type** field, select the expected type of the referenced destination. For example, for a JMS destination, the type might be `javax.jms.Queue`. The value entered in this field is used in the `message-destination-ref-type` entry in the

deployment descriptor.

7. In the **Usage** field, indicate whether the message destination consumes or produces messages. The value specified here is used for the `message-destination-ref-usage` entry in the deployment descriptor.
8. In the **Description** text area, enter a description for the reference. Click **Finish**.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Defining references in Java EE modules (J2EE 1.4)
Adding EJB references (EJB 1.x)
Adding resource manager connection factory references (J2EE 1.x)
Adding resource environment references (J2EE 1.4)
Adding security role references (J2EE 1.4)
Adding Web service references (J2EE 1.4)
Adding message destinations

Adding message destinations

The J2EE 1.4 specification provides a means for J2EE components to refer to message destination objects by using "logical" names called message destination references. You can use the deployment descriptor editors to define message destination references.

About this task

Message destinations are elements that specify logical message destinations within an application. The `message-destination` element defines a `message-destination-name`, which is used for linking. Message destinations can be defined in any module in the J2EE application as the referencing component. A message destination link on a message destination reference or a message-driven bean points to the name of a message destination.

At deployment, the message destination references are bound to the administered message destinations in the target operational environment.

Restriction: Only the following minimum project levels can include message destinations:

- J2EE 1.4 Application Clients
- EJB 2.1 projects
- 2.4 Web Applications

For each message destination that you define, a `message-destination` element is added to the deployment descriptor for that application component.

The following code shows an example message destination defined in a deployment descriptor:

```
<message-destination>
 <description></description>
 <message-destination-name>MyDest</message-destination-name>
</message-destination>
```

Procedure

1. Open the deployment descriptor editor for the module project where you want to add a message destination. To do this, in the Enterprise Explorer of the Java™ EE perspective, double-click the deployment descriptor node for your project.
2. In the **Message Destinations** section of the editor, click the **Add** button. This section is on different pages of the editor depending on the module type:
   - Application client projects: Overview page
   - Dynamic Web projects: Variables page
   - EJB projects: Assembly page
3. In the **Name** field, specify a name for the message destination. The value entered in the **Name** field is used in the `message-destination-name` element in the deployment descriptor and is the value that could be used by a message destination reference or message-driven bean as its `message-destination-link` element. The message destination allows an EJB to send a message to a specific message-driven bean in the same application. The name is arbitrary, and both sender and receiver specify the same name as the value for their <message-destination-link> elements. Here is an example of code containing a sample <message-destination> value:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
 version="3.0">
 <display-name>TestEJB</display-name>
 <enterprise-beans>
  <message-driven>
```

```
    <ejb-name>ConsumerBean</ejb-name>
    <message-destination-link>ConsumerDestination</message-destination-link>
   </message-driven>
   <session>
    <ejb-name>ProducerBean</ejb-name>
    <message-destination-ref>
     <message-destination-ref-name>beans.ProducerBean/destination</message-destination-ref-name>
     <message-destination-link>ConsumerDestination</message-destination-link>
    </message-destination-ref>
   </session>
  </enterprise-beans>
  <assembly-descriptor>
   <message-destination>
    <description></description>
    <message-destination-name>ConsumerDestination</message-destination-name>
   </message-destination>
  </assembly-descriptor>
</ejb-jar>
```

4. In the **Description** text area, enter a description for the message destination. Click **Finish**.
5. Click **Finish**.

Results

The message destination is added to the deployment descriptor. Now, when you define message destination references, you can link to this message destination. In the **Message Driven Destinations** section of the editor, you can select the message destination and change the name and description. You can also remove the message destination.

Related tasks:

Defining references in Java EE modules (J2EE 1.4)

Adding message destination references (J2EE 1.4)

Adding resource environment references (J2EE 1.4)

The J2EE specification provides a means for J2EE application components to refer to administered objects that are associated with a resource. The reference is done by using "logical" names called resource environment references. You can use the deployment descriptor editors to define resource environment references.

About this task

At deployment, the resource environment references are bound to the actual administered objects in the target operational environment.

For each resource environment reference that you define, a `resourece-env-ref` element is added to the deployment descriptor for that application component. Resource environment references are scoped to the application component where they are defined, so they are not accessible to other application components during run time. Other components can define resource environment references with the same name without causing a name conflict.

Resource environment references are available only for J2EE 1.3 or later, and EJB 2.x.

To define a resource environment reference, complete the following steps:

Procedure

1. Open the Add Reference wizard from the deployment descriptor editor for your J2EE module.
2. Select **Resource environment reference** and click **Next**.
3. In the **Name** field, specify a name for the reference. The value entered in the **Name** field is used in the `resource-env-ref-name` entry in the deployment descriptor. The name is the environment entry name that is used in the application code.**Note:** The name of the reference is relative to the `java:comp/env` context. For example, the name should be `jms/StockHistoryDB` rather than `java:comp/env/jms/StockHistoryDB`.
4. In the **Type** field, select the expected type of the referenced object. The value entered in this field is used in the `resource-env-ref-type` entry in the deployment descriptor.
5. In the **Description** text area, enter a description for the reference. Click **Finish**.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Defining references in Java EE modules (J2EE 1.4)
Adding EJB references (EJB 1.x)
Adding message destination references (J2EE 1.4)
Adding resource manager connection factory references (J2EE 1.x)
Adding security role references (J2EE 1.4)
Adding Web service references (J2EE 1.4)

Adding security role references (J2EE 1.4)

EJB modules must define security role references for all the security role names that are used in the enterprise bean code or that are defined in the deployment descriptor. You can use the EJB deployment descriptor editor to declare the security role references.

About this task

Each security role reference that you declare in your EJB module uses the `security-role-ref` element in deployment descriptor. If the security role is declared in the deployment descriptor, you must use the `role-link` element in the `security-role-ref` element to link the reference to the declared security role.

Procedure

1. Open the Add Reference wizard from the EJB module where you are declaring the security role reference.
2. Select **Security role reference** and click **Next**.
3. In the **Name** field, specify a name for the reference, or accept the default name provided by the wizard (recommended). The value entered in the **Name** field is used in the `role-name` entry in the deployment descriptor. The name must be the security role name that is used as a parameter to `javax.ejb.EJBContext.isCallerInRole(String roleName)`.
4. In the **Link** drop down, if the security role that you are referencing is declared in the deployment descriptor, select the security role. The value in the Link field is used in the `role-link` entry in the deployment descriptor.
5. In the **Description** text area, enter a description for the reference.
6. Click **Finish**.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Defining references in Java EE modules (J2EE 1.4)
Adding EJB references (EJB 1.x)
Adding message destination references (J2EE 1.4)
Adding resource manager connection factory references (J2EE 1.x)
Adding resource environment references (J2EE 1.4)
Adding Web service references (J2EE 1.4)

Defining Web service handlers (J2EE 1.x)

For J2EE 1.3 and 1.4 modules that include Web service references, you can use the deployment descriptor editor to define and configure Web service handlers for each Web service reference. Handlers allow you to process SOAP message header traffic for the remote call to the Web service. The SOAP header is defined by the SOAP specification. Handlers are defined by JAX-RPC and the Web Services for J2EE specification. You can use the deployment descriptor editor to to define Web service handler for a Web service reference:

Before you begin

In order to define Web service handlers, you must first define a Web service reference.

About this task

When you define a handler for a Web service reference, the `handler` element is added to the related service reference (the `service-ref` element). For J2EE 1.3, the `service-ref` element is in the webservicesclient.xml file. For J2EE 1.4, it is in the deployment descriptor.

Procedure

1. In the Enterprise Explorer of the Java™ EE perspective, expand the project tree and select the deployment descriptor for your project where you want to define a Web service handler. You can define Web service handlers for Web service references in EJB projects, application client projects, and web projects.
2. Right-click the deployment descriptor and select **Open With** > **Deployment Descriptor Editor** from the pop-up menu. The appropriate deployment descriptor for your project type opens. For example, an EJB module opens in the EJB deployment descriptor editor.
3. Click the **WS Handler** tab to go to the Handlers page of the deployment descriptor editor.
4. In the **Service references** drop-down list, select the Web service reference that you want to define the handler for.
5. Click **Add** to open the New Handler dialog box and define the handler:
   A. In the **Display name** field, type a value for the `display-name` element in the deployment descriptor.
   B. In the **Description** field, type a value for the `description` element for the handler.
   C. In the **Handler name** field, type a value for the `handler-name` element.
   D. In the **Handler class** field, enter the fully qualified name of the Java class for the handler. This value is used in the `handler-class` element.
6. Click **Finish**. The handler is added to the list of handlers defined for the selected reference. If you select the handler, you can see and modify the properties.
7. Optional: In the **Icons** section, you can specify small and large GIF or JPEG images to help identify the Web service handler at run time. The images must first be imported into the project.
8. Optional: In the **Initial parameters** section, click the **Add** button to define a name and value for an initialization parameter for the selected handler.
9. Optional: In the **SOAP headers** section, click the **Add** button to define the namespace URL and local part for a new SOAP header for the handler.

Related concepts:

Application Client Deployment Descriptor editor

Defining WebSphere extensions and bindings for application clients

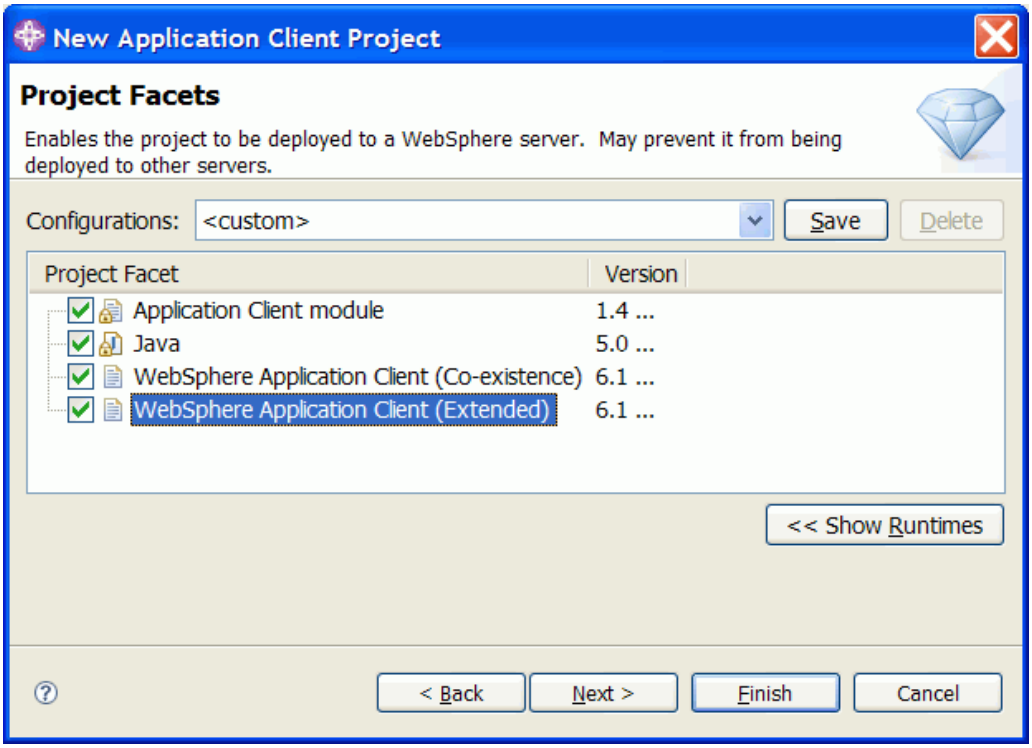
The application server requires binding information to bind the deployment information specified in the application to a specific resource.

Editing deployment descriptors

A deployment descriptor is an extensible markup language (XML) file that describes how to deploy a module or application by specifying configuration and container options. The Application Client deployment descriptor editor helps you to define deployment information in an Application Client deployment descriptor for modules that you create in the Application Client development environment.

Before you begin

This topic assumes that you have assembled code artifacts into an Application Client module that you want to deploy onto an application server. In order for the WebSphere® extensions and bindings elements to show up in the Application Client project's deployment descriptor, you must ensure that you select `WebSphere Application Client(Extended)` facet when creating your new Application Client project.



Why and when to perform this task

When you use an assembly tool to create a module, the assembly tool creates deployment descriptor files for the module automatically.

You can edit a deployment descriptor file manually. However, it is preferable to edit a deployment descriptor using an assembly tool deployment descriptor editor to ensure that the deployment descriptor contains valid properties and that its references hold appropriate values.

Table 1. Application Client Deployment Descriptor

Deployment descriptor editor	Resources modified in the editor
Application Client deployment descriptor editor	application-client.xml
	ibm-application-client-bnd.xmi
	ibm-application-client-ext.xmi

The application server requires binding information to bind the deployment information specified in the application to a specific resource. For example, it can map a logical name of an external dependency or resource to the actual physical JNDI name of the resource. It also can map security role information to a set of groups or users. IBM® extensions are additions to the standard descriptors for Application client projects.
These pages provide explanation of the controls to set numerous application parameters related to paths, and variables referenced, security, and other general deployment settings. As you specify deployment information, the editor incorporates the appropriate tagging.

- **Defining JNDI bindings of references for Application Client Projects**
  Use the deployment descriptor editor to define the JNDI bindings of references for an application client project.

Defining JNDI bindings of references for Application Client Projects

Use the deployment descriptor editor to define the JNDI bindings of references for an application client project.

Procedure

1. In Enterprise Explorer, expand the Application Client Projects and select your application client project.
2. Expand your project name and double click the **Deployment Descriptor**.
3. On the **References** page of the editor, in the **References** section, go to the WebSphere® Bindings section.
4. In the **JNDI name** field, type a JNDI name.

Specifying dependent JAR files or modules

You can use the Manifest editor to specify JAR files or modules that are required by a module. The dependencies are defined in the MANIFEST.MF file for your module.

About this task

When you are specifying required JAR files or modules, you first specify the enterprise application (EAR) that your project is a part of. Typically, the project is referenced by one EAR project in the workspace. However, it is possible that you have multiple enterprise applications that contain a reference to the same module or utility JAR project. If so, then you should ensure that you give the JAR or module the same Uniform Resource Identifier (URI) in each application, so that the class path is valid for all applications.

It is also possible that your module is a standalone project and is not currently referenced by any enterprise application. In this case, since there is no enterprise application scope defined, you cannot use the Manifest editor to update the dependencies. To add the module to an enterprise application, see Adding modules to an enterprise application.

Procedure

1. In the Enterprise Explorer view of the Java™ EE perspective, right-click your project's MANIFEST.MF file, and select **Open With** > **Manifest Editor** from the pop-up menu. The MANIFEST.MF file is located in the following locations for the different module types:

Project type	Location of manifest file
EJB projects	ejbModule/META-INF/MANIFEST.MF
Application client projects	appClientModule/META-INF/MANIFEST.MF
Web projects	WebContent/META-INF/MANIFEST.MF
Connector projects	connectorModule/META-INF/MANIFEST.MF

   Tip: For application client modules, you can also click **Edit** in the Main Class section of the Client Deployment Descriptor editor to launch the Manifest Editor.
2. In the **Classpath Scope** section of the editor, select the enterprise application to use for class path editing. Because the Manifest editor is designed for class path editing, not all the attributes or information contained in the manifest file are represented on this page. The Classpath Scope section lists all enterprise applications that contain a reference to the selected project as a module or utility JAR. Use the **Refresh** button to update the list if a change is made in a separate editor.
3. In the **Dependencies** section, select the JAR files or modules that are required or dependent. You can also move them up and down in the list to specify their order on the MANIFEST class path as well as the Java build path.In cases where your module depends on an EJB module that has an EJB client JAR file, you can choose whether you want to depend on the EJB JAR file or the EJB client JAR file. If you select the **Use EJB JARs** radio button, the table does not show any EJB client JAR files. If you select the **Use EJB client JARs** radio button, the table does not show any EJB JAR files that have corresponding EJB client JAR files. If you select the **Allow both** radio button, the table shows EJB JAR files and EJB client JAR files and allows you to select both types.

   Tip: The **Dependencies** section automatically switches dependencies based on which radio button you select. For example, if you have a dependency set on an EJB JAR file and you select the **Use EJB client JARs** radio button, the dependency switches to the appropriate EJB client JAR file. If you select the **Allow both** radio button, none of your dependency selections change automatically.
4. In the **Implementation Version** section, type the version of the module you are implementing.
5. In the **Annotation Scanning** section, you can specify archives and packages that you want to exclude from annotation scanning. For more information on annotation scanning see: Excluding files from annotation scanning.

6. In the **Main Class** section, you can use the manifest editor to set the main Java class as the entry point for the application.
7. Click **File** > **Save** to save your changes.

Results

Tip: If you need to compile against a server's runtime JAR files during development, you do not need to add these JAR files as dependent JAR files. The workbench manages this by using the target server property for the project. The workbench adds the appropriate libraries to your project's build and class path based on the target server. For more information, see Specifying target servers for J2EE projects.

Related concepts:

Application Client Deployment Descriptor editor
Manifest editor
Cyclical dependencies between J2EE modules

Related tasks:

Defining the Main Class for J2EE modules
Adding project utility JAR files (J2EE 1.4)
Importing an enterprise application EAR file
Correcting cyclical dependencies after an EAR is imported

Manifest editor

The Manifest editor updates the manifest class path for a utility JAR or module in an enterprise application. The Java™ build path is updated accordingly for the containing project. It also sets the Main Class attribute of the manifest, which is typically only used for application client modules.

To use the Manifest editor, right-click your project's MANIFEST.MF file and select **Open With** > **Manifest Editor**. The Manifest editor consists of the following pages and sections:

Dependencies page

The Manifest editor is designed for classpath editing. Therefore, not all the attributes or information contained in the manifest file (MANIFEST.MF) are represented in this topic.

- Classpath Scope section

- Use this section to select the enterprise application to use for class path editing. The list contains all enterprise applications which contain a reference to the current project as a module or utility JAR. Use the **Refresh** button to update the list if a change is made in a separate editor.

- Dependencies section

- Use this section to select other JAR files or modules contained by the enterprise application that are required by the JAR file or module you are currently working with. You can see existing class path entries in the manifest file, and all available, valid entries in the selected application in the preceding section.In cases where your module depends on an EJB module that has an EJB client JAR file, you can choose whether you want to depend on the EJB JAR file or the EJB client JAR file. If you select the **Use EJB JARs** radio button, the table does not show any EJB client JAR files. If you select the **Use EJB client JARs** radio button, the table does not show any EJB JAR files that have corresponding EJB client JAR files. If you select the **Allow both** radio button, the table shows EJB JAR files and EJB client JAR files and allows you to select both types.

  Tip: The Dependencies section automatically switches dependencies based on which radio button you select. For example, if you have a dependency set on an EJB JAR file and you select the **Use EJB client JARs** radio button, the dependency switches to the appropriate EJB client JAR file. If you select the **Allow both** radio button, none of your dependency selections changes automatically.

- Implementation Version section

- Use this section to specify a version for the JAR packaging.

- Annotation Scanning

- Use this section to specify archives and packages that you want to exclude from annotation scanning. For more information on annotation scanning see: Excluding files from annotation scanning

- Main Class section

- Use this section to set the main-class, or entry point, of the application. The Main-Class attribute specifies the Java class of the application entry point.

.

Source page

It is not necessary to manually edit manifest files in J2EE projects. However, if you choose to manually edit these files, you should be aware of some common limitations from the Manifest Format specification:

- The relative path of the file must be `META-INF/MANIFEST.MF`, in all capital letters.
- Lines in the file must be no greater than 72 characters. Continuation of long lines is indicated by a carriage return and a space.
- The last line in the file must end in a carriage return, or it does not get parsed.

For more information, refer to the official Manifest Format specification. The standard Java APIs are used for reading and writing manifest files.

Adding a directory to the MANIFEST.MF file

When you add a directory to the MANIFEST.MF file, you receive an error message like the one below: `IWAE0024W The`

