Java^{tm} For RPG Programmers

Hands On Lab

Fall Common 1998

Phil Coulthard, George Farr and AS/400 Team

IBM Canada Ltd

Table of Contents

Introduction

Java is the hot new language kid on the block. This lab will give you some hands-on experience with Java and point out, where appropriate, the contrasts to the RPG language.

Goal

In this lab we will produce a Java equivalent of an existing "green screen" application. This will be a client Java program that runs on the workstation. The "final" version accesses data on the AS/400, but we will not have time to complete the entire Java program in our two hours, so we will concentrate on coding the client user interface code only.

Tool

In this lab we will use the **VisualAge for Java** tool for our Java development. We will not spend much time describing the tool and its capabilities - really any Java development tool will work for the purposes of this lab. We are simply interested in editing and running Java here so we can concentrate on the language, not the tool.

There are other labs at COMMON that act as a good introduction to the **VisualAge for Java** (VAJava) product. One of the key pieces of VisualAge is the ***Visual Composition Editor***, or ***VisualBuilder*** as it is known by. This is, like **SDA**, a **WYSIWYG** *(What You See Is What You Get)* tool for designing your user interfaces visually. While we highly recommend this part of VAJava, to focus on our goal of showing you the language we will write all our user interface code by hand in this lab. We will ***not*** use the VisualBuilder here.

Java and the AS/400

Java is an object oriented programming language that is, compared to other OO languages like C++, is "easy to digest". Of course, this is a relative statement. As an AS/400 programmer, Java applies to you in the following ways:

- Today, as a programming language for you client user interfaces.
- Tomorrow (V4R2 and beyond), as a programming language on the AS/400 itself. That is, an alternative for RPG.

Java Applets

Java can be used to write applets, which are small programs that can only run inside web browsers such as **Netscape Navigator** or **Microsoft Internet Explorer**. These are mini-programs, but they have full user interface capabilities. They run right inside the browser. Java is traditionally an interpreted language, like **Visual Basic** and **Smalltalk**, and the web browsers today all include a Java interpreter engine.

Java applets can be used inside a traditional **HTML** (*HyperText Markup Language*) web page to add logic, graphics or user interaction. They can even be used to access data from a host, such as **DB2/400**.

The key things to remember about applets are:

- They only run inside a browser. They have no “main window” of their own, but rather use the real estate of the web browser.
- They physically live on the same server as the web page itself. The web browser, upon encountering an HTML “**APPLET**” tag inside the HTML source for a web page will return to the server to retrieve the applet (as pointed to by the **APPLET** tag), and download it into memory where it will be run.
- They are not permitted to access the local client’s hard drive or run programs on the local client. They are also not allowed to communicate back to any host server except the one they came from (the restrictions can be waived with “signed” applets that are run by consenting users).

Java applets can target AS/400 data and programs. This can be done using built-in Java communications support for TCP/IP sockets programming, or it can be done using the **AS/400 Toolbox for Java** set of classes written by IBM Rochester. This Java code offers a significantly easier means to access AS/400 services than raw communications coding.

Java Applications

While the early excitement around Java was due to its unique ability to program web pages with live code, this is not Java’s only role. It is also a full fledged application programming language, and can be used effectively to write full applications, which are invoked from the command line as with traditional language applications.

Using Java to write applications offers all the functionality and portability benefits of Java applets, but:

- Removes the security “sandbox” restrictions that applets have.
- Does not offer, yet, the exceptional benefit of being loaded on demand that applets enjoy. This means distribution and maintenance are bigger considerations, for *client* Java applications.

Note that the **AS/400 Toolbox for Java** code can be used for Java applications or applets.

To run a Java application on a particular operating system, you must have a Java Virtual Machine (JVM - interpreter) on that operating system. All current operating systems have now, or will soon have, a JVM built into them.

Java on the AS/400

Java, in combination with the **AS/400 Toolbox for Java** classes, can be used to write compelling Java client graphical user interfaces that easily access your AS/400 data, programs and commands.

But that is not all. As of **Version 4 Release 2 of OS/400**, you are able to use Java for server applications too with the advent of a Java interpreter and Java static compiler on the AS/400.

The **AS/400 Toolbox for Java** classes are also usable directly on the AS/400 as they are on the client.

The Lab - Section 1: VisualAge for Java and “Hello World!”

Section Introduction

In this section we will start slowly, and introduce you to:

- **VisualAge for Java**, and its main ***WorkBench*** window.
- VisualAge for Java **projects** for organizing your Java code by application.
- Java **packages**, which group related Java code for easy distribution and reuse.
- Java **classes**, which contain all the “meat” in Java - variables and methods (code).
- Java **methods**, which is where all executable code in Java is placed.

You will:

- 1. Create a project.
- 2. Create a package.
- 3. Create a class.
- 4. Type in a method inside that class.
- 5. Save and run the class.

Let us begin our journey of Java, the language...

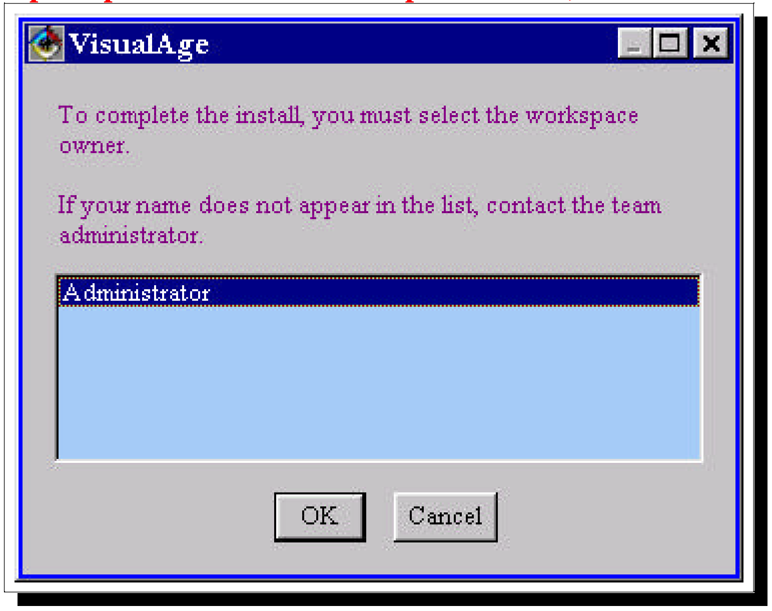
Step 1. Starting VisualAge for Java

INSTRUCTIONS:

NOTE: "**Mouse button 1**" is the left mouse button, "**Mouse button 2**" is the right mouse button.

1a. If there is no "**Workbench**" entry in your Windows task bar at the bottom of the screen, then start **VisualAge for Java** by:

- Selecting the **Start** button in the left side of the task bar
- Selecting **Programs -> IBM VisualAge for Java for Windows -> IBM VisualAge for Java**
  NOTE: If you are prompted to select a Workspace Owner, select Administrator



VisualAge

To complete the install, you must select the workspace owner.

If your name does not appear in the list, contact the team administrator.

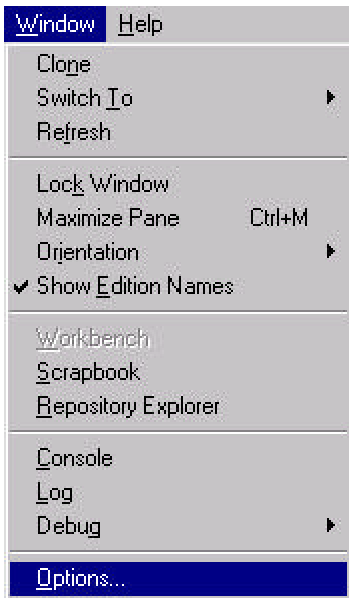
Administrator

NOTE: If you are prompted for a user password, type anything, such as admin
NOTE: You will see a "What would you like to do" dialog - just press Close on it.

