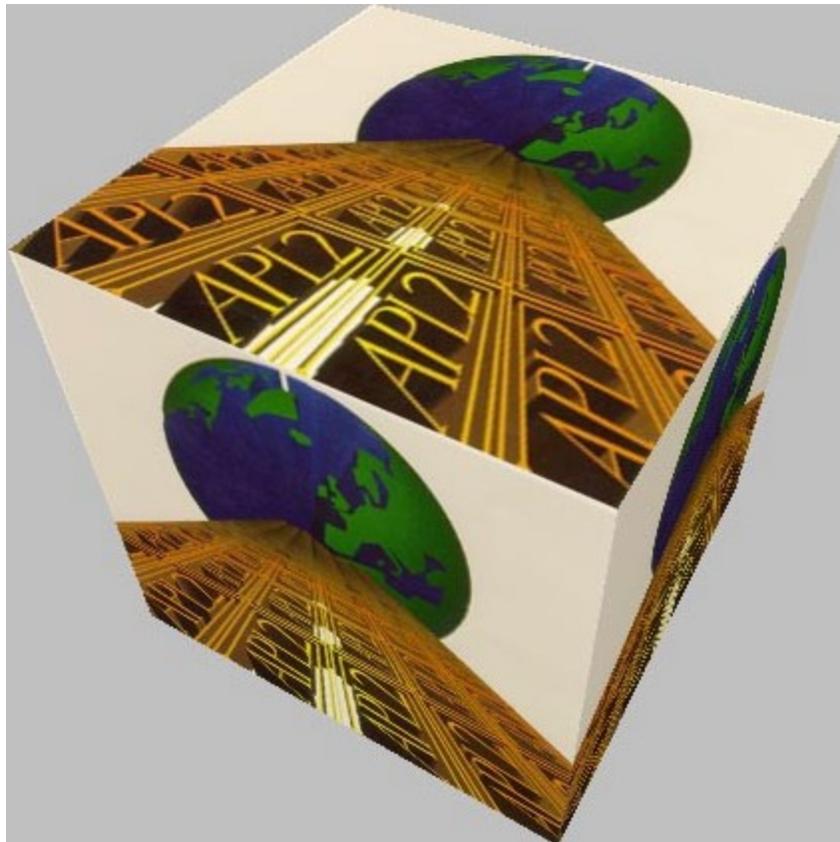# APL2 Programming:
Using APL2 with WebSphere
SC18-9442-02

APL Products and Services

IBM Silicon Valley Laboratory

555 Bailey Avenue

San Jose, California 95141

APL2@vnet.ibm.com

Copyrights

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe on any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject material in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Programming Interface Information

This user's guide is intended to help programmers write applications in APL2. It documents General-Use Programming Interface and Associated Guidance Information provided by APL2. General-use programming interfaces allow the customer to write programs that obtain the services of APL2.

Trademarks

IBM Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
APL2
IBM
WebSphere

Other Trademarks

The following terms are trademarks of other companies:

J2EE	Sun Microsystems, Inc.
Java	Sun Microsystems, Inc.
Linux	Linus Torvalds
Windows	Microsoft Corporation
Solaris	Sun Microsystems, Inc.
Sun	Sun Microsystems, Inc.

Required Software

The following software was used to build and run the examples in this document:

Workstation APL2 for Multiplatforms Version 2.0 with Service Level 4
WebSphere Studio Application Developer 5.1.2
WebSphere Application Server Version 5.1

Other versions of the WebSphere software can be used except that WebSphere Application Server Express does not support the Enterprise Java Beans demonstrated in the later examples.

The examples were built and deployed on Windows XP. The WebSphere Studio Application Developer windows may have different appearances on other systems.

The examples use an APL2 namespace. The full Workstation APL2 development environment is required to build namespaces. Either the full Workstation APL2 development environment or the APL2 Runtime Library can be used to run APL2 namespaces.

Because APL2 namespace files are platform dependent, a different namespace file must be built for each operating system (AIX, Linux, Solaris or Windows) on which the namespace will be used. The full Workstation APL2 development environment must be installed on at least one machine for each target operating system so that the appropriate namespace file can be built.

Installation of either the full Workstation APL2 development environment or the APL2 Runtime Library is required on each machine that will call APL2.

Motivation

Workstation APL2 Version 2.0 Service Level 4 includes a new feature that supports calling APL2 from Java. Using this facility, it is now possible to deploy APL2 applications for use under WebSphere Application Server. There are a variety of excellent reasons why you should learn to use APL2 under WebSphere Application Server. For example,

If you're a Data Center IT manager who has APL2 programmers as well as Java programmers in your shop, how do you,,,

Cost-effectively Web-enable APL2 software in which you've devoted years of IT investment?

Leverage your IT shop's application deployment expertise?

Reduce the amount of specialized code you need to deploy APL2 applications?

Reduce your need for specialists in deploying APL2 applications?

Focus your expensive APL2 programmers on business applications rather than deployment code?

Reduce the security risks of APL2 web servers and deployment tools?

Load balance your APL2 applications across multiple machines?

If you're an APL2 programmer, how do you,,,

Deploy APL2 applications across the Web?

Respond to multiple client requests asynchronously?

Leverage tools for security control and HTTP request and XML parsing?

Get the IT shop to stop asking you to stop using APL2?

The answer to all these questions is to use WebSphere Application Server and the new APL2 Java interface to call APL2! The IBM WebSphere Application Server can call standard Java components which in turn can call APL2 applications. By leveraging WebSphere Application Server you can almost completely avoid writing and maintaining specialized APL2 code to deploy APL2 applications. In addition, you gain the benefits of WebSphere's extensive request management, security, and load balancing features.

Abstract

Workstation APL2 Version 2.0 Service Level 4 supports calling APL2 from Java. Using this facility, it is now possible to deploy APL2 applications for use under WebSphere Application Server. This document explains how to use WebSphere Studio Application Developer to build Java components that use APL2 and can be used under WebSphere Application Server.

This document shows how to build an APL2 namespace and use it in the following types of Java components:

Java servlets
Java Server Pages (JSPs)
Java Beans
Enterprise Java Beans (EJBs)

Use of the Model-View-Controller (MVC) design pattern is illustrated using all four types of Java components.

WebSphere Overview

The IBM web site includes this definition of WebSphere Application Server:

"IBM WebSphere Application Server is a high-performance and extremely scalable transaction engine for dynamic e-business applications."

What does this mean? A little history of web servers will help explain:

Traditional HTTP web page servers originally only provided clients with access to static HTML pages. Today, it is still common that most content on web sites is static. So, this is a good model for a large amount of web content.

However, people quickly realized that some content needed to be dynamic. Responses to user requests would need to include customized results. Over the years, several features have been added to web servers to address this problem.

Early web servers supported the Common Gateway Interface (CGI) protocol. CGI enabled the servers to execute programs in response to user requests. However, these were stand-alone programs which were not integrated with the server. They had to provide all their own services such as database access, messaging, and security.

A further development was the addition of using client side code. *Applets* are Java programs that run on the client's machine. Applets were often used to simply provide an enhanced user interface. Sometimes though, they were used to generate true dynamic pages. But to do this, they often had to reach back to the server to access server data. This introduced security and management concerns.

The next development was the use of *servlets*, or server side code, to generate dynamic pages. Servlets are called directly by the application server. Because servlets are called in the context of the server, they can exploit services provided by the server. No longer would application writers have to provide their own security, messaging, and database access services. However, servlets too had a problem. Servlets contained code both to process the logical content of a request and code to generate HTML. This mixture of code and HTML was hard to manage. The solution to this problem was the introduction of *Java Server Pages*, or JSPs.

Java Server Pages format Java output and produce HTML. Like servlets, JSPs contain both HTML and Java code. However, JSPs typically do not include Java code that processes the logical content of requests.. Rather, the Java code in JSPs merely retrieves and formats servlet results so they can be included in the output HTML.

By judicious use of servlets and JSPs, application developers could separate the logic portions of their applications from the presentation portions. This enabled development of applications that were more easily maintained and reused.

Eventually developers recognized that applications typically have two kinds of logic: application logic and business logic. Application logic includes decisions about how to process a request. For example, determining the type of a request is an application logic decision. Business logic includes the core calculations of an application. For example, calculating a dividend would be a business logic operation. The final step in our history is the introduction of components which allow the separation of application and business logic. *Java*

Beans and *Enterprise Java Beans*, or EJBs, are used to perform business logic. Servlets are now only used to perform application logic.

Java Beans and EJBs are like servlets; they are written in Java. Java Beans are simple Java classes. EJBs are more complicated but offer some real benefits; EJBs are reusable and so can be load managed across multiple machines. Typically, EJBs are used to perform core business logic operations. Java beans are used to handle the interface between servlets and EJBs.

So, the current state of enterprise application development is this:

Model-View-Controller Design Pattern
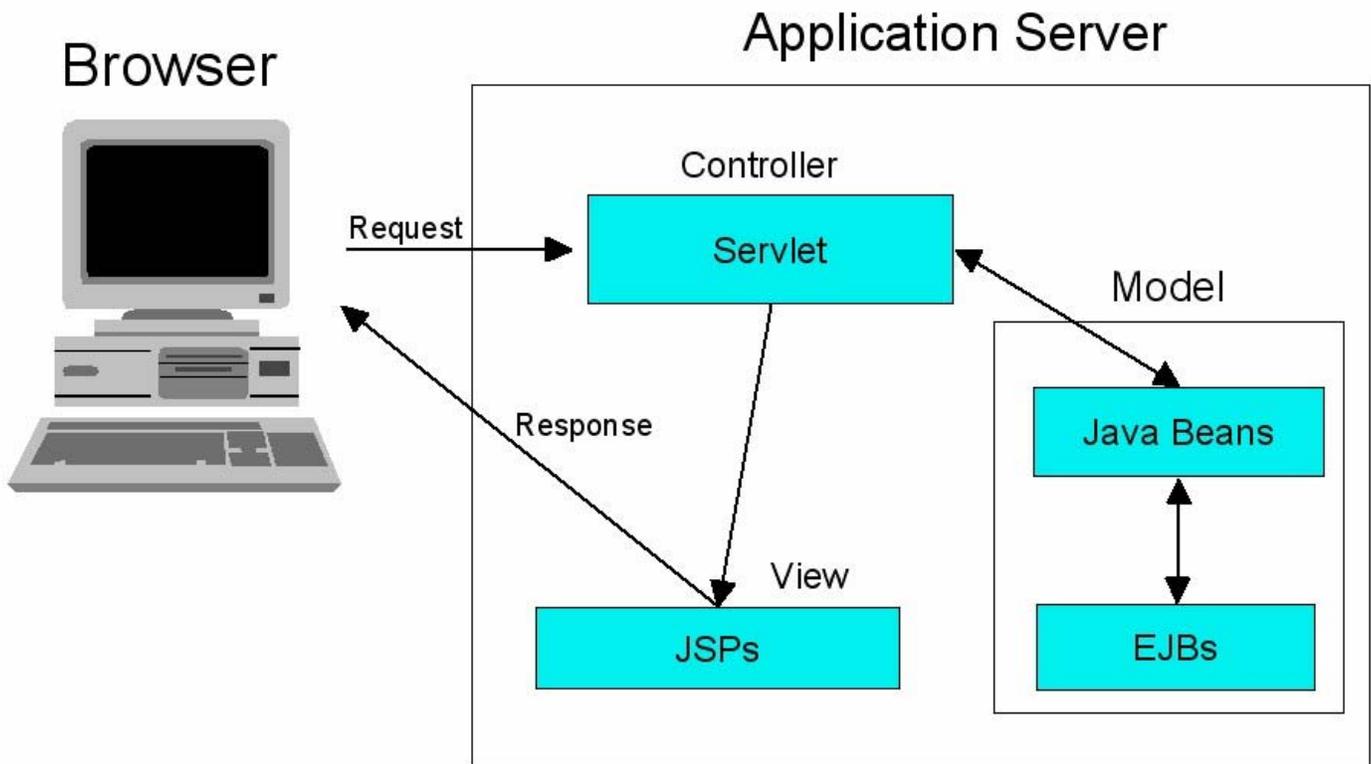
Servlets are used to accept requests, perform application logic and determine how the request should be processed. The servlets then call one or more Java beans, and in turn EJBs, to perform the core business logic necessary to fulfill the request. The servlets then pass the beans' results on to JSPs to generate HTML. This separation of tasks is so common it has a name, the *Model-View-Controller*, or MVC, design pattern. The servlet acts as a *controller* which manages the processing of the request, the Java beans and EJBs implement the business *model*, and the JSP generates the user's output, or *view* layer of the application.

Because the MVC design pattern separates the application logic, business logic, and view portions of applications, it supports easier maintenance and better reusability. It also allows more efficient allocation of resources. Experts in business logic can focus on the business's algorithms; experts in application logic can focus on building applications, and experts in page design can focus on the user interface.

The WebSphere Application Server, or WAS, manages all four types of components. WAS provides a container in which the components operate and which provides services to them. WAS also provides tools for administering the applications it is serving. These tools include security control, access control, and load balancing, among others.

The rest of this document will demonstrate how to use APL2 from servlets, Java Server Pages, Java Beans, and Enterprise Java Beans. Although it will be shown that it is possible to use APL2 from all these types of components, the final example should be your guide for robust application development and deployment. The final example follows the MVC design pattern and uses a servlet to accept requests, a Java Bean and an EJB to perform the business logic (including a call to APL2) and a Java Server Page to format the output.

WebSphere Studio Application Developer Introduction

The Java 2 Platform, Enterprise Edition (J2EE) specification describes the structure and use of servlets, Java Server Pages, Java Beans, and Enterprise Java Beans to build J2EE compliant applications. WebSphere Studio Application Developer, WSAD, is an Integrated Development Environment, an IDE, used for creating and testing J2EE compliant applications. The J2EE specification also includes rules for how to combine components into archive files and how the contents of these files should be described for deployment on J2EE compliant servers such as WebSphere Application Server. WSAD includes tools for managing archive and employment descriptor files.

All the resources used in a WSAD session are stored in a directory called a *workspace*. When you first start WSAD, you will be prompted for the directory you want to use as your workspace. WSAD will then display the contents of the workspace on the WSAD *workbench*. The following picture shows a typical WSAD workbench:
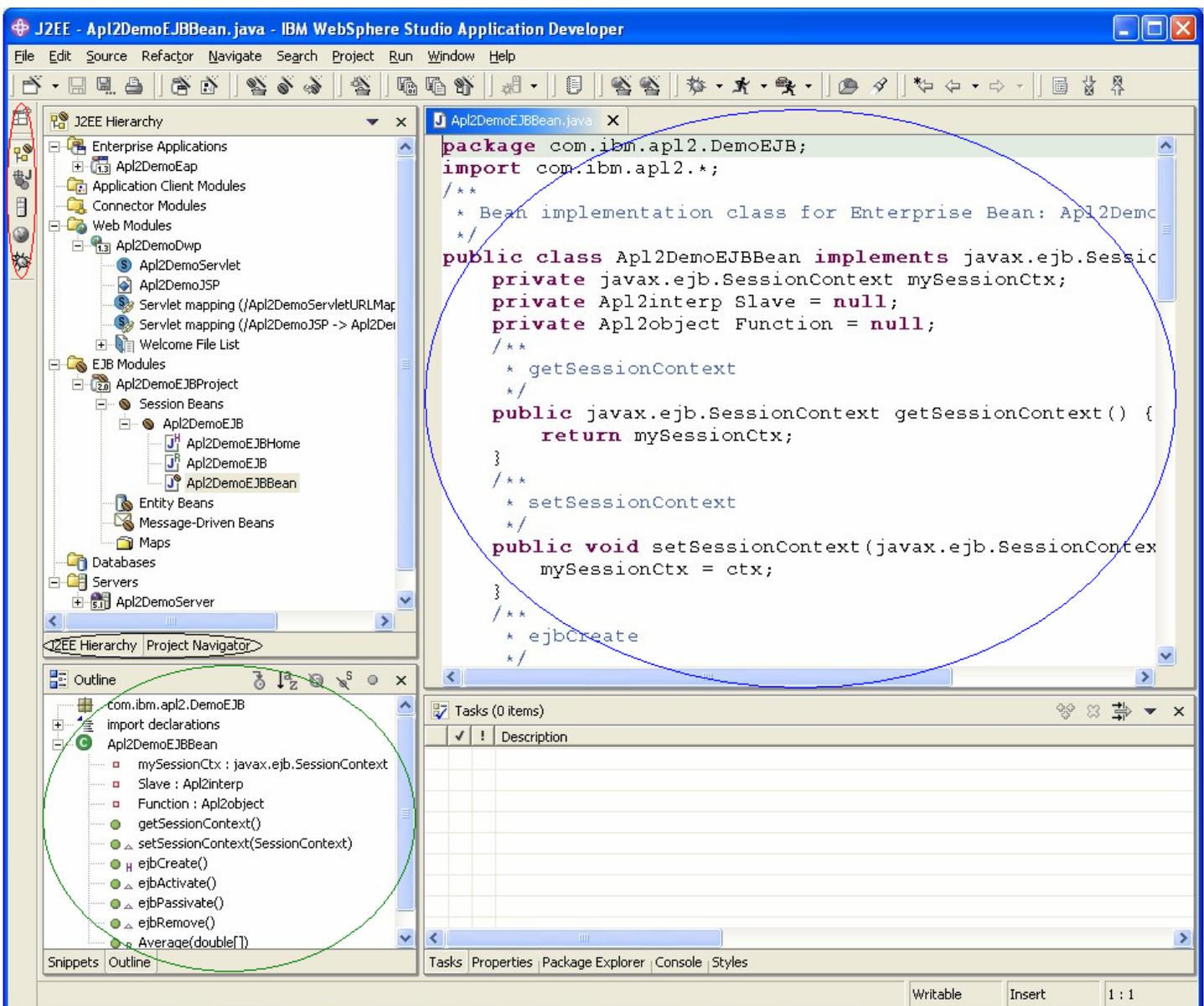
WSAD provides a variety of tools for working with different types of resources. Each tool is presented in a *view*. This workbench shows four view windows. A Java Editor view shows some Java source code; it is in the upper right hand corner and circled in blue. The Outline view shows a list of the classes, fields, and methods in the currently displayed Java source code; it is in the lower left corner and circled in green. Views of the same resource are frequently stacked in tabbed notebooks allowing the user to easily switch between different views. The J2EE Hierarchy and Project Navigator views are stacked in a tabbed notebook; their tabs are in the left center and circled in black.

WSAD allows the user to work with projects from several *perspectives*. Each perspective provides a set of capabilities aimed at accomplishing a specific type of task or works with specific types of resources. For example, the Java perspective combines views that you would commonly use while editing Java source files, while the Debug perspective contains the views that you would use while debugging Java programs. The picture shows the J2EE Perspective. The Perspective toolbar is in the upper left corner and circled in red; it is used to switch perspectives.

The J2EE Hierarchy view is the upper left window. It shows the resources in the workspace as they are arranged in the J2EE Hierarchy. WSAD uses the J2EE Hierarchy to organize applications' files.

WSAD organizes application files into *projects* of which there are several types including:

Enterprise Application Project   Contains a list of all the projects in an application
Web Project   Contains servlets, Java Server Pages, and Java Beans
EJB Project   Contains Enterprise Java Beans
Server Projects   Contains web server configuration information

The J2EE Hierarchy view shows the workspace's projects and their components in a tree structure. The workspace displayed in the picture contains four projects:

An Enterprise Application named Apl2DemoEap
A Web Module named Apl2DemoDwp
An EJB Module named Apl2DemoEJBProject
A Server named Apl2DemoServer

You can create a new project or component by pulling down the File menu and selecting New. You will be prompted with a list of project and component types. Selecting one of them will lead you to a wizard for building the resource.

You can open a resource using the default view by double clicking on it. You can use an alternate view by right-clicking on the resource name and selecting Open With.

The J2EE hierarchy shows the workspace's resources and their J2EE hierarchical relationships. If you double click on a resource in the J2EE Hierarchy, the default view will show the resource's *deployment descriptor*. A deployment descriptor lists the components in a resource, its relationships to other resources, and how the resource should be deployed.

By switching to another workspace perspective or view, you can change the default view for different types of resources. For example, in the J2EE Perspective's Project Navigator view, the default view for servlets is the Java Editor.

This is necessarily a very brief introduction to WebSphere Studio Application Developer. For further information consult the product's online help and tutorials.

Introduction to Examples

The rest of this document is a series of examples that show how to use WebSphere Studio Application Developer to use APL2 in servlets, Java Server Pages, Java Beans and Enterprise Java beans, and how to deploy an Enterprise Application on WebSphere Application Server.

All the examples use a simple APL2 namespace that calculates the average of a vector of numbers. Build the namespace like this:

```
      )CLEAR
CLEAR WS
      ∇Z←AVERAGE VECTOR
[1]    Z←(+/VECTOR)÷ρVECTOR∇
      )WSID WSNAME
WAS CLEAR WS
      )SAVE
2004-08-05 15.16.39 (GMT-4) WSNAME
      3 11 ⎕NA 'CNS'
1
      CNS 'WSNAME' 'C:\PROGRAM FILES\IBMAPL2W\BIN'
C:\PROGRAM FILES\IBMAPL2W\BIN\WSNAME.ans
```

The namespace should placed in a directory that is included in the PATH environment variable.

This namespace is clearly a trivial example; it does not attempt to demonstrate the power of APL2 and such a simple operation could easily be performed in Java. However, the namespace is sufficient for showing how to call APL2 from different types of components under WebSphere without adding any confusing complexities that a sophisticated example might require.

Finally, as you work through the examples, you will find that you are performing some of the same steps over and over. This is deliberate. WebSphere Studio Application Developer can seem daunting at first. But you will probably find that after you work through a few examples, the process of building applications is actually not that complicated. By following the steps of all the examples, you hopefully will gain enough experience with WSAD that you will feel comfortable navigating through the tool's facilities by yourself.

Example 1: Using APL2 from a Servlet

A servlet is a Java class that implements methods described by the J2EE specification. These include methods to handle servlet initialization, destruction, and configuration in addition to methods for processing different types of client requests. By implementing the appropriate methods, a servlet will be suitable for use by a J2EE web server. The server calls the servlet's methods passing argument objects as described by the J2EE specification. These objects can be used to call services provided by the server.