```
Manifest Class-Path for archive xyz.jar contains an entry,
 properties, that is not resolveable to a file or module in the EAR:
 sample...
```

The EJB specification is not explicit on this issue. However, it does suggest that loose files within an EAR are invalid. Currently, this configuration does work in the WebSphere® Application Server, but you should not rely on this configuration working in the future. You can create a Java project and add the property files to a source folder (or the project if the project is the source folder). On the Modules page of the application deployment descriptor editor, you can add this Java project as a Project Utility JAR, then you can run the EAR file in the WebSphere Test Environment. When you export the EAR file, the Java project is automatically added to a JAR and included in the EAR.

For information about Annotation scanning, see Excluding files from annotation scanning.

Related concepts:

Application Client Deployment Descriptor editor

Related tasks:

Specifying dependent JAR files or modules

Defining the Main Class for J2EE modules

Defining the Main Class for J2EE modules

You can use the manifest editor to specify the Main Class attribute in the manifest (MANIFEST.MF) for a module.

About this task

The Main Class attribute in the MANIFEST.MF file specifies the Java™ class of the application entry point. This attribute is typically used for application client modules. For application client modules, the main class attribute is also displayed on the Overview page of the Client Deployment Descriptor editor.

Procedure

1. In the Enterprise Explorer view of the Java EE perspective, right-click your project's MANIFEST.MF file, and select **Open With** > **Manifest Editor** from the pop-up menu. The MANIFEST.MF file is located in the following locations for the different module types:

Project type	Location of manifest file
EJB projects	ejbModule/META-INF/MANIFEST.MF
Application client projects	appClientModule/META-INF/MANIFEST.MF
Web projects	WebContent/META-INF/MANIFEST.MF
Connector projects	connectorModule/META-INF/MANIFEST.MF

   Tip: For application client modules, you can also click **Edit** in the Main Class section of the client deployment descriptor editor to launch the Manifest Editor.
2. In the Dependencies page of the editor, scroll to the Main Class section and choose from the following options:
   - Type the package name and file name for the main class.
   - Click **Browse** to use the Type Selection window to locate the main class.
   - Click **Create** to use the Create Java Class wizard to create a new main class.
   - Click **Remove** to remove the current definition for the main class. Clicking **Remove** does not delete the actual Java class. You can remove it manually in the Enterprise Explorer view.

Related concepts:

Application Client Deployment Descriptor editor

Manifest editor

Related tasks:

Specifying dependent JAR files or modules

Specifying dependent JAR files or modules

You can use the Java™ EE Module Dependencies page to specify JAR files or modules that are required by a module. The dependencies are defined in the MANIFEST.MF file for your module.

About this task

When you are specifying required JAR files or modules, you first specify the enterprise application (EAR) that your project is a part of. Typically, the project is referenced by one EAR project in the workspace. However, it is possible that you have multiple enterprise applications that contain a reference to the same module or utility JAR project. If so, then ensure that you give the JAR or module the same Uniform Resource Identifier (URI) in each application, so that the class path is valid for all applications.

It is also possible that your module is a standalone project and is not currently referenced by any enterprise application. In this case, since there is no enterprise application scope defined, you cannot use the Manifest editor to update the dependencies. To add the module to an enterprise application, see Adding modules to an enterprise application.

Procedure

1. In the Enterprise Explorer view of the Java EE perspective, right-click your project file, and select **Properties**. On the Properties page, select **Deployment Assembly**. If you have a deployment descriptor, you can open **Deployment Assembly** by clicking the Manage Utility Jars link on the bottom of the deployment descriptor page.
2. Click **OK**.

Results

Tip: If you need to compile against a server runtime JAR files during development, you do not need to add these JAR files as dependent JAR files. The workbench manages this by using the target server property for the project. The workbench adds the appropriate libraries to your project's build and class path based on the target server. For more information, see Specifying target servers for J2EE projects.

Related concepts:
Application deployment descriptor editor
Defining Java EE applications

Adding modules to an enterprise application

You can use the Application Deployment Descriptor editor to add existing projects in your workspace to your enterprise application as new modules.

About this task

For each module that you add to the enterprise application, a `<module>` element is defined in the deployment descriptor (application.xml).

Tip: There are shortcuts for adding modules or utility JAR files to an enterprise application. In the Enterprise Explorer view, drag a project and drop it into:

- The modules box on the **Overview** section of the deployment descriptor for the enterprise application.

Procedure

1. In the Enterprise Explorer view of the Java™ EE perspective, right-click the deployment descriptor for your enterprise application project and select **Open With** > **Deployment Descriptor Editor** to open the Application Deployment Descriptor editor.
2. In the modules box on the **Overview** section, click **Add** > **Module**.
3. Select the existing project in your workspace that you want to add as a module to the application.
4. Click **OK**. The selected module is added to the enterprise application.
5. If the target server specified for the module is different from the target server specified for the enterprise application, the Change Target Server dialog opens. Click **Yes** to update the target server for the module to be the same target server specified for the enterprise application.

What to do next

To remove a module from the enterprise application, select the module from the list and click **Remove**.

Exporting and importing binary modules

You can add binary modules into and remove binary modules from your workspace.

Before you begin

Binary modules are Java™ EE modules (for example, EJB, web, connector, application client, and utility jars) that are left in binary state, that is, in a single .jar, .war, or .rar file) within the EAR.

About this task

Using binary modules in your workspace allows for better performance, so it is possible for a developer to only load that active component he is working on in source mode (that is, as a project) while leaving all other modules in binary state. This reduces workspace size by requiring projects for only those modules being modified. It improves performance because no builders or validators execute on binary modules.
None of the deployment descriptor files, .java files, mapping files, or the MANIFEST.MF files can be modified; however everything can be read from them in their respective editors in read-only mode.

Procedure

1. To export a module as a binary, follow these steps:
   A. If the application includes EJBs, then you need to ensure that the EAR has been prepared for deployment before the export. Right-click the EJB project, and select **Java EE** > **Prepare for deployment**.
   B. Right-click your project, and select **Export** > **Shared Ear File**.
   C. On the Shared Ear Export page, click **Browse** to choose a destination folder for the EAR file.
   D. Select **Optimize for a specific server runtime** to optimize server performance, or clear **Optimize for a specific server runtime** if you do not want to optimize performance. Accept the default server runtime, or select a different runtime.**Note:** If you have selected WebSphere® Application Server as your runtime, the **Optimize for a specific server runtime** has no effect on your deployment, because this feature is not implemented for WebSphere Application Server.
   E. Select **Overwrite existing file** or clear **Overwrite existing file** if you do not want to overwrite existent files.
   F. Click **Finish**.
2. To import a module as a binary, follow these steps:
   A. Right-click your project, and select **Import** > **Shared Ear File**.
   B. On the Shared Ear Import page, click **Browse** to locate the EAR file that you want to import.
   C. Select the **Ear project** associated with the Ear file.
   D. Select the **Target runtime** for your imported project.
   E. Click **Finish**.

Related concepts:
Application deployment descriptor editor
Defining Java EE applications

Java EE deployment assembly

Deployment assembly property pages allow you to add flexible resource and dependency mapping to your applications.

To access the deployment assembly page, right-click your Java™ EE project, and select **Properties** > **Deployment Assembly**. For Java EE modules the page consists of two tabs:

- the Deployment Assembly tab
- the Manifest Entries tab

The Deployment Assembly tab

The Deployment Assembly table consists of 2 columns: the Deploy Path column and the Source column.

- The Deploy Path: The Deploy Path column represents the path where the reference will be located within the packaged archive. You can modify this location to customize how you want your packaged archive to be organized. However, removing default folder mappings or modifying their deploy paths should be done carefully, since issues with deployment might be encountered if the changes violate the Java EE specification requirements.
- The Source ColumnThe Source column represents the location of the resource relative to the project, the file system, or the workspace, depending on which type of dependency was added.

Adding dependencies

- **Archive**: Adding a new archive reference allows you to add a reference to an archive file that resides inside any project in the workspace. Adding this reference places the chosen archive in the specified runtime location.
- **External Archive**: Adding a new External Archive reference allows you to add a reference to an archive file that resides anywhere in the file system. Adding this reference places the chosen archive in the specified runtime location.