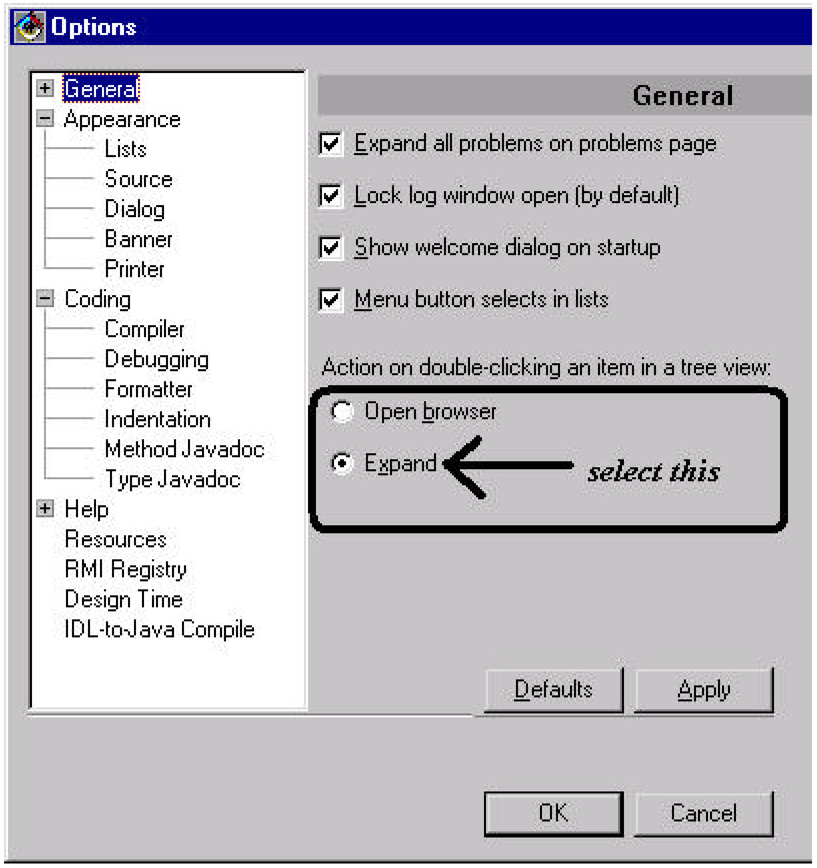
1b. If there is already a **Workbench** entry, click on it with mouse button 1 to give it focus.

1c. To ensure the lab runs smoothly for you, please make or verify the following:

- Under the **Windows** pulldown, select **Options**:



- Select the first entry in the tree "**General**" and ensure the "**Expand** list" radio button is selected



General

- [✓] Expand all problems on problems page
- [✓] Lock log window open (by default)
- [✓] Show welcome dialog on startup
- [✓] Menu button selects in lists

Action on double-clicking an item in a tree view:
Open browser [✓ Expand]

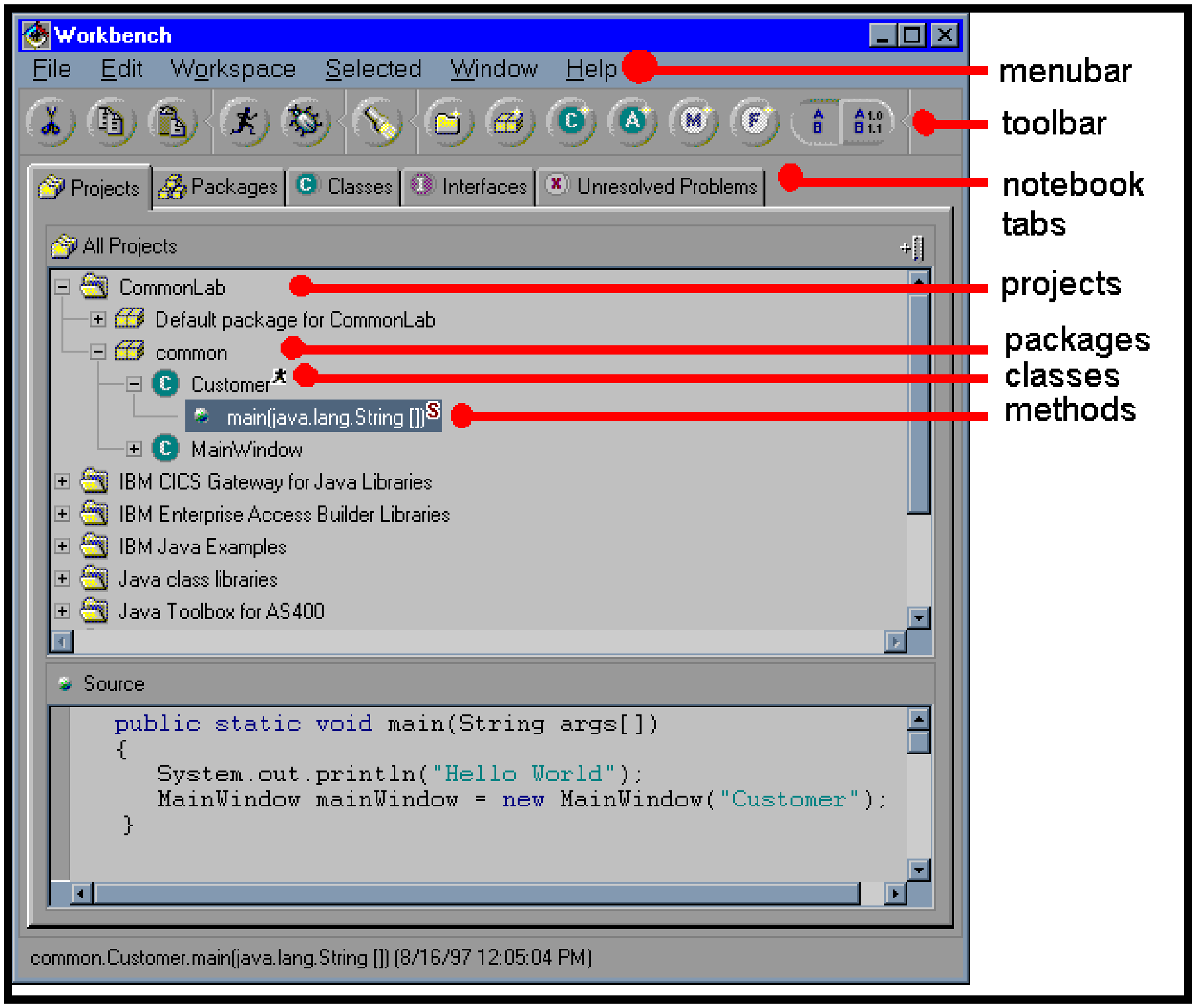
- Close the Options dialog by pressing **OK**.

NOTES:

The VAJava Workbench window is a multiple paned **IDE** (*Integrated Development Environment*) for Java development. It divides your Java applications into:

- **Projects**. These, like AS/400 libraries, allow you to partition your applications into manageable units. These are VAJava-unique constructs. Projects contain multiple packages.
- **Packages**. These, like AS/400 ILE RPG service programs, allow you to divide your application pieces into easily reused units. These are Java language constructs. Packages contain multiple classes.
- **Classes**. These, like AS/400 ILE RPG modules, allow you to divide your source code into functions (methods in Java, procedures and subroutines in RPG) and variables those functions need. These are typically self-contained groupings. Classes contain multiple fields (variables) and methods.
- **Methods**. These, like AS/400 ILE RPG procedures and subroutines, contain all the actual code your program or application will use. Unlike RPG, in Java executable code can only exist in methods. And methods can only exist inside classes.

VAJava WorkBench


Workbench
File Edit Workspace Selected Window Help
menubar
toolbar
Projects Packages Classes Interfaces Unresolved Problems
notebook tabs
All Projects
CommonLab
Default package for CommonLab
common
Customer
main(java.lang.String [])
MainWindow
IBM CICS Gateway for Java Libraries
IBM Enterprise Access Builder Libraries
IBM Java Examples
Java class libraries
Java Toolbox for AS400
projects
packages
classes
methods
Source

```
public static void main(String args[])
{
   System.out.println("Hello World");
   MainWindow mainWindow = new MainWindow("Customer");
}
```

common.Customer.main(java.lang.String []) (8/16/97 12:05:04 PM)

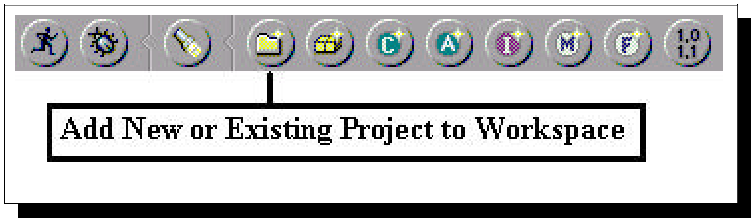
Note: if there is already a project listed at the top called **CommonLab**, it is left over from the previous lab. Delete it now by:

- Selecting it with mouse button 1.
- Right clicking with mouse button 2, and selecting **Delete** from the popup menu.