Servlet authors typically do not write all the different types of methods described by the J2EE specification. Instead, they write classes that extend the `javax.servlet.http.HttpServlet` class. The HttpServlet class provides implementations for all the methods required by J2EE. Servlet authors merely have to provide their own implementations of those methods they want to override. The `doGet` and `doPost` methods, which process client requests, are typically the only methods overridden. `doGet` and `doPost` are passed objects representing the request and response. The methods use these arguments to determine the contents of the request and to build the response.

The example demonstrates how to build a servlet that implements the `doGet` and `doPost` methods. These methods are typically used to process requests built by HTML forms. The example servlet expects a request to contain two parameters named `VALUE_1` and `VALUE_2`. The servlet converts the parameters to numeric values, computes their average, and builds HTML containing the result.

There are 10 steps in this example:

Create an Enterprise Application Project
Create a Web Project
Create a Java Package
Create a Servlet
Edit the Servlet's Source Code
Create a Server
Test the Servlet
Change the Context Roots (Optional)
Change the URL Mapping (Optional)
Shutting Down the Server

Create an Enterprise Application Project

The first step is to build an Enterprise Application Project. The Enterprise Application Project will eventually contain a list of all the other projects in the application.

Pull down the File menu and then select New and Enterprise Application Project. This displays the New Enterprise Application Project wizard:
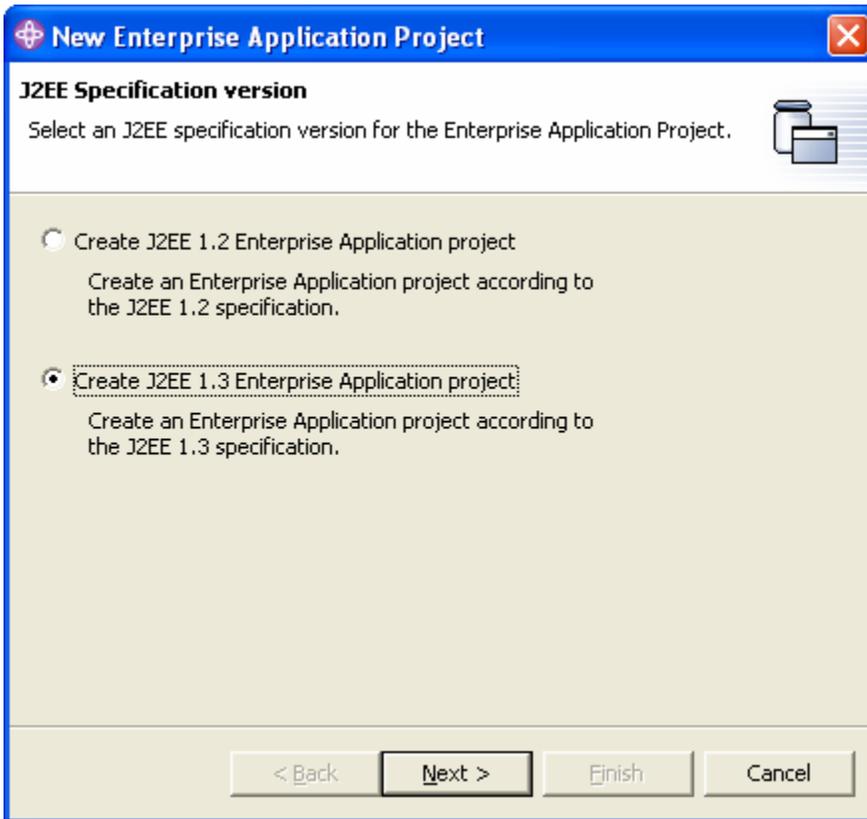
Select Create J2EE 1.3 Enterprise Application project and push Next

The wizard now prompts you for the project name:
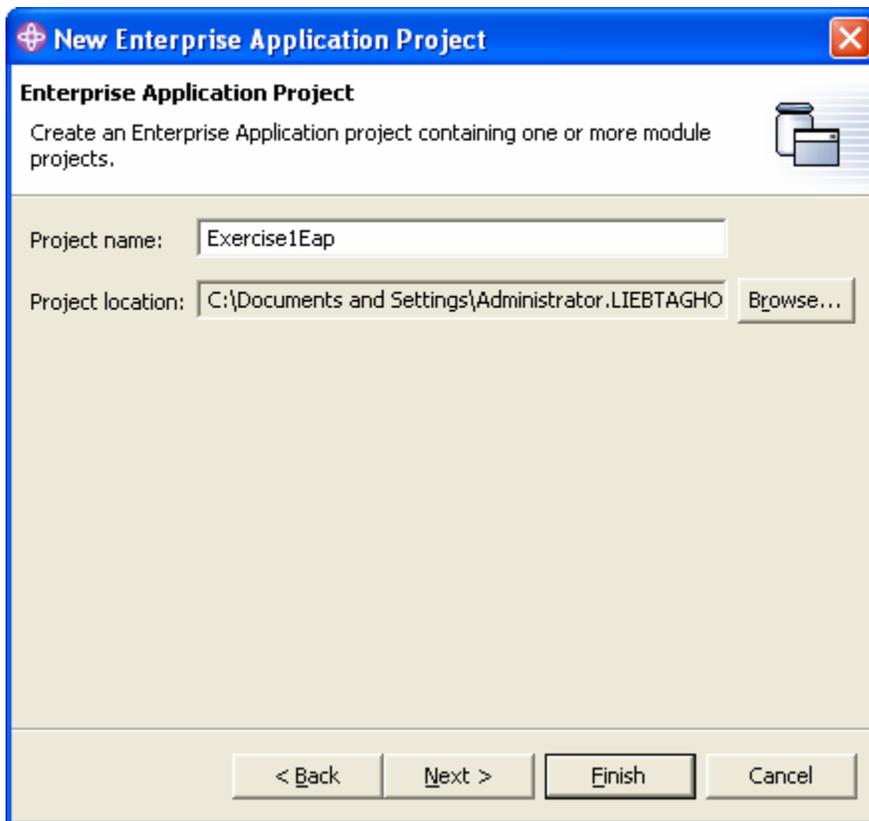
Enter `Exercise1Eap` and push Finish.

The `Exercise1Eap` project should now appear in the list of projects in the Enterprise Applications folder in the J2EE Hierarchy view.

Create a Web Project

Next create a web project. The web project will eventually contain all the web components in the application.
(The Exercise1 application only contains one web component, a servlet.)

Pull down the File menu
Select New
Select Dynamic Web Project

This displays the New Dynamic Web Project wizard:
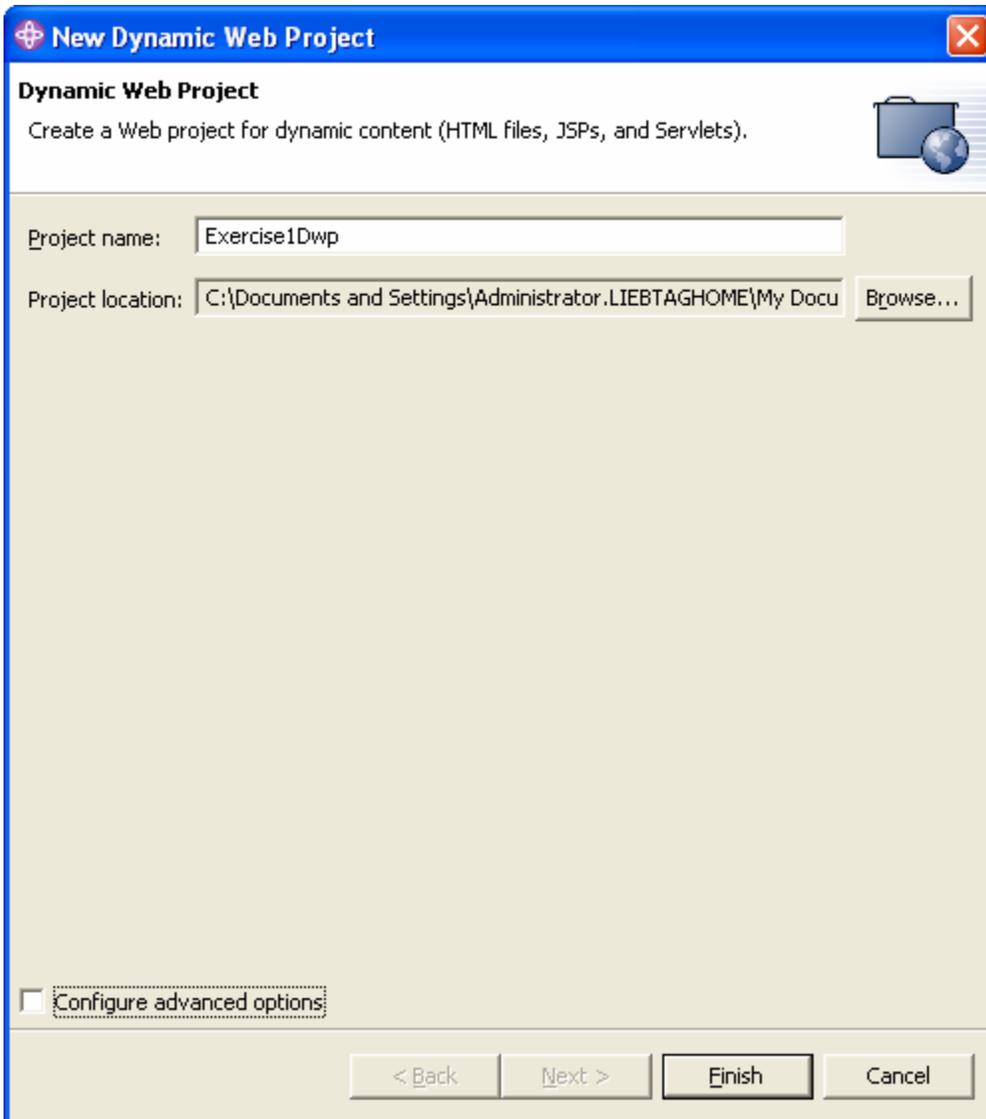
Enter `Exercise1Dwp` in the Project name field and press Finish.

You will be prompted to switch to the Web Perspective.
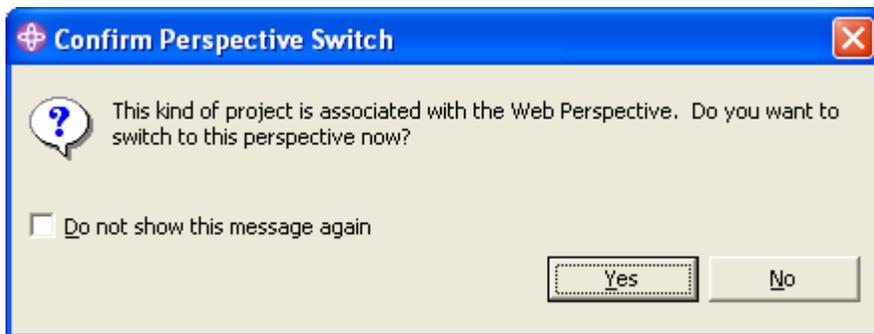
Since the Web Perspective will be useful in a little while, press Yes.

The `Exercise1Dwp` project should now appear in the list of projects in the Web Perspective's Project Navigator view.

Create a Java Package

You must now create a Java package. Java packages contain classes that are related and work together. You will create a package that will eventually contain just one class, a servlet class.

Pull down the File menu and select New and then Other… to display the New wizard:
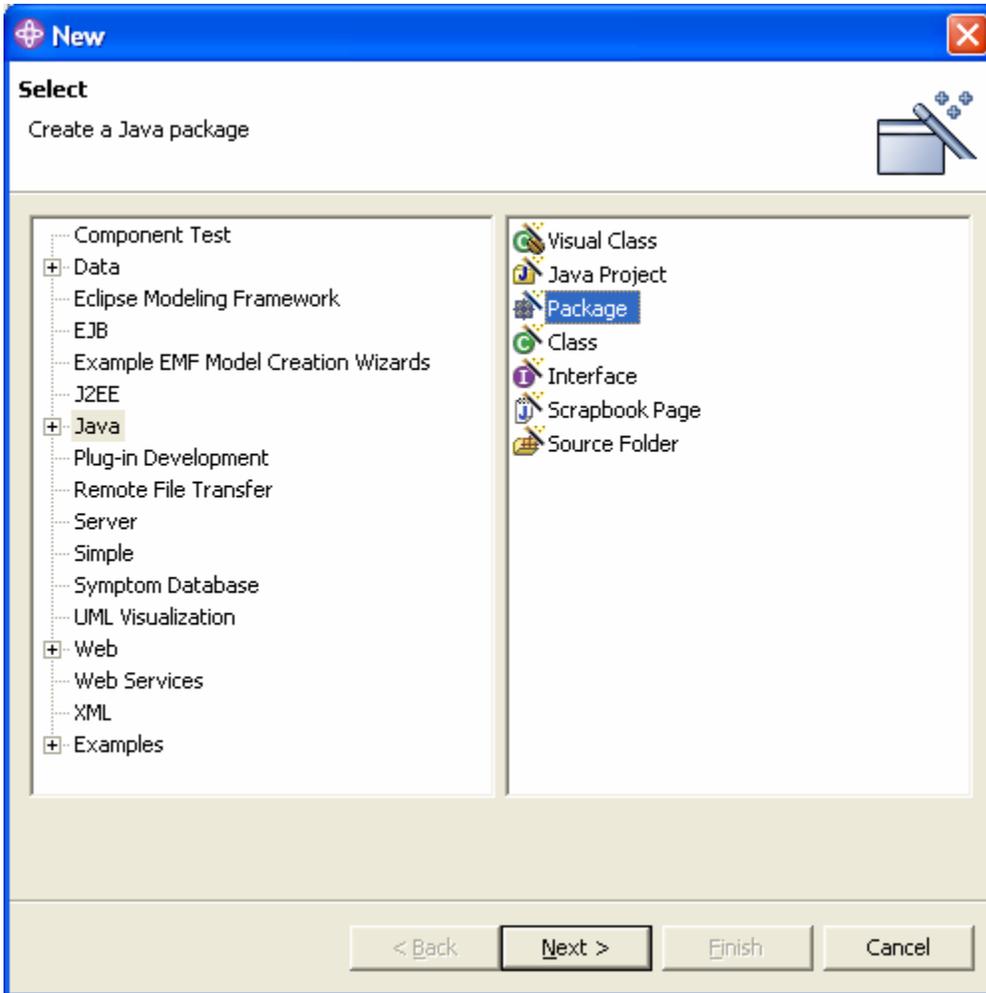
Select Java on the left and Package on the right. Then press Next.

The wizard now prompts you for the package name:
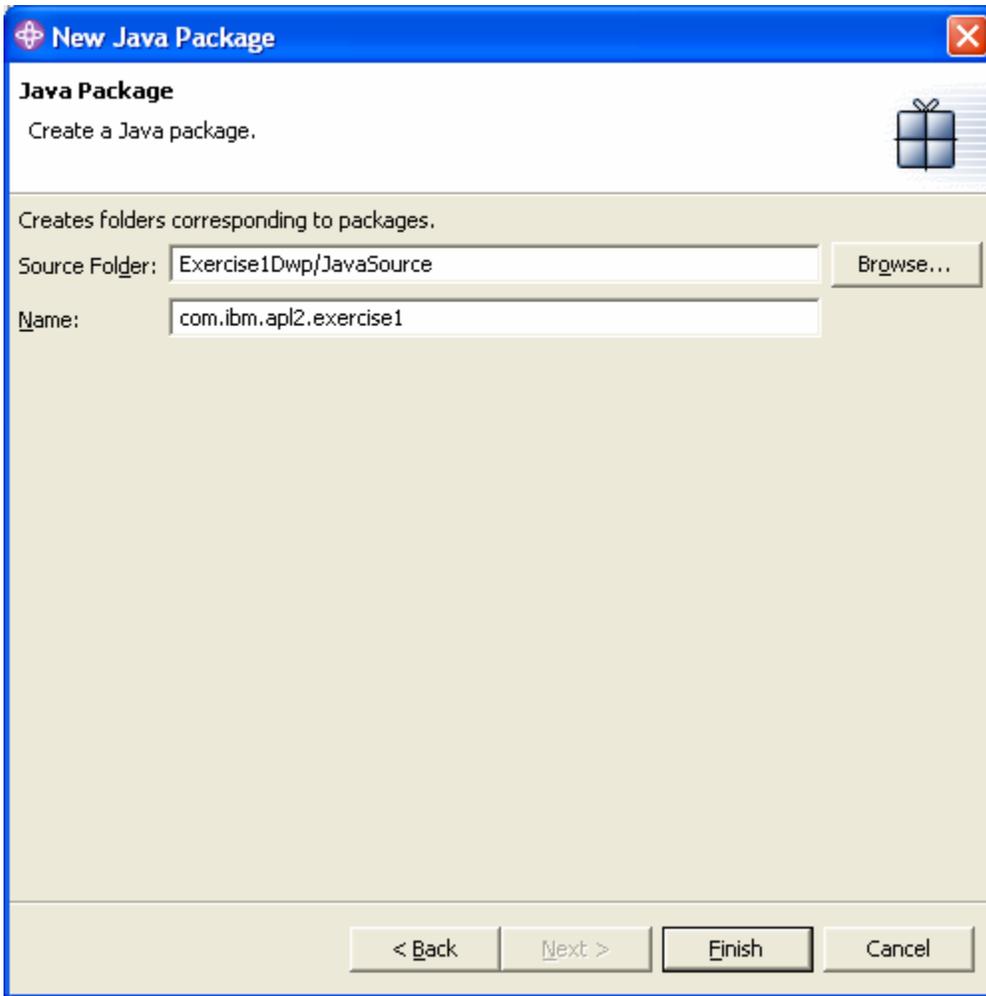
Make sure the Source Folder field shows `Exercise1Dwp/JavaSource`.
Then enter `com.ibm.apl2.exercise1` in the Name field and press Finish

The `com.ibm.apl2.exercise1` package should now appear in the Java Resources folder of the
`Exercise1Dwp` project in the Web Perspective's Project Navigator view.

Create a Servlet

You have now finished creating your application's projects and are ready to create your servlet.

Pull down the File menu and select New and then Servlet to display the New Servlet wizard:
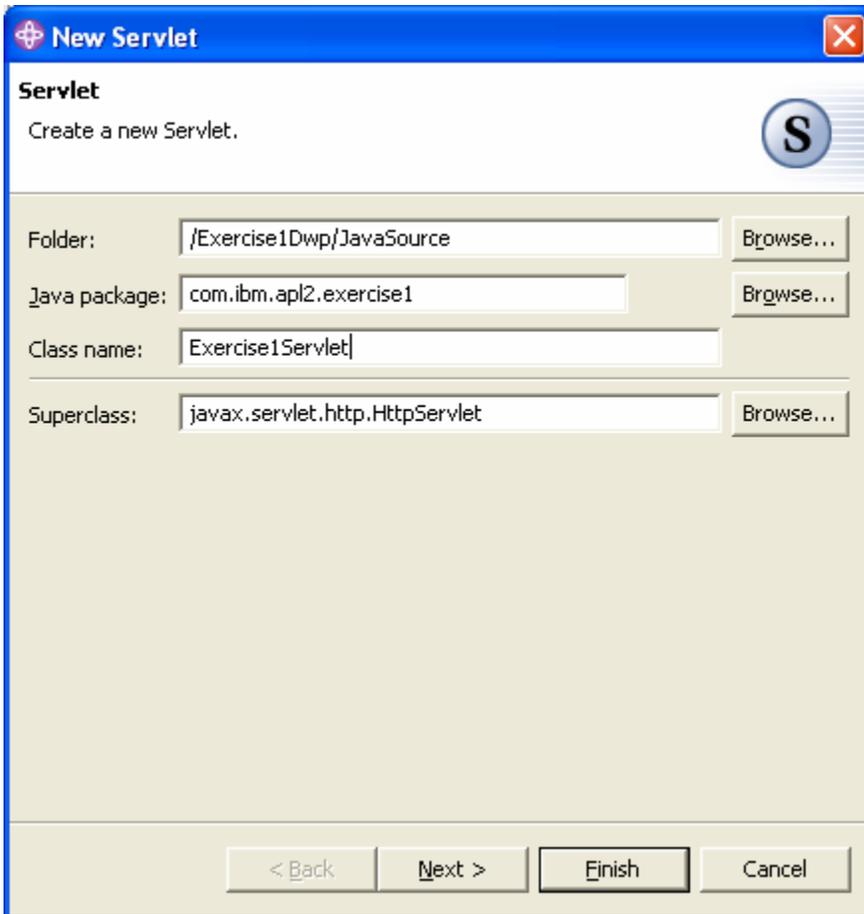
The wizard prompts you to provide a folder, a package name, and a class name. WSAD uses this information to determine where to put the servlet.

Press Browse next to the folder and navigate to the `/Exercise1Dwp/JavaSource` or type it in the Folder field.

Press Browse next to the Java package and navigate to the `com.ibm.apl2.exercise1` or type it in the Java package field.

Type `Exercise1Servlet` in the Class name field.

Notice the servlet's superclass is `javax.servlet.http.HttpServlet`. This is the class that was mentioned earlier and includes most of the servlet's function. We will only override a few methods. Press Next to proceed to this stage in the wizard.