- **Variable**: Adding a new variable reference allows you to choose from the existing classpath variables that have been defined in the workspace preferences (which can be accessed through the **Window** > **Preferences** > **Java** > **Build Path** > **Classpath Variables** preference page), and add a reference to the archive file represented by the variable. Adding this reference places the chosen archive in the specified runtime location.
- **Classpath Container**: Adding a new classpath container reference allows you to choose from existing libraries and add a reference to the classpath container. Adding this reference places all archive files contained in the library into the specified runtime location. This feature should be used mainly to add user libraries to the classpaths.
- **Referenced Projects Classpath Entries**: This type of reference is rarely used, and not recommended, but allows configuring loose JAR files that are added to a project classpath directly by the Java Build Path property sheet. Once a JAR file is added to a Java module (EJB, WAR, Application Client, Connector), this type of reference can be added to the EAR project. Select the JAR file under the Java module to include in deployment.

- **Project**: Adding a new project reference allows you to bundle the chosen workspace project into an archive, and to place the archive in the specified runtime location.
- **Folder Mapping**: Adding a new folder mapping reference allows you to select any project folder and map the folder into a runtime location within the archive. By default, the added folder mappings' deploy paths are set to the root folder of the archive.

the Manifest Entries tab

Java EE classpath management

The Java EE development experience in your workspace simulates the runtime environment closely, reducing the possibility of unforeseen issues that may appear after you publish your application. Each module's MANIFEST file is managed capturing the runtime visibility by duplicating dependent JAR files or other projects on the project classpath. The MANIFEST tab of the deployment assembly captures the existing entries, and allows you to add additional entries within the scope of the parent EAR module. The list of entries available is limited to files of JAR type that reside within the deployed EAR module, and are not included in the designated EAR lib directory. In Java EE 5, the library folder was introduced to EAR modules as a simple technique for sharing JAR files used by other contained modules, but are no longer required to add entries in each respective MANIFEST file. The Deployment Assembly page for EAR projects includes a field for changing the default location of this folder. By default any jar in the `/lib` folder is shared, and your project classpath will include these JAR files automatically. In addition, the JAR file is not required to physically reside in this folder if a mapping is created from its development location to the EAR file's runtime `/lib` folder.
Web modules also have a special folder that allows the sharing of libraries under `WEB-INF/lib`. Similar to the EAR's library folder, any loose JAR file or workspace project mapped to this location will automatically appear on the Web project's classpath.

WebSphere Application Server Loose configuration

The WebSphere® test environment prepares your project for deployment as you are developing, and uses the flat project contents as they are, meanwhile mapping to a standard Java EE runtime structure that WebSphere Application Server can understand. This mapping in WebSphere Application Server is called loose configuration, and allows hot deployment of Java EE applications without the need for special packaging prior to publish. The new assembly capability allows you to use non-default project layouts, and artifact mappings that may require some packaging at publish time, and could impact publishing performance. In these cases, validation messages warn you of these potential issues. If no messages appear, then your application can run as-is, with no performance penalty.

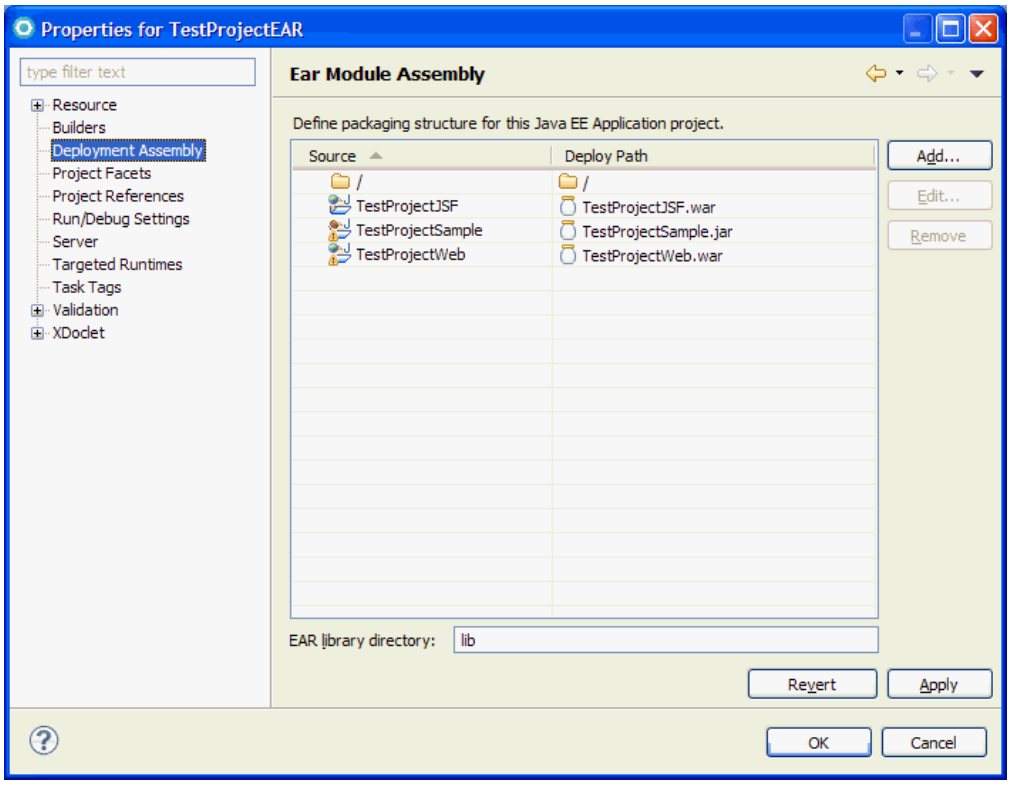
Related tasks:
Editing the Java EE deployment assembly page

Editing the Java EE deployment assembly page

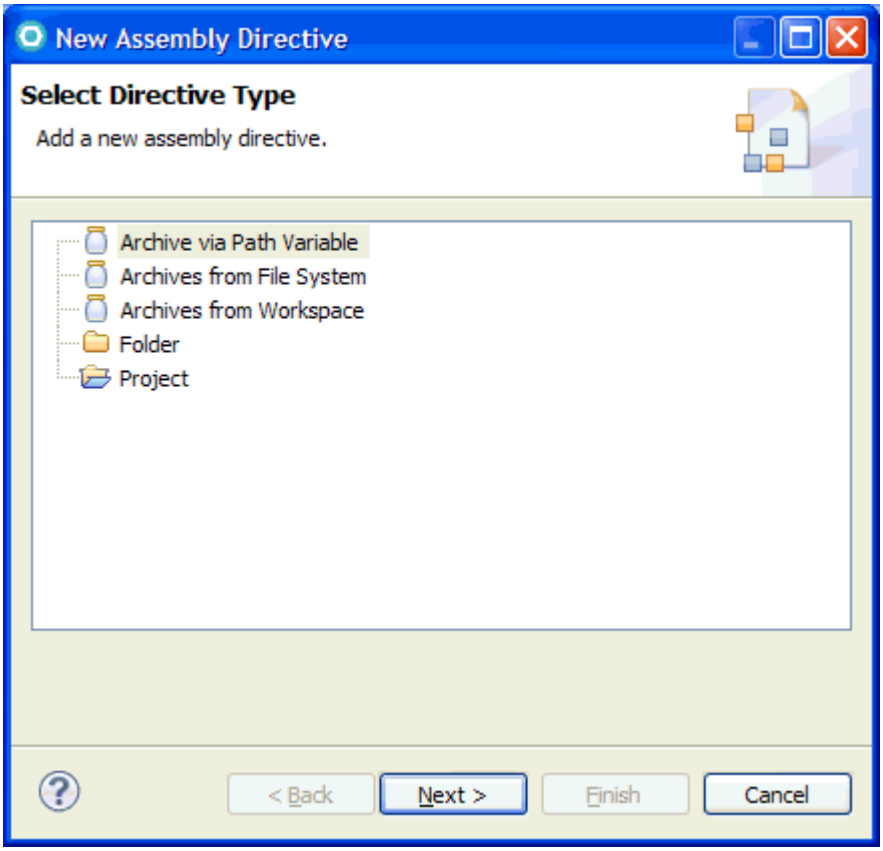
The Deployment Assembly project property page is shared among all Java™ EE project types, including Aries projects.

Procedure

1. Editing your EAR project deployment assembly page:
   A. To use the Deployment Assembly page, right-click your EAR project and select **Properties** > **Deployment Assembly**. The Deployment Assembly page consists of a table containing a Deploy Path column and a Source column:



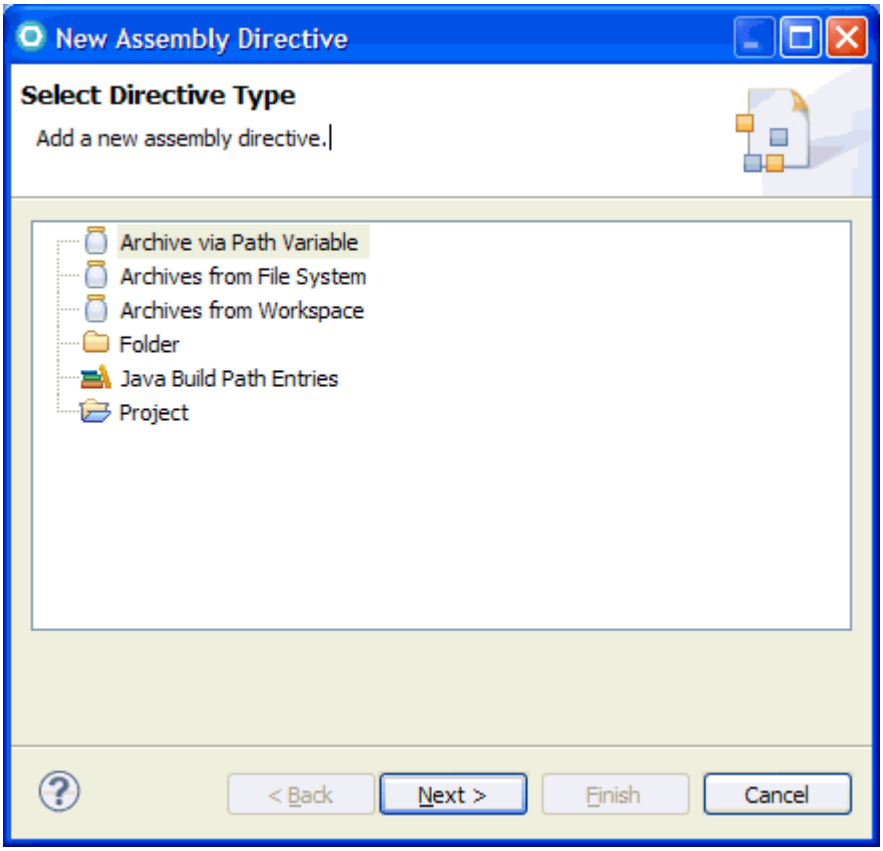
   B. The deploy path column represents the path within the packaged archive. Click **Add** to add mappings:



These are the possible references you can add:
- **Archive via path Variable**
- **Archives from File System**
- **Archives from Workspace**
- **Folder**
- **Project**: allows bundling of the project into an archive, and placed in the runtime location specified. Project references are often used to bundle utility projects into a web project (WEB-INF/lib) location, which is automatically included on the War's runtime classpath.

C. The Source column represents the project path. For information about Annotation scanning, see Excluding files from annotation scanning.

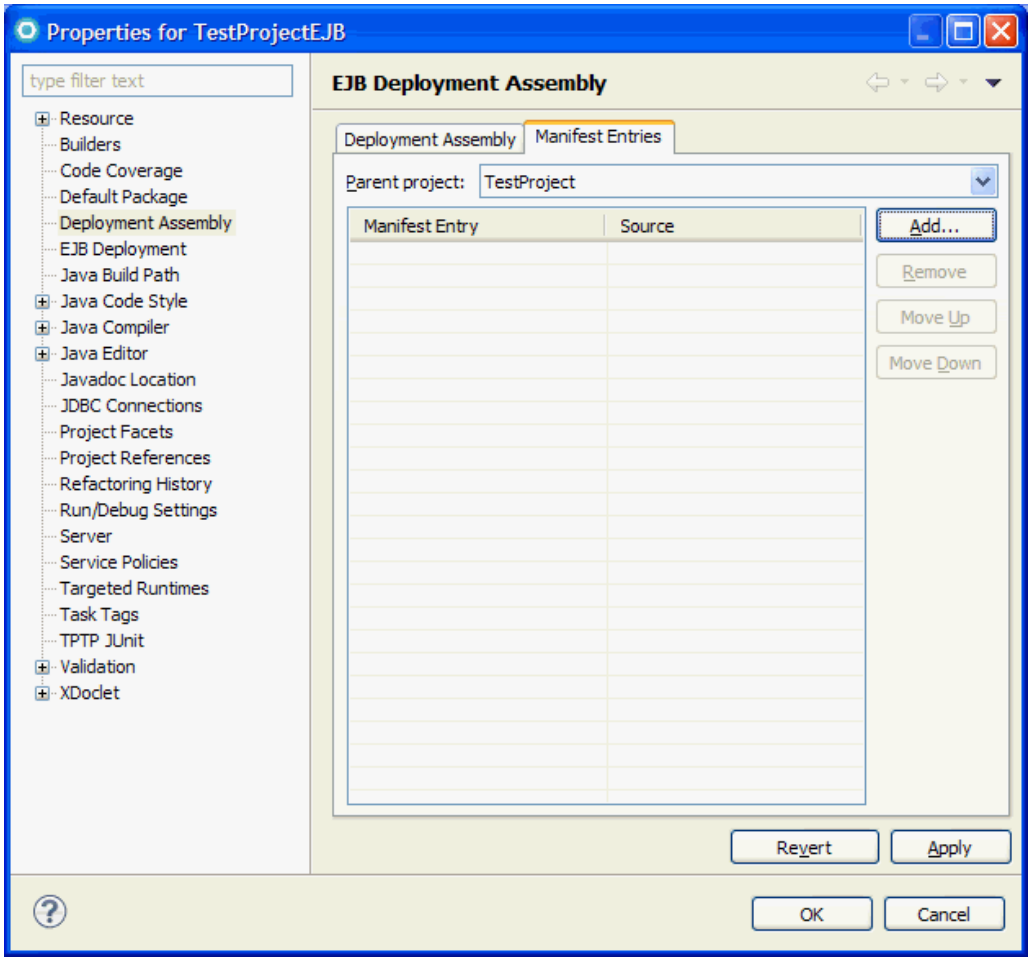
2. Editing your EJB or web project deployment assembly page:
   A. To use the Deployment Assembly page, right-click your Java EE project and select **Properties** > **Deployment Assembly**. For Java EE modules the page consists of two tabs:
      - The Deployment Assembly tab: the Deployment Assembly page consists of a table containing a Deploy Path column and a Source column.
      - The Manifest Entries tab
   B. On the Deployment Assembly tab, click **Add** to add mappings:



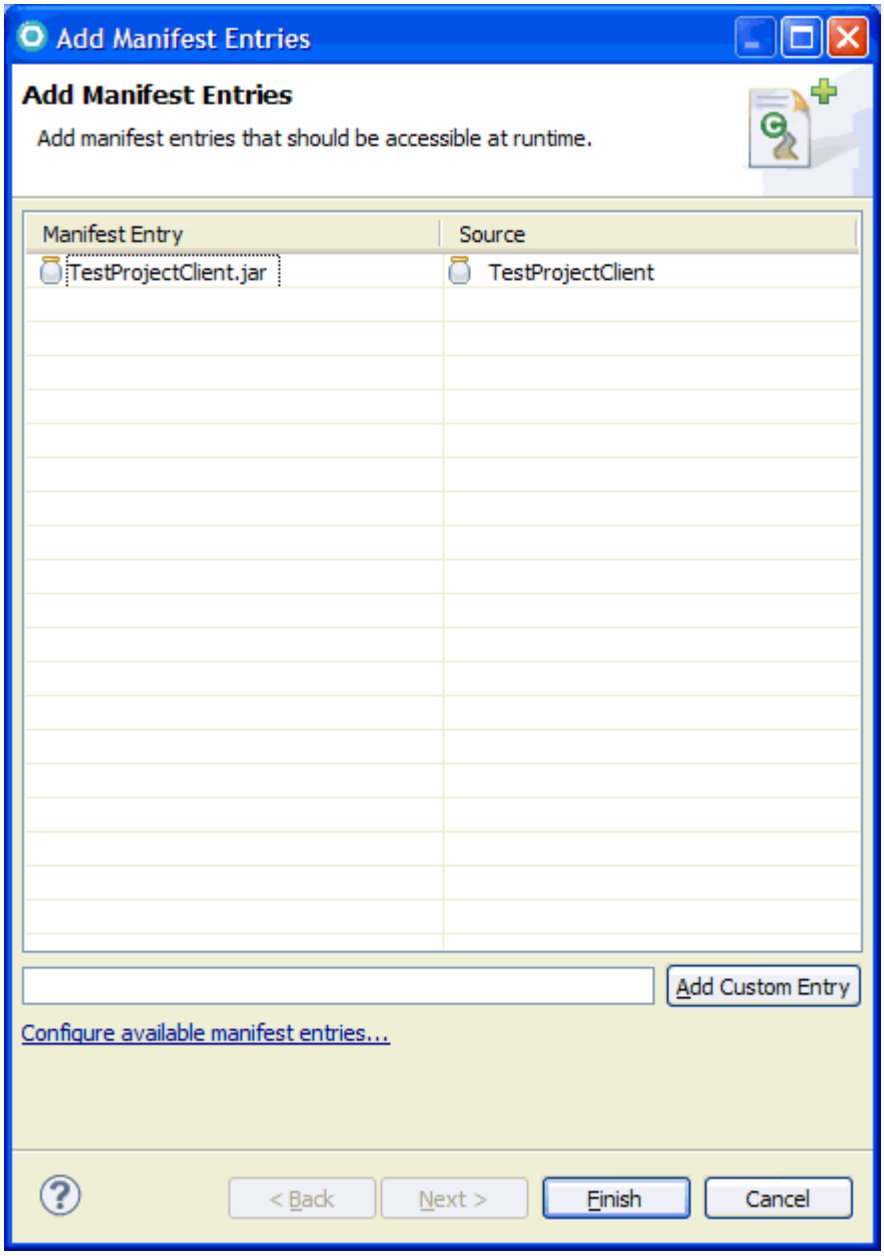
These are the possible references you can add:

- **Archive via path Variable**
- **Archives from File System**
- **Archives from Workspace**
- **Folder**
- **Java Build Path Entries**
- **Project**: allows bundling of the project into an archive, and placed in the runtime location specified. Project references are often used to bundle utility projects into a web project (WEB-INF/lib) location, which is automatically included on the War's runtime classpath.

3. On the Manifest Entry page, click add to add manifest entries:



Click **Add** to add manifest entries:



Related concepts:
Java EE deployment assembly

Securing enterprise applications

You can provide security for your Java™ EE enterprise application using annotations or using deployment descriptors.

Security is important in the Java EE environment, and is accomplished through authentication and authorization. Authentication verifies the identity of a given user, typically by requiring the user to enter a user name and password. In the Java EE environment, authentication is associated with a realm. The realm can store user identity information in many ways, including files, LDAP directories, and even databases accessed through JDBC. Authorization grants access control permissions based not only on what software is running but also on identity of the authenticated user who is running it. Each time a user logs in, he or she is granted a set of permissions for each application.

Before Java EE 5, if you wanted to use authorization for a given application, you needed to specify authorization information in the application deployment descriptors ejb-jar.xml or web.xml. One of the main focuses of Java EE is to simplify development of Java EE applications. Starting in Java EE 5, developers can specify annotations in Java source files instead of putting metadata in deployment descriptors. Annotations simplify the development of Java EE applications, shortening development cycles and reducing the total cost of ownership. You can secure your enterprise application using annotations, or, if you prefer, using deployment descriptions.

You can secure your enterprise application using annotations, or, if you prefer, using deployment descriptions. For a web module, you still need to specify a <security-constraint> in the web.xml application deployment descriptor in order to have authorization constraints, just as you did in J2EE 1.4. In the Java EE 5 environment, the permissions-related annotations are only defined for EJB modules. For EJB security, see Securing EJBs

Related concepts:

Developing EJB 3.1 applications
Differences between EJB 3.1 and EJB 2.1
EJB modules
Editing EJB 3.1 applications
Testing EJB 3.1 applications
Deploying EJB 3.1 applications
Creating enterprise beans
Developing Java EE Applications
Java EE: Overview
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications

Related tasks:

Creating EJB projects
Setting Java EE preferences

Using annotations to secure Java EE applications

You can provide security for your Java™ EE enterprise application directly in your source code using annotations.

About this task

Common security annotations: JSR 250 defines a number of common security annotations. Five security annotations are defined:

Procedure

1. **javax.annotation.security.PermitAll**:
   - Can be used at type or method level.
   - Indicates that the given method or all business methods of the given EJB are accessible by everyone.
2. **javax.annotation.security.DenyAll**:
   - Can be used at method level.
   - Indicates that the given method in the EJB cannot be accessed by anyone.
3. **javax.annotation.security.RolesAllowed**:
   - Can be used at type or method level.
   - Indicates that the given method or all business methods in the EJB can be accessed by users associated with the list of roles.
4. **javax.annotation.security.DeclareRoles**:
   - Can be used at type level.
   - Defines roles for security checking. To be used by EJBContext.isCallerInRole, HttpServletRequest.isUserInRole, and WebServiceContext.isUserInRole.
5. **javax.annotation.security.RunAs**:
   - Can be used at type level.
   - Specifies the run-as role for the given components.

Example

Using security annotations

- For the annotations @PermitAll, @DenyAll, and @RolesAllowed, a class-level annotation applies to a class and a method-level annotation applies to a method. Method-level annotation overrides the behavior of class level annotation. `@Stateless`