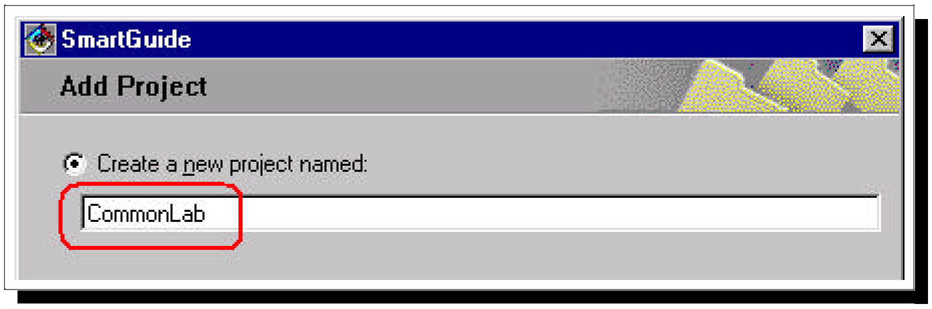
Step 2. Creating New Project - "CommonLab"

INSTRUCTIONS:

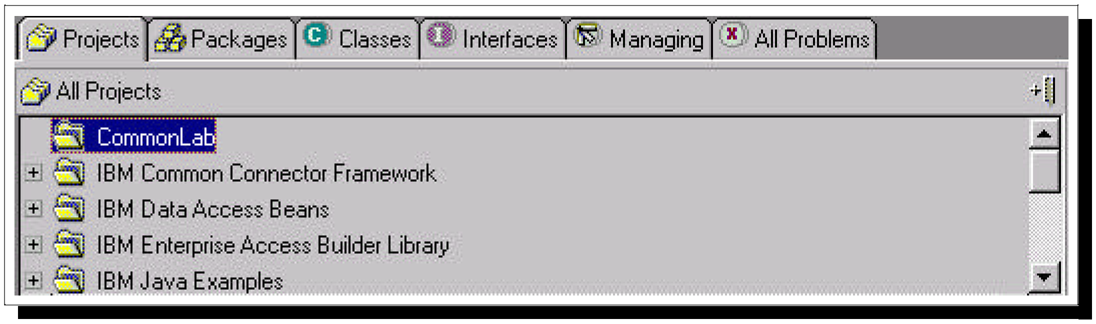
2a. Select the **Project** "SmartGuide" (a.k.a. "wizard") icon in the toolbar:



2b. In the resulting dialog box, type in **CommonLab** and select **Finish**:



After pressing **Finish**, you will see the new project *CommonLab* in the tree view, sorted alphabetically:

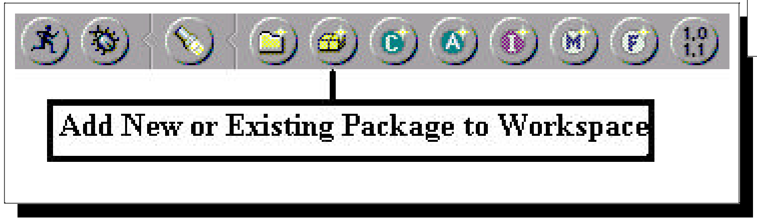


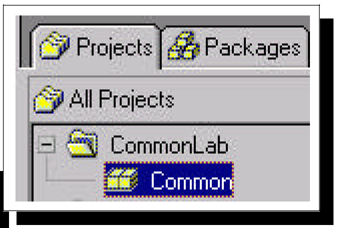
2c. Ensure the project is selected by clicking on it with mouse button 1.

Step 3. Creating New Package - "Common"

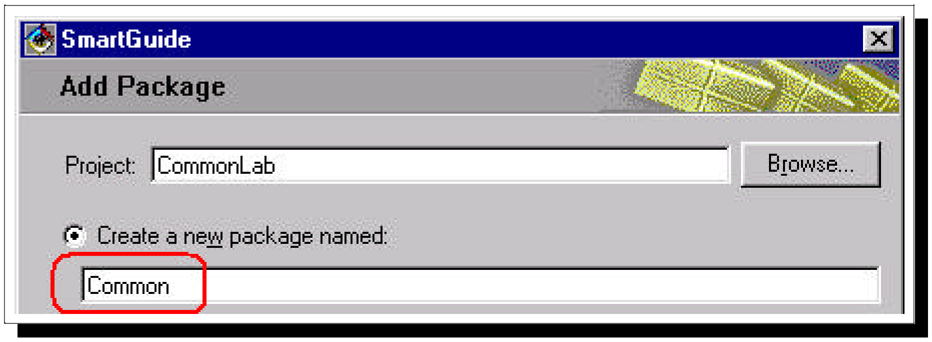
INSTRUCTIONS:

3a. Click on the **Package** icon in the *toolbar* to create a new package:





3b. In the resulting dialog box, type in **Common** and select **Finish**:



Project: CommonLab

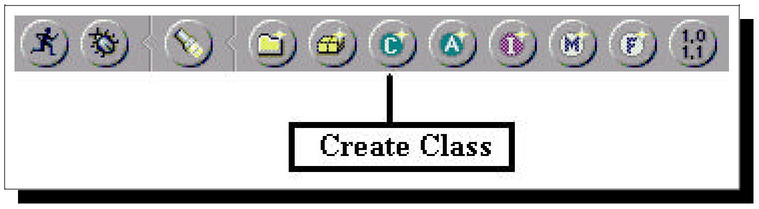
Create a new package named: Common

After the package is created, you will see the new package ***Common*** in the tree view - under the ***CommonLab*** project.

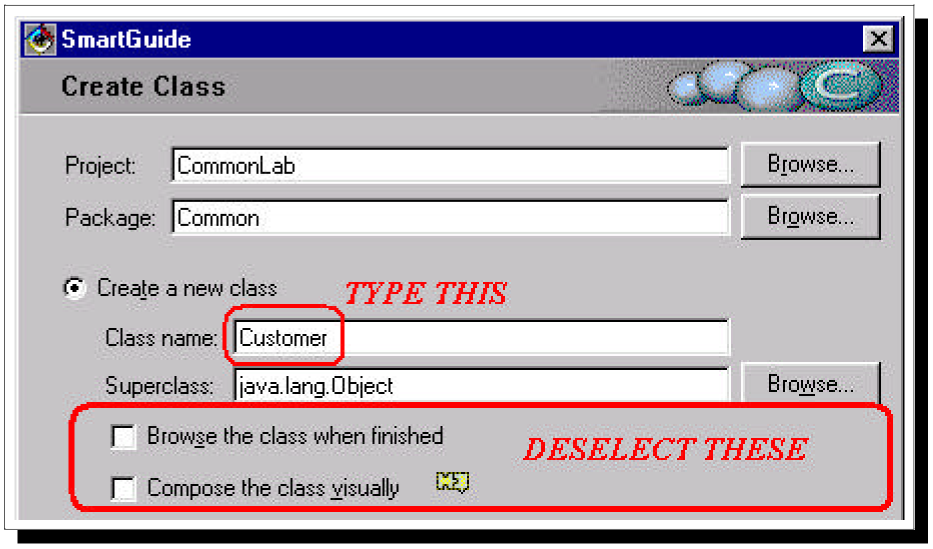
Step 4. Creating New Class - "Customer"

INSTRUCTIONS:

4a. Click on the **Class** icon in the *toolbar* to create a new class inside the ***Common*** package (make sure the ***Common*** package from the above step is selected!):



4b. In the resulting dialog box, type in **Customer** for the class name, deselect the two checkboxes, and select **Finish**:


SmartGuide

Create Class

Project: CommonLab

Package: Common

Create a new class

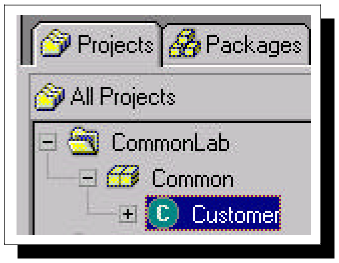
Class name: Customer

Superclass: java.lang.Object

Browse the class when finished

Compose the class visually

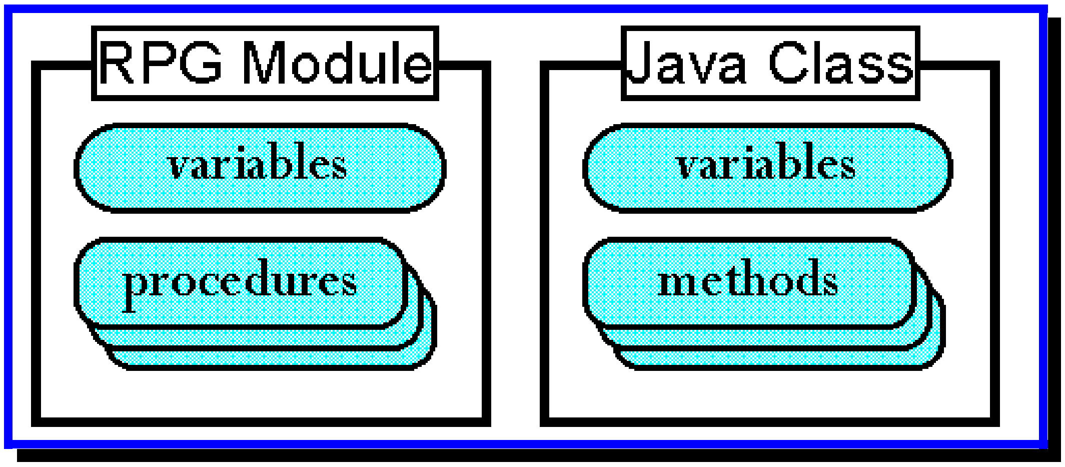
After the class is created, you will see the new class named ***Customer*** in the tree view, under the ***Common*** package.



NOTES:

What is a **class**? It is a key construct in Java: *all* code and *all* variables exist *only* inside classes. In fact, code must exist inside methods which must exist inside classes.