The wizard now prompts you for information about what type of methods the servlet should support. The wizard will automatically generate source code stubs for the methods selected here.
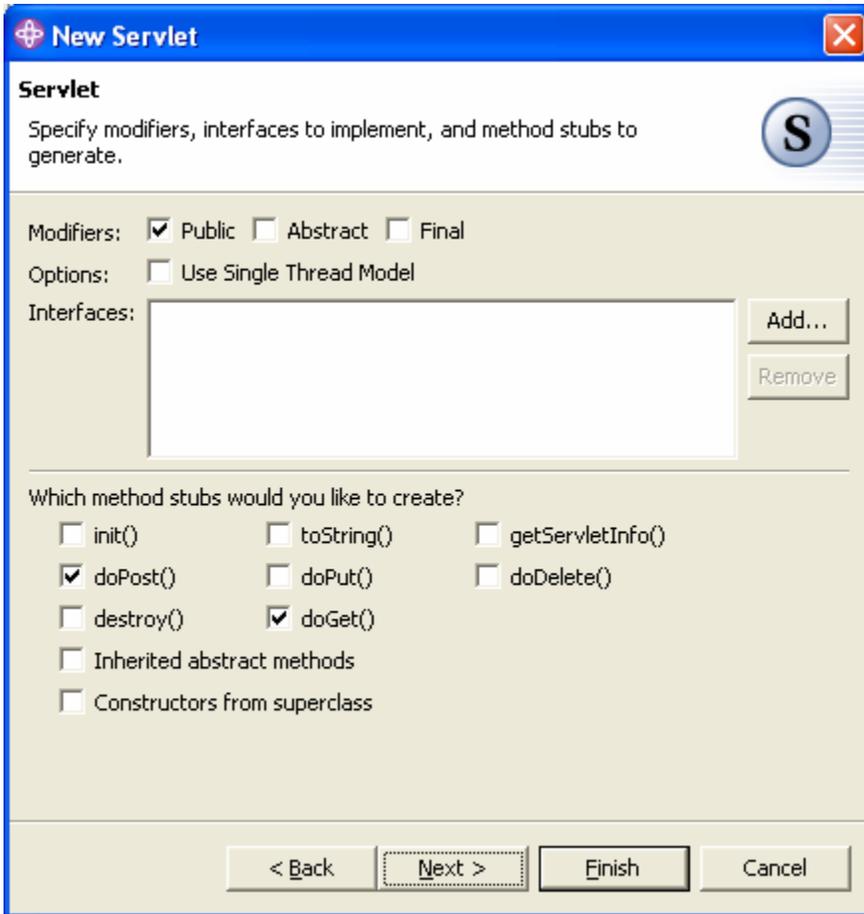
Check Public, doPost, and doGet. Clear all other check boxes and press Next

The wizard now prompts you for whether the servlet should be added to the `web.xml` file.

The `web.xml` file contains the project's deployment descriptor information. The servlet needs to be added to the deployment descriptor in order for it to be deployed to the web server. Since we want this servlet to be deployed, leave this box checked, and press Finish.

The `Exercise1Servlet.java` file should now appear in the `com.ibm.apl2.exercise1` package in the Java Resources folder of the `Exercise1Dwp` project in the Web Perspective's Project Navigator view. The wizard will also open the `Exercise1Servlet.java` file in a Java Editor. It is ready for editing.

Edit the Servlet's Source Code

When the new servlet wizard finished creating the servlet, it opened a new view showing the servlet's source code. You now need to edit this source code.

Add the following line of code to both the `doGet` and `doPost` methods (between the curly braces):

processRequest(req, resp);

Notice both the added lines have an error icon on the left side of the editor view. Position the mouse pointer over the error icon and hold it there a moment. An error message will be displayed.

Press Ctrl+S (or select Save from the File pull-down menu) to save the source code changes.

Click on the Tasks tab in the bottom view. Notice the Tasks view shows the same two error messages. The WebSphere Application Developer performs incremental compilation every time you save changes to files. Any errors discovered during compilation are displayed in the Tasks view. You can double click on an error message in the task view to navigate to the line with the error. You can also use the Tasks view to fix errors.

Right click on either of the error messages.
Select Quick Fix… from the popup menu
Select Create method 'processRequest(..)' and push Ok.

Switch to the Java editor view and notice the `processRequest` method has been added.

Press Ctrl+S and notice the error messages disappear.

Now you need to provide the actual code for the `processRequest` method. Copy and paste the following code into the `processRequest` method:

```
    //* Use the servlet's request object to extract the parameters
    String String1 = req.getParameter("VALUE_1");
    String String2 = req.getParameter("VALUE_2");

    //* Make sure they're not empty
    if (String1 == null) String1 = "0";
    if (String2 == null) String2 = "0";
    if (String1.compareTo("") == 0) String1 = "0";
    if (String2.compareTo("") == 0) String2 = "0";

    //* Extract their numeric values
    double Value1 = 0;
    double Value2 = 0;
    try {Value1 = java.lang.Double.parseDouble(String1);}
    catch (NumberFormatException e) {}
    try {Value2 = java.lang.Double.parseDouble(String2);}
    catch (NumberFormatException e) {}

    //* Calculate the average
    double Average = 0;
    Apl2interp Apl2 = null;
    try {
    Apl2 = new Apl2interp();
            Apl2object Vector = new Apl2object(Apl2,
    new double[]{ Value1, Value2 });
            Apl2.Associate("WSNAME", 11, "AVERAGE");
            Apl2object Result = Apl2.Execute("AVERAGE", Vector);
            Average = Result.doubleValue();
    }
    catch (Apl2exception e) {}
    if (Apl2 != null) try {Apl2.Stop();} catch (Apl2exception e) {}

    //* Use the response object to get a PrintWriter object
    //* and build the HTML result.
    PrintWriter out = resp.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Exercise 1 Average</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<P>");
    out.println("Average: " + Average);
    out.println("</BODY>");
    out.println("</HTML>");
```

Press Ctrl+S and notice several new error messages appear in the Tasks view. The Apl2 and PrintWriter classes can not be resolved because we have not yet imported the packages that contain them. Add the following statements after the package statement at the beginning of the file:

import com.ibm.apl2.*;
import java.io.PrintWriter;

Press Ctrl+S again to save the changes.

Notice the PrintWriter error message disappeared, but the Apl2 messages did not. This is because the compiler does not know where to find the Apl2 classes. To fix this, we must add the apl2.jar file to the path used by the compiler when building classes.

In the Project Navigator view, right click on Exercise1Dwp and select Properties. This will display the Properties dialog for the Exercise1Dwp project:
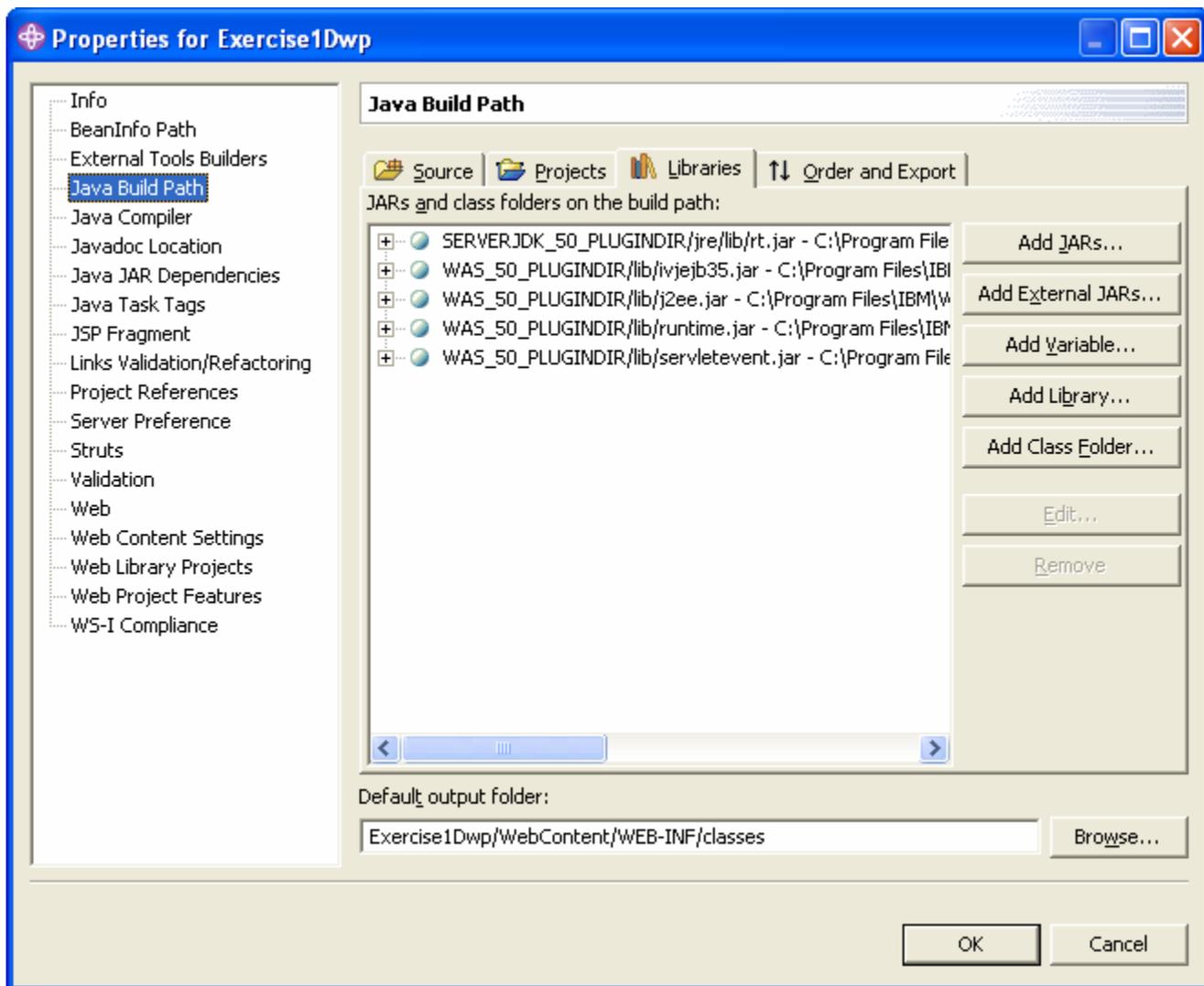
Select Java Build path on the left and the Libraries tab on the right.

Press the Add External JARs… button to open the JAR Selection dialog:
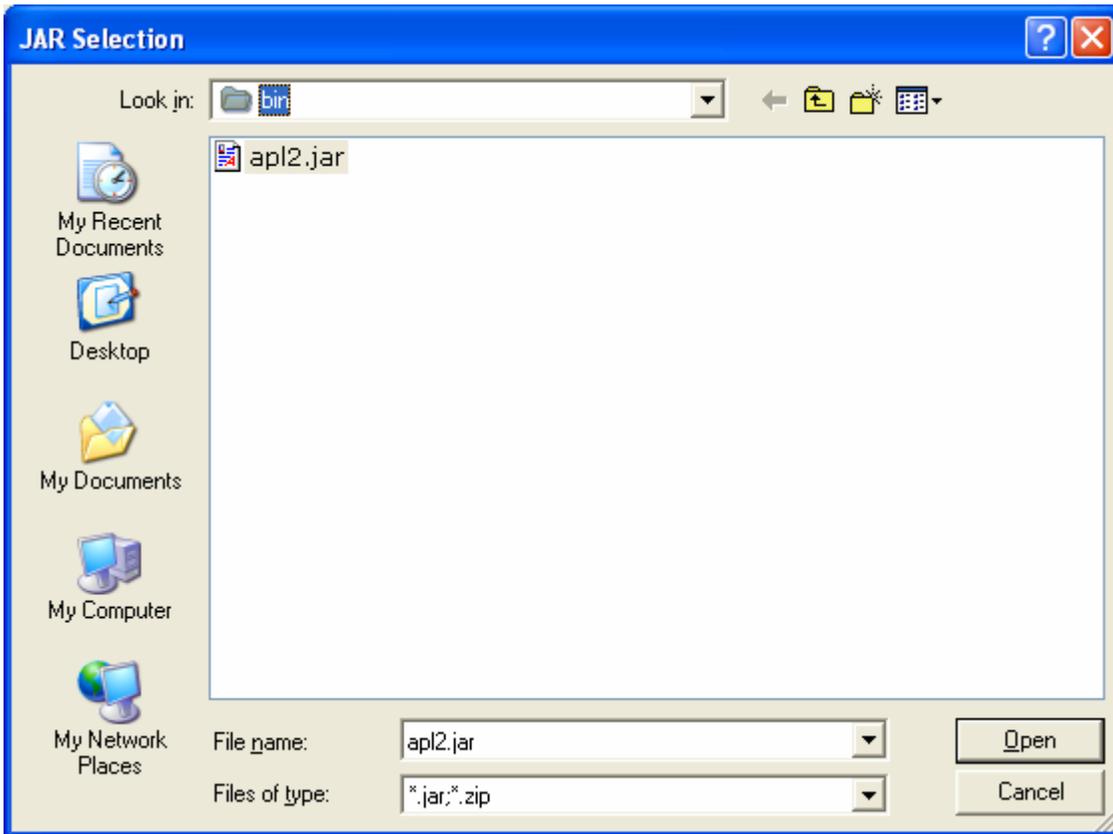
Navigate to the \ibmapl2w\bin folder and select the apl2.jar file. Press Open to close the JAR Selection dialog and then press Ok to close the project's Properties dialog.

The Apl2 error messages should all disappear. However, one new error should appear. The code does not handle IOException exceptions. Use the Tasks view to fix this problem too. Right click on the error message and select Quick Fix…
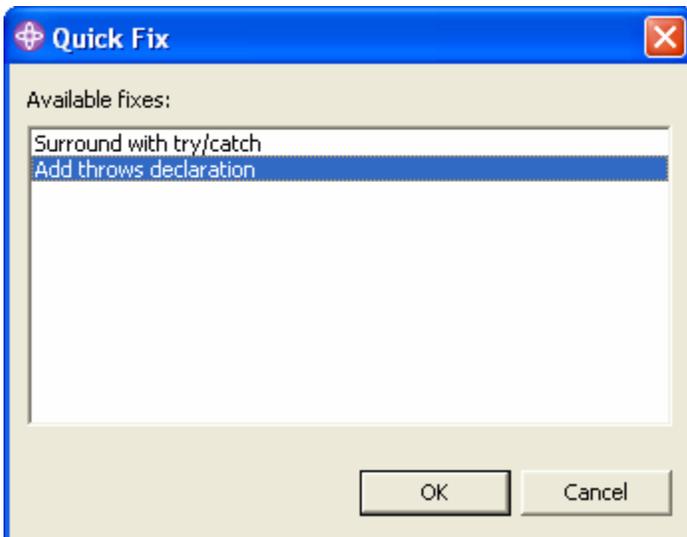
Select Add throws declaration and press Ok. Then, press Ctrl+S to save the changes.. There should now be no error messages in the Tasks view. Close the Exercise1Servlet.java file.

Create a Server

You are now almost ready to test your servlet. The last thing to do is create a server.

First, switch to the Server Perspective by pulling down the Window menu and selecting Open Perspective and then Server

In the Server Configuration view (in the lower left corner of the workbench), right click on the Servers folder. Select New and then Server and Server Configuration to display the Create a New Server and Server Configuration wizard:
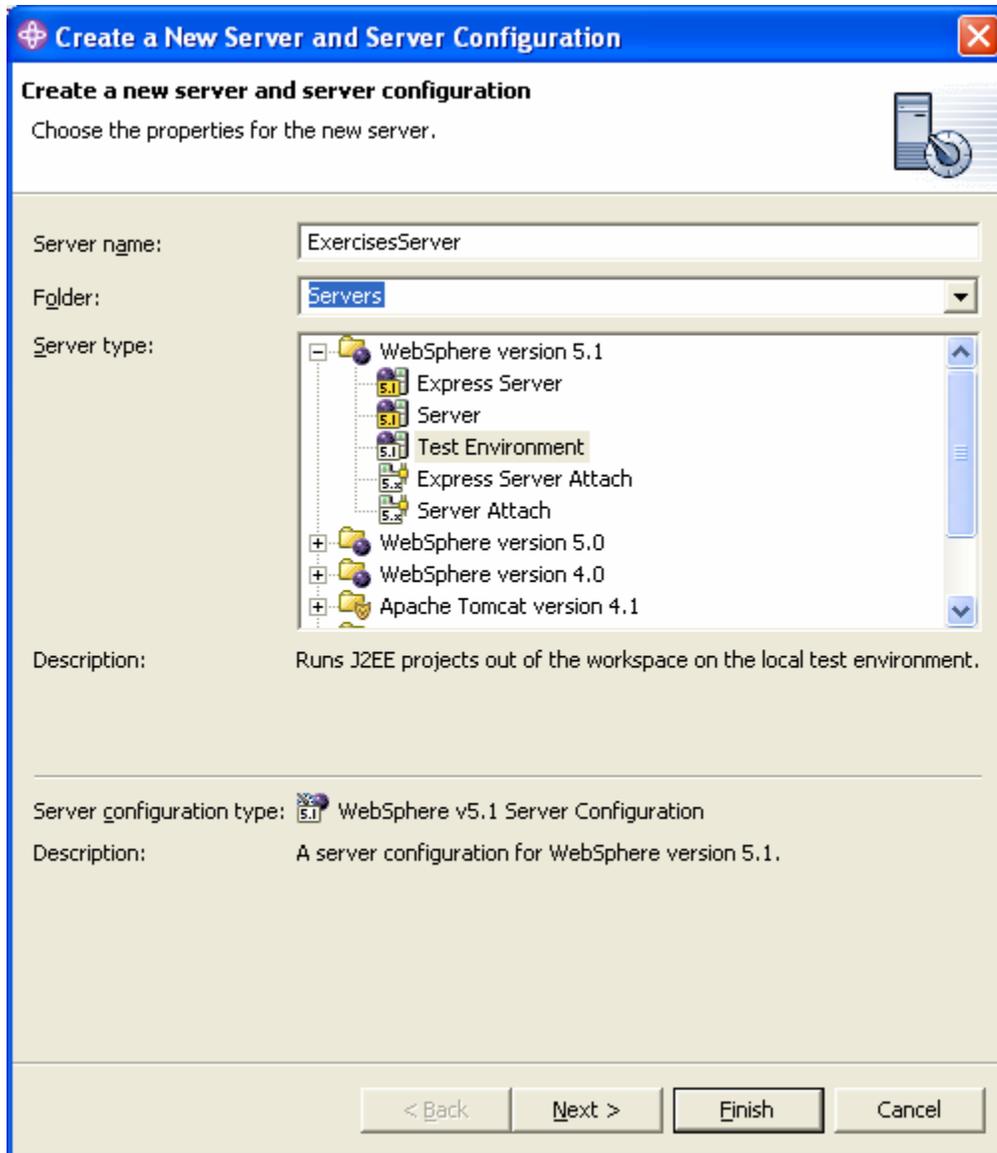
Enter `ExercisesServer` in the server name field. Then, make sure WebSphere version 5.1 and Test Environment is selected, and press Finish.

The `ExercisesServer` should now appear in the Servers folder of the Server Configuration view.

Recall that when we added the servlet's source code that used the Apl2 classes, the compiler needed to be instructed where to find the APL2.jar file. The server will also need this information.

In the Server Configuration view, right click on `ExercisesServer` and select Open. This opens the server's stacked configuration views. Select the Environment tab at the bottom of the stacked views. Add the apl2.jar file to the server's Class Path:

Collapse the ws.ext.dirs section.
Expand the Class Path section.
Press Add External JARs…
Navigate to `\ibmapl2w\bin` and select `apl2.jar`.
Press Open

Notice that there is an asterisk next to the text in the titlebar of the Environment view. This indicates there is an unsaved change in this view. Press Ctrl+S to save the change. The asterisk should disappear.

Close the Environment view by clicking the X in the view's titlebar.

Test the Servlet

Now, you are finally ready to test your servlet.

Switch to the J2EE Perspective. You can click on the J2EE Perspective icon on the Perspective Toolbar on the left edge of the workbench. Then, select the Project Navigator view.

In the Project Navigator view, right click on `Exercise1Servlet.java` and select Run on Server… This displays the Server Selection dialog:
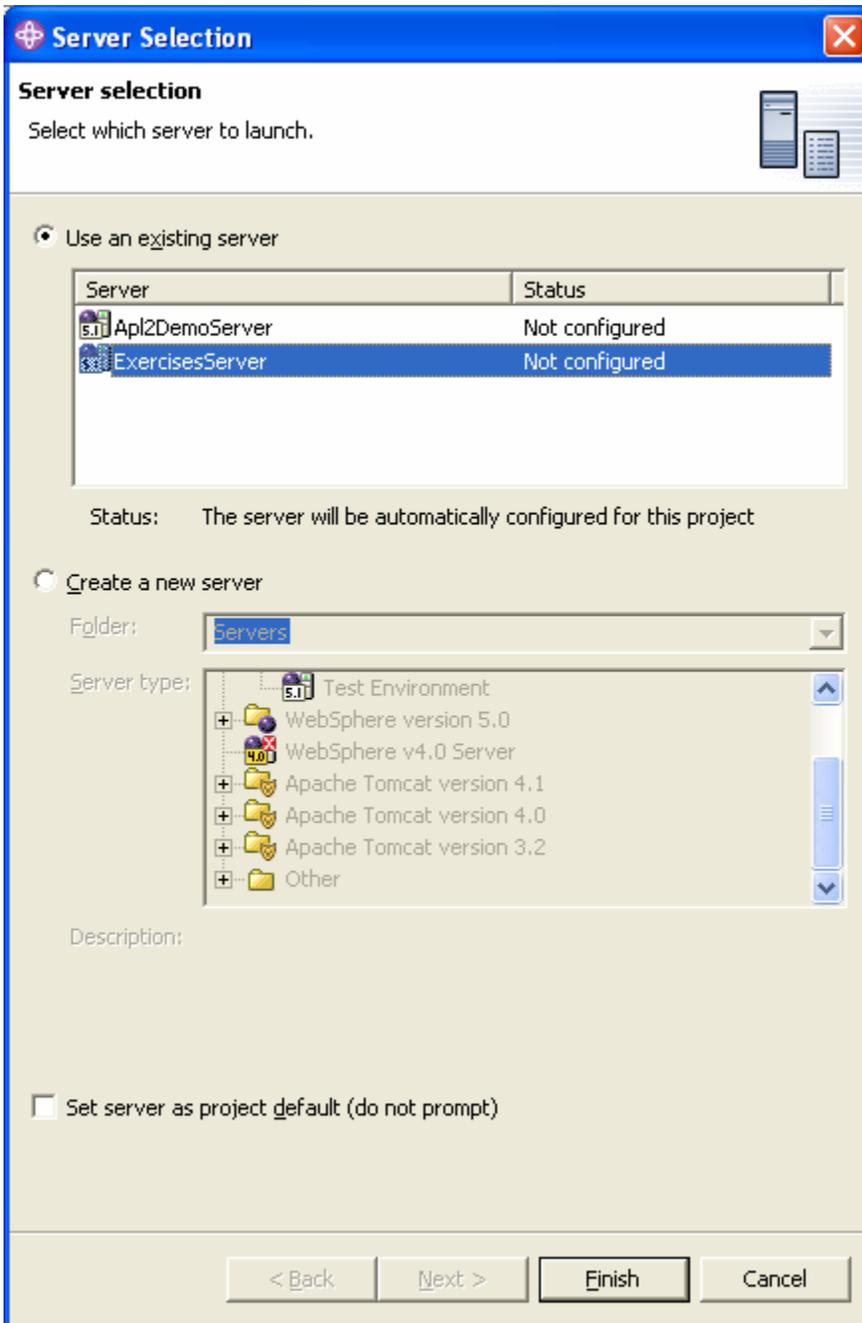
Check Use an Existing Server and select the `ExercisesServer`. Then press Finish to start the server.