```
@RolesAllowed("team")
public class TestEJB implements Test {
 @PermitAll
 public String hello(String msg) {
  return "Hello, " + msg;
 }

 public String goodbye(String msg) {
  return "Goodbye, " + msg;
 }
}
```

In this example, the hello() method is accessible by everyone, and the goodbye()

method is accessible by users of role team.

- The @DeclareRoles annotation defines a list of roles to be used by a given component. In the Java EE 5 environment, you can look up the resource by using the @javax.annotation.Resource and verify whether the user is of the given role by invoking the following APIs: *Table 1.*

Component	API used to check role
EJB	`javax.ejb.EJBContext.isCallerInRole(role)`
Servlet	`javax.servlet.http.HttpServletRequest.isUserInRole(role)`
Web Service	`javax.xml.ws.WebServiceContext.isUserInRole(role)`

- The @PermitAll, @DenyAll, and @RolesAllowed annotations allow you to implement most of the authorization decision. However, to accomplish more complicated logic, use the @DeclareRoles annotation. For example, suppose the hello method is to be accessible by a user who is in role A and who is not in role B at the same time. The following code clip achieves this goal: `@Stateless`

```
@DeclaresRoles({"A", "B"})
public class TestEJB implements Test {
 @Resource private SessionContext sc;
 public String hello(String msg) {
  if (sc.isCallerInRole("A") && !sc.isCallerInRole("B")) {
   ...
  } else {
   ...
  }
 }
}
```

Invalid use of security annotations

- More than one of @DenyAll, @PermitAll, @RolesAllowed cannot apply to the same method. For example, the following usage is not valid: `@PermitAll`

```
@DenyAll
public String test()
```

- Two @RolesAllowed annotations cannot apply to the same methods. For example, the following usage is not valid and fails in compilation: `@RolesAllowed("team")`

```
@RolesAllowed("otherteam")
public String hello()
```

Instead, use this structure: `@RolesAllowed({"team", "otherteam"})`