Java classes are similar to ILE RPG IV **modules**! Modules contain variables and RPG procedures and subroutines. Java classes contain variables and methods. ***Methods*** are like RPG ***procedures***.



A class in Java typically looks like this:

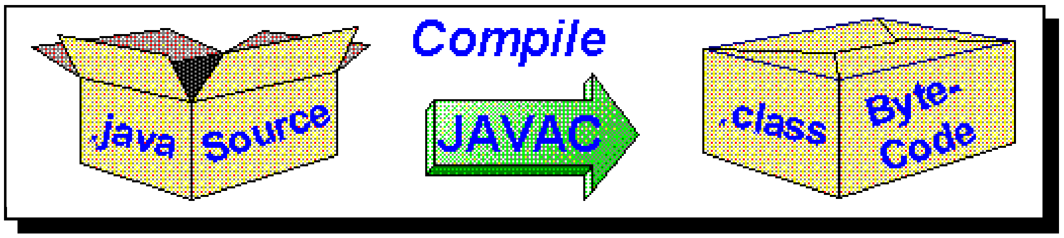
```
public class MyClass
{
   // variables
   // methods
}
```

Note the keyword **class**, and the braces delimiting the begin and end of the class. In this example, "`MyClass`" is the user-supplied name of the class. The Java keyword **public** indicates this class is accessible by everyone. This is an optional keyword - without it only other classes in *this* package have access to this class.

NOTE: THE BRACES ARE FREE FORM, SO THIS IS ALSO VALID:

```
        public class MyClass { // mycode }
```

When using Java outside of tools like **VisualAge for Java**, you will typically have one class per source file **(.Java)**. This will then be compiled into one ByteCode **(.class)** file with the same name as the class (`MyClass` in this case). The compiler is called **JAVAC** and it converts source to easily interpreted ByteCode.



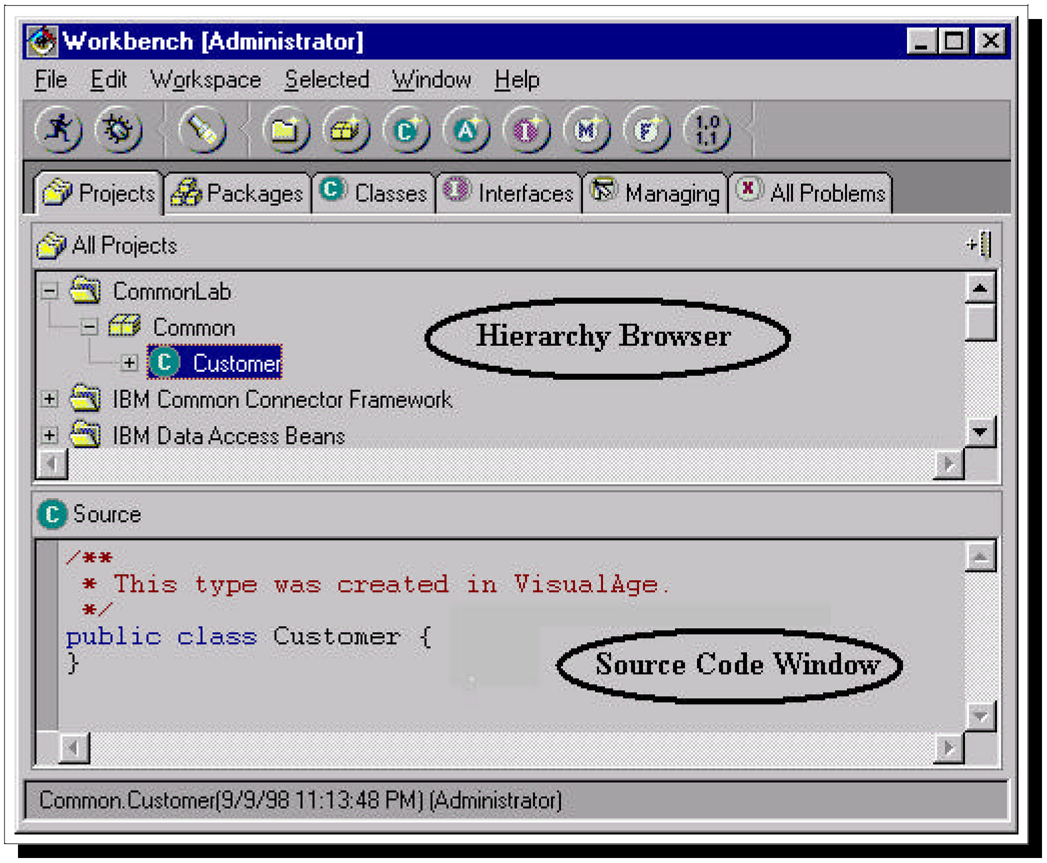
VAJava does not require you to do this compile step while working within the tool.

If we were not working inside VAJava, we would indicate that our class is part of a particular package by adding the line “**`package MyPackage;`**” at the top of our source file (which VAJava generates when you export out of the tool).

Step 5. Coding Our First Method

INSTRUCTIONS:

5a. Ensure the newly created class **Customer** is selected in the **"All Projects"** or ***hierarchy browser*** pane (top) and click in the "***Source code window***" pane at the bottom:


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
CommonLab
Common
Customer
IBM Common Connector Framework
IBM Data Access Beans
Hierarchy Browser
Source
Source Code Window
Common.Customer(9/9/98 11:13:48 PM) (Administrator)

```
/**
 * This type was created in VisualAge.
 */
public class Customer {
}
```

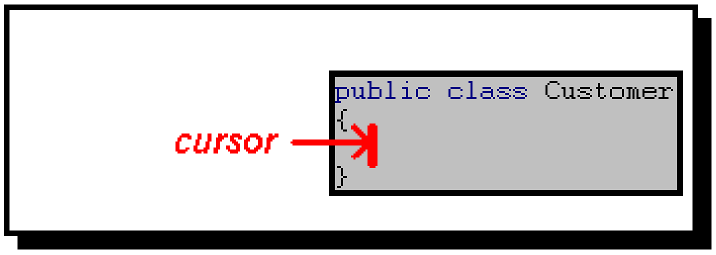
We will type our first code here. Note the comments at the top of the window. Comments in Java come in two forms:

- **Multiple line**: These start with "`/*`" and continue until an ending "`*/`" pair is found.
- **Single line**: To put a comment on a line or end of a line, start it with `//`

The comments here were generated by the ***Create Class SmartGuide*** we used to create this class, as was the empty class "shell" code.

5b. Position your cursor, just before the first brace - **{** - and press **Enter** to move it down one line. This lines up our braces - a style some people prefer.

5c. Press the **End** key to move the cursor to the end of the line and press **Enter** again to insert a new line between the braces...

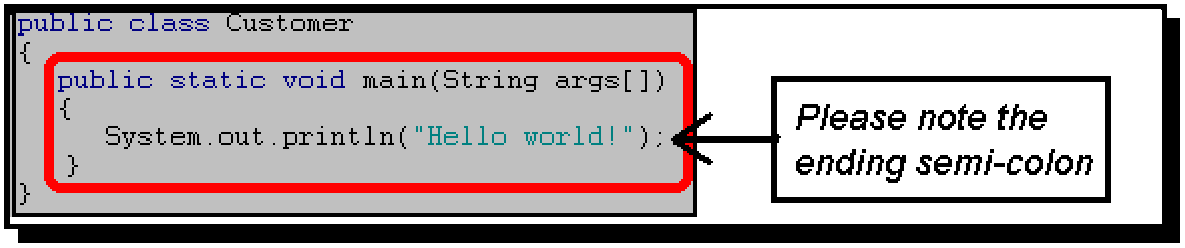


```
public class Customer
{

}
```

5d. Now, we will type our first method! Type in the following, exactly as shown:

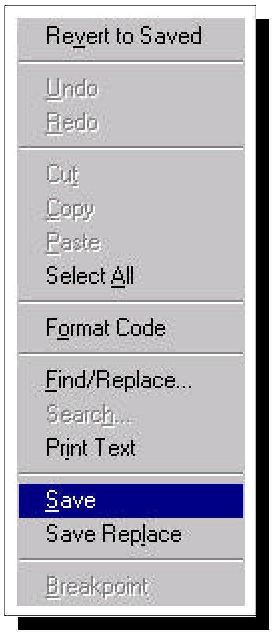
NOTES ABOUT TYPING:::

- ***Case is important***. Java names are case sensitive. “`MyVar`” does ***not*** equal “`myvar`”.
- ***White space is not important.*** Leave/insert as many blanks as you like.
- ***Watch for the semi-colons*** (;) at the end of executable lines of code! They are important.
- ***Note the colors*** in the VAJava editor. This coloring can aid in readability, and help easily see missing comment delimiters.