Note: The server will attempt to open a port and listen for connections through the network. If you have firewall software installed, you may be prompted to allow the program to have internet access. Answer yes.

Starting the server will take a while. Please be patient. As it is starting, you can select the Console view in the bottom stacked notebook. The Console view shows the progress and error messages produced by the server.

Eventually, the server will start and produce the following message:

Server server1 open for e-business

Once the server starts, a Web Browser view will open showing the results of the servlet.. It should simply display the word Average and the value 0.0. This is because the servlet was not passed any parameters and it used the default values of 0. You could build an HTML page with a form that prompted the user for two values and passed them to the servlet. For now though, to test the servlet, type the following URL in the web browser's address bar:

http://localhost:9080/Exercise1Dwp/Exercise1Servlet?VALUE_1=4&VALUE_2=6

This should display the average of 4 and 6 as the floating point value 5.0.

Note: If you start the APL2 SVP Monitor window and turn on trace, you can see that each time you hit enter in the Web Browser view, APL2 signs on and signs off.

Change the Context Root (Optional)

Notice that the URL in the Web Browser's address field appears to include the name of the `Exercise1Dwp` Dynamic Web Project. Actually, it is not the project name; it is the project's *context root*. The context root is used to group your application's components in the same URL domain. It defaults to the project name.

You may want to change the context root to a more user-friendly name. For example, if the project name is accountingversion2.1, you could use a version independent name like OnlineBanking. To change the context root, use the web project's Properties dialog:

Right click on `Exercise1Dwp` in the Project Navigator view and select Properties.
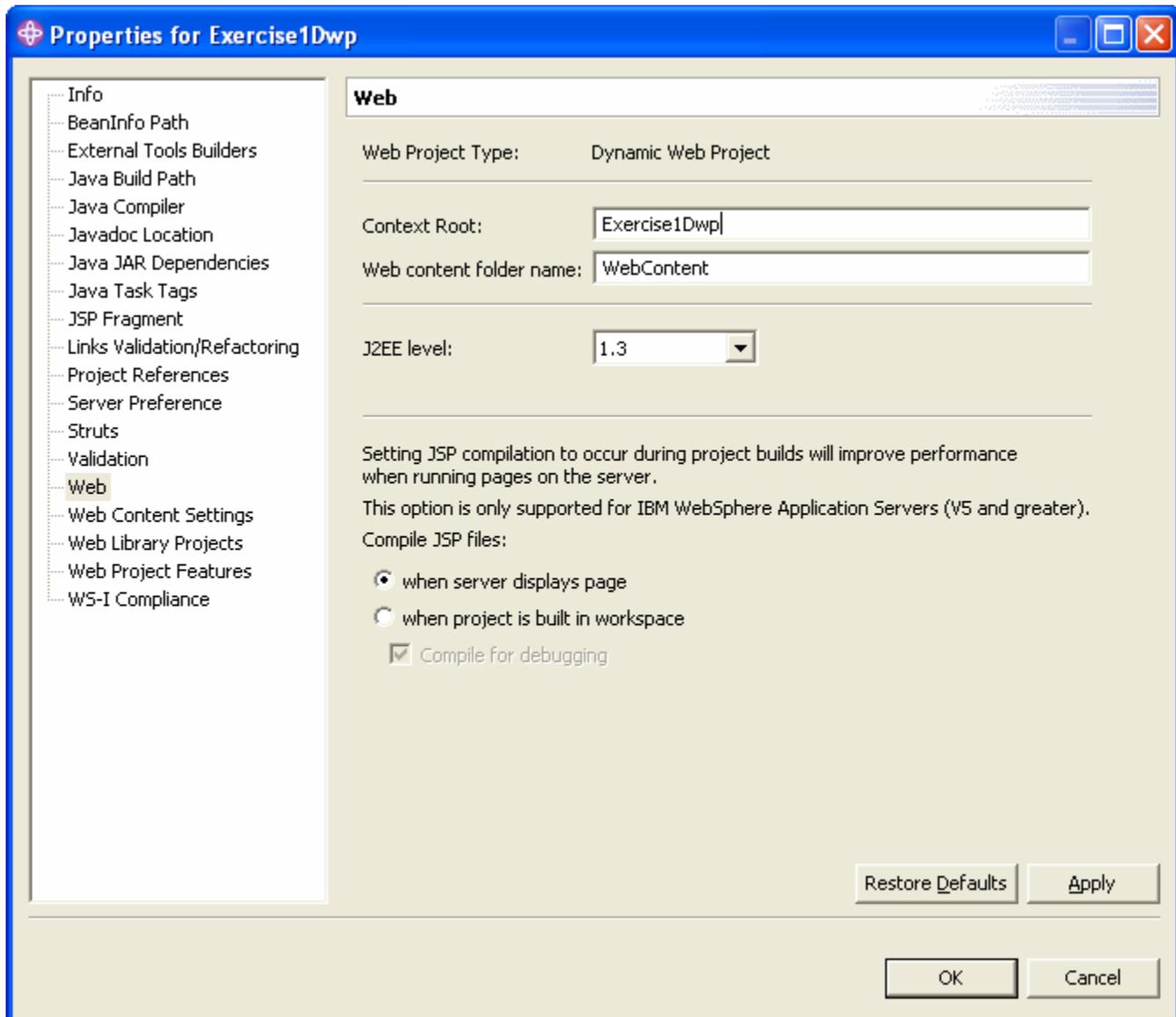
Select Web on the left side of the Properties dialog. The Context Root field displays the context root for this project. You can change it to something more to your liking. Then, press Apply. You will be prompted whether you want links that reference the context root fixed. This is because the `Exercise1Eap` Enterprise

Application Project's deployment descriptor includes the `Exercise1Dwp` Dynamic Web Project's context root. When you are prompted, press Yes.

Press Ok to close the Properties dialog.

When you change the project's properties, this does not effect the already running server. To have your changes effect the server, you must restart either the project or the server.

To restart the project, right click on `ExercisesServer` in the Server Perspective's Server Configuration view and select Restart Project… and then select `Exercise1Eap`. You will see messages in the Console view indicating the project is restarted. You can then type your modified context root to use the servlet.

Change URL Mapping (Optional)

The URL in the Web Browser's address field also appears to include the name of the servlet. Again, it is not the servlet name. Rather, it is the servlet's *URL mapping*. The URL mapping is used to map the text used in URLs to the servlet name. The URL mapping defaults to the name of the servlet.

You may want to change the URL mapping to a more user-friendly name. For example, if the servlet name is balanceservletversion7.3, you could use a version independent name like AccountBalances. You can modify the URL mapping in the project's employment descriptor.

Navigate to the J2EE Perspective and double click on `Exercise1Dwp` in the J2EE hierarchy. This opens the employment descriptor views for the project. They are stacked in a tabbed notebook.

Select the Servlets tab and then select `Exercise1Servlet`.

Notice the servlet's URL mapping is listed on the right. You can change it here. For example, if you replaced `/Exercise1Servlet` with `/MyServlet`, the URL to use would become:

http://localhost:9080/Exercise1Dwp/MyServlet?VALUE_1=4&VALUE_2=6

Use the Add button to add new context roots. Use the Remove button to remove them. After making changes, press Ctrl+S to save your changes.

When you change the URL mapping in the project's deployment descriptor, the change does not effect the already running server. To have your changes take effect, you must again either restart the project or the server.

To restart the project, right click on `ExercisesServer` in the Server Configuration view and select Restart Project… and then select `Exercise1Ezp`. You will see the messages in the Console view indicating the project is restarted. You can then use your modified context root.

Shutting Down the Server

This completes exercise 1 except for one last step.

When you are satisfied that the servlet is working properly you can close the Web Browser view and stop the server. To stop the server, navigate to the Server Perspective and the Server Configuration view. Right click on the `ExercisesServer` and select Stop. You can monitor the processing of the stop request in the Console view.

Example 2: Using APL2 from a Java Server Page

A Java Server Page, or JSP, is an HTML file with imbedded Java code. When a request is received for a JSP, the application server detects tags that delimit Java code in the HTML. The server automatically generates a servlet from the code and the HTML. The server then runs the servlet.

There are several tags to delimit Java code. These include:

```
<%@   %>   Directives to control the servlet generation and compilation
<%!   %>   Declarations of variables and methods
<%    %>   Code fragments
<%=   %>   Code expressions
<%--  --%> Comment
```

The servlet generated and run by the server automatically includes objects named request and response which the JSP code can use to call services provided by the server.

For more information about JSPs, consult `http://java.sun.com/products/jsp`.

The example demonstrates how to build a JSP that prompts the user for two values. The JSP converts the values to numbers, uses APL2 to compute their average, and returns HTML containing the result.

There are 6 steps in this example:

Create an Enterprise Application Project
Create a Web Project
Create a Java Server Page
Edit the JSP's Source Code
Test the JSP
Shutting Down the Server

Create an Enterprise Application Project

Navigate to the J2EE Perspective and the J2EE Hierarchy view.

Select File and New and Enterprise Application Project to display the New Enterprise Application Project dialog.

Select Create J2EE 1.3 Enterprise Application project and press Next.

Type `Exercise2Eap` in the Project name field and press Finish.

`Exercise2Eap` should now appear in the Enterprise Applications folder in the J2EE Hierarchy view.

Create a Web Project

Select File and New and Dynamic Web Project to display the New Dynamic Web Project dialog.

Type `Exercise2Dwp` in the Project name field.

Check Configure advanced options and press Next.

Ensure the EAR project field contains `Exercise2Eap`.

Notice you could set the project's context root in this dialog.

Press Finish.

You will be prompted to switch to the Web Perspective. Since the Web Perspective will be useful in a little while, press Yes.

`Exercise2Dwp` should now appear in the Project Navigator view of the Web Perspective.

Create a Java Server Page

You have now finished creating the exercise's projects and are ready to create your JSP.

Select File and New and JSP File to display the New JSP File dialog.
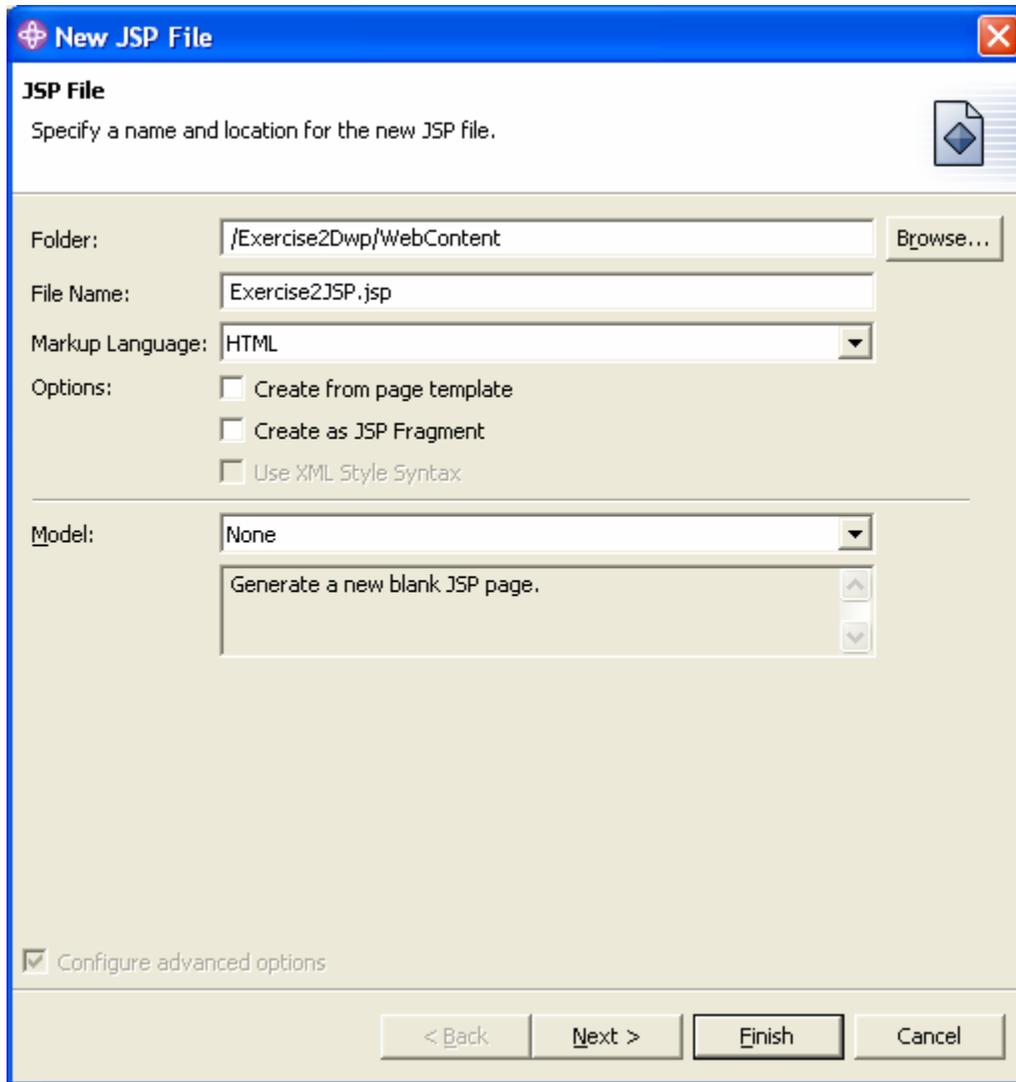
Ensure /`Exercise2Dwp/WebContent` is in the Folder field.

Type `Exercise2JSP.jsp` in the File name field and press Finish.

The `Exercise2JSP.jsp` file should now appear in the `Exercise2Dwp/WebContent` folder in the Web Perspective's Project Navigator view.

A new editor view should also open showing the JSP. Notice there are three tabs in the stack of editor views. The Design tab is a WYSIWYG page editor. The Source tab is a source code editor. The Preview tab displays how the page will look in a browser. Switch to the Source view of the JSP.

Edit the JSP's Source code

You now need to edit the JSP's source code.

The automatically generated JSP's body contains one line: `<P>Place content here.</P>`

Use copy and paste to replace the body with the following code:

```
<H1>Exercise 2 JSP</H1>
<H2>Average of Two Numbers</H2>
<FORM method="POST" action="/Exercise2Dwp/Exercise2JSP.jsp">
<%
String String1 = request.getParameter("VALUE_1");
String String2 = request.getParameter("VALUE_2");
if (String1 == null) String1 = "0";
if (String2 == null) String2 = "0";
if (String1.compareTo("") == 0) String1 = "0";
if (String2.compareTo("") == 0) String2 = "0";
double Value1 = 0;
double Value2 = 0;
try { Value1 = java.lang.Double.parseDouble(String1); }
catch (NumberFormatException e) {}
try { Value2 = java.lang.Double.parseDouble(String2); }
catch (NumberFormatException e) {}
%>
<P>Value 1: <INPUT name="VALUE_1" value="<%=Value1%>" maxlength="20"
        size="20" type="text">
<P>Value 2: <INPUT name="VALUE_2" value="<%=Value2%>" maxlength="20"
        size="20" type="text">
<P><INPUT name="SUBMIT" type="submit" value="Calculate">
</FORM>
<%
double Average = 0;
Apl2interp Apl2 = null;
try {
        Apl2 = new Apl2interp();
        Apl2.Associate("WSNAME", 11, "AVERAGE");
        Apl2object Vector = new Apl2object(Apl2, new double[] { Value1, Value2 });
        Apl2object Result = Apl2.Execute("AVERAGE", Vector);
        Average = Result.doubleValue();
} catch (Apl2exception e) {}
if (Apl2 != null)
        try {Apl2.Stop();} catch (Apl2exception e) {}
%>
<P>
<H4>Calculated Average: <%=Average%></H4>
```

Press Ctrl+S to save your changes. Notice that several messages appear in the Tasks view indicating the compiler can not resolve the Apl2 classes. Add the apl2.jar file to the `Exercise2Dwp` properties:

Right click on `Exercise2Dwp` in the Project Navigator view. Select Properties to display the Properties for Exercise2Dwp dialog.

Select Java Build Path on the left. Select the Libraries tab on the right.
Push the Add External JARs… button.
Navigate to `\ibmapl2w\bin`, select `apl2.jar`, and press Open.
Press Ok to close the Properties for Exercise2Dwp dialog.

Notice the messages still do not go away. This is because we need to add code to the JSP to import the classes. There is a `<%@ page` directive near the top of the JSP. This directive provides information to the server about how to generate the servlet. Add a line with the following import value to the page directive:

import="com.ibm.apl2.*"

Press Ctrl+S to save the change. There should now be no messages in the Tasks view.

Close the `Exercise2JSP.jsp` file.

Test the JSP

You are now ready to test your JSP.

In the Project Navigator view, right click on `Exercise2JSP.jsp` and select Run on Server… to display the Server Selection dialog:

Check Use an Existing Server and select the `ExercisesServer`. Then press Finish to start the server.

Starting the server will again take a while. When the server has started and is ready for e-business, the Web Browser view will show the JSP's form. The input fields and result will display the default values of 0. Type some numbers and press enter.

Notice the URL again appears to contain the project and JSP names. Once again, these are not resource names. Rather, they are the context root and URL mapping. They can be changed for JSPs just as they were for servlets.

Shutting Down the Server

This completes exercise 2 except for shutting down the server.

Close the Web Browser view.
Navigate to the Server Perspective and the Server Configuration view.
Right click on the `ExercisesServer` and select Stop.

Example 3: Using APL2 from an Enterprise Java Bean

A Enterprise Java Bean, or EJB, is a set of Java classes that together implement business logic and can operate in an EJB container. An EJB container is a portion of an application server which provides a set of services that is defined by the J2EE specification.

EJB containers provide a wealth of services including EJB location, transaction processing, messaging, mail, XML parsing, multithreading, resource management, persistent data management, and security control. By using these services, authors can reuse standard code rather than having to rewrite these facilities. In addition, EJBs are platform independent; they are written to use the J2EE EJB container platform rather than any particular operating system platform. In addition, using the server's services, as provided by the container, an EJB can be concurrently used by many different types of applications. EJBs can be written so they can be distributed across multiple machines and so load balanced by the server.

There are three types of Enterprise Java Beans:

Session beans represent business logic
Entity beans represent state information
Message beans process messages

Since APL2 is well suited for implementation of complicated algorithms such as are used in business logic, the example demonstrates how to use APL2 in a session bean.

EJBs are designed to be distributed. The EJB architecture includes features which enable a consistent approach to this distribution. EJBs are never used directly by clients. Instead, they provide a set of interfaces through which clients can access the EJB. These interfaces work with the server and handle EJB location and network communication (in the case of distributed EJBs.)

The first two interfaces are called the *Home* and *Object* interfaces. The home interface's create method is used to create instances of EJB classes. The create method returns an object interface. The client uses the object interface to make method calls to the instance of the EJB.

The technique used for locating the home interface depends on whether the EJB is on the same or another machine. The *Local* interface is used to locate EJBs on the same machine. The *Remote* interface is used to locate EJBs on other machines. The EJB container provides a lookup service called the *JNDI* service. The client uses a JNDI service and the EJB name to locate a local or remote interface. The client then uses the local or remote interface to locate the EJB's home interface.

When an EJB is written, application methods must be placed on either the local or remote interface, or both, before clients can access the methods.

This use of home, object, local, and remote interfaces enables the EJB container running in the server to manage the creation and lifetime of EJBs. Unlike servlets, an EJB is not created for each request. Instead, the container creates and reuses EJBs as they are needed and manages them in a pool. The lifetime of an EJB may be as long as an application is deployed and started on a server. An EJB may, and often will be, used by many clients during its lifetime. This is useful for APL2. The APL2 interpreter can be started when the EJB is created and reused for multiple requests rather than started and stopped for each request.

Although the EJB architecture sounds complicated, and it is, developers do not have to write all these different interfaces and their supporting code. WSAD generates almost all the code. Developers merely have to write the code that implements their business logic and identify those methods which should be accessible to local and remote clients.

The example demonstrates how to build a simple EJB that starts an APL2 interpreter and associates a name with a function in a namespace when it is created, has a method on the remote interface which calls the APL2 function, and shuts down APL2 when the EJB is destroyed. The example uses the Universal Test Client, a component of WSAD which can be used to test EJBs.

There are 7 steps in this example:

Create an Enterprise Application Project
Create an EJB Project
Create a Java Package
Create an Enterprise Java Bean
Edit the EJB's Source code
Generate the Deployment and RMIC Code
Test the EJB
Shutting Down the Server

Create an Enterprise Application Project

Navigate to the J2EE Perspective and the J2EE Hierarchy view.

Select File and New and Enterprise Application Project to display the New Enterprise Application Project dialog.

Select Create J2EE 1.3 Enterprise Application project and press Next.

Type `Exercise3Eap` in the Project name field and press Finish.

`Exercise3Eap` should now appear in the Enterprise Applications folder in the J2EE Hierarchy view.

Create an EJB Project

Unlike servlets and JSPs, EJBs are not stored in Web Projects.   EJBs are stored in EJB projects. This is because EJBs are not web components and are managed by EJB containers rather than web containers.