```
public String hello()
```

Related concepts:

Securing enterprise applications

Using deployment descriptors to secure enterprise applications version 1.4

You can provide security for your Java™ EE enterprise application version 1.4 projects using deployment descriptors.

Related tasks:
Defining security roles for enterprise applications using deployment descriptors

Defining security roles for enterprise applications (J2EE 1.4)

You can use the application deployment descriptor editor to define security roles for your enterprise applications.

About this task

A security role is a logical grouping of principals. Access to operations (such as EJB methods) is controlled by granting access to a role. You can grant access to users individually or in groups.
For each security role that you define in the deployment descriptor editor, a `<security-role>` element is added to the application.xml file.

Procedure

1. In the Enterprise Explorer view of the Java™ EE perspective, right-click the deployment descriptor for your enterprise application project and select **Open With** > **Deployment Descriptor Editor** to open the application deployment descriptor editor.
2. On the Security page of the editor, click **Add**.
3. Type a name and description for the new role and click **Finish**.

What to do next

To remove a security role, select the role and click **Remove**.

- **Gathering security roles (J2EE 1.4)**
  For an enterprise application, you can roll up all the security roles that are defined in the application's modules. You can then combine and remove redundant or unnecessary security roles. You can use the deployment descriptor to gather security roles.
- **Replacing security roles (J2EE 1.4)**
  You can use the application deployment descriptor editor to replace redundant or unnecessary security roles with preferred roles.
- **Adding users to security role bindings**
  You can use the application deployment descriptor editor to add users to security role bindings to your EJB module.
- **Adding groups to security role bindings**
  You can add groups to security role bindings
- **Adding security role run as bindings**
  You can specify a user ID and password that are required to access a bean from another bean. This additional security level is a WebSphere® Application Server binding.

Related concepts:

Application Deployment Descriptor editor

Related tasks:

Gathering security roles (J2EE 1.4)
Replacing security roles (J2EE 1.4)

Gathering security roles (J2EE 1.4)

For an enterprise application, you can roll up all the security roles that are defined in the application's modules. You can then combine and remove redundant or unnecessary security roles. You can use the deployment descriptor to gather security roles.

About this task

A security role is a logical grouping of principals. Access to operations (such as EJB methods) is controlled by granting access to a role. The **Gather** option combines all security roles defined in modules that are included in the application.

Procedure

1. In the Enterprise Explorer view of the Java™ EE perspective, right-click the deployment descriptor for your enterprise application project and select **Open With** > **Deployment Descriptor Editor** to open the application deployment descriptor editor.
2. On the Security page of the editor, click **Gather**. Any security roles that are defined in the enterprise application's modules are added to the enterprise application. The security roles are not removed from the modules.

Related concepts:

Application Deployment Descriptor editor

Related tasks:

Defining security roles for enterprise applications (J2EE 1.4)

Replacing security roles (J2EE 1.4)

Replacing security roles (J2EE 1.4)

You can use the application deployment descriptor editor to replace redundant or unnecessary security roles with preferred roles.

About this task

In some application development situations, you might encounter redundant roles that are serving the same purpose throughout the enterprise application or in the modules. In this case, you can use a wizard to replace the redundant roles throughout the enterprise application and modules with the security roles that you want to keep.For example, you have a security role Boss that is defined in the enterprise application. The Boss security role is serving the same purpose as another security role called Manager that is gathered from the EJB module. To remove the redundancy, you can replace all usages of the role Manager with the role Boss. After the Manager role is replaced by Boss, the Manager role is removed from the application and is deleted. The Manager role is also removed from the EJB module where it was gathered from and replaced with the Boss role in the module.

Procedure

1. In the Enterprise Explorer view of the Java™ EE perspective, right-click the deployment descriptor for your enterprise application project and select **Open With** > **Deployment Descriptor Editor** to open the application deployment descriptor editor.
2. On the Security page of the editor, click **Replace**.
3. Select the security roles that you want to keep, then click **Next**.
4. For the security roles that you did not select to keep, define the replacement scheme by completing the following steps:
   A. In the **Security roles to be replaced** table, select a security role.
   B. In the **Replacement scheme** table on the left, select the replacement security role that you want to use to replace the security role or roles that you selected in the right table.
   C. Click the left arrow (**<**) button to move the security roles to be replaced into the **Replacement scheme** table.
   Note: In the **Replacement scheme** table, security roles that appear as children in the node tree is replaced by the security role that appears as their parent.
5. When you finish defining the replacement scheme for all the security roles, click **Finish**.

Results

The wizard replaces the security roles throughout the enterprise application and modules based on the replacement scheme that you defined.

Related concepts:

Application Deployment Descriptor editor

Related tasks:

Defining security roles for enterprise applications (J2EE 1.4)

Gathering security roles (J2EE 1.4)

Adding users to security role bindings

You can use the application deployment descriptor editor to add users to security role bindings to your EJB module.

Procedure

1. Switch to the Java™ EE perspective.
2. In the Enterprise Explorer view, select the desired EJB module.
3. Right click the deployment descriptor, and select **Open With** > **Deployment Descriptor Editor**.
4. On the **Security** page of the editor, scroll to the WebSphere® Bindings section.
5. Select a security role to be bound to a user.
6. Select the **Users/Groups** check box.
7. In the **Users** section, click **Add**. The **Add Users** wizard appears.
8. Type a user name.
9. Click **Finish**.

Results

For additional information about security roles, see the WebSphere Application Server documentation.

Adding groups to security role bindings

You can add groups to security role bindings

Procedure

1. Switch to the Java™ EE perspective.
2. In the Enterprise Explorer view, select the desired EJB module.
3. Right click the deployment descriptor, and select **Open With** > **Deployment Descriptor Editor**.
4. Select **Open With** > **Deployment Descriptor Editor** from the pop-up menu.
5. On the **Security** page of the editor, scroll to the WebSphere® Bindings section.
6. Select a security role to be bound to a group.
7. Select the **Users/Groups** check box.
8. In the **Groups** section, click **Add**. The Add Group wizard appears.
9. Type a name for the new group.
10. Click **Finish**.

Adding security role run as bindings

You can specify a user ID and password that are required to access a bean from another bean. This additional security level is a WebSphere® Application Server binding.

Before you begin

Before you can add a security role "run as" binding, you must first create a security role on the **Assembly Descriptor** page of the EJB deployment descriptor editor for an EJB module. Then you need to add a security identity on the **Access** page of the EJB deployment descriptor editor. This security identity must be set to use the identity of a security role. Then, in the application deployment descriptor editor, gather up the security roles and select the role that you created for your enterprise beans. This enables the **Security Role Run as Bindings** section in the application deployment descriptor editor.

About this task

This setting is a WebSphere Application Server binding for an enterprise application that allows you to specify a user ID and password that are required in order to execute an enterprise bean.

Procedure

1. Switch to the Java™ EE perspective.
2. In the Enterprise Explorer view, select the desired EJB module.
3. Right click the deployment descriptor, and select **Open With** > **Deployment Descriptor Editor**.
4. On the **Security** page of the editor, select a security role that was gathered from an EJB module. The gathered security role must also be used as the identity of a security identity that you also specified in for the EJB module. When you select a valid security role, the **Security Role Run As Bindings** section is enabled.
5. Click the **Add** button next to the **Security Role Run As Bindings** section. The Add Security Role Run As Binding wizard opens.
6. Type a **User ID** and **Password** for the security role.
7. Click **Finish**. The "run as" binding for the selected security role is added.

Results

For additional information about security roles, see the WebSphere Application Server documentation.

Validating code in enterprise applications

The workbench includes validators that check certain files in your enterprise application module projects for errors.

About this task

By default, the workbench validates your files automatically after any build, including automatic builds. You can also begin the validation process manually without building.

On the workbench Preferences window, you can enable or disable validators to be used on your projects. Also, you can enable or disable validators for each enterprise application module project individually on the Properties page for that project.

Each validator can apply to certain types of files, certain project natures, and certain project facets. When a validator applies to a project facet or nature, the workbench uses that validator only on projects that have that facet or nature. Likewise, most validators apply only to certain types of files, so the workbench uses those validators only on those types of files.

Procedure

1. Click **Window** > **Preferences** and select **Validation** in the left pane. The Validation page of the Preferences window lists the validators available in your project and their settings.
2. Optional: Review the check box options available near the top of the window to customize your validation settings:

Option	Description
Allow projects to override these preference settings	Select to set individual validation settings for one or more of your projects.
Suspend all validators	Select to prevent validation at the global level.
Save all modified resources automatically prior to validating	Select to save resources you have modified before the validation begins.
Show a confirmation dialog when performing manual validation	Select to show an informational dialog after a manual validation request has completed.

3. In the list of validators, select the check boxes next to each validator you want to use at the global level. Each validator has a check box to specify whether it is used on manual validation and/or on build validation.
4. Tune a validator by clicking the button in the **Settings** column. Not all validators have additional settings.
5. Begin the validation process by one of the following methods:
   - Right-click a project and click **Validate**.
   - Start a build.

Results

Any errors found by the validators are listed in the Problems view.If you want to set individual validation settings for one or more of your projects, see Overriding global validation preferences for more information.

What to do next

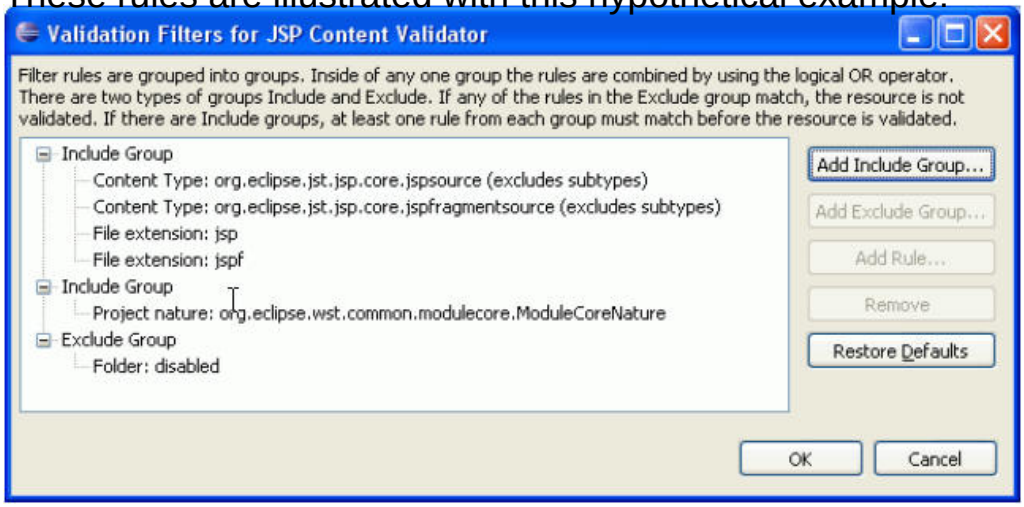
Tuning validators

Whether or not a validator validates a particular resource depends on the filters that are in place for that validator.
When a validator is first developed, the implementer of the validator defines a default set of filters. These filters may be based on:
- file extensions
- folder or file names
- project natures
- project facets
- content types

Through the Validation Filters dialog, you are able to further tune these settings. Normally you would simply keep the defaults; however, two reasons why you may want to tune validation are:
- Performance: if you have a very large workspace, you could reduce the amount of validation.
- Non standard conventions: if you use a non standard naming convention (for example stores XML in files with an .acme-xml extension), you could still enable the appropriate validators to run against those files.

You can access this dialog by clicking **Window** > **Preferences** > **Validation** and then clicking **Settings** beside each validator.
Filters are stored in groups. There are two types of groups: Include groups and Exclude groups. You can have as many Include groups as you like. Filters inside of an Include group cause resources to be validated. If any rule matches then the entire group matches. Inside of a group the filter rules are OR'd together. However individual Include groups are AND'ed together. You can have one Exclude group. If any of its filter rules match, then the resource is excluded. Exclusion takes precedence over inclusion.

These rules are illustrated with this hypothetical example:



- If the resource is in the disabled folder, it will be excluded because exclusion takes precedence over everything else.
- If the resource does not have the JSP source content type, and it does not have the JSP fragment source content type, and it does not have a file extension of .jsp or .jspf then it will be excluded because none of the rules in the first group matched.

- If the project does not have the module core nature then it will be excluded because the single rule in the second group did not match.
- Otherwise the resource will be validated by this particular validator.

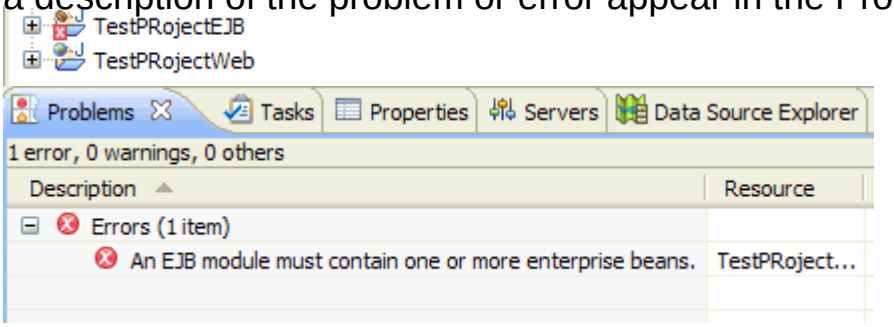
To add a rule to a group, select the group on the left, and click **Add Rule**.

Validating Java EE projects

The workbench includes validators that check certain files in your Java™ EE enterprise application projects for errors.

Procedure

After you create a Java EE project, the Java EE validators check the code. A red error mark appears in the Enterprise Explorer view beside the problem module, and a description of the problem or error appear in the Problems view:



Validating for Java EE projects created in version 6 of the product

The Java™ EE tools provide a validation to detect projects from version 6 of the product

The version 6 validator checks to see if any V6 projects are found in the workspace and alerts you that these project are not supported and must be migrated.

The validator also checks to see if any obsolete V6 artifacts (.j2ee, .websettings, natures) still exist in the V7.x projects and offers a quick fix to remove this obsolete data

Java EE validators

This table lists the Java™ EE validators that are available for the different project types and gives a brief description of each validator.

Validator name	Description
Application client 5.0 and 6.0 Validator	The application client 5.0 and 6.0 validator validates the following application client project resources: Main-Class: Main: If no main class is defined, this error appears in the Problems view: `The Main-Class attribute must be defined in the application client module.` EJB Reference: `EJB Reference name cannot be empty.` Service Reference: `Service Reference name cannot be empty.` Message Destination Reference: `Message Destination Reference name cannot be empty.` Resource Reference: `Resource Reference name cannot be empty.` Resource environment reference: `Resource environment reference name cannot be empty.`

EJB 3.0 and 3.1 validator	The EJB 3.0 and 3.1 validator verifies that enterprise beans contained in an EJB project comply with the Sun Enterprise JavaBeans Specifications (3.0 and 3.1), depending on the level of the bean. Specifically, the EJB validator validates the following resources: EJB project contains at least one bean (created in a deployment descriptor or using annotations).Deployment descriptor validation only:Session bean ejb-class is specified and exists.Message bean ejb-class is specified and exists.The following classes specified for each session bean in the deployment descriptor exist:Business local interfaceBusiness remote interfaceHome interfaceLocal interfaceLocal Home interface Remote interface Service endpoint All interfaces and classes listed in references exist:EJB references (ejb-ref and ejb-local-ref) LocalLocal HomeHome Remote Injection classesresource references Injection classes message destination references Injection classes Service references Service interface Injection classes Duplicate references do not exist.Security role is not empty.Duplicate Security Roles do not exist.
EAR 5.0 and 6.0 validator	The EAR 5.0 and 6.0 validator validates the following: Each module, including utility jars, are backed by a physical resource.The Context root of each Web module is unique.Module URIs: Web URIs end with .war, connector URIs end in .rar, other URIs end in .jar (Applies for Deployment Descriptor case only).Duplicate URIs do not exist. Two modules with web1.war cannot exist. customer.war and customer.jar are allowed.Security roles cannot be empty.Duplicate security roles do not exist.The EAR validator only ensures the validity and dependency of the module projects with respect to the enterprise application project.

Web 2.5 and 3.0 validator	The Web 2.5 and 3.0 validator validates the following items relating to the deployment descriptor:Servlet class exists.Duplicate servlets do not exist.Servlet mapping is not duplicated.All interfaces and classes listed in references exist.Duplicate references do not exist.Security role is not empty.Duplicate Security Roles do not exist.

Manually validating code

When you run a manual validation, all resources in the selected project are validated according to the validation settings.

About this task

The validators used depend on the global and project validation settings. When you validate a project manually, the global settings are used unless both of the following are true:

- The **Allow projects to override these preference settings** check box is selected on the global validation preferences page.
- The **Enable project specific settings** check box is selected on the project's validation preferences page.