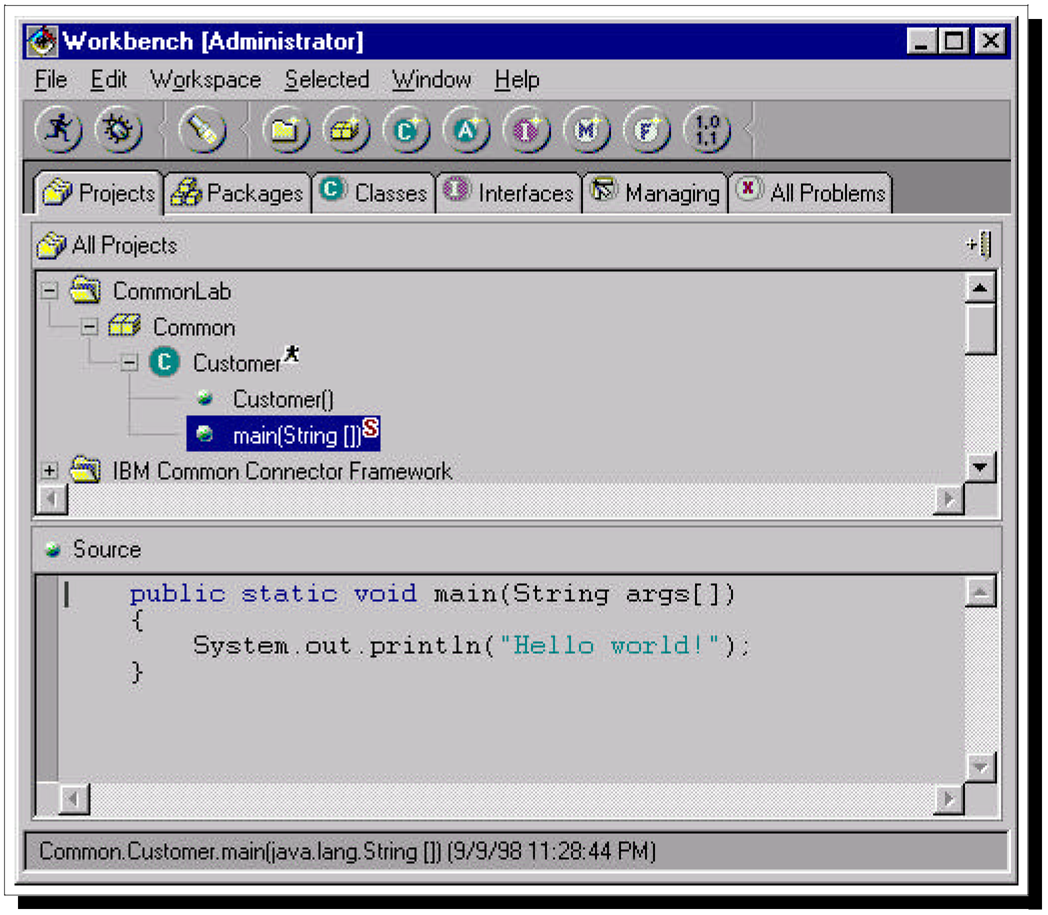


```
public class Customer
{
  public static void main(String args[])
  {
     System.out.println("Hello world!");
  }
}
```

5e. Right click (mouse button 2) in the source area, and select **Save** from the popup...



This will take a few moments as **VisualAge for Java** “***incorporates***” your changes. After it has finished you will notice that the source window has changed, and so has the hierarchy tree at the top...


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
CommonLab
Common
Customer
Customer()
main(String [])
IBM Common Connector Framework
Source

```
public static void main(String args[])
{
    System.out.println("Hello world!");
}
```

Common.Customer.main(java.lang.String []) (9/9/98 11:28:44 PM)

<u>What has happened here?</u>: The code you typed was for a new method. Typically, in VAJava you would use the ***New Method*** SmartGuide from the toolbar to create a new method, but we are "getting our feet wet" and writing it by hand instead. The method you created is called ***main***, and VAJava recognized that you have written a method and so updated the hierarchy to reflect that new method (inside the Customer class where you wrote it). It then automatically selected that method so you are looking at the source code for only it. The source window pane only reflects the source for the selected item - in this case, the method ***main***.
<u>Note:</u> VAJava also automatically created the "Customer()" method for us, when we created the class.

<u>Note:</u> *if there were errors in what you typed VAJava will inform you of this and display the error message on the message line at the bottom of the Workbench window. Ensure you have typed exactly what is shown here and retry the Save operation until it saves without error. Did you honor the case? Did you type the* ***semicolon*** *after the* ***System.out.println*** *statement?*

Before describing what it is you typed in, we are actually going to run it!! See that little running man icon beside the **Customer** class in the tree view? That implies this class is *runnable* - due to the existence of the **"public static void main(String args[])"** method we just created.

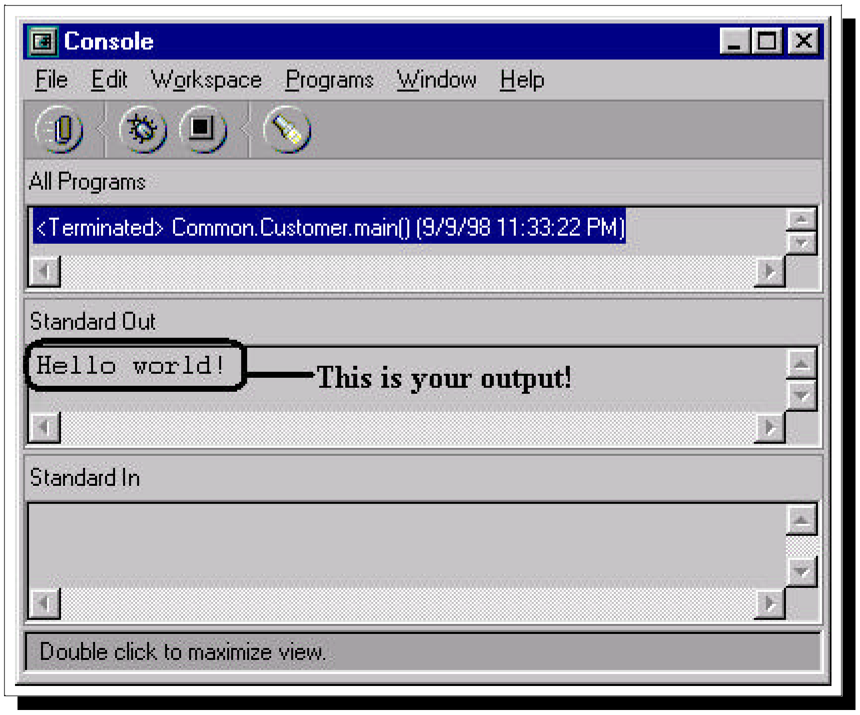
This is a special method that Java recognizes - and which it looks for any time you try to "run" a class.

5f. To run *this* class, from the toolbar select the "running man" icon

... this will run your code!

Note: when you "run" a Java class like this, Java looks for and runs the main explicitly named "main" in it, which is the one we just created.

What do you see? You should see a ***Console*** window come up - this is where VAJava shows anything written to standard output (as we did with ***"System.out.println(...)"***).



```
<Terminated> Common.Customer.main() (9/9/98 11:33:22 PM)
```

```
Hello world!
```

Double click to maximize view.

NOTE: If your Console window shows additional text, it is from previous labs. Select the Edit->Clear pulldown item to clear the window.

5g. Close this console window (click the Windows' "x" in the top right corner).

NOTES:

How do you feel? You have written and run your first bit of Java code. Here is what you wrote:

- **public** - this method is publicly accessible. In this case, it will be called from the command line.
- **static** - this method is static. That means it does not require an *instance* of the class to run it - this is just like regular procedures in RPG. We will cover the more interesting non-static methods shortly. Note, the "S" superscript beside **main** in the tree view is because of this.
- **void** - this method does not return anything. You must explicitly state this in Java using the keyword void.
- ***main*** - the name of this method. "**main**" is a special case method that is runnable directly from the command line by typing **"`Java classname`"** - in this case the Java runtime looks inside the specified `.class` file for the method called **"`main`"** and runs it.
- **String *args*[]** - this is the input to this method. It is an array (hence the square brackets []) of Strings - the Java equivalent of RPG alphabetic or character fields. If you pass parameters to the program via the command line, they go into this array - one entry per word passed:

```
                    JAVA MYCLASS HEY THIS "IS NEAT"
                         args[0] == "HEY"
                         args[1] == "THIS"
                         args[2] == "IS NEAT"
```

- **System.out.println(...)** - this is how you write strings to "standard out" in Java. Any strings passed to this as a parameter are displayed on the command line where the Java program was invoked. This is equivalent to sending a program message to **`*EXT`** on the AS/400, or using the RPG **`DSPLY`** opcode.
- *SemiColon* (**;**) - in Java, all statements end with a semicolon. As Java is a completely free format language, an explicit end of statement delimiter is important for the compiler. Contrast this to RPG, where a statement terminator is not required because it is a column oriented language.

What you have written will print out the string "Hello world!" on the command line when run from a command line. When run from within **VisualAge for Java** as we did, "standard out" output is shown in the console window, as we saw.