Select File and New and EJB Project to display the New EJB Project dialog:
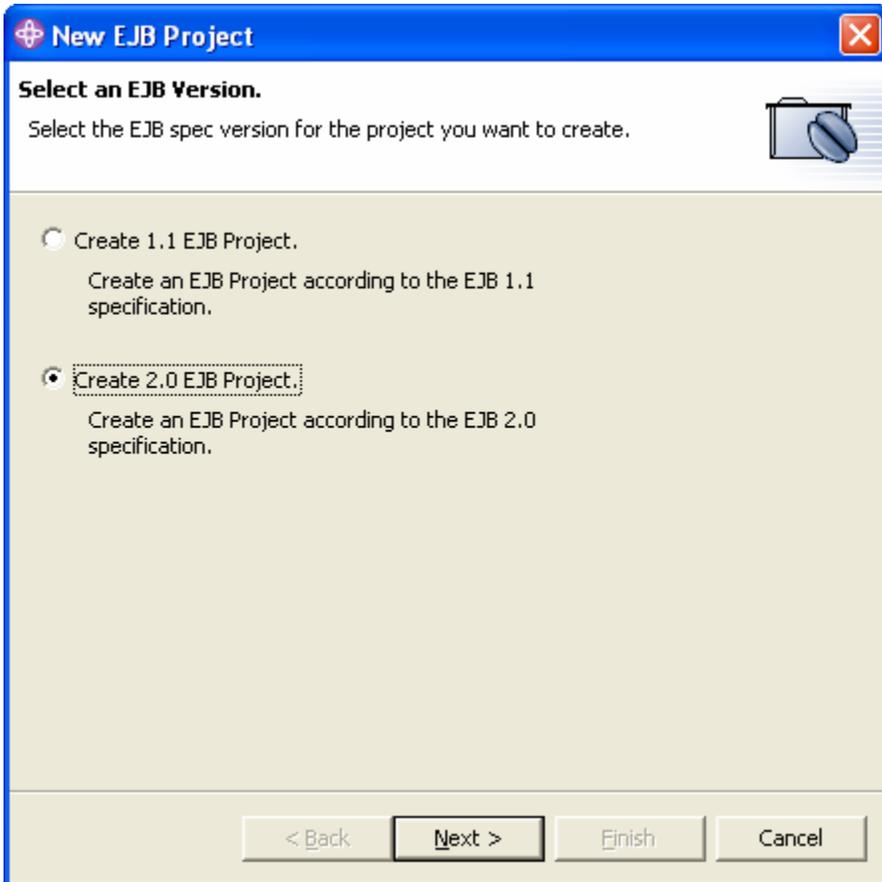
Select Create 2.0 EJB Project and press Next to proceed to the next page of the wizard where you will enter the project name.
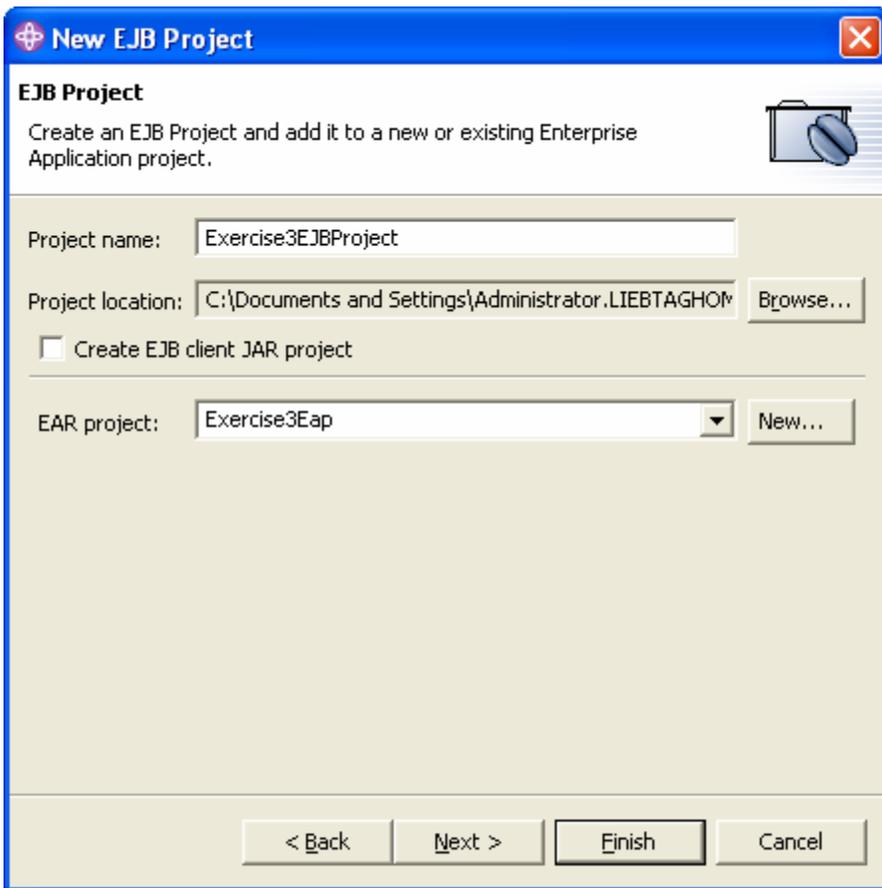
Type `Exercise3EJBProject` in the Project name field.

Select `Exercise3Eap` in the EAR project field. The field is named the EAR project field because Enterprise Application Projects are stored in Enterprise Archives, or EAR, files.

Press Finish.

`Exercise3EJBProject` should now appear in the EJB Modules folder in the J2EE Hierarchy view of the J2EE Perspective.

Create a Java Package

You must now create a Java package to hold your EJB's source code.

Select File and New and Other… to display the New dialog:
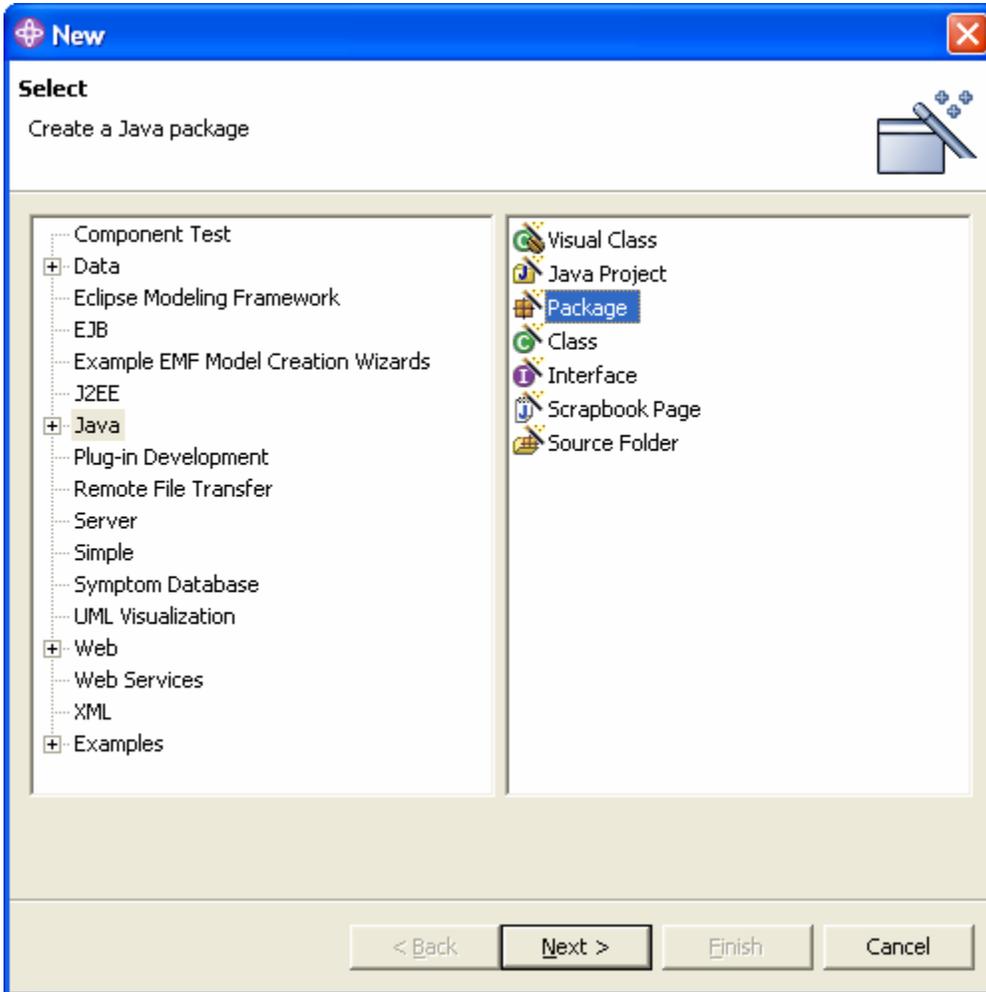
Select Java on the left and Package on the right and press Next.
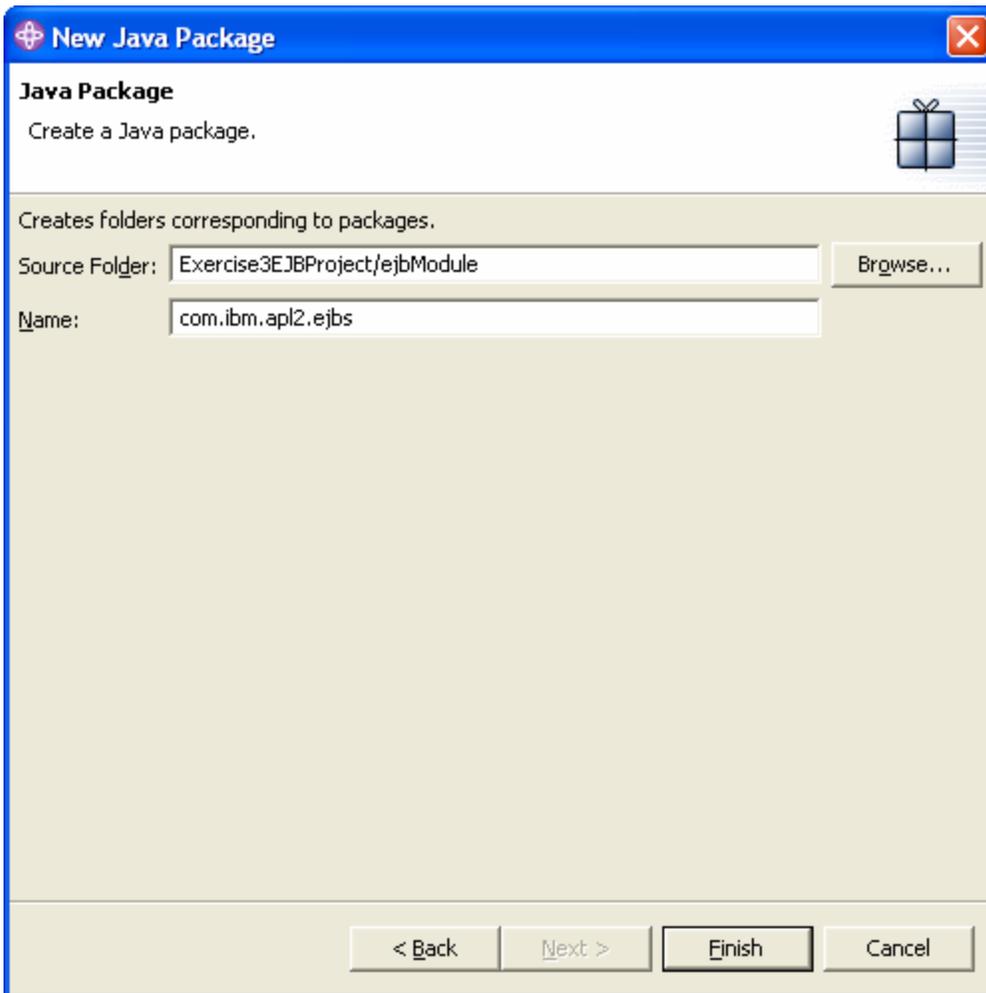
Type `Exercise3EJBProject/ejbModule` in the Source folder field. (You can also use the Browse…
button to navigate to this folder.)

Type `com.ibm.apl2.ejbs` in the Name field and press Finish.

If you switch to the Project Navigator view, you can see that the `com.ibm.apl2.ejbs` package now appears
in the `Exercise3EJBProject` project's `ejbModule` folder.

Create an Enterprise Java Bean

You have now finished creating the exercise's projects and are ready to create your EJB.

Select File and New and Enterprise Bean to display the New Enterprise Bean dialog:
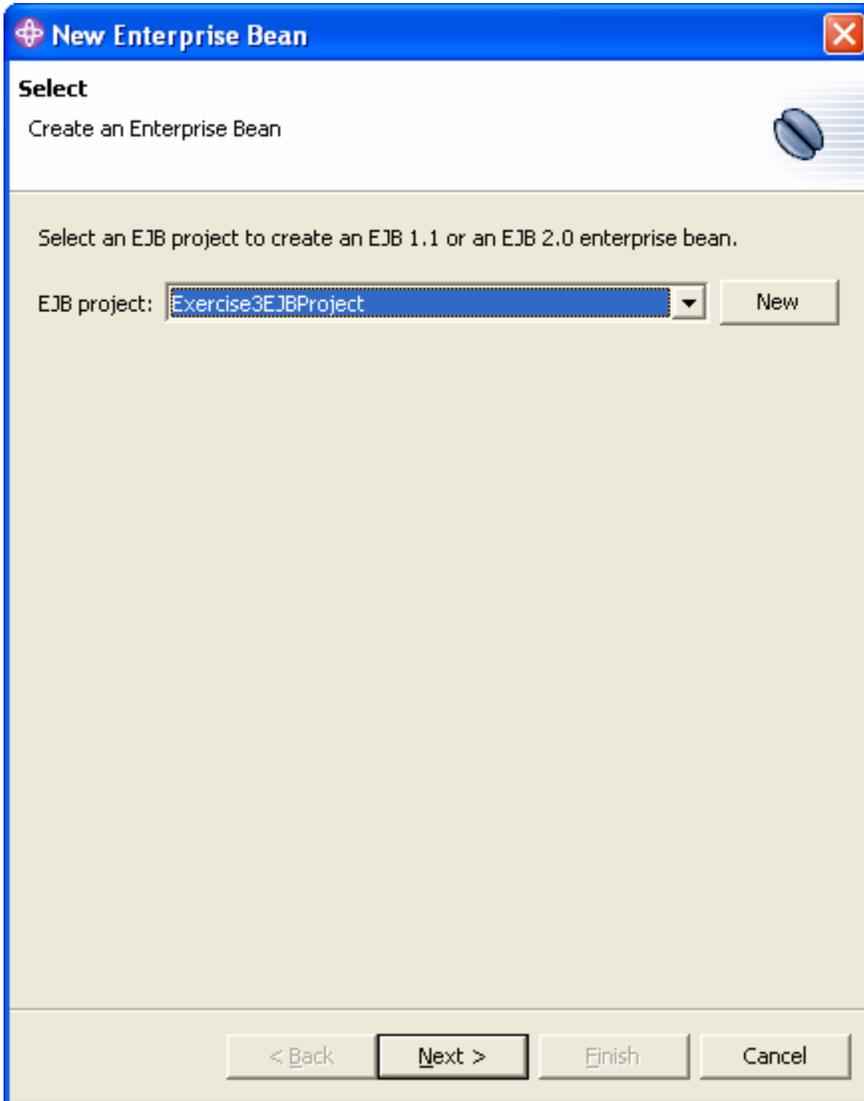
Select `Exercise3EJBProject` in the EJB project field and press Next to proceed to the next stage of the wizard in which you will select the EJB type.
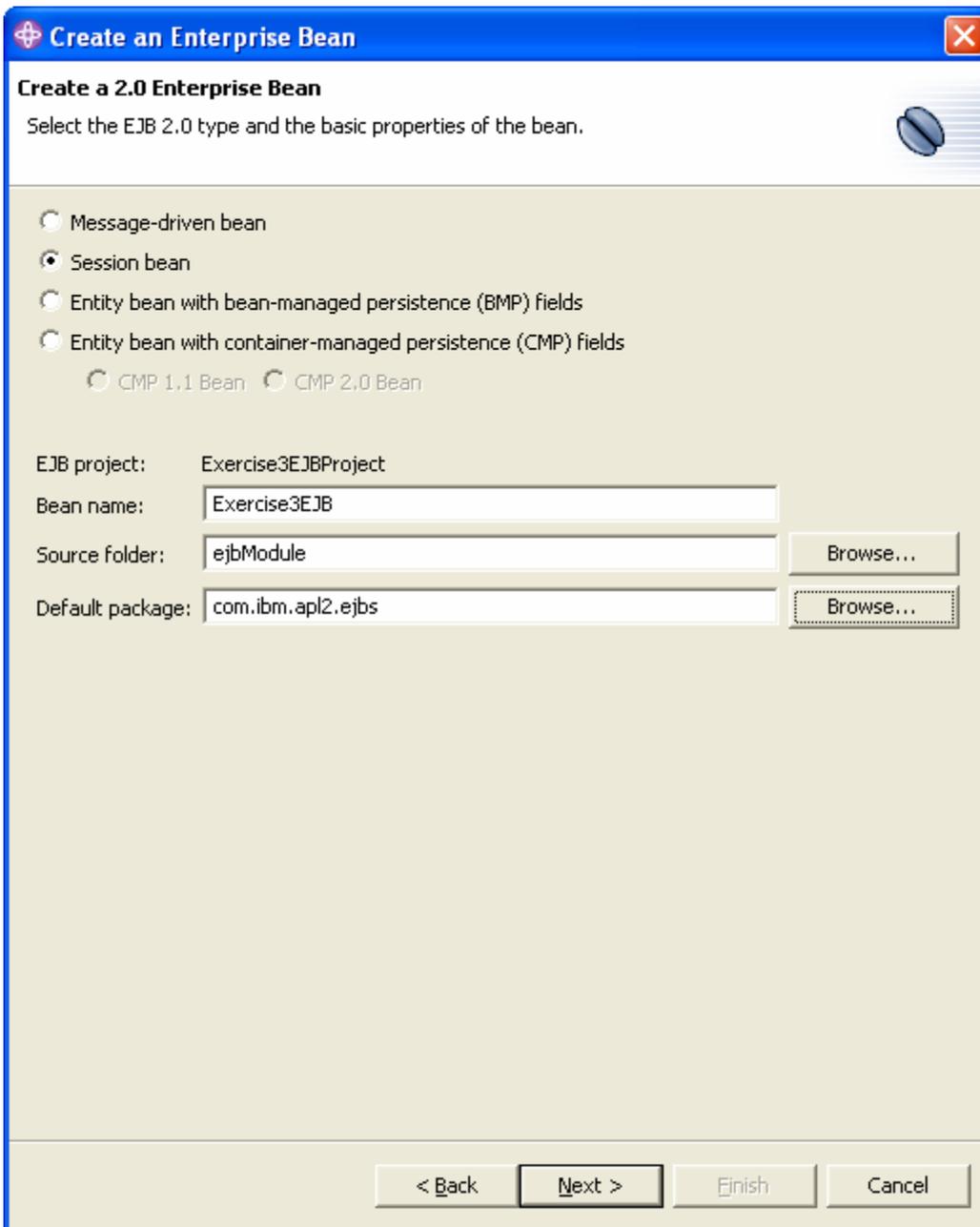
Recall that there are four types of Enterprise Java Beans: message-driven beans, session beans, and entity beans. Session beans are the appropriate type to use with APL2.

Check Session Bean.

Type `Exercise3EJB` in the Bean name field.

Verify that the Source folder field contains `ejbModule`.

Type `com.ibm.apl2.ejbs` in the Default package field and press Next to proceed to the next stage in which you will specify the attributes of the session bean.

There are two types of session beans: stateful and stateless. Stateful beans use EJB container services to store data that persists across uses; stateless beans do not. These services can not be used to store APL2 data because APL2 data is not serializable so session beans that use APL2 should be stateless.

Select the Session type: Stateless.

Each time a client uses an object interface to call an EJB method, it is called a transaction. Session beans can either manage their own transactions or let the container do it. Our bean will use the container's transaction management.

Select the Transaction type: Container.

Although the example uses WSAD's Universal Test Client which runs on the same machine as the EJB, the steps shown later in the example demonstrates building an EJB with a method on the remote interface.

Check Remote client view.

Notice that the wizard has used the bean name to generate a lot of other types of names. These names will be automatically used as the client locates the local, remote, and home interfaces.

Press Finish to generate the EJB.

Switch to the J2EE Hierarchy view. Notice that the Exercise3EJB bean now appears in the Session Beans folder in the `Exercise3EJBProject`. Notice also that three resources appear inside `Exercise3EJB` bean. These three files contain the Java source code that make up the EJB.

Edit the EJB's Source code

The J2EE hierarchy view shows that the New Enterprise bean wizard generated three resources:

Exercise3EJBHome
Exercise3EJB
Exercise3EJBBean

For this example, only the third, `Exercise3EJBBean` needs to be edited. So, double click on `Exercise3EJBBean` to open the Java Editor. Notice the wizard generated several methods for you. These methods are called by the server to initialize, terminate, and otherwise manage the EJB. There are two we care about: `ejbCreate` and `ejbRemove`. These methods are called when the EJB is created and destroyed. We will insert code in them to start and stop an APL2 interpreter. We will also add some instance variables and a new method which a client can call.

Add the following statement after the package statement at the top of the file:

import com.ibm.apl2.*;

Add the following two statements after the declaration of mySessionCtx:

private Apl2interp Slave = null;
private Apl2object Function = null;

Add the following code to the `ejbCreate` method:

```
    Slave = null;
    try {
            Slave = new Apl2interp();
            Function = new Apl2object(Slave, "AVERAGE");
            Slave.Associate("WSNAME", 11, Function);
    } catch (Apl2exception Exception) {
            if (Slave != null) {
                    try {
                            Slave.Stop();
                    } catch (Apl2exception StopException) {
                    }
                    Slave = null;
            }
            throw new javax.ejb.CreateException(
                    "Unable to initialize APL2 environment");
    }
```

Add the following code to the `ejbRemove` method:

```
    try {
            Function.Free();
            Slave.Stop();
    } catch (Apl2exception Exception) {
    }
```

And finally, add the following method to the end of the file (before the last curly brace):

```
    public double Average(double[] Array) {
            double Result = 0;
            try {
                    Apl2object aplintArray = new Apl2object(Slave, Array);
                    Apl2object aplAverage = Slave.Execute(Function,
                            aplintArray);
                    aplintArray.Free();
                    Result = aplAverage.doubleValue();
                    aplAverage.Free();
            } catch (Apl2exception e) {
                    System.out.println("Apl2exception caught");
                    System.out.println("Exception message: " +
                            e.getMessage());
                    System.out.println("Event: " + e.Type + " " + e.Code);
            }
            return Result;
    }
```

Notice that many of the inserted lines are flagged with error icons on the left side of the editor. Press Ctrl+S to save the changes and notice that many error messages also appear in the Tasks view.   They are all produced because the compiler can not find the Apl2 classes. Fix this by adding the apl2.jar file to the project:

Right click on `Exercise3EJBProject` and select Properties.
Select Java Build Path on the left and the Libraries tab on the right.
Press the Add External JARs… button.
Navigate to `\ibmapl2w\bin`, select `apl2.jar`, and press Open.
Press Ok to close the Properties for Exercise3EJBProject dialog.