Whether the workbench uses the global or project validation preferences, only the validators selected to run on manual validation are used when you run a manual validation.

Procedure

1. Select the project that you want to validate.
2. Right-click the project and then click **Validate**. If this option is not available, validation is disabled or there are no validators enabled for the project. To enable validation at the global level, see Validating code in enterprise applications. To enable validators for this project, see Overriding global validation preferences.

Results

The workbench validates the project using the enabled validators. Any errors found by the validators are listed in the Problems view.**Note:** The errors listed in the validation results dialog do not necessarily correspond one for one with the errors listed in the problems view. The validation results display all of the errors found, whereas duplicate errors appear only once in the Problems view.

What to do next

Common validation errors and solutions

This table lists the common error messages you may encounter when you validate your projects.

Message prefix	Message	Explanation
Application Client validator		
CHKJ1000	Validation failed because the application client file is not valid. Ensure that the deployment descriptor is valid.	The application-client.xml file cannot be loaded. The project metadata cannot be initialized from the application-client.xml file. Ensure the following: that the META-INF folder exists in the application client projectthat META-INF contains the application-client.xml filethat META-INF is in the project's classpath.Validate the syntax of the application-client.xml file: in the Navigator view, highlight the application-client.xml file, right-click, and select **Validate XML file**.If both 1) and 2) are okay, close the project, reopen the project, and rebuild the project. The project metadata will refresh.
EAR validator		

CHKJ1001	The EAR project {0} is invalid.	The application.xml file cannot be loaded. The project metadata cannot be initialized from the application.xml file. Ensure the following: that the META-INF folder exists in the EAR projectthat META-INF contains `application.xml`that META-INF is in the project's classpath.Validate the syntax of the application.xml file: in the Navigator view, highlight the application.xml file, right-click, and select **Validate XML file**.If both 1) and 2) are okay, close the project, reopen the project, and rebuild the project. The project metadata will refresh.
EJB validator		
CHKJ2019	The {0} key class must be serializable at runtime.	The EJB is compliant with the EJB specification. This message is a warning that problems may occur. The warning appears when a type needs to be serializable at runtime and when serializability cannot be verified at compile-time. A type is serializable if, at runtime, it is a primitive type, a primitive array, a remote object, or if it implements java.io.Serializable. This message flags java.lang.Object and it cannot be disabled. You can either make the object serializable at compile-time or ignore the warning.
CHKJ2412	The return type must be serializable at runtime.	
CHKJ2413	Argument {1} of {0} must be serializable at runtime.	
CHKJ2102	Either a finder descriptor, or a matching custom finder method on the {0} class, must be defined.	A finder descriptor must exist for every finder method.

CHKJ2873	Migrate this bean's datasource binding to a CMP Connection Factory binding.	
CHKJ2874	Migrate this EJB module's default datasource binding to a default CMP Connection Factory binding.	
CHKJ2875E	<ejb-client-jar> {0} must exist in every EAR file that contains this EJB module.	If `<ejb-client-jar>` is specified in `ejb-jar.xml`, a corresponding EJB client project must contain the home and remote interfaces and any other types that a client will need. If these types are all contained in a single EJB project, delete the `<ejb-client-jar>` line in the deployment descriptor. Otherwise, ensure that the EJB client project exists, is open, and is a project utility JAR in every EAR that uses this EJB project as a module.
CHKJ2905	The EJB validator did not run because ejb-jar.xml could not be loaded. Run the XML validator for more information.	CHKJ2905 means that the project's metadata could not be initialized from ejb-jar.xml. Ensure the following: that the META-INF folder exists in the EJB projectthat META-INF contains ejb-jar.xmlthat META-INF is in the project's classpath.Validate the syntax of the ejb-jar.xml file: in the Navigator view, highlight the ejb-jar.xml file, right-click, and select **Validate XML file**.If both 1) and 2) are okay, close the project, reopen the project, and rebuild the project. The project metadata will refresh.
JSP validator		

IWAW0482	No valid JspTranslator	There is a path problem with the project; the JSP Validator needs access to the WAS runtime code. If IWAW0482E appears on all web projects, check the Variable or JRE path: Check the global preferences (**Window > Preferences > Java >Installed JREs**) and make sure that the location for the JRE is pointing to a valid JRE directory. Ensure that the classpath variables (**Window > Preferences > Java > Classpath Variables**) are set correctly.
WAR validator		
CHKJ3008	Missing or invalid WAR file.	The web.xml file cannot be loaded. The project metadata cannot be initialized from the web.xml file. Ensure the following: that the WEB-INF folder exists in the web projectthat WEB-INF contains the web.xml filethat WEB-INF is in the project's classpath.Validate the syntax of the web.xml file: in the Navigator view, highlight the web.xml file, right-click, and select **Validate XML file**.If both 1) and 2) are okay, close the project, reopen the project, and rebuild the project. The project metadata will refresh.
XML validator		

	The content of element type "ejb-jar" is incomplete, it must match "(description?,display-name?,small-icon?,large-icon?,enterprise-beans,assembly-descriptor?,ejb-client-jar?)".	The EJB 1.1 and 2.0 specifications mandate that at least one enterprise bean must exist in an EJB .jar file. This error message is normal during development of EJB .jar files and can be ignored until you perform a production action, such as exporting or deploying code. Define at least one enterprise bean in the project.

Disabling validation

You can disable one or more validators individually or disable validation entirely. Also, you can set validation settings for your entire workspace and for individual projects.

About this task

To disable validators in your project or workspace, complete the following steps:

Procedure

1. Click **Window** > **Preferences** and select **Validation** in the left pane. The Validation page of the Preferences window lists the validators available in your project and their settings.
2. To disable individual validators, clear the check boxes next to each validator that you want to disable. Each validator has a check box to specify whether it is enabled for manual validation or on a build.
3. Optional: You can also change the following check box options on this page:

Option	Description
Allow projects to override these preference settings	Select to set individual validation settings for one or more of your projects.
Suspend all validators	Select to prevent validation at the global level. If you select this check box, you can still enable validation at the project level.

4. Click **OK**.

Results

If you want to set individual validation settings for one or more of your projects, see Overriding global validation preferences

What to do next

Selecting code validators

You can select specific validators to run during manual and build code validation. You can set each validator to run on manual validation, build validation, both, or neither.

About this task

To choose the validators that you want to use for a project, complete the following steps:

Procedure

1. Click **Window** > **Preferences** and select **Validation** in the left pane. The Validation page of the Preferences window lists the validators available in your project and their settings.
2. Clear the **Suspend all validators** check box.
3. Optional: If you want to set individual validation settings for one or more of your projects, select the **Allow projects to override these preference settings** check box.
4. In the list of validators, select the check boxes next to each validator you want to use at the global level. Each validator has a check box to specify whether it is used on manual validation or on a build.**Note:** If you deselect any validator that is currently selected, any messages associated with the deselected validator will be removed from the task list, the next time you perform a full build.
5. Tune a validator by clicking the button in the **Settings** column. Not all validators have additional settings.

Results

If you want to set individual validation settings for one or more of your projects, see Overriding global validation preferences. Like the global validation preferences, you can set validators at the project level to run on manual validation, build validation, both, or neither.

What to do next

Overriding global validation preferences

For a given project, you can override the global validation preferences.

About this task

The default validation preferences are specified globally on the Validation page of the Preferences dialog.

Procedure

1. Click **Window** > **Preferences** and select **Validation** in the left pane. The Validation page of the Preferences window lists the validators available in your project and their settings.
2. Select the **Allow projects to override these preference settings** check box and click **OK**. Now individual projects can override the global preferences.
3. Right-click a project and then click **Properties**.
4. In the left pane of the Properties window, click **Validation**.
5. Select the **Override validation preferences** check box. This check box is available only if you checked the **Allow projects to override these preference settings** check box on the workbench validation preferences page.
6. To prevent validation for this project, select the **Suspend all validators** check box.
7. In the list of validators, select the check boxes next to each validator you want to use. Each validator has a check box to specify whether it is used on manual validation or on a build.
8. Choose an alternate implementation for a validator by clicking the button in the **Settings** column, then click **OK**. Not all validators have alternate implementations.

What to do next

Deploying Java EE applications

You can deploy your Java™ EE application using WebSphere® Application Server v7, V8, V8.5 orV6.1 with the Feature Pack for Enterprise Java beans 3.0, with a profile created for your selected server.

About this task

When deploying on an EAR file, all projects defined in the EAR file are deployed.

Procedure

You can deploy your Java EE application using WebSphere Application Server v7, V8, V8.5 orV6.1 with the Feature Pack for Enterprise Java beans 3.0, with a profile created for your selected server.

Related concepts:
Developing Java EE Applications
Java EE: Overview
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications
Securing enterprise applications

Migrating the specification level of Java EE projects

The Java™ EE upgrade wizard supports the migration from J2EE 1.2, 1.3 and 1.4 specifications to the Java EE 5.0 specification level for all Java EE module types. The wizard also supports migration from Java EE 5.0 to Java EE 6.0 specification level for all Java EE module types.

About this task

Refer to J2EE 1.3 to 1.4 specification level upgrade and J2EE 1.2 to 1.4 specification level migration for details on the migration of J2EE 1.3 and 1.2 specification level artifacts to J2EE 1.4 specification level. Using the Java EE specifications upgrade wizard, you can upgrade any Java EE 5 and pre-Java EE 5 module to Java EE 6. See the following table for more module version details:

Table 1. Java EE version numbers

Module type	J2EE	Java EE 5	Java EE 6
Web	2.2	2.5	3.0
EJB	1.1You can migrate EJB 1.1 to EJB 3 or EJB 3.1, providing that there are no CMP or BMP beans (EJB 1. x and 2. x projects with CMP or BMP beans can be left as is in a Java EE EAR).	3.0	3.1
JCA	1.0	1.5	1.6
EAR	1.2	5.0	6.0
Application Client	1.2	5.0	6.0

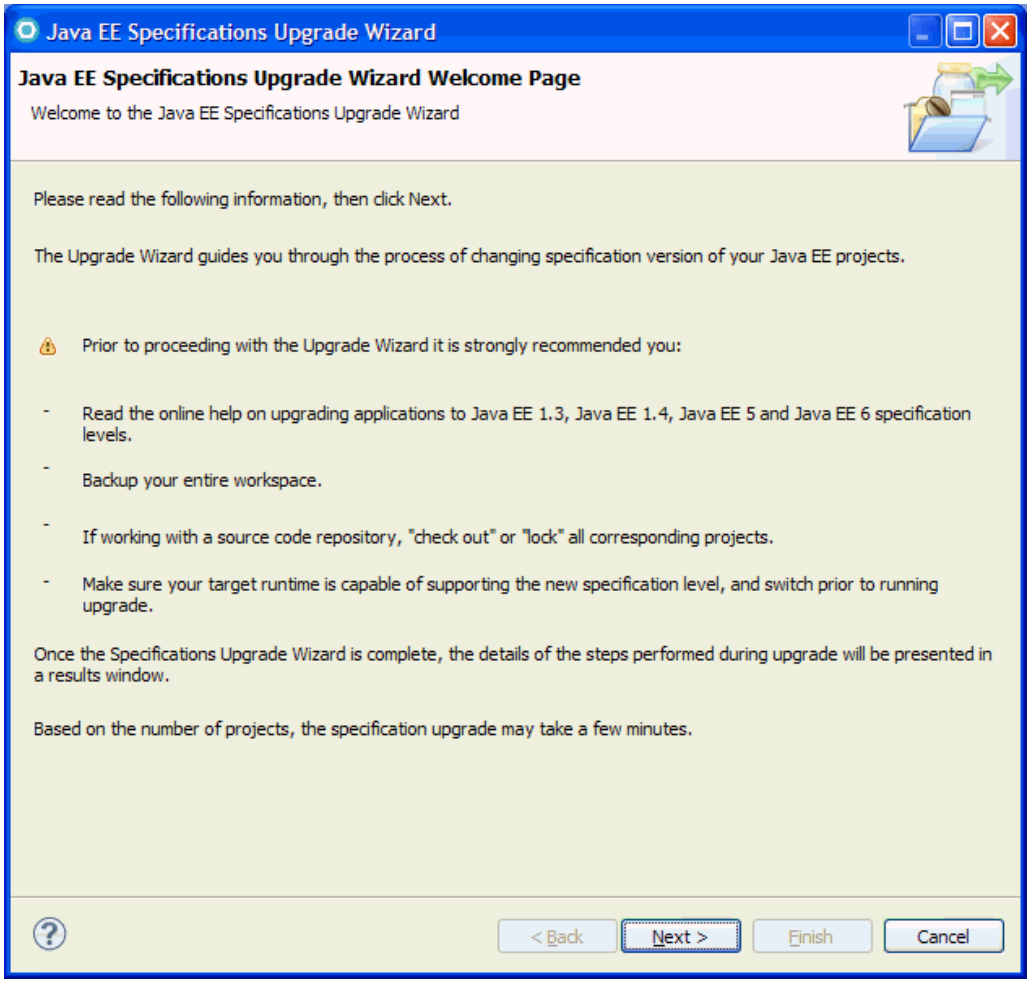
For more information, see EJB 3.0 module considerations

Note: Before using the Java EE upgrade wizard, it is recommended that you take the following steps:

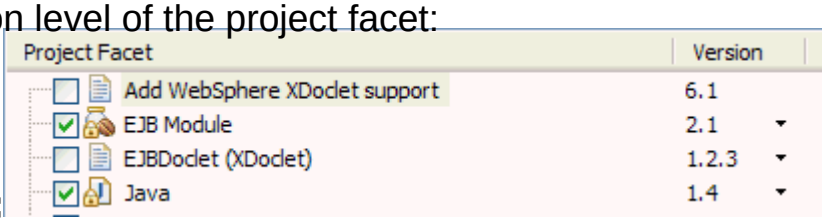
- Back up your entire workspace.
- If you are working with a shared repository, check out or lock all corresponding projects.

Procedure

1. In the Enterprise Explorer view of the Java EE perspective, right-click the enterprise application project that you want to upgrade.
2. Right-click the project that you want to upgrade, and select **Java EE** > **Specifications Upgrade wizard**.
3. Follow the instructions on the Java EE Specifications Upgrade wizard Welcome Page before proceeding with the migration process, and click **Next**.

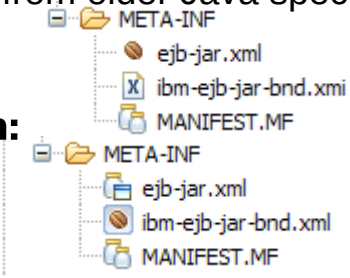


4. In the **J2EE version** field, select the Java EE version level you want to upgrade to, and click **Next**.
5. Select the modules you want to upgrade, and click **Finish**. **Modifications made by the migration process:**
   - Increases the version level of the project facet:



   - **Before migration:**
   - **After migration:**

- Converts *.xmi* files from older Java specification levels into .xml files:
  - **Before migration:**
  - **After migration:**



Modifications not made by the migration process:
- Migration does not create new EJB 3.0 or EJB 3.1 business interfaces.
- Migration does not convert deployment descriptor values into Java EE annotations within your Java code.

Related concepts:
Developing Java EE Applications
Java EE: Overview
Tools for Java EE development
Project facets
Creating and configuring Java EE modules using annotations
Defining Java EE applications
Securing enterprise applications

Related reference:
J2EE 1.3 to 1.4 specification level upgrade
J2EE 1.2 to 1.4 specification level migration

Changes in the J2EE Migration Wizard in version 7.5

Changes have been made to the J2EE Migration Wizard in version 7.5 that are common to the migration of all Java™ EE specification levels.

Project structure migration option is removed (version 6.0.x to version 7.5 difference)

The check box **Migrate project structure** in the Java EE Migration wizard in version 6.0.x is removed in version 7.5. This feature in earlier versions of the wizard allowed you to choose to migrate the folder structure of J2EE projects created in earlier versions of the product to the required project structure for that version of the product. The option to Migrate project structure is no longer needed because in version 7.5 the folder structure for Java EE projects is flexible; you can have multiple source folders and give them any name. Regardless of what folder structure you use in your Java EE projects, when you deploy your projects, the product creates the correct deployment artifacts in the correct format.

J2EE 1.3 to 1.4 specification level upgrade

Java™ EE projects can be upgraded from the J2EE 1.3 to J2EE 1.4 specification level. This section describes, for each type of Java EE project, the artifacts upgraded from J2EE 1.3 to J2EE 1.4 by the Java EE upgrade Wizard.

Related tasks:
Migrating the specification level of Java EE projects
Migrating secure web services
Configuring an EJB 2.1 message-driven bean to use a JCA adapter

Migrating secure web services

Secure web services are not migrated by the J2EE Migration wizard when web services are migrated from J2EE 1.3 to J2EE 1.4. The migration of secure web services requires manual steps.

About this task

After the J2EE migration, the secure binding and extension files must be migrated manually to J2EE 1.4 as follows:

Procedure

1. Double click the webservices.xml file to open the Web Services editor.
2. Select the **Binding Configurations** tab to edit the binding file.
3. Add all the necessary binding configurations under the new sections **Request Consumer Binding Configuration Details** and **Response Generator Binding Configuration Details**.
4. Select the **Extension** tab to edit the extension file.
5. Add all the necessary extension configurations under the new sections **Request Consumer Service Configuration Details** and **Response Generator Service Configuration Details**.
6. Save and exit the editor.

Related reference:
J2EE 1.3 to 1.4 specification level upgrade
Enterprise Java bean projects (EJB 2.0 to EJB 2.1)

Enterprise Java bean projects (EJB 2.0 to EJB 2.1)

The J2EE migration wizard supports the migration of enterprise bean deployment descriptors from the J2EE 1.3 specification level EJB resource to J2EE 1.4. Stateless session beans and message-driven beans are migrated to J2EE 1.4.

Migrating session beans

The J2EE migration wizard migrates stateless session beans that are defined as service endpoint interfaces (SEI) in the webservices.xml descriptor of an EJB project in J2EE 1.3 to the J2EE 1.4 specification level by setting the service endpoint interfaces on the stateless session bean. The J2EE 1.4 specification requires a SEI be defined on a stateless session bean if the session bean is to be used as a Web services endpoint. During the migration of an EJB JAR file, all session beans in the EJB project get the service endpoint set to the name used in the webservices.xml descriptor of the EJB project. Here is an example of how the metadata of an EJB project looks before and after migration to the J2EE 1.4 specification level.

EJB project in J2EE 1.3: `webservices.xml` descriptor with a stateless session bean used as a Web service endpoint interface before migration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE webservices PUBLIC "-//IBM Corporation, Inc.//DTD J2EE Web services 1.0//EN"
"http://www.ibm.com/webservices/dtd/j2ee_web_services_1_0.dtd">
  <webservices id="WebServices_1084831328093">
     <webservice-description id="WebServiceDescription_1084831328093">
        <webservice-description-name>EchoEJBService</webservice-description-name>
        <wsdl-file>META-INF/wsdl/EchoEJB.wsdl</wsdl-file>
        <jaxrpc-mapping-file>META-INF/EchoEJB_mapping.xml</jaxrpc-mapping-file>
        <port-component id="PortComponent_1084831328103">
           <port-component-name>EchoEJB</port-component-name>
           <wsdl-port id="WSDLPort_1084831328103">
              <namespaceURI>http://test</namespaceURI>
              <localpart>EchoEJB</localpart>
           </wsdl-port>
           <service-endpoint-interface>test.EchoEJB</service-endpoint-interface>
           <service-impl-bean id="ServiceImplBean_1084831328103">
              <ejb-link>EchoEJB</ejb-link>
           </service-impl-bean>
        </port-component>
     </webservice-description>
  </webservices>
```