A note about braces in Java

Braces - **'{'** and **'}'** - are used to begin and end methods (eg, subroutines) as well as to begin and end methods. In fact, they also begin and end any block such as those inside "**`if`**" and "**`while`**" statements:

```
public class MyClass
{
   int myMethod()
   {
     if (aVariable > 10)
       {
       }
   }
}
```

Note that braces and all Java source is totally free format. Some prefer this style for braces:

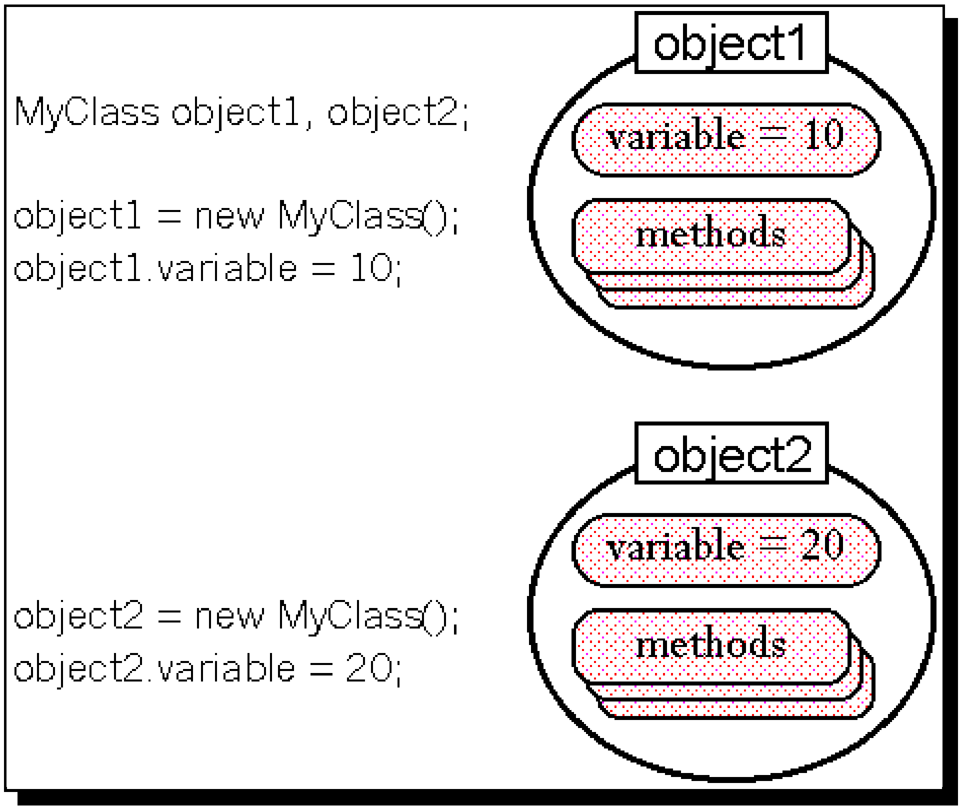
```
public class MyClass {
   ...
}
```

The Lab - Section 2: Objects and GUI

Section Introduction

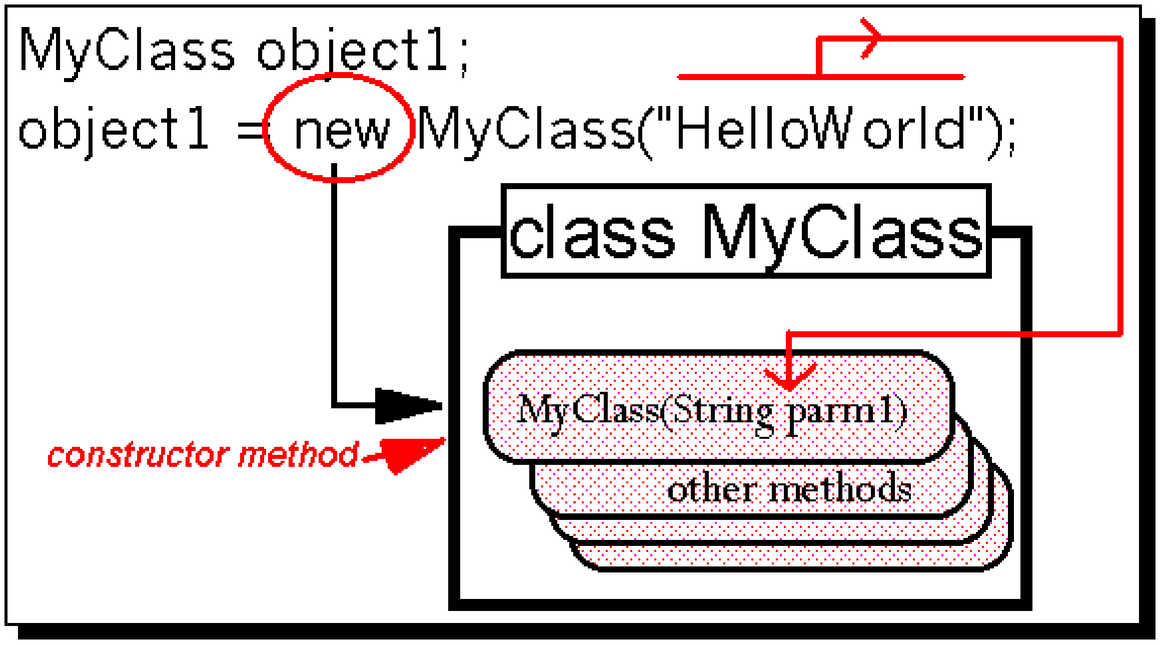
In this section we will pick up the pace considerably, and introduce you to:

- **Objects**. These are "*instances*" of classes, and are necessary to use classes that contain non-static methods or variables. They are created by defining a variable, specifying the class as the type, and equating the variable to an *instance* or *allocation* of the class using the **new** operator in Java.



```
MyClass object1, object2;

object1 = new MyClass();
object1.variable = 10;

object2 = new MyClass();
object2.variable = 20;
```

- **Instance variables**. These are non-static variables declared at the class level and available to all methods in the class. Each instance (object) of the class gets its own copy of these variables. Compare to global variables in RPG.
- **Local variables**. These are variables declared inside a method and are local to that method. They are only "alive" as long as the method is running.
- **Constructors**. These are special methods that each class can optionally have that are called by Java when the class is first "*instantiated*" (an instance is allocated). They are used to initialize variables and state, similar to RPG's ***INZSR** subroutine. They are identified by their name - it is the same as the class.



- **GUI**. Graphical User Interface. We will create a window to show our **“Hello World”** string in this time.

You will:

- Create a new “*helper*” class, called **FrameListener**, for handling the close action on our window. This class was previously coded for you, so you will just have to ***copy it*** in to your package.
- Create another new class called **MainWindow**, that implements a Java-supplied interface for handling GUI events, and uses the first helper class to process the close-window event.
- Define instance variables in this class for, among other things, a Java window (“**Frame**”).
- Define a constructor for the class which takes a string as input, and displays it in the title of the window. Recall that a constructor is just a method with the same name as the class.
- Instantiate the Java Frame class instance (create an ***object***).
- Back in the previous “**main**” method of class **Customer**, add code to instantiate and use this new **MainWindow** class.

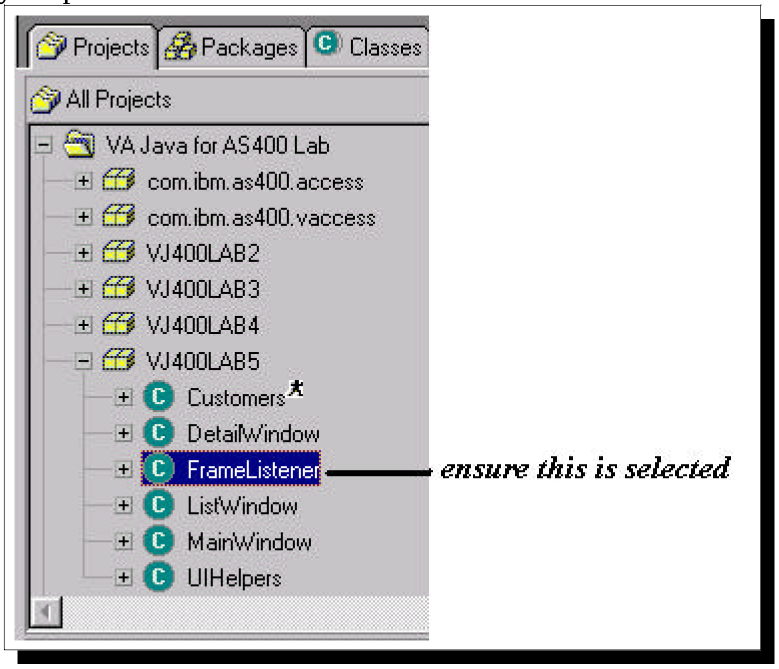
Ready? Let us continue our journey of Java, the language...