The messages should all disappear.

If it is not already active, switch to the J2EE Perspective. Then, select the Outline view in the stack of views in the lower left corner.

The Outline view shows all the fields and methods in the EJB you are editing. Notice that the list includes the `Average` method you added to the EJB.

The `Average` method needs to be added to the remote interface so that it can be used by clients.

Right click on the `Average` method in the Outline view. Select Enterprise Bean, and Promote to Remote Interface.

Notice the `Average` method now has a small R next to it. This indicates the method is available through the remote interface.

Close the Exercise3EJBBean.java file.

Generate the Deployment and RMIC Code

Although we have specified the EJB should be a stateless session bean and promoted the Average method to the remote interface, we do not yet have all the utility code in place to actually implement these protocols and enable the EJB to communicate with the EJB container and for clients to use Remote Method Invocation (RMI) to call the EJB. To generate this code, we instruct WSAD to generate deployment and RMIC code.

In the J2EEHierachy view, right click `Exercise3EJBProject` and select Generate and then Deployment and RMIC Code… to display the Generate Deployment and RMIC Code dialog:
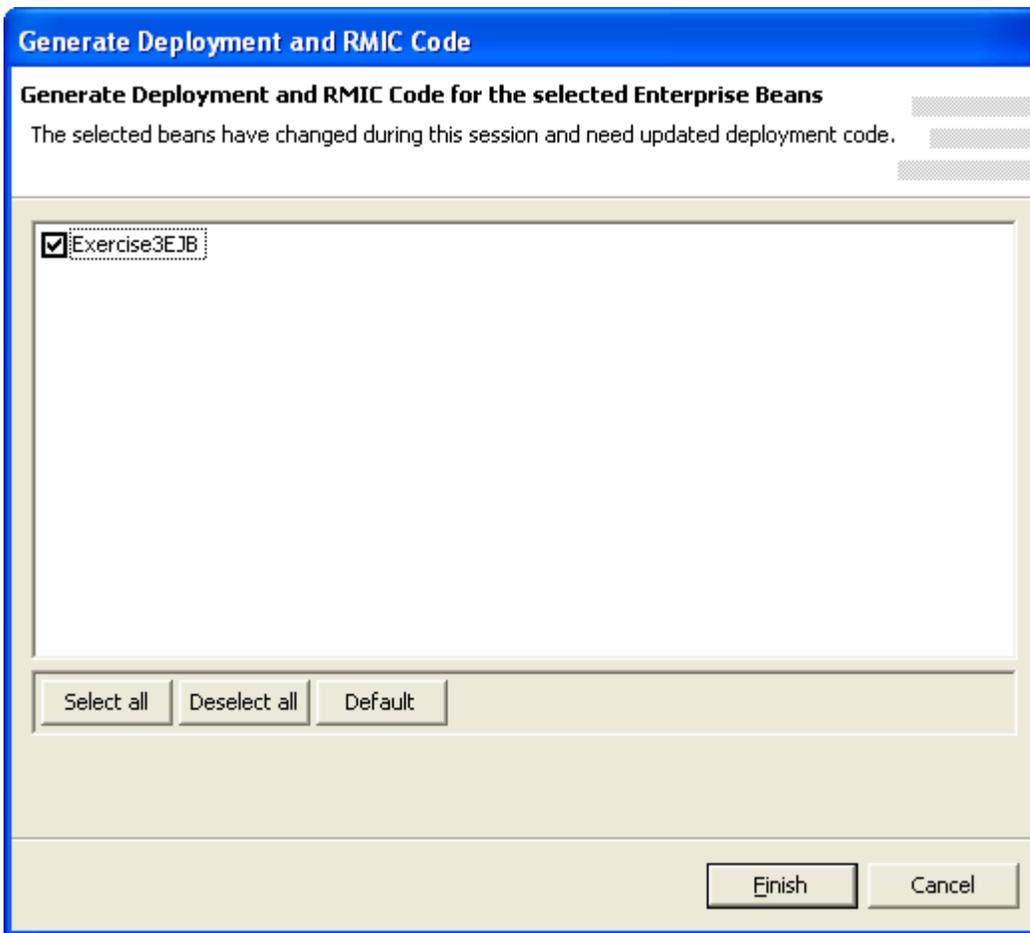
Select `Exercise3EJB` and press Finish.

Using the Project Navigator view, look in the `com.ibm.apl2.ejbs` package. Notice there are now a large number of Java source files. These files implement the required interfaces.

Test the EJB

You are now ready to test your EJB.

Switch to the Web Perspective's Project Navigator view.
Expand Exercise3EJBProject, ejbModule, and com.ibm.apl2.ejbs.
Right click `Exercise3EJB.java` and select Run on Server… to display the Server Selection dialog.
Check Use an Existing Server, select `ExercisesServer`, and press Next to display the Select Tasks dialog:

Check Deploy EJB Beans and press Finish.

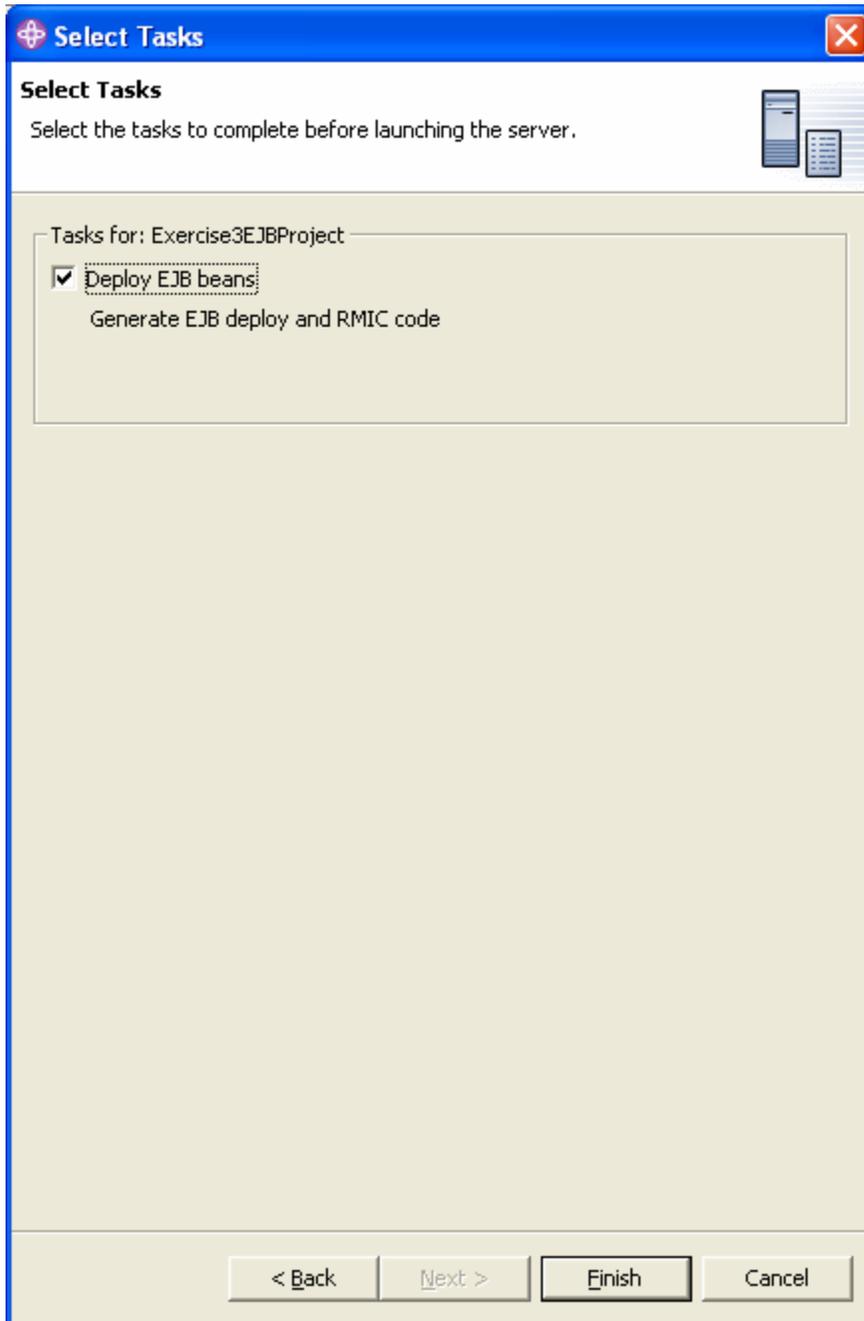Starting the server will again take a while. Please be patient again.

Unlike the JSP, the EJB does not include any HTML. It can't even be called directly by a client like the servlet. So, WSAD will automatically start a program called the Universal Test Client. The Universal Test Client is used to test EJBs. It enables you to manually call an EJB's home interface's create method to create an instance of the EJB and then call methods on the remote interface.

In the left pane of the Test Client, is a section called EJB References which should contain `Exercise3EJB`. Expand `Exercise3EJB`. This will display the `Exercise3EJBHome` interface. Expand it too. This will display the `Exercise3_ejbCreate` method. Click on this method.
This will make an Invoke button appear in the right pane.

The Test Client view should now look like this:
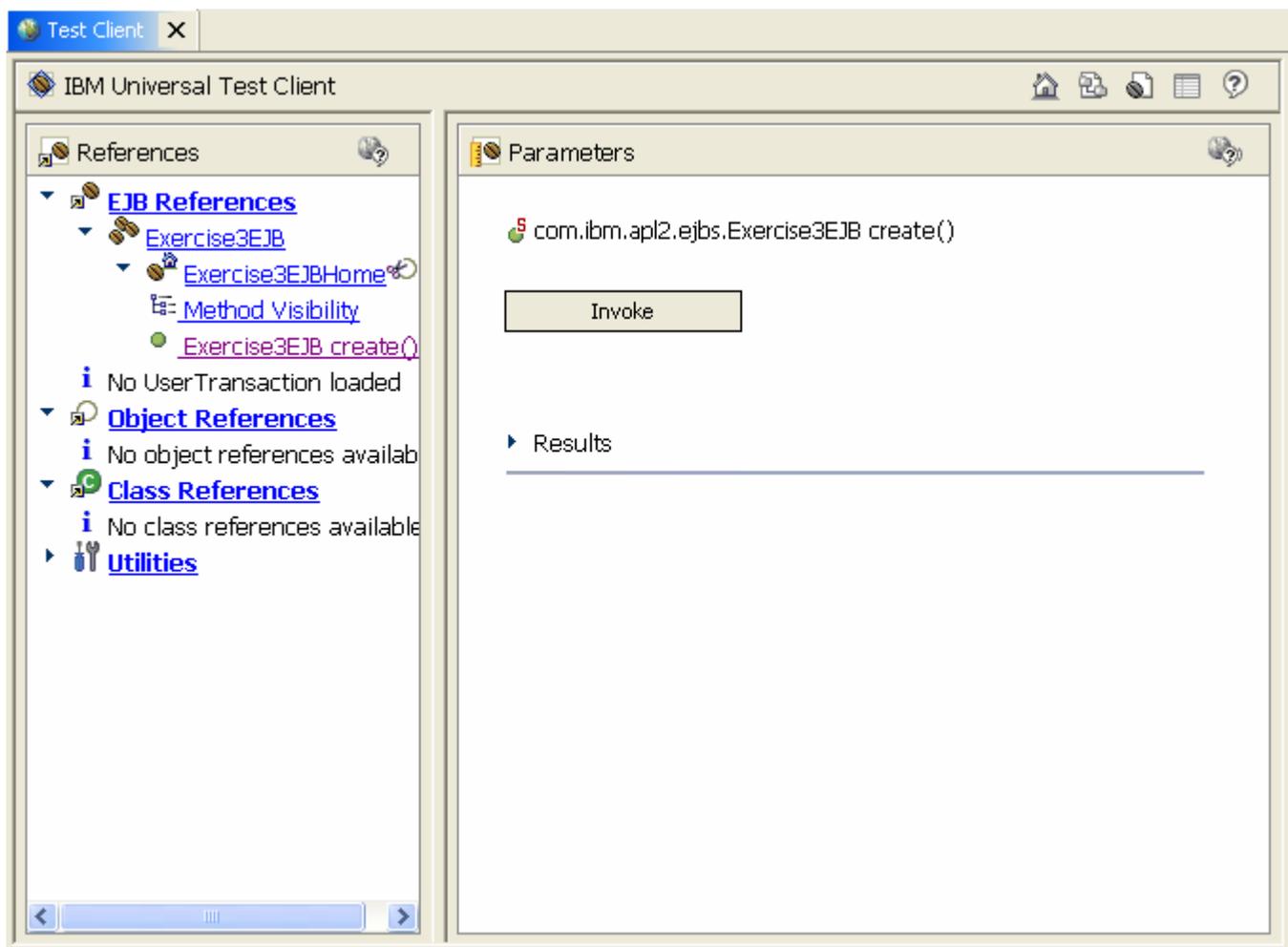
Press the Invoke button to invoke the `ejbCreate` method to create an instance of the EJB.

Invoking the `ejbCreate` method will return an `Exercise3EJB` object. Press the Work with Object button to work with the EJB. This will make an `Exercise3EJB_1` reference appear in the list of EJB references.

Expand `Exercise3EJB_1`. The `Average` method will now appear.

Click on the `Average` method. The right pane should now change and prompt you for parameters for the `Average` method.

Click on the word Expand.

Click on the word Add twice.

Type two numbers in the parameter entry fields.

Press the Invoke button.

The Test Client should now look something like this:
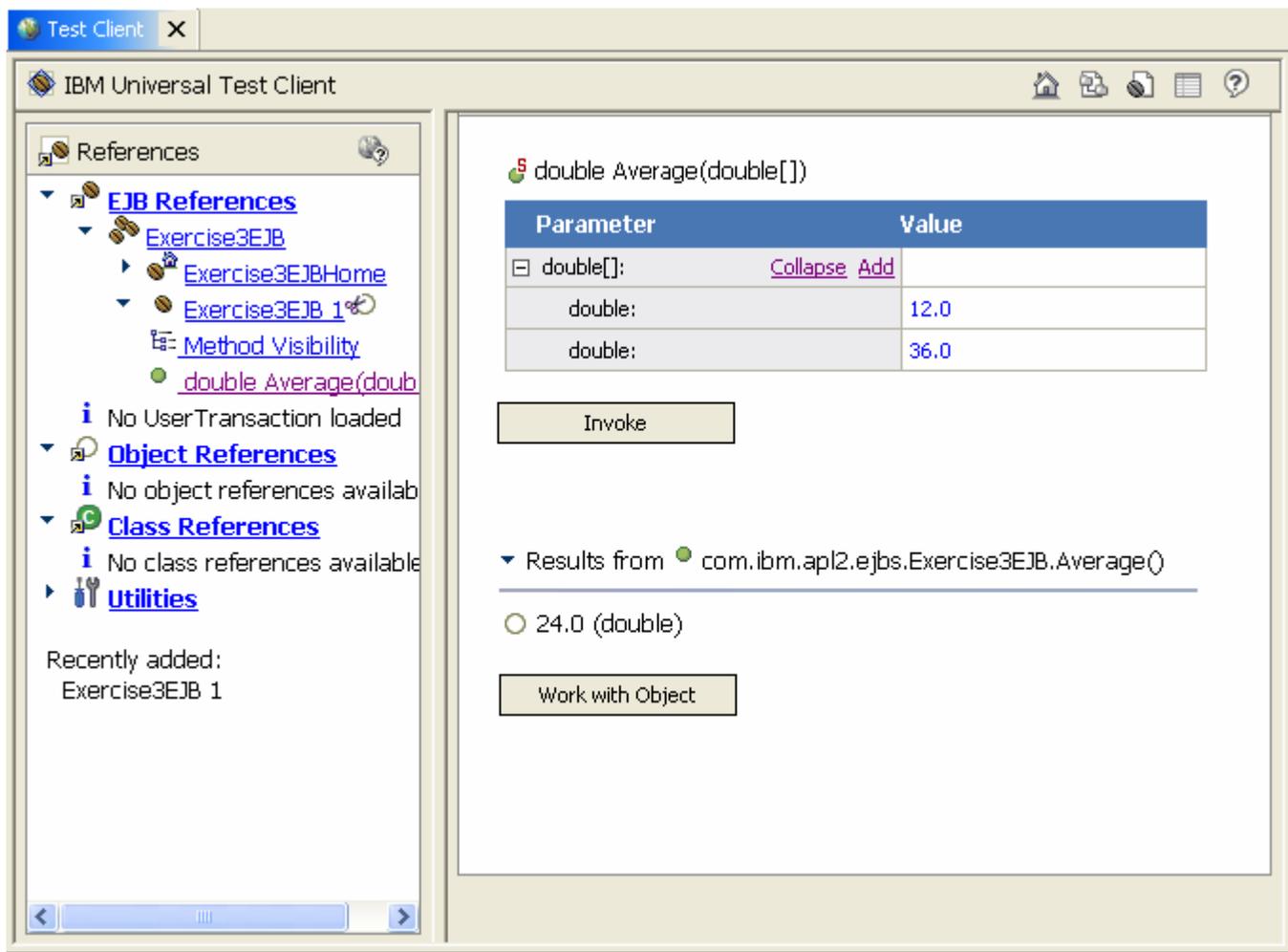
Notice that if you start the APL2 SVP Monitor window the Processors dialog (on the Options menu) shows that APL2 is signed on. And, if you turn on trace, no trace messages appear when you invoke the Average method.

Shutting Down the Server

This completes exercise 3 except for shutting down the server.

Close the Test Client view.
Navigate to the Server Perspective and the Server Configuration view.
Right click on the `ExercisesServer` and select Stop.

Example 4: Using APL2 with the Model-View-Controller Design Pattern

The examples so far have introduced you to using servlets and JSPs and the basic concepts behind EJBs. As you have seen, it is possible to build an application using APL2 and just servlets and JSPs. But to really leverage the capabilities of WebSphere Application Server, it is recommended that you use the Model-View-Controller design pattern and use servlets for application logic, EJBs for business logic, and JSPs for the view layer. This example will illustrate how to build an application using MVC.

The example application will use a servlet to accept incoming requests, parse the input parameters, and pass them to an EJB. The EJB in turn will use APL2 to perform the calculation (which will again be a simple average) and return a numeric result. The servlet will then pass the result on to a JSP for formatting and generation of the response HTML.

Actually, the application servlet will not call an EJB directly. The complexities of using JNDI to look up EJB interfaces and then the code to use the local, remote, and home interfaces are too great for inclusion in a servlet; they would clutter up the servlet and conceal the application logic. Instead, a simpler type of component called a *Java Bean* will be used as an interface between the servlet and the EJB. The Java bean will perform the operations necessary to call the EJB.

A Java bean is just a Java class that follows a few simple rules. Briefly, Java beans are serializable; they implement the Serializable interface defined in the `java.io.serializable` package. This means all their contents must able to be written to disk using Java's serializing facilities. Secondly, Java bean fields are implemented as *Properties*. Properties are private fields for which there are two methods, a *getter* and a *setter*. For example, If there is a field named Value, there will be methods named getValue and setValue. Clients of the bean call the methods to specify and reference the property. They do not access the field directly. Although properties are very common Java beans, we won't need them in this example and so we won't use this facility.

There are 16 steps in this example:

Create an Enterprise Application Project
Create an EJB Project
Create a Web Project
Create a Java Package for the EJB
Create a Java Package for the servlet and Java bean
Create an Enterprise Java Bean
Edit the EJB's Source code
Generate the Deployment and RMIC Code
Create a Java bean
Edit the Java bean's code
Create a servlet
Edit the servlet's code
Create a JSP
Edit the JSP's code
Test the Application
Shutting Down the Server

Create an Enterprise Application Project

Navigate to the J2EE Perspective and the J2EE Hierarchy view.

Select File and New and Enterprise Application Project to display the New Enterprise Application Project dialog.

Select Create J2EE 1.3 Enterprise Application project and press Next.

Type `Exercise4Eap` in the Project name field and press Finish.

`Exercise4Eap` should now appear in the Enterprise Applications folder in the J2EE Hierarchy view.

Create an EJB Project

Select File and New and EJB Project to display the New EJB Project dialog:

Select Create 2.0 EJB Project and press Next.

Type `Exercise4EJBProject` in the Project name field.

Select Exercise4Eap in the EAR project field.

Press Finish.

`Exercise4EJBProject` should now appear in the EJB Modules folder in the J2EE Hierarchy view of the J2EE Perspective.

Add the apl2.jar file to the project's build path:

Right click on Exercise4EJBProject and select Properties.
Select Java Build Path on the left and the Libraries tab on the right.
Press the Add External JARs… button.
Navigate to `\ibmapl2w\bin`, select `apl2.jar`, and press Open.
Press Ok to close the Properties for Exercise4EJBProject dialog.

Create a Web Project

Select File and New and Dynamic Web Project to display the New Dynamic Web Project dialog.

Type `Exercise4Dwp` in the Project name field.

Check Configure advanced options and press Next.

Ensure the EAR project field contains Exercise4Eap and press Finish.

You will be prompted to switch to the Web Perspective. Press Yes.

`Exercise4Dwp` should now appear in the Project Navigator view of the Web Perspective.

Create a Java Package for the EJB

Select File and New and Other… to display the New dialog:

Select Java on the left and Package on the right and press Next.

Make sure the Source Folder field shows `Exercise4EJBProject/ejbModule.`

Type `com.ibm.apl2.ejbs` in the Name field and press Finish.

Verify the `com.ibm.apl2.ejbs` package should now appear in the `ejbModule` folder of the `Exercise4EJBProject` project in the Web Perspective's Project Navigator view.

Create a Java Package for the Servlet and Java bean

Select File and New and Other… to display the New dialog:

Select Java on the left and Package on the right and press Next.

Make sure the Source Folder field shows `Exercise4Dwp/JavaSource`.

Then enter `com.ibm.apl2.exercise4` in the Name field and press Finish

Verify the `com.ibm.apl2.exercise4` package should now appear in the Java Resources folder of the `Exercise4Dwp` project in the Web Perspective's Project Navigator view.

Create an Enterprise Java Bean

Switch to the J2EE Perspective.

Select File and New and Enterprise Bean to display the New Enterprise Bean dialog.

Select `Exercise4EJBProject` in the EJB project field and press Next.

Check Session Bean.