The `<service-endpoint-interface>` and `<service-impl-bean>` tags in the preceding example define stateless session bean "EchoEJB" as a service endpoint in the Web services descriptor at the J2EE 1.3 specification level before migration.

EJB project in J2EE 1.4: EJB deployment descriptor for the same stateless session bean "EchoEJB" with service endpoint interface created by the migration process

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar>
<ejb-jar id="ejb-jar_ID" version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
 <display-name>
 EchoEJBProject</display-name>
 <enterprise-beans>
  <session id="EchoEJB">
   <ejb-name>EchoEJB</ejb-name>
   <home>test.EchoEJBHome</home>
   <remote>test.EchoEJB</remote>
   <service-endpoint>test.EchoEJB</service-endpoint>
   <ejb-class>test.EchoEJBBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>
</ejb-jar>
```

The`<service-endpoint>` tag in the preceding example defines "EchoEJB" as a service endpoint in the J2EE 1.4 specification level after migration.

Migrating message-driven beans

The J2EE migration wizard supports the migration of EJB 2.0 to EJB 2.1 message-driven beans.Message-driven beans were introduced in EJB 2.0 to support the processing of asynchronous messages from a Java™ Message Service (JMS). The EJB 2.1 specification expands the definition of the message-driven bean so that it can support any messaging system, not just JMS.

Note: Message-driven beans that have been migrated from the EJB 2.0 specification level to EJB 2.1 and are deployed to WebSphere® Application Server version 6 must be deployed against a Java Connector Architecture (JCA) 1.5 resource adapter instead of a listener port (as in WebSphere Application Server version 5). You must use the Deployment Descriptor Editor to change the WebSphere Bindings settings for the EJB 2.1 message-driven beans to use a JCA adapter. See Configuring an EJB 2.1 message-driven bean to use a JCA adapter.

The EJB 2.0 message-driven bean artifacts migrated are:

- acknowledgeMode
- messageSelector
- destinationType
- subscriptionDurablity

Some of the EJB 2.0 message-driven bean elements are replaced with `activation-config` properties. The property names and values used in the `activation-config` property to describe the messaging service varies depending on the type of message service used. However, EJB 2.1 defines a set of fixed properties for JMS-based message-driven beans.

The following example compares the elements of a sample bean in EJB 2.0 with how the elements appear in EJB 2.1.

An example of message-driven bean elements in EJB 2.0:

```
<message-driven id="Mdb20">
   <ejb-name>Mdb</ejb-name>
   <ejb-class>ejbs.MdbBean</ejb-class>
```

```
   <transaction-type>Bean</transaction-type>
   <message-selector>mdbMessage</message-selector>
   <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
   <message-driven-destination>
  <destination-type>javax.jms.Topic</destination-type>
  <subscription-durability>Durable</subscription-durability>
    </message-driven-destination>
</message-driven>
```

An example of message-driven bean elements in EJB 2.1:

```
    <message-driven id="Mdb21">
 <ejb-name>Foo/ejb-name>
 <ejb-class>ejbs.FooBean</ejb-class>
  <messaging-type>javax.jms.MessageListener</messaging-type>
  <transaction-type>Bean/transaction-type>
  <message-destination-type>javax.jms.Topic</message-destination-type>
   <activation-config>
  <activation-config-property>
    <activation-config-property-name>destinationType</activation-config-property-name>
    <activation-config-property-value>javax.jms.Topic</activation-config-property-value>
  </activation-config-property>
  <activation-config-property>
    <activation-config-property-name>subscriptionDurability</activation-config-property-name>
      <activation-config-property-value>Durable</activation-config-property-value>
   </activation-config-property>
   <activation-config-property>
      <activation-config-property-name>acknowledgeMode</activation-config-property-name>
      <activation-config-property-value>AutoAcknowledge</activation-config-property-value>
   </activation-config-property>
   <activation-config-property>
  <activation-config-property-name>messageSelector</activation-config-property-name>
  <activation-config-property-value>fooSelector</activation-config-property-value>
   </activation-config-property>
</activation-config>
</message-driven>
```

Related tasks:

Migrating secure web services
Configuring an EJB 2.1 message-driven bean to use a JCA adapter

Configuring an EJB 2.1 message-driven bean to use a JCA adapter

Message-driven beans that have been migrated from Enterprise Java™ Bean (EJB) 2.0 to the EJB 2.1 specification level and are deployed to WebSphere® Application Server version 6 must be deployed against a Java Connector Architecture (JCA) 1.5 resource adapter instead of a listener port.

Procedure

1. Open the EJB project in the Enterprise Explorer.
2. Double-click the EJB project **Deployment Descriptor** file in the Enterprise Explorer. The EJB deployment descriptor editor opens.
3. Click the **Bean** tab to open the **Bean** page.
4. For each EJB 2.1 message-driven bean in the EJB project, do the following:
   A. Select the EJB 2.1 message-driven bean in the list of beans on the left side of the Bean page.
   B. Under the heading **WebSphere Bindings**, select the **JCA Adapter** button.
   C. Specify bindings deployment properties:
      1. **ActivationSpec JNDI name**.Type the JNDI name of the J2C activation specification that is to be used to deploy this message-driven bean. This name must match the name of a J2C activation specification that you define to WebSphere Application Server.
      2. **ActivationSpec Authorization Alias**.The name of a J2C authentication alias used for authentication of connections to the JCA resource adapter. A J2C authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.
      3. **Destination JNDI name**.Type the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI name space.
5. Save the changes and close the Deployment Descriptor editor.

Related reference:
J2EE 1.3 to 1.4 specification level upgrade
Enterprise Java bean projects (EJB 2.0 to EJB 2.1)

Web projects (Servlet level 2.3 to Servlet level 2.4)

Artifacts of a Web deployment descriptor are migrated by the J2EE Migration Wizard when a J2EE 1.3 level Web project is migrated to J2EE 1.4.
The following Web application artifacts are migrated:

Authentication constraints

J2EE 1.4 includes a Description object that has two attributes: *language* and *value*. This Description object did not exist in J2EE 1.3; the description was an attribute on the Authentication Constraint. So, when the artifacts of a Web deployment descriptor are migrated to J2EE 1.4, the *value* of the Description object is taken from the description attribute of the Authentication constraint.

Security constraints

Similarly, in J2EE 1.3 the description was an attribute on Security Constraint. In J2EE 1.4, there is a new Description object with the attributes *language* and *value*. So, the *value* of the Description object is taken from the description attribute of the Security Constraint.

Web application

The description string attribute of the ContextParam object in the J2EE 1.3 specification level has been moved to a Description object in ParamValue in J2EE 1.4.
The TagLib object in J2EE 1.3 has been moved to the JSPConfig object in J2EE 1.4. The JSPConfig objects belonged to the Web root object in 1.3.

Connector projects (JCA 1.0 to JCA 1.5)

The J2EE mMigration wizard migrates the artifacts of a J2EE connector architecture (JCA) 1.0 descriptor to JCA 1.5. The migrated artifacts are related to the elements of the ResourceAdaptor object as two new objects, OutboundResourceAdaptor and ConnectionDefinition, were added to the ResourceAdaptor object in J2EE 1.4 specification level for Connector projects.

The mapping of the migrated elements:

- The following elements are moved from the ResourceAdaptor object to the OutboundResourceAdaptor object:
  - reauthenticationSupport
  - transactionSupport
- The following elements are moved from the ResourceAdaptor object to the ConnectionDefinition object:
  - managedConnectionFactoryClass
  - connectionFactoryInterface
  - connectionFactoryImplClass
  - connectionInterface
  - connectionImplClass
- The outboundResourceAdaptor object contains a list of ConnectionDefinition definitions. Hence the ConnectionDefinition is added to the ConnectionDefinition list held by OutboundResourceAdaptor.
- The OutboundResourceAdaptor object is contained by the ResourceAdaptor object.
- The AuthenticationMechanism object has undergone some changes in JCA 1.5 where the customAuthMechType element is replaced by the authenticationMechanism and the authenticationType element is replaced by authenticationMechanism
- The description attribute in the ResourceAdaptor object is replaced with a Description object where the description attribute string is set to value element in the Description object for the following objects:
  - SecurityPermission
  - AuthenticationMechanism
  - ConfigurationProperties

Web services (J2EE 1.3 to J2EE 1.4)

The J2EE 1.4 specification has added support for web services through the new JAX-RPC 1.0 API.
The JAX-RPC API supports service endpoints through:
- servlets with JAXR-RPC
- servlets with direct SOAP
- stateless session beans

The J2EE 1.4 specification supports the web services for J2EE specification (JSR 109). JSR 109 defines the deployment requirements for web services and uses the JAX-RPC programming model.The following web services artifacts are migrated using the J2EE Migration wizard:
- Web services descriptor
- Web services client descriptor
- JAX-RPC mapping descriptor

Migrating web service deployment descriptors

Any web service deployment descriptors contained in J2EE 1.3 projects that are migrated to the J2EE 1.4 specification level can be migrated from JSR-109 V1.0 (for J2EE 1.3) to J2EE 1.4. Web service deployment descriptors, as defined by JSR-109 V1.0, consist of the files webservices.xml, webservicesclient.xml, and all JAX-RPC mapping deployment descriptors that are referenced by the webservices.xml and webservicesclient.xml files. As with other J2EE deployment descriptors, migration modifies the structure of the information contained in the descriptors in order to make them comply with the J2EE 1.4 specification. One structural change which is particular to the web service deployment descriptors is the change to the way in which qualified names are represented. In JSR-109 V1.0, qualified names are represented using a sequence of two elements, `<namespaceURI>` and `<localpart>`, which contain the namespace URI and the local part of the name respectively. Qualified names in J2EE 1.4 are based on the XMLSchema QName type, which uses XML namespaces.
Further details on the migration of each of the web service deployment descriptors:
- **Web services descriptor (webservices.xml)**The webservices.xml deployment descriptor is present in web projects and EJB projects that contain J2EE web services. Both the `<wsdl-port>` element and the `<soap-header>` element contain qualified names and their contents are migrated to the J2EE 1.4 format.
  For example, if `<wsdl-port>` is represented as follows before migration,

```
<wsdl-port>
 <namespaceURI>http://addressbook.webservice</namespaceURI>
 <localpart>AddressBook</localpart>
</wsdl-port>
```

  After migration `<wsdl-port>` are represented as:

```
<wsdl-port xmlns:pfx="http://addressbook.webservice">pfx:AddressBook</wsdl-port>
```

  The prefix "pfx" is used as the namespace prefix for all qualified names that are migrated.

- **Web services client descriptor (webservicesclient.xml)**The webservicesclient.xml deployment descriptor is present in J2EE 1.3 web projects, EJB projects, and application client projects that contain J2EE web service clients. During migration from J2EE 1.3 to 1.4, the contents of webservicesclient.xml are migrated and moved to the deployment descriptor for the project. The process that occurs is as follows:
  - For web projects, all `<service-ref>` elements in webserivcesclient.xml are moved under the `<web-app>` element in web.xml.
  - For application client projects, all `<service-ref>` elements in webservicesclient.xml are moved under the `<application-client>` element in application-client.xml.
  - For EJB projects, all `<service-ref>` elements within a `<component-scoped-refs>` in the webserviceclient.xml are moved under the corresponding `<enterprise-bean>` in ejb-jar.xml.
  - Webservicesclient.xml is deleted.

  Both the `<service-qname>` element and the `<soap-header>` element contain qualified names and their contents are migrated to the J2EE 1.4 format. For example, if `<service-qname>` is represented as follows before migration,

```
<service-qname>
 <namespaceURI>http://addressbook.webservice</namespaceURI>
 <localpart>AddressBookService</localpart>
</service-qname>
```

  After migration `<service-qname>` are represented as:

```
<service-qname xmlns:pfx="http://addressbook.webservice">pfx:AddressBookService</service-qname>
```

  The prefix "pfx" is used as the namespace prefix for all qualified names that are migrated.
- **JAX-RPC mapping descriptor**Both the webservices.xml and webservicesclient.xml deployment descriptors can reference one or more JAX-RPC mapping deployment descriptors.
  In the webservices.xml file, these references are contained in the `<jaxrpc-mapping-file>` element under each `<webservice-description>` element. In the webservicesclient.xml file, these references are contained in the `<jaxrpc-mapping-file>` element under each `<service-ref>` element.
  During migration from J2EE 1.3 to 1.4, all the JAX-RPC mapping deployment descriptors referenced in webservices.xml and webservicesclient.xml are migrated. Migration includes migrating all qualified names to the J2EE 1.4 format (see the sections on webservices.xml and webservicesclient.xml for examples of qualified names following migration).

J2EE 1.2 to 1.4 specification level migration

J2EE modules can be migrated from the J2EE 1.2 specification level to J2EE 1.4. This section describes the artifacts migrated for J2EE projects from J2EE 1.2 to J2EE 1.4 specification level by the J2EE upgrade wizard.
For details on using the J2EE upgrade wizard, refer to Migrating the specification level of Java EE projects.

Related tasks:
Migrating the specification level of Java EE projects
Migrating Enterprise JavaBeans projects (EJB 1.1 to EJB 2.x)

Migrating Enterprise JavaBeans projects (EJB 1.1 to EJB 2.x)

Migrating an EJB project from the EJB 1.1 to EJB 2x specification level is described in this section.

About this task

Migrating an EJB project from EJB 1.1 to EJB 2.x involves the migration of container-managed persistence (CMP) entity beans.
There have been no changes in entity beans between J2EE 1.3 and J2EE 1.4 specification level. Migrating CMP entity beans from the EJB 1.1 to EJB 2.1 specification level is accomplished the same way as migrating a CMP entity bean from EJB 1.1 to EJB 2.0. Migrating container-managed entity beans from EJB 1.1 to EJB 2.x specification level requires the following steps:
1. Convert the EJB project from EJB 1.1 to EJB 2.x.
2. Migrate the EJB code from EJB 1.1 to EJB 2.x.
3. Migrate any user-defined EJB 1.1 references to local references in EJB 2.x.

Related reference:
J2EE 1.2 to 1.4 specification level migration
Merge of method elements during project structure migration
Web projects (Servlet level 2.2 to Servlet level 2.4)

Converting projects from EJB 1.1 to EJB 2.x

An EJB 1.1 project can be converted to an EJB 2.x project using the J2EE Migration Wizard.

About this task

- In the Java™ EE view, right-click on the 1.1 project, and then select **Java EE** > **Specifications Upgrade Wizard**.

Or, if you want to preserve the original EJB 1.1 project, you can create a new 2.x project and then import the existing project JAR file into it (**File** > **Import** > **EJB JAR**).
Although the project is an EJB 2.x project, the existing (or imported) EJB 1.1 container-managed persistence (CMP) entity beans remain EJB 1.1 beans. That is, the CMP entity beans are not converted to EJB 2.x.
The J2EE Migration wizard migrates the enterprise beans within an EJB 2.x project from 1.1 into 2.x. (If you choose to migrate your 1.1 CMP entity beans to 2.x, all beans in the 2.x project must be migrated. However, you can selectively choose to add local client views to these migrated 2.x beans.)
- The wizard maintains the existing EJB 1.1 inheritance in the EJB 2.x project.
- The wizard maintains EJB 1.1 (proprietary) relationships into EJB 2.x (standard) relationships, plus other benefits.

Note: If you have any mapped associations, EJB 2.x associations are created for the associations themselves, but the role maps for those associations become invalid. If you run validation, you see an error occur. To get around this, open the mapping editor first and save the map. The role maps are removed. You may then run validation again and remap the roles.

Related reference:
Merge of method elements during project structure migration
Web projects (Servlet level 2.2 to Servlet level 2.4)

Migrating code from EJB 1.1 to EJB 2.x

For projects converted from EJB 1.1 to EJB 2.x, steps must be taken to migrate existing EJB 1.1 code to EJB 2.x.

About this task

Note: EJB 2.x beans are only supported in an EJB 2.x project (although a 2.x project also supports 1.1 beans).

1. For any CMP 1.1 bean, replace each CMP field with abstract `getXXX` and `setXXX` methods. (Then the bean class needs to be abstract.)
2. For any CMP, create an abstract `getXXX` and `setXXX` method for the primary key.
3. For any CMP 1.1 finder method, create an `EJBQL` (EJB Query Language) method for each finder method. **Note:** EJB Query Language has the following limitations in version 7.5 V6.0:
   - EJB Query Language queries involving EJBs with keys made up of relationships to other EJBs appear as invalid and cause errors at deployment time.
   - The IBM® EJB Query Language support extends the EJB 2.x specification in various ways, including relaxing some restrictions, adding support for more DB2® functions, and so on. If portability across various vendor databases or EJB deployment tool is a concern, then care should be taken to write all EJB Query Language queries strictly according to instructions described in Chapter 11 of the EJB 2.x specification.

   Note: EJB Query Language has the following limitations in Rational® Application Developer V7.5:
   - For EJB 1.1 level beans, IBM provides an extension for supporting finders based on EJBQL. When EJBs which use this are migrated to be 2.x EJBs, the old abstract schema name used in the EJBQL query does not match the abstract schema name that the migrator gives to the EJB. This results in errors in the problems view like "WQRY0025E: The abstract schema name ResultBean is not defined". The fix is to manually change the abstract schema name in the query to match that of the migrated EJB.

4. For any CMP 1.1 finder, return `java.util.Collection` instead of `java.util.Enumeration`.
5. For any CMP 1.1 bean, change all occurrences of `this.field = value` to `setField(value)` in `ejbCreate()` and elsewhere throughout the code.
6. Update your exception handling (rollback behavior) for non-application exceptions:
   - Throw `javax.ejb.EJBException` instead of `java.rmi.RemoteException` to report non-application exceptions.
   - In EJB 2.x and 1.1, all non-application exceptions thrown by the instance result in the rollback of the transaction in which the instance executed, and in discarding the instance.
7. Update your Exception handling (rollback behavior) for application exceptions:
   - In EJB 2.x and 1.1, an application exception does not cause the container to automatically roll back a transaction.

- In EJB 1.1, the container performs the rollback only if the instance was invoked using the `setRollbackOnly()` method on its EJBContext object.

8. Update any CMP setting of application-specific default values to be inside `ejbCreate` (not using global variables, since EJB 1.1 containers set all fields to generic default values before calling `ejbCreate` which overwrites any previous application-specific defaults).

Related reference:
Merge of method elements during project structure migration
Web projects (Servlet level 2.2 to Servlet level 2.4)

Migrating EJB references for EJB 1.1 relationships

When EJB 1.1 relationships were created, EJB references were created to the Remote Client view. During the EJB 1.1 project migration using the J2EE Migration wizard, these EJB remote references for the EJB 1.1 relationships are removed and replaced with EJB local references. Local references for user-defined EJB references must be recreated manually.

About this task

Note: In EJB 2.x, EJB relationships can be created only if the beans have Local Client view defined on them and the EJB local references are created for the EJB 2.x relationships.

For user-defined EJB references, migration is *not* done using the J2EE Migration wizard. However, follow these steps to set up the local references:

1. Delete the existing EJB remote references in References page of the deployment descriptor editor.
2. Add an EJB local reference in References page of the deployment descriptor editor.

Related reference:

Merge of method elements during project structure migration

Web projects (Servlet level 2.2 to Servlet level 2.4)

Merge of method elements during project structure migration

During the project structure migration using the J2EE Migration wizard, the method elements (that include security identity, container transaction, method permissions, access intent, and isolation levels) of the same type for all the beans are merged to make them logically grouped.

A sample of the method elements before and after the project structure migration.

Here is a sample of the method permission in the source page of the deployment descriptor editor before project structure migration. `<method-permission>`

```
  <role-name>rol1</role-name>
  <role-name>rol2</role-name>
  <method>
   <ejb-name>TestBean1</ejb-name>
   <method-intf>Home</method-intf>
   <method-name>getEJBMetaData</method-name>
   <method-params>
   </method-params>
  </method>
  <method>
   <ejb-name>TestBean1</ejb-name>
   <method-intf>Home</method-intf>
   <method-name>getHomeHandle</method-name>
   <method-params>
   </method-params>
  </method>
  <method>
   <ejb-name>TestBean2</ejb-name>
   <method-intf>Home</method-intf>
   <method-namae>remove</method-name>
   <method-params>
    <method-param>java.lang.Object</method-param>
   </method-params>
  </method>
  <method>
   <ejb-name>TestBean2</ejb-name>
   <method-intf>Home</method-intf>
   <method-name>remove</method-name>
   <method-params>
    <method-param>javax.ejb.Handle</method-param>
   </method-params>
  </method>
</method-permission>
<method-permission>
 <role-name>rol1</role-name>
 <role-name>rol2</role-name>
 <method>
```

```
  <ejb-name>TestBean2</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>isIdentical</method-name>
  <method-params>
   <method-param>javax.ejb.EJBObject</method-param>
  </method-params>
 </method>
</method-permission>
```

Here is a sample of the method permission in the source page of the deployment descriptor editor after project structure migration. `<method-permission>`

```
 <role-name>rol1</role-name>
 <role-name>rol2</role-name>
 <method>
  <ejb-name>TestBean1</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>getEJBMetaData</method-name>
  <method-params>
  </method-params>
 </method>
 <method>
  <ejb-name>TestBean1</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>getHomeHandle</method-name>
  <method-params>
  </method-params>
 </method>
 <method>
  <ejb-name>TestBean2</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>remove</method-name>
  <method-params>
   <method-param>>java.lang.Object</method-param>
  </method-params>
 </method>
 <method>
  <ejb-name>TestBean2</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>remove</method-name>
  <method-params>
   <method-param>javax.ejb.Handle</method-param>
  </method-params>
 </method>
 <method>
  <ejb-name>TestBean2</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>isIdentical</method-name>
  <method-params>
```

```
    <method-param>javax.ejb.EJBObject</method-param>
   </method-params>
  </method>
 </method-permission>
```

Note: When the CMP 1.x to CMP 2.x bean migration is also selected along with the project structure migration in the J2EE Migration wizard, the access intent and isolation levels are removed but everything else is merged during migration. The reason that access intents and isolation level are removed is that they are no longer valid due the changes in the extensions model. The new model has both access intents and isolation level defined in access intents and we have bean-level access intents and method-level access intents. It is always advised to use the bean-level access intents rather than the method-level access intents.

Related tasks:
Migrating Enterprise JavaBeans projects (EJB 1.1 to EJB 2.x)
Converting projects from EJB 1.1 to EJB 2.x
Migrating code from EJB 1.1 to EJB 2.x
Migrating EJB references for EJB 1.1 relationships

Web projects (Servlet level 2.2 to Servlet level 2.4)

Artifacts of a web deployment descriptor are migrated by the J2EE Migration wizard when a J2EE 1.2 web project is migrated to the J2EE 1.4 specification level.
The following web application artifacts are migrated:

Authentication constraints

J2EE 1.4 includes a Description object that has two attributes: *language* and *value*. This Description object did not exist in J2EE 1.2; the description was an attribute on the Authentication Constraint. So, when the artifacts of a web deployment descriptor are migrated to J2EE 1.4, the *value* of the Description object is taken from the description attribute of the Authentication constraint.

Security constraints

Similarly, in J2EE 1.2 the description was an attribute on Security Constraint. In J2EE 1.4 there is new Description object with the attributes *language* and *value*. So, the *value* of the Description object is taken from the description attribute of the Security Constraint.

Web application

The description string attribute of the ContextParam object in the J2EE 1.2 specification level has been moved to a Description object in ParamValue in J2EE 1.4.
The TagLib object in J2EE 1.2 has been moved to the JSPConfig object in J2EE 1.4. The JSPConfig objects belonged to the web root object in 1.2.

Related tasks:
Migrating Enterprise JavaBeans projects (EJB 1.1 to EJB 2.x)
Converting projects from EJB 1.1 to EJB 2.x
Migrating code from EJB 1.1 to EJB 2.x
Migrating EJB references for EJB 1.1 relationships

Migrating a CMP EJB project from WebLogic

When migrating a CMP EJB project from WebLogic you need to change the database schema name to NULLID, in order to made the generated code independent of the database schema name.

Limitation in specifying schema name

CMP EJB v1.x or v2.x mapping created against a data model using a specific schema name results in generated SQL code using the schema name prefix (schema_name.table). The database could be DB2® or Oracle or other vendor. This does not allow for flexibility when specifying a corresponding data source in WebSphere® Application Server or v7.x to be independent of the schema name in the generated code, by having the schema name to be automatically set to be the same as the login id used to connect to the database.

Flexible schema names

Rather than hard coding a schema name into a CMP EJB 1.x or 2.x mapping, use NULLID. If the schema name NULLID is used in the database model, then the generated EJB deploy code SQL statements are independent of the schema name, allowing flexibility in the data source specified on the WebSphere Application Server V7.x server. The NULLID prefix of the generated SQL statements (NULLID.table) is changed at runtime by the application server ejb/datasource runtime to correspond to the login id specified for the server data source.

If an EJB mapping is done against a database connection that results in a Data Model in the EJB backend folder in which the schema name is other than NULLID, you can change the schema name in this way:

1. For Rational® Application Developer V7.0, there is a logical Data Models folder in an EJB project in the Enterprise Explorer view of the Java™ EE perspective. When expanded, the Data Models folder shows the database model file: {<name>).dbm. This file is physically located in the EJB project folder: *ejbModule/META-INF/backends/{database vendor backend folder}*.
2. Double click the *.dbm data model file to open the Physical Data Model editor; or right click the file and select **Open** .
3. Click the Schema, and its information will show in Properties view. Change the schema name to `NULLID`. Save the file.
4. Right-click your CMP EJB project, and select **Java EE** > **Prepare for deployment**. The generated SQL code is prefixed with NULLID, and this prefix is modified at runtime by the WebSphere Application server to send SQL statements having a schema prefix corresponding to the login id **Note:** There are some database vendor and version scenarios where it is not acceptable to have the schema name correspond to the login id of the server data source and requires setting custom properties. For z/OS® DB2, see Specifying a runtime DB2 z/OS data source Schema for a CMP EJB Project with a NULLID mapped Schema.

Developing EJB 3.1 applications

Java™ EE specification makes the creation of EJB 3.1 applications simpler than previous EJB specifications.

New in EJB 3.1 (JSR 318)

- Singleton beans (@Singleton)
  - Before EJB 3.1, there was no easy way to share data throughout the application; the EJB 3.1 Singleton is an application-wide singleton.
  - Concurrency can be managed either by the container or by the Bean Developer

Java EE streamlines EJB development in the following ways:

- **Fewer required classes and interfaces**
  - Home and object interfaces are no longer required – you need the business interface only
  - No need to implement javax.ejb.SessionBean
  - No need to declare checked exceptions
- **Optional deployment descriptors**
  - Annotations provide component definition and dependency injection
- **Simple lookups**
  - new EJBContext() interface method replaces JNDI calls
- **Lightweight persistence for object-relational mapping**
  - Entities are POJOs that provide an object-oriented view of the data stored in relational database
- **New Interceptors class (new in Java EE 5)**
  - Interceptors are objects that can intercept a call to a business method (to handle security for example)
  - Similar in purpose and action to Servlet filter or Web services handler
  - Provide limited form of aspect-oriented programming

Related concepts:

EJB 3.1 overview
Differences between EJB 3.1 and EJB 2.1
EJB modules
Editing EJB 3.1 applications
Securing enterprise applications
Testing EJB 3.1 applications
Deploying EJB 3.1 applications
Creating enterprise beans

Related tasks:

Creating EJB projects