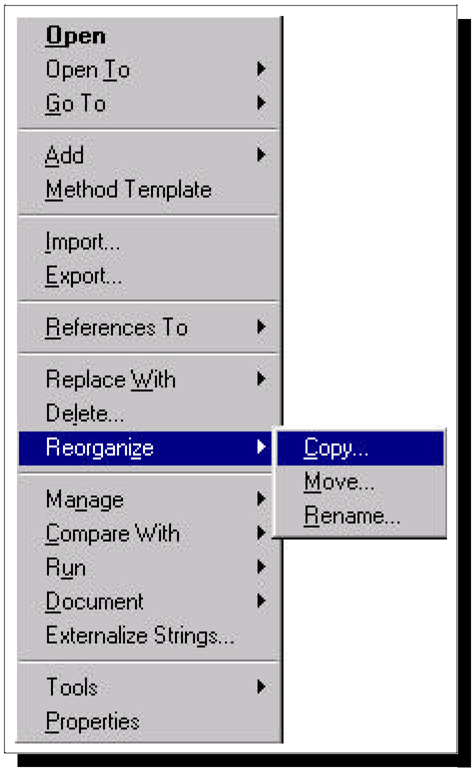
Step 1. Creating New Helper Class - "FrameListener"

When you create a ***Graphical User Interface*** (GUI) window in Java, unless you supply some specific code to handle the closing of it - there is no way you will be able to close it! So, before we create our window class we need to ***copy*** in a "pre-cooked" class that is explicitly used to process a window "close event".

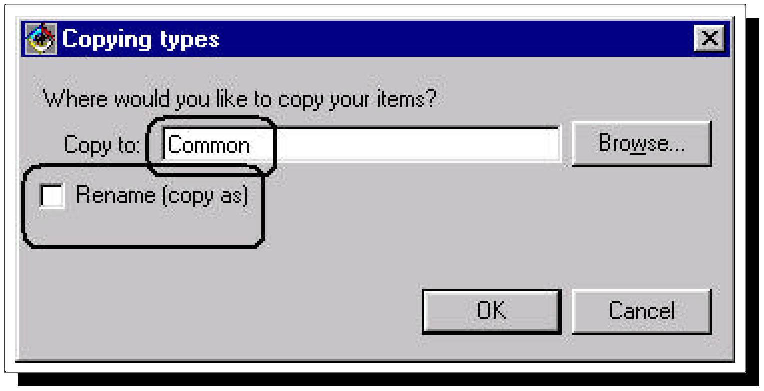
Note: ***in "real life" you would not copy but rather simply reuse it in-place.***

1a. Look for a ***project*** called **VA Java for AS400 Lab**, and expand it (by selecting the plus sign '+' beside it).
1b. Inside the project, expand the ***package*** called **VJ400LAB5**
1c. ***Left click and then right click*** (mouse button 2) on ***class*** **FrameListener** and select the **Reorganize->Copy...** option.





1d. In the Copying Types window, type in **Common** for the target package...
1e. Now ***deselect Rename*** and press the ***OK*** push button in bottom right.

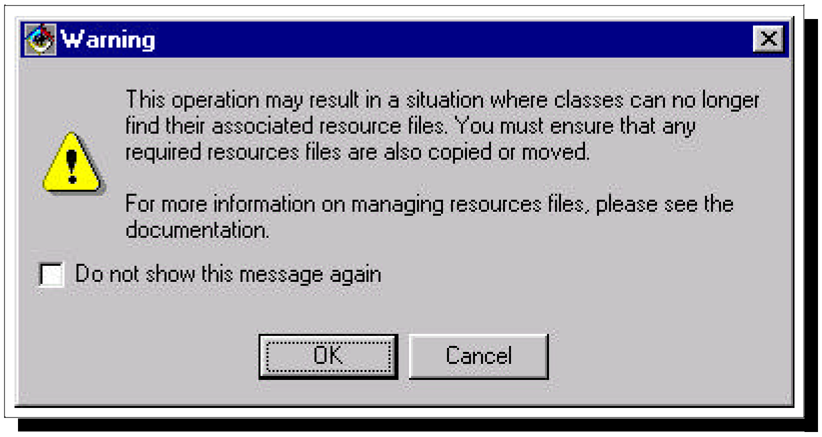

Copying types

Where would you like to copy your items?

Copy to: Common

☐ Rename (copy as)

NOTE: You will probably get the following warning message...



This operation may result in a situation where classes can no longer find their associated resource files. You must ensure that any required resources files are also copied or moved.

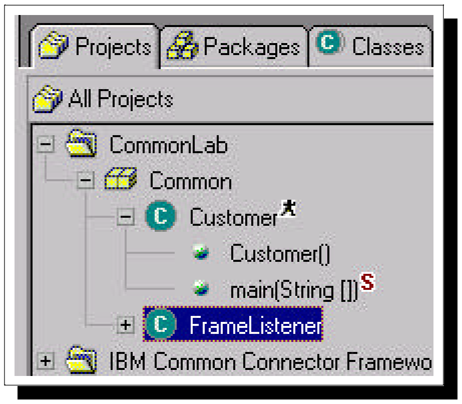
For more information on managing resources files, please see the documentation.

☐ Do not show this message again

THIS IS OK! So just press OK.

OTHER PROBLEMS? Make sure the FrameWindow class is selected!

1f. Now go back to your **CommonLab *project***, and you should see the new class, **FrameListener**, you just copied in:

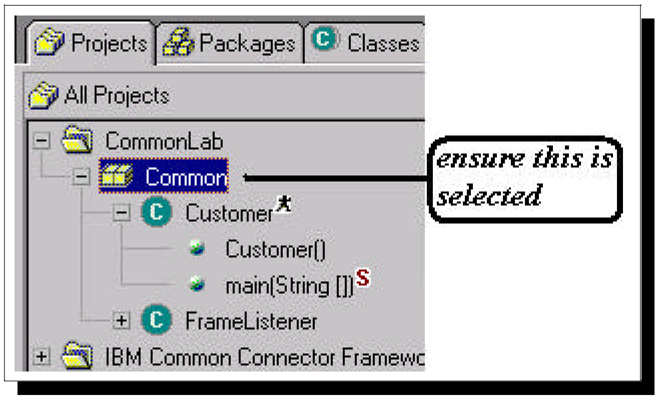


Step 2. Creating A New Window Class

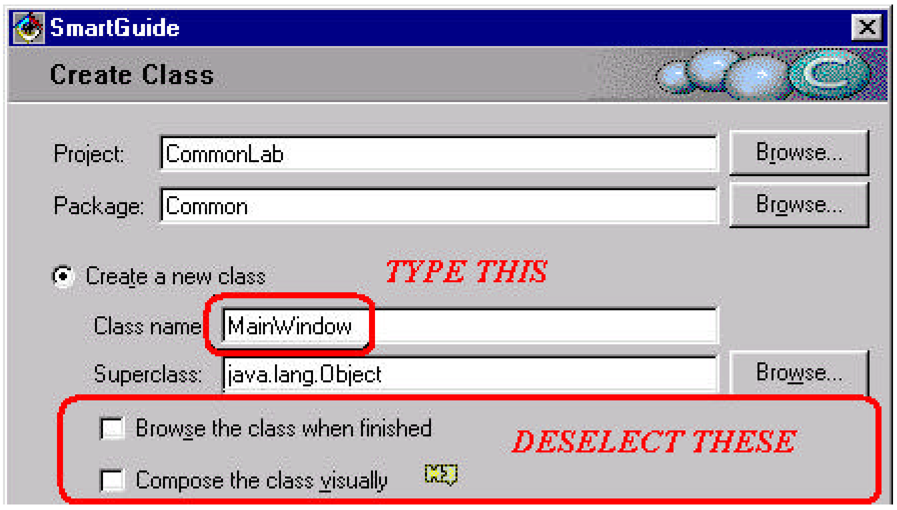
Now we will create yet another class, our third so far. This class will instantiate a Java window object and display the window. The window's title bar will be set to the passed-in string. We will then expand on this.

For our user interface code, we will use a Java supplied "package" (collection of classes, like an ILE service program which is a collection of modules). This package is called **AWT** (***Abstract Windowing Toolkit***), and it's fully qualified name is java.awt, as you will see.

2a. Select the package "**Common**" in the tree view:



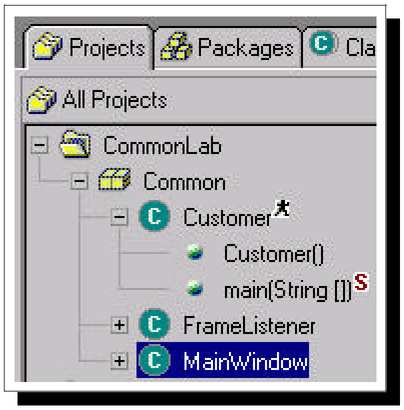
2b. Now, select the ***New Class*** icon in the toolbar , and specify `MainWindow` for the name of the class, and deselect the checkboxes:



Project: CommonLab
Package: Common
● Create a new class
Class name: MainWindow
Superclass: java.lang.Object
☐ Browse the class when finished
☐ Compose the class visually

2c. Press "**Finish**" to have VAJava create this new class.

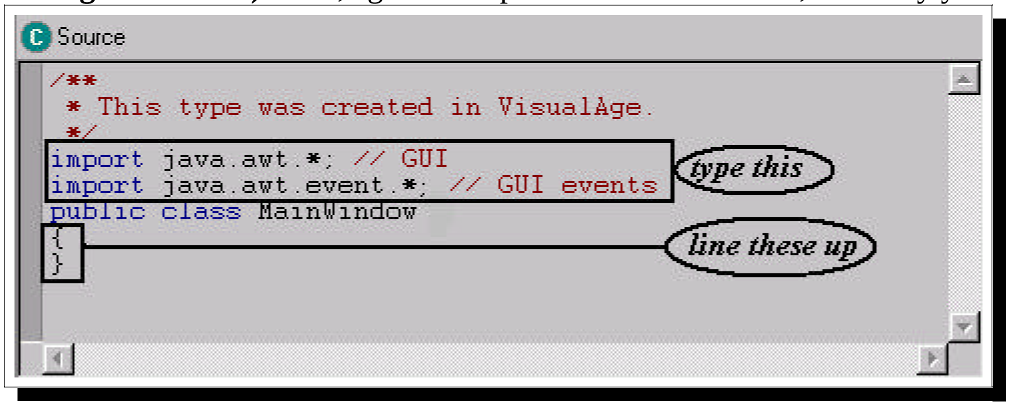
You will now see the new **MainWindow** class in the tree view, under the **Common** package, and the source for the class will be shown in the ***Source*** window pane.



2d. It is time again to write some code. Position the cursor in the source window pane, and before the start of the **`class`** statement, add the following **import** statements:

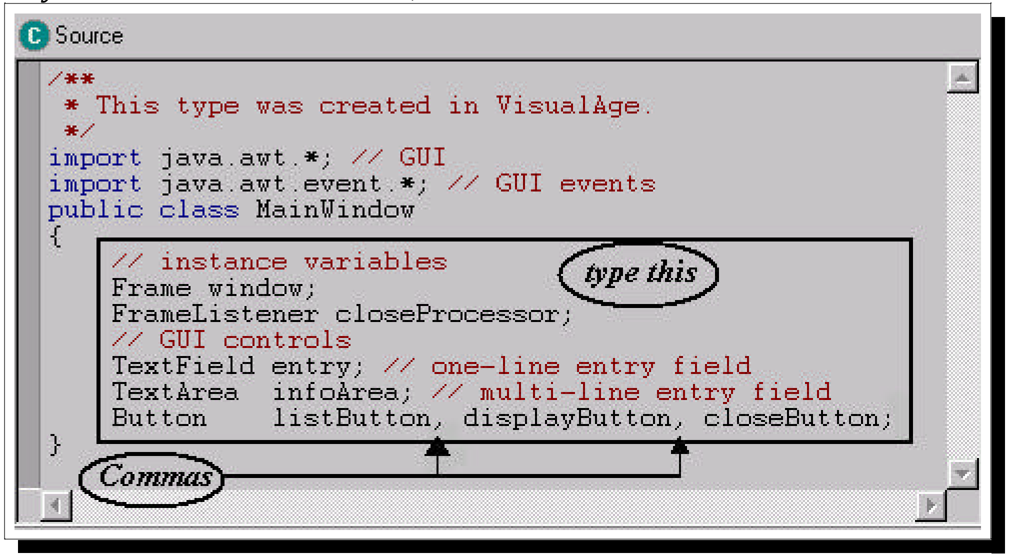
```
    import java.awt.*; // GUI
    import java.awt.event.*; // GUI events
```

(note the ending semicolons). Also, again line up the braces of the class, so finally you have:



```
/**
 * This type was created in VisualAge.
 */
import java.awt.*; // GUI
import java.awt.event.*; // GUI events
public class MainWindow
{
}
```

2e. Now, between the braces we will declare some class instance variables. Type in the **following** source ***exactly*** as shown in the boxed area, between the braces:



```
/**
 * This type was created in VisualAge.
 */
import java.awt.*; // GUI
import java.awt.event.*; // GUI events
public class MainWindow
{
    // instance variables
    Frame window;
    FrameListener closeProcessor;
    // GUI controls
    TextField entry; // one-line entry field
    TextArea  infoArea; // multi-line entry field
    Button    listButton, displayButton, closeButton;
}
```

2f. Press **CTRL+S** to **SAVE YOUR CHANGES**!

Did you get an error? That is, is there a message on the message line at the bottom to the effect "Field type XXXX is missing" and an X beside the MainWindow class name in the tree view? If so, it probably means you typed an uppercase letter as lowercase or vice versa. Eg, "Textfield" versus "TextField". Change it and resave.

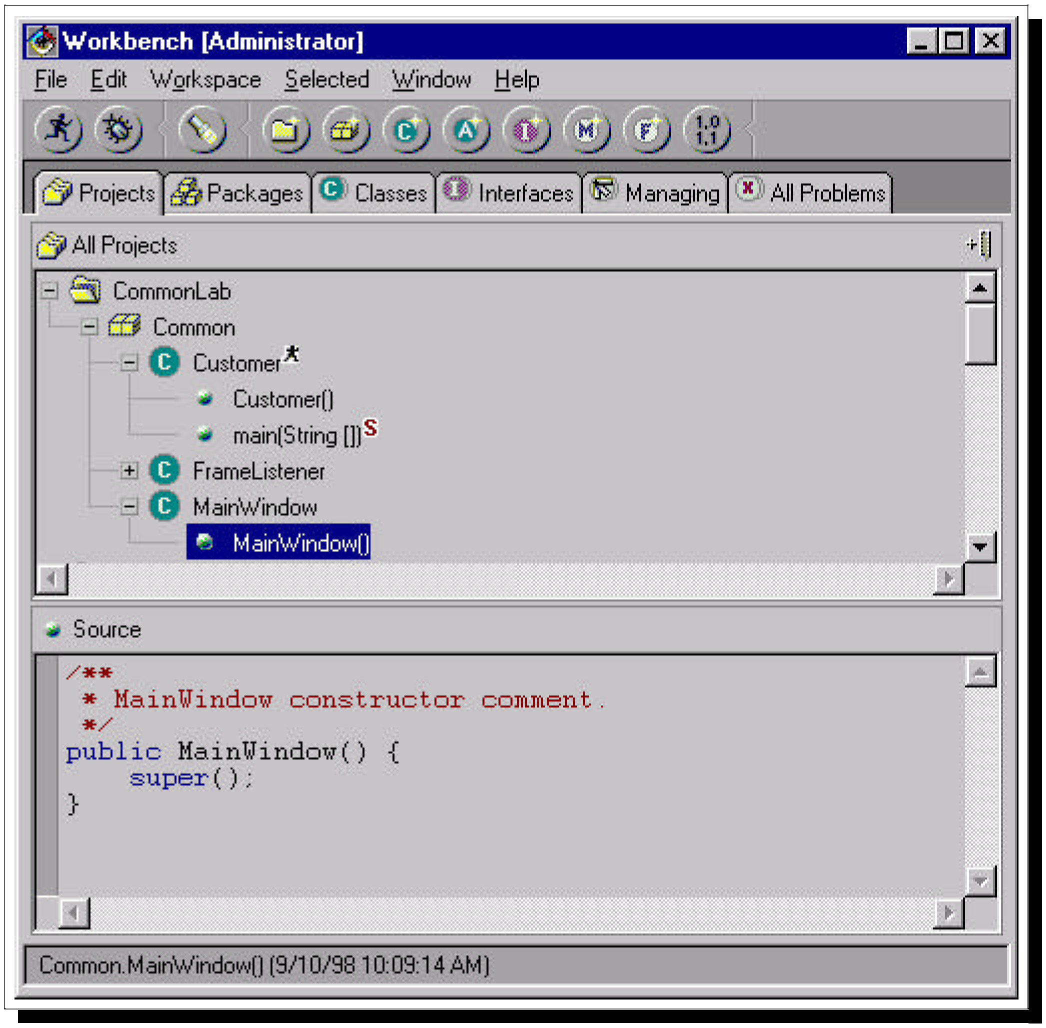
Note: you just declared *instance* variables for this new class:

- The variable ***window*** is of class type **Frame**. Frame is a Java-supplied class. In the constructor we will actually allocate an instance (*instantiate*) of **`Frame`** and assign it to this variable.
- The variable ***closeProcessor*** is of class type **FrameListener**. This is the class we copied in. We are going to *instantiate* an instance of this class in the constructor as well.
- The variables ***entry***, ***infoArea*** and ***listButton***, ***displayButton***, ***closeButton*** are graphical user interface controls or components that we will eventually be showing in our **Frame** window.
- **TextField**, **TextArea** and **Button** are classes defined in the **`java.awt`** package we *imported*.

Now we will create a **constructor** for this class. We are