Type `Exercise4EJB` in the Bean name field.

Verify that the Source folder field contains ejbModule.

Type `com.ibm.apl2.ejbs` in the Default package field and press Next.

Select the Session type: Stateless.

Select the Transaction type: Container.

Check Remote client view.

Press Finish to generate the EJB.

Verify the `Exercise4EJB` bean now appears in the Session Beans folder in the `Exercise4EJBProject` in the J2EE Hierarchy view.

Edit the EJB's Source code

Expand Exercise4EJB.

Double click on `Exercise4EJBBean` to open the Java Editor.

Add the following statement after the package statement at the top of the file:

import com.ibm.apl2.*;

Add the following two statements after the declaration of `mySessionCtx`:

private Apl2interp Slave = null;
private Apl2object Function = null;

Add the following code to the `ejbCreate` method:

```
    Slave = null;
    try {
            Slave = new Apl2interp();
            Function = new Apl2object(Slave, "AVERAGE");
            Slave.Associate("WSNAME", 11, Function);
    } catch (Apl2exception Exception) {
            if (Slave != null) {
                    try {
                            Slave.Stop();
                    } catch (Apl2exception StopException) {
                    }
                    Slave = null;
            }
            throw new javax.ejb.CreateException(
                    "Unable to initialize APL2 environment");
    }
```

Add the following code to the `ejbRemove` method:

```
    try {
            Function.Free();
            Slave.Stop();
    } catch (Apl2exception Exception) {
    }
```

Add the following method to the end of the file:

```
    public double Average(double[] Array) {
            double Result = 0;
            try {
                    Apl2object aplintArray = new Apl2object(Slave, Array);
                    Apl2object aplAverage = Slave.Execute(Function,
                            aplintArray);
                    aplintArray.Free();
                    Result = aplAverage.doubleValue();
                    aplAverage.Free();
            } catch (Apl2exception e) {
                    System.out.println("Apl2exception caught");
                    System.out.println("Exception message: " +
                            e.getMessage());
                    System.out.println("Event: " + e.Type + " " + e.Code);
            }
            return Result;
    }
```

Press Ctrl+S to save your changes.

Add the `Average` method to the remote interface so that is can be used by clients:

If it is not already active, switch to the J2EE Perspective and select the Outline view.

Right click on the `Average` method in the Outline view. Select Enterprise Bean, and Promote to Remote Interface.

Close the Exercise4EJBBean.java file.

Generate the Deployment and RMIC Code

Right click `Exercise4EJBProject` and select Generate and then Deployment and RMIC Code… to display the Generate Deployment and RMIC Code dialog:

Select `Exercise4EJB` and press Finish.

Create a Java Bean

A Java Bean is a Java class that implements the Serializable interface and provides getters and setters for properties. Java beans provide a convenient mechanism for encapsulating the code required to call EJBs.

Select File and New and Other… to display the New dialog:
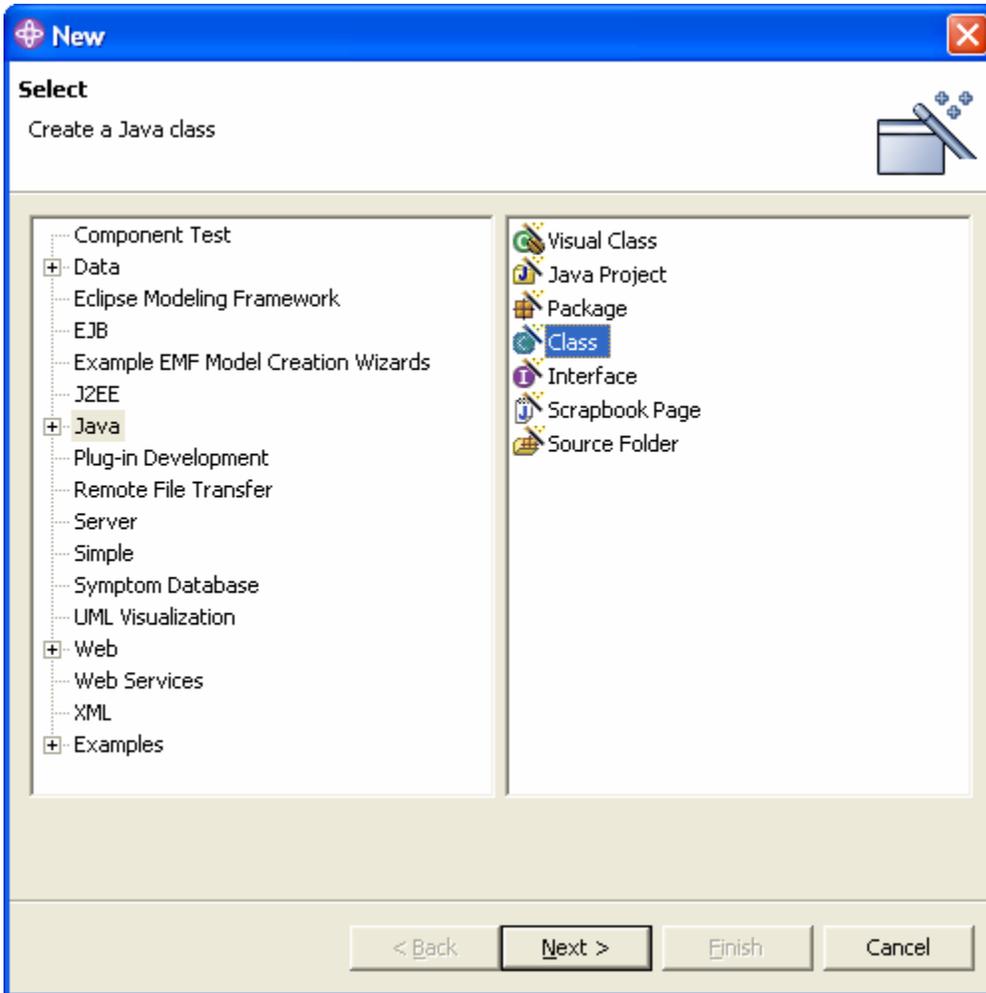
Select Java on the left and Class on the right and press Next to display the New Java Class dialog:
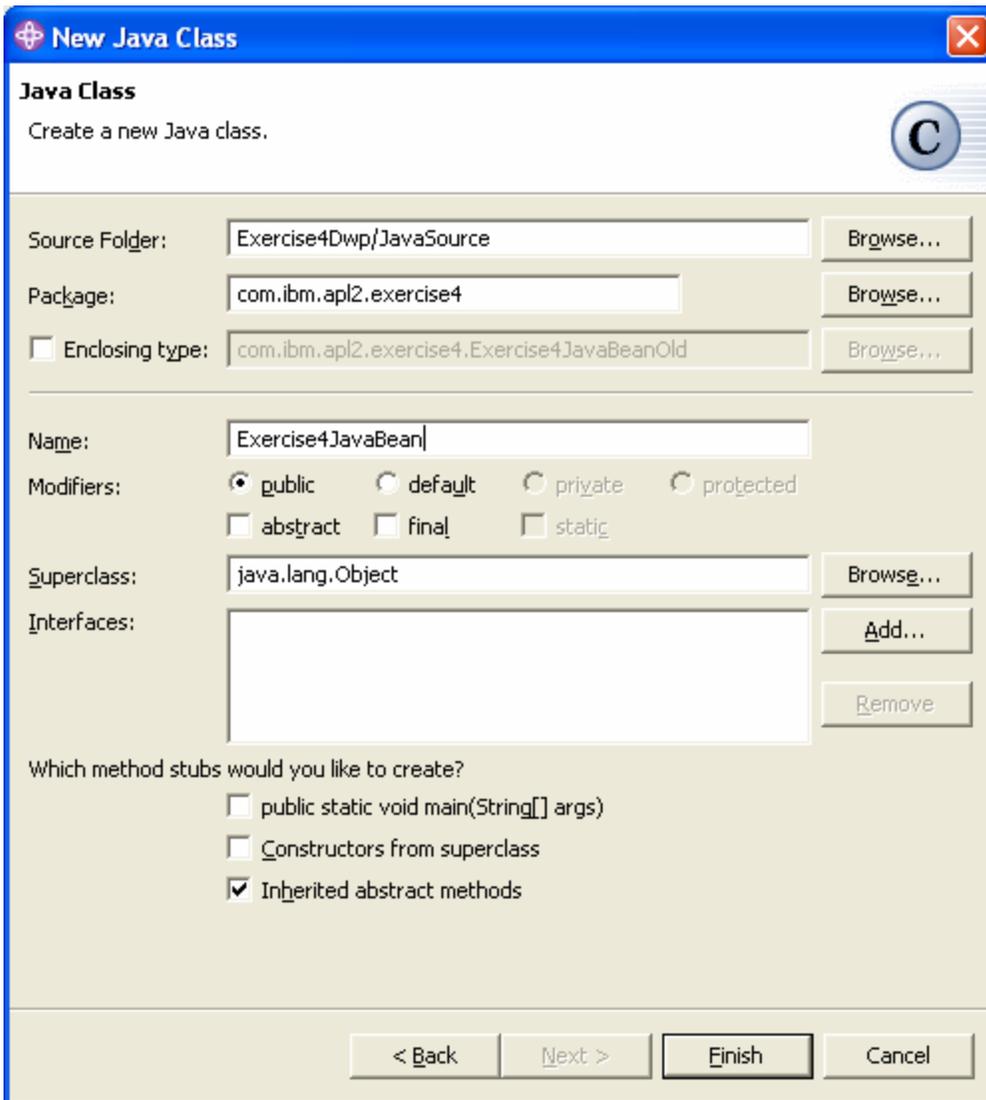
Ensure the Source folder contains `Exercise4Dwp/JavaSource`.

Ensure the Package field contains `com.ibm.apl2.exercise4`.

Type `Exercise4JavaBean` in the Name field.

Press the Add… button to display the Implemented Interface Selection dialog:
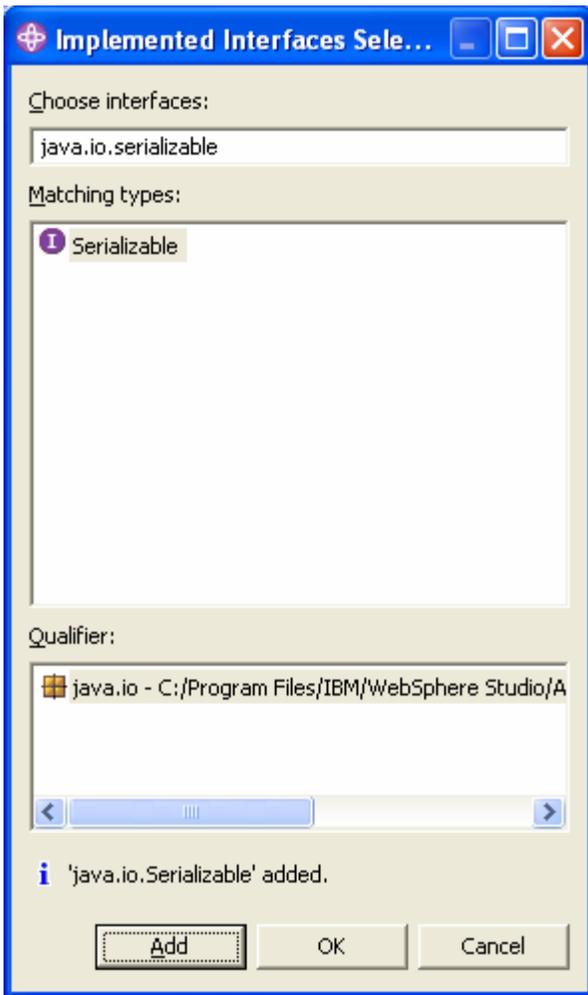
Type `java.io.serializable` in the Choose interface field.

Press Add… and then press Ok

Press Finish to close the New Java Class dialog.

Edit the Java Bean's Source Code

Copy the following statements after the import statement at the top of the file:

```
import com.ibm.apl2.ejbs.Exercise4EJB;
import com.ibm.apl2.ejbs.Exercise4EJBHome;
import java.rmi.RemoteException;
import java.util.Hashtable;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

Copy the following method to the end of the file (before the last curly brace):

```
public double Average(double Value1, double Value2) {
        final String WEBSPHERE_FACTORY =
                "com.ibm.websphere.naming.WsnInitialContextFactory";
        final String LIBRARY_APP_NAMING_PROVIDER =
                "corbaloc:iiop:localhost:2809";
        final String JndiName = "ejb/com/ibm/apl2/ejbs/Exercise4EJBHome";
        final String ClassName = "com.ibm.apl2.ejbs.Exercise4EJBHome";

        double Average = 0;

        try {
                Hashtable Environment = new Hashtable();
                Environment.put(Context.INITIAL_CONTEXT_FACTORY,
                        WEBSPHERE_FACTORY);
                Environment.put(Context.PROVIDER_URL,
                        LIBRARY_APP_NAMING_PROVIDER);

                javax.naming.InitialContext InitialContext = new
                        InitialContext(Environment);

                java.lang.Object RemoteObject =
                        InitialContext.lookup(JndiName);

                Class EjbClass = Class.forName(ClassName);
                Exercise4EJBHome EjbHome =
                        (Exercise4EJBHome) javax.rmi.PortableRemoteObject.narrow(
                                RemoteObject,
                                EjbClass);
                Exercise4EJB Ejb = EjbHome.create();
                Average = Ejb.Average(new double[] { Value1, Value2 });
        } catch (NamingException e) {
                System.out.println("NamingException caught.");
        } catch (RemoteException e) {
                System.out.println("RemoteException caught.");
        } catch (CreateException e) {
                System.out.println("CreateException caught.");
        } catch (ClassNotFoundException e) {
                System.out.println("ClassNotFoundException caught.");
        }
        return Average;
```

Press Ctrl+S to save the changes.

Five messages should appear in the Tasks view. They are produced because the Web project is not configured to use the EJB project's files. Add this configuration information now:

Switch to the Project Navigator view and right click on `Exercise4Dwp` and select Properties to display the Properties dialog for the Exercise4Dwp project:
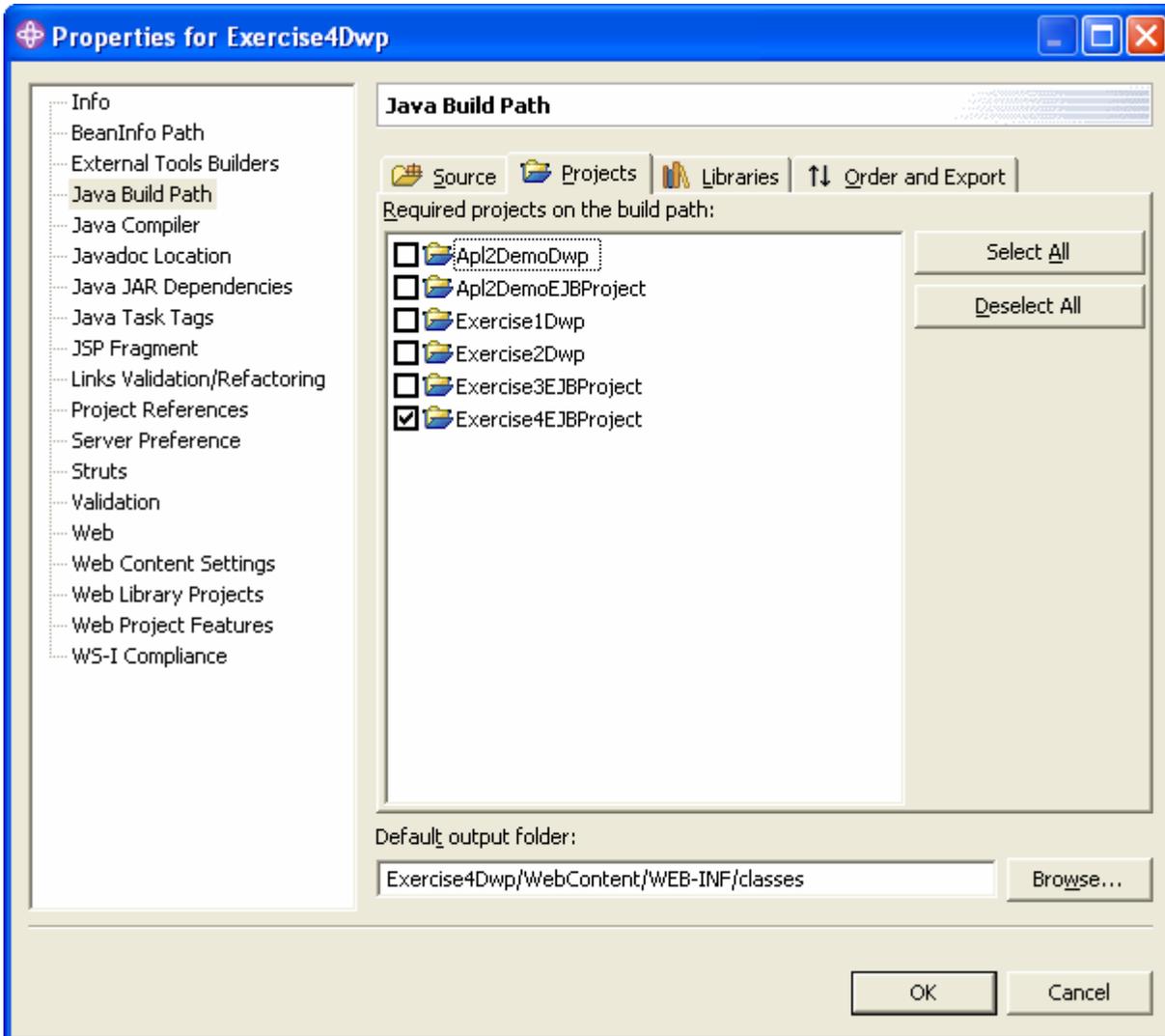
Select Java Build Path on the left and Projects on the right.

Check `Exercise4EJBProject` and press Ok.

The messages in the Tasks view should disappear.

Notes on the code:

The bulk of the Java bean's code is devoted to locating and establishing a connection with the EJB. The creation of the InitialContext object establishes a connection with the container's JNDI service. The initialContext object is then used to locate, or lookup, the remote interface for the EJB. The PortableRemoteObject class's narrow method is then used to locate the EJB's home interface. Finally, the home interface's create method is used to create an instance of the EJB. You should realize that although these steps are always required when accessing an EJB through a remote interface, you typically would not perform them in a Java bean called directly from a servlet. You would usually write utility classes and methods to

perform these steps rather than imbedding them in the Java bean. They are only presented here to give you a flavor of the steps that are required.

Close the Exercise4JavaBean.java file.

Create a Servlet

Switch to the Web Perspective and pull down the File menu and select New and then Servlet to display the New Servlet wizard.

Type `/Exercise4Dwp/JavaSource` in the Folder field (or use Browse…).

Type `com.ibm.apl2.exercise4` in the Java package field.

Type `Exercise4Servlet` in the Class name field.

Press Next.

Check Public, doPost, and doGet. Clear all other check boxes and press Next

Ensure Add to web.xml is checked and press Finish.

Verify the `Exercise4Servlet.java` file appears in the `com.ibm.apl2.exercise4` package in the Java Resources folder of the `Exercise4Dwp` project.

Editing the Servlet's Source Code

Add the following line of code to both the `doGet` and `doPost` methods:

processRequest(req, resp);

Copy and paste the following method to the end of the file (before the last curly brace):

```
private void processRequest(
        HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException {
        String String1 = req.getParameter("VALUE_1");
        String String2 = req.getParameter("VALUE_2");

        //* Make sure they're not empty
        if (String1 == null) String1 = "0";
        if (String2 == null) String2 = "0";
        if (String1.compareTo("") == 0) String1 = "0";
        if (String2.compareTo("") == 0) String2 = "0";

        //* Extract their numeric values
        double Value1 = 0;
        double Value2 = 0;
        try {
                Value1 = java.lang.Double.parseDouble(String1);
        } catch (NumberFormatException e) {}
        try {
                Value2 = java.lang.Double.parseDouble(String2);
        } catch (NumberFormatException e) {}

        //* Use a Java Bean to calculate the average
        Exercise4JavaBean Bean = new Exercise4JavaBean();
        double Average = Bean.Average(Value1, Value2);

        //* Save the result in the request object
        req.setAttribute("Average", new Double(Average));

        //* Forward request to the JSP for output formatting
                        getServletContext().getRequestDispatcher("Exercise4JSP.jsp").forward(
        req,    resp);
    }
```

Press Ctrl+S to save the changes.

Notes on the code:

In the first exercise we glossed over the servlet's code. This servlet is slightly more complicated and you should understand how it works.

Servlets get passed two parameters: an HttpServletRequest and an HttpServletResponse object. These objects are used to retrieve information about the request and to build the response. The servlet calls the request object's getParameter method to get the contents of two form parameters, VALUE_1 and VALUE_2. These values are passed in the request's URL. The response object is not used until the very end of the servlet.

The servlet then uses standard Java facilities to parse the parameter values.

Once the values are parsed, the servlet creates an instance of the Java bean and calls its Average method. The Java bean returns a double value.

The servlet then uses the request object's setAttribute method to put the Java bean's result into a request attribute named Average. In addition to parameters included in a request's URL, each request can have attributes set and referenced by application components. The servlet uses an attribute to pass the bean's result to the JSP that builds the response HTML.

The final line of the servlet forwards the request to the JSP. The JSP will then extract the bean's result from the attribute and build the response HTML. Servlets always run in a web container. The web container provides a *context* through which servlets can call container services. The servlet class includes a method named getServletContext which returns a reference to this context. The context's getRequestDispatcher service returns a reference to a dispatcher service which can be used to forward a request to another component.

WSAD provides a useful facility for exploring Java classes and services provided by objects. WSAD can help you type method and field names. Try this:

Insert a new line at the end of the servlet's Average method.
Position the cursor on the new line and type getServletContext(). and pause.

Notice that when you type the period, WSAD displays the list of fields and methods that are available in the object returned by getServletContext(). If you keep typing and type getR, WSAD displays the list of methods that begin with getR. You can then use the arrow keys to select the GetRequestDispatcher method. When you press Enter, the method name is inserted at the end of the line and the cursor is positioned within the method's parentheses, ready for you to type parameters. After typing a character string, you can move the cursor after the closing parenthesis and type another period to get a list of the methods available in the object returned by getRequestDispatcher.

This is a handy method for remembering the services available in request, response, and context objects.

Once you are done experimenting with the typing aids, delete any extra lines you added and close the `Exercise4Servlet.java` file.

Create a Java Server Page

Select File and New and JSP File to display the New JSP File dialog.

Ensure `/Exercise4Dwp/WebContent` is in the Folder field.

Type `Exercise4JSP.jsp` in the File name field and press Finish.

Verify the `Exercise4JSP.jsp` file appears in the `Exercise4Dwp/WebContent` folder.

Edit the JSP's Source code

Use copy and paste to replace the body of the HTML with the following code:

```
<H1>Exercise 4 JSP</H1>
<H2>Average of Two Numbers</H2>
<FORM method="POST" action="/Exercise4Dwp/Exercise4Servlet">
<%
String Value1 = request.getParameter("VALUE_1");
String Value2 = request.getParameter("VALUE_2");
if(Value1 == null) Value1 = "0";
if(Value2 == null) Value2 = "0";
Double Average = (Double)request.getAttribute("Average");
if(Average == null) Average = new Double(0.0);
%>
<P>Value 1: <INPUT name="VALUE_1" value="<%= Value1 %>" maxlength="20" size="20"
type="text">
<P>Value 2: <INPUT name="VALUE_2" value="<%= Value2 %>" maxlength="20" size="20"
type="text">
<P><INPUT name="SUBMIT" type="submit" value="Average">
</FORM>
<P><H4>Calculated Average: <%= Average %></H4>
```

Press Ctrl+S to save the changes.

Notes on the code:

Like servlets, JSPs are provided with request and response objects that are used to retrieve information about the request and to build the response. The JSP objects are always named request and response. The JSP's code uses the request object to again get the VALUE_1 and VALUE_2 parameters. These values are used to display the user's previous entries in the new HTML that the JSP will build. The JSP uses the request object's getAttribute method to retrieve the value of the attribute named Average. Recall, this attribute was set by the servlet and contains the result returned by the Java bean.

When you are finished examining the code, close the `Exercise4JSP.jsp` file.

Test the Application

You are now ready to test your application.

In the Project Navigator view, expand `Exercise4Dwp`, `Java Resource`, and `com.ibm.apl2.exercise4`.

Right click on `Exercise4Servlet.java` and select Run on Server… to display the Server Selection dialog:

Check Use an Existing Server and select the `ExercisesServer` and press Next.
Check Deploy EJB beans and press Finish to start the sever.

Starting the server will again take a while. (In fact it will probably take longer than with previous exercises since more components are starting.) When the server has started and is ready for e-business, the Web Browser view will show the JSP's form. The input fields and result will display the default values of 0. Type some numbers and press enter.

Notice that the URL once again appears to contain the project and servlet names. Once again, these are not resource names. Rather, they are the context root and URL mapping. They can be changed just as before.

Shutting Down the Server

This completes exercise 4 except for shutting down the server.

Close the Web Browser view.
Navigate to the Server Perspective and the Server Configuration view.
Right click on the `ExercisesServer` and select Stop.

Example 5: Deploying an Enterprise Application on WAS

So far, you have used WebSphere Studio Application Developer's built-in server to test your applications. Eventually, you will probably want to put an application into production. To do this, you can deploy your application to WebSphere Application Server. This example illustrates that process.

There are 7 steps in this example:

Export the Project
Start the Server
Start the Administrative Console
Configure the Server to use APL2
Install the Application
Restart the Server
Test the Application

Export the Project

The first step in application deployment is to export your enterprise project to a file that can be read by WebSphere Application Server. WSAD can export to many different types of files. The appropriate type for WAS is the Enterprise Archive, or EAR file.

Select File and Export… to display the Export dialog:
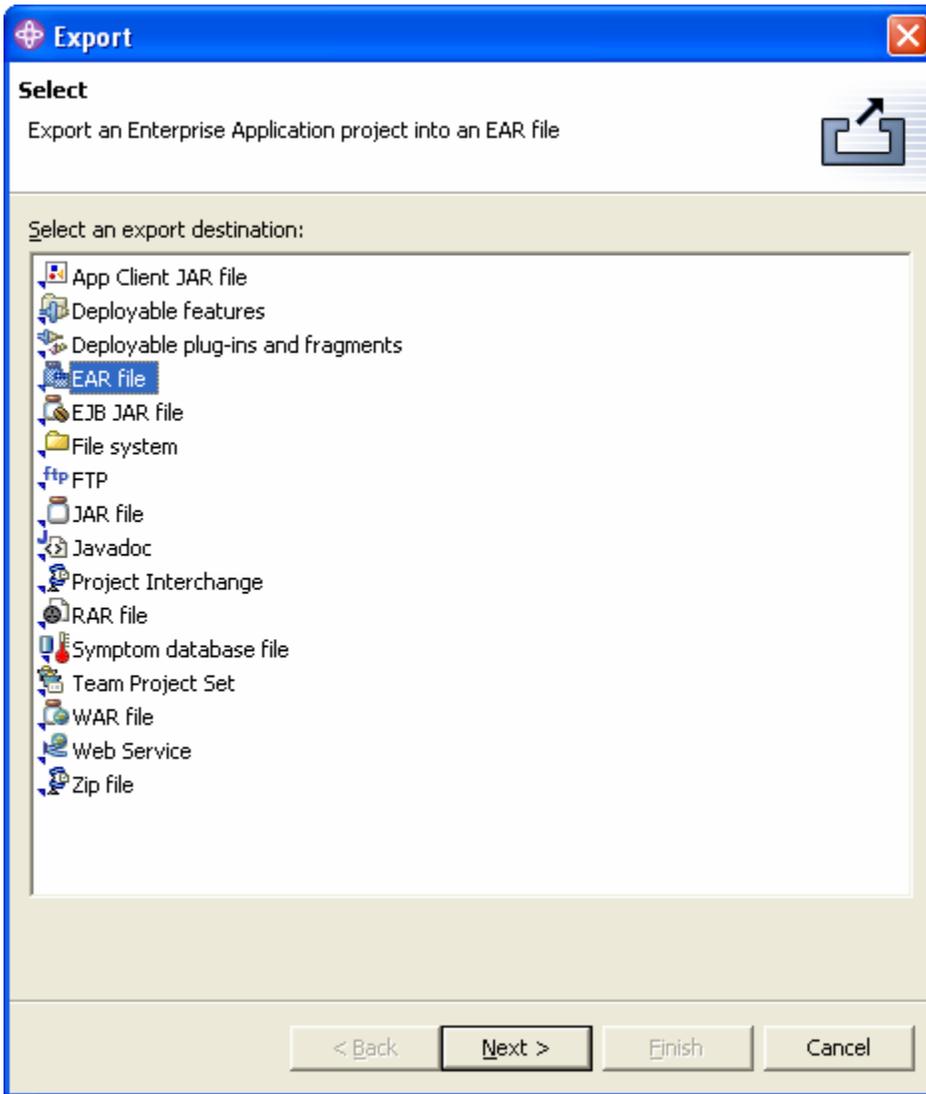
Select EAR file and press Next.
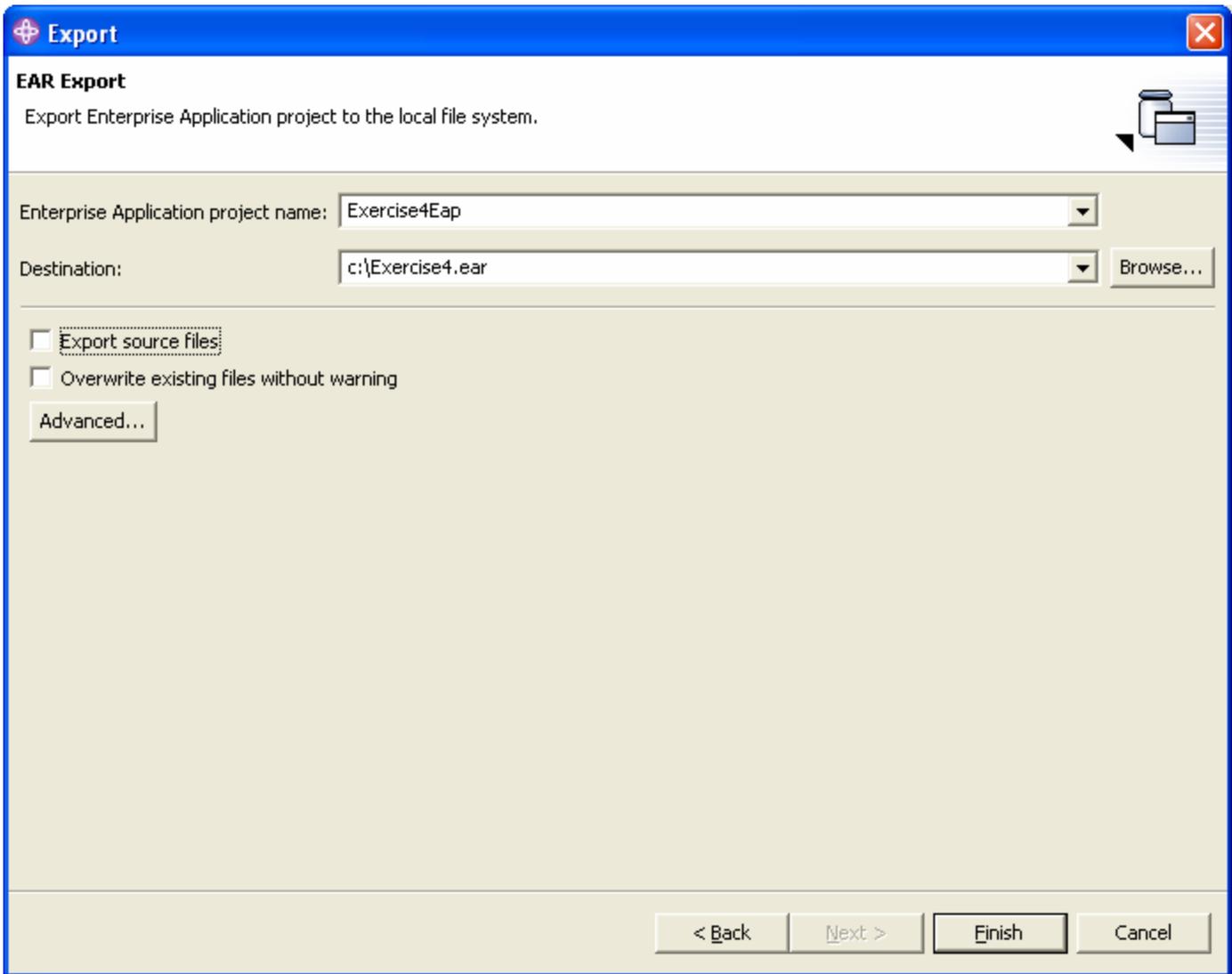
Select `Exercise4Eap` in the Enterprise Application project name field.

Type a path and EAR file name in the Destination field. Make a note of the path and filename you use; you will need it later.

Press Finish.

You should now have an EAR file which can be deployed. You can shut down WSAD.

Start the Server

In order to deploy the project, WebSphere Application Server will need to be running. So, start it:

Select Start, Programs, IBM WebSphere, Application Server v5.1, Start the Server.

It will take a while for the server to start. You will see progress messages in a console window. When the server initialization is complete, the console window will automatically close.

Start the Administrative Console

The administrative console is used to configure the server and install applications. Start it:

Select Start, Programs, IBM WebSphere, Application Server v5.1, Administrative Console.

The administrative console is a browser based application. It will initially prompt you for a userid. You can use any userid. It is not used for security access; it is only used for logging purposes.

Configure the Server to use APL2

Just like the WSAD test servers, the WebSphere Application server needs to be configured so that applications can load classes from apl2.jar. Configure the server's classpath:

Expand Servers
Select Application Servers
Select server1
Select the configuration tab
Scroll down and select Process Definition
Select Java Virtual Machine
Type the path and filename of apl2.jar in the Classpath field.
Press Apply
Click the Save link
Press Save

Install the Application

Now you are ready to install the enterprise application contained in the EAR file you exported from WSAD:

Expand Applications
Select Install New Application
Type the path and file name of the EAR file you exported from WSAD in the Local path field.
Press Next
Keep pressing Next until you have seen all 6 installation steps and then press Finish.
Select Save to Master Configuration
Press the Save button.

Press the Administrative console's Logout link and shut down the browser.

Restart the Server

To enable your changes, you must stop and restart the server:

Select Start, Programs, IBM WebSphere, Application Server v5.1, Stop the Server.
Select Start, Programs, IBM WebSphere, Application Server v5.1, Start the Server.

Test the Application

Open a browser and type the following URL:

http://localhost:9080/Exercise4Dwp/Exercise4Servlet

You should see the exercise 4 JSP and be able to calculate the average of two numbers.

When you are satisfied that your application is working properly, you can stop the server:

Select Start, Programs, IBM WebSphere, Application Server v5.1, Stop the Server.

Summary

These examples have given you a brief introduction to using APL2 in J2EE Enterprise Applications using WebSphere Studio Application Developer and WebSphere Application Server.   You have learned how to build servlets, Java Server Pages, Java Beans, and Enterprise Java Beans and how to deploy Enterprise Applications. You have learned that the Model-View-Control design pattern encourages separation of tasks for more efficient utilization of resources, opportunities for code reuse, and ease of application maintenance.

J2EE and the WebSphere suite of products support an enormous number of features and facilities. These examples have provided just a small sampling of their capabilities. Although the techniques demonstrated are sufficient for building full-fledged applications, you may want to learn more about J2EE and WebSphere. Useful topics for further study include JSP tags, STRUTS, web container services, and entity and message-driven EJBs. IBM offers a variety of classroom and online courses about WebSphere. For a complete list, consult this URL:

http://www.ibm.com/services/learning

The authors particularly recommend the following courses:

Servlet and JSP Development with IBM WebSphere Studio V5.1.1
EJB Development using WebSphere Studio Application Developer V5.x

Appendix: Importing the Examples

After completing the examples, you may decide to clean up your workspace and delete them. If you later want to review the examples, you can import them rather than typing them all in again. The examples are available in the apl2web.zip file. The file is available in the APL2 \samples directory on Windows and in the APL2 /examples/java directory on Unix systems.

Select File and Import… to display the Import dialog:
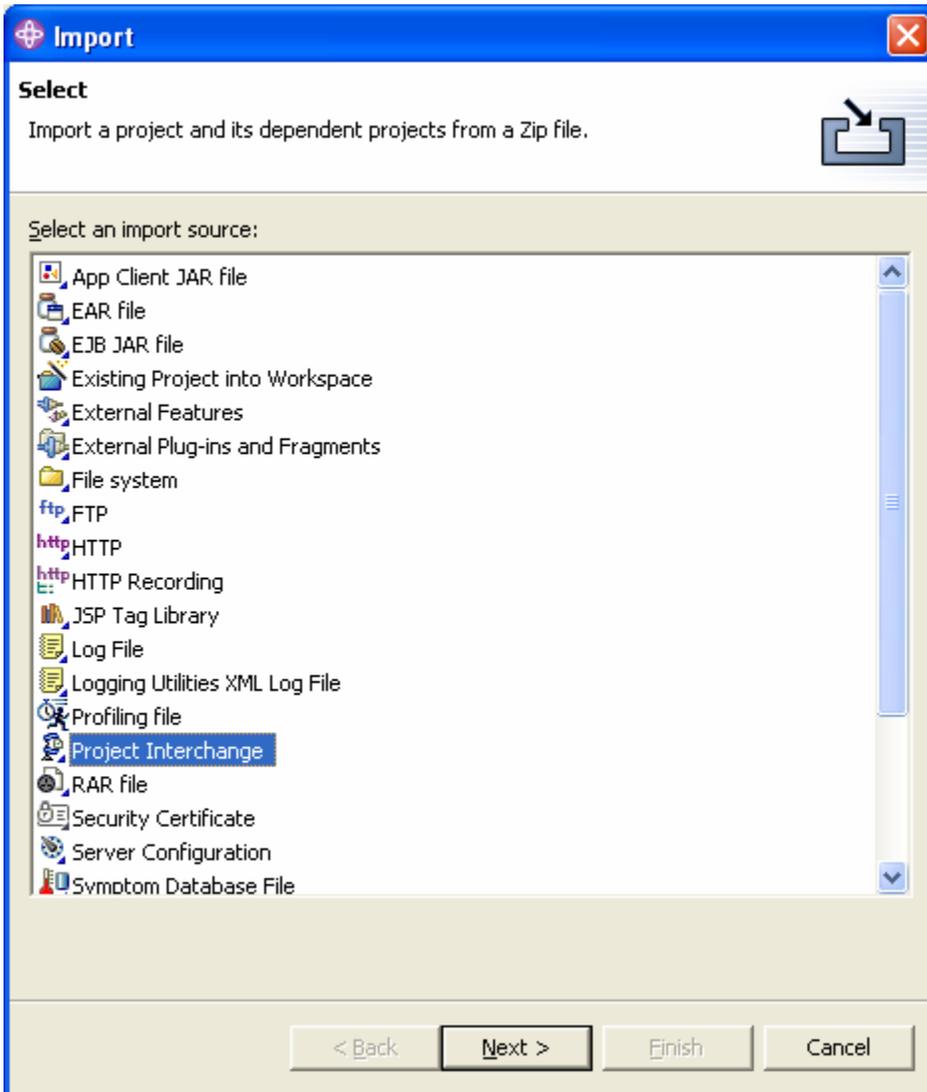
Select Project Interchange and press Next.
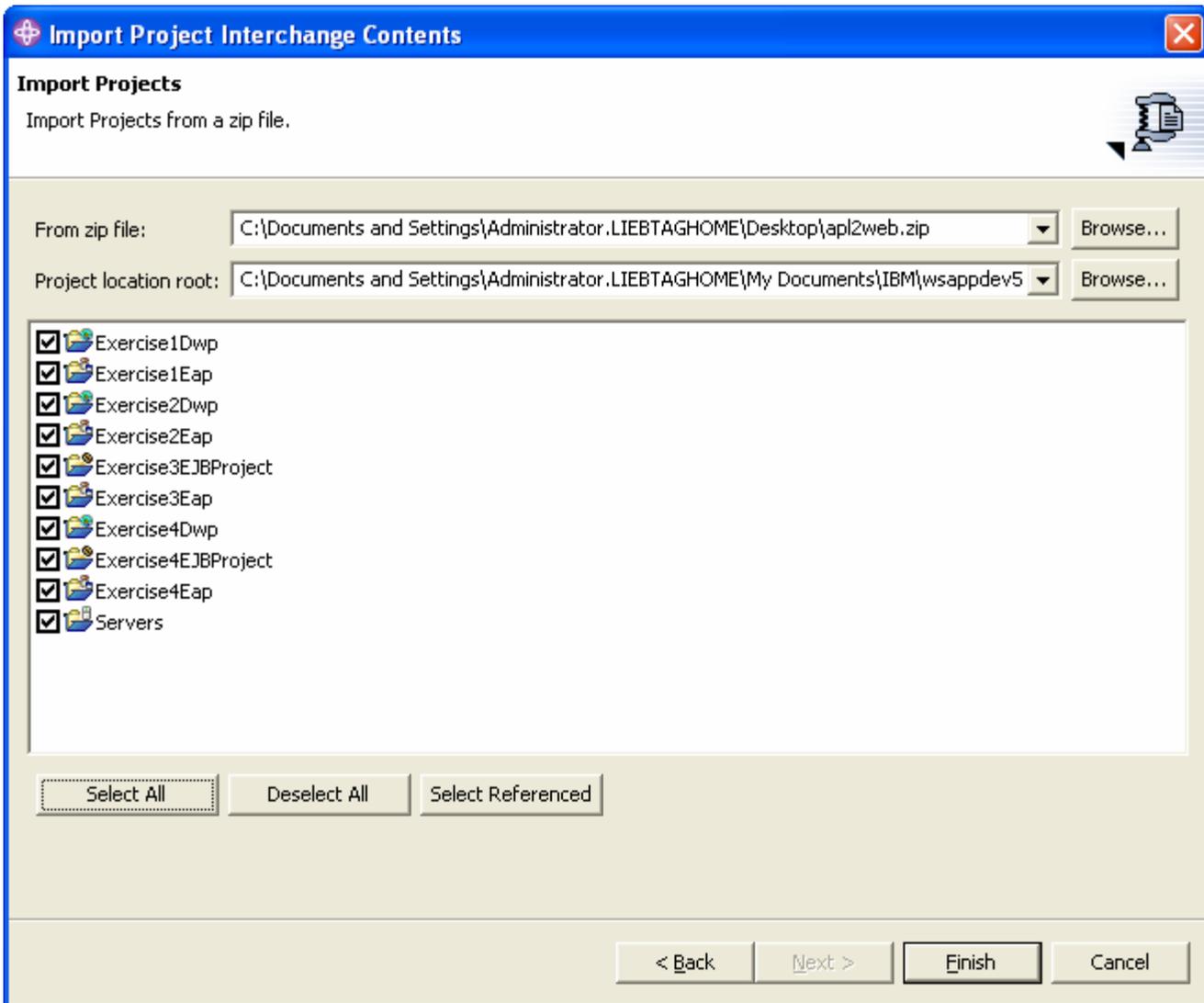
Type the path and file name of the apl2web.zip file in the From zip file field.

Press Select All and press Finish.

If you already have some of the example projects, you will be prompted to confirm that you want them overwritten. Press Ok.

All the example projects should now appear in your workspace.

Testing Imported Exercises

The process for running imported exercises is slightly different than was used for the hand built exercises.

The project interchange zip file contains all four exercises' and the server's configuration files. The server's configuration files contain references to all four exercises. This means that when you try to run any of the exercises on the server, the server will actually try to start all four enterprise applications. This is fine but to avoid error messages, it requires that you perform a step that was not originally required for exercises 1 and 2.

Recall that when you tested exercises 3 and 4, in the Server Selection dialog you pressed the Next button and then checked Deploy EJB Beans. This was required to give the server access to the projects' EJB beans.

Since the imported server will try to start all four exercises, it will require access to the EJB beans for both exercises 3 and 4, even if you are only trying to run exercise 1 or 2. So, when you reach the Server Selection dialog, you should press the Next button and then check Deploy EJB Beans for both the Exercise3EJBProject and the Exercise4EJBProject projects. Although exercises 1 and 2 will still work, if you do not deploy exercise 3 and 4's EJB beans, those applications will fail to start and error messages will appear in the Console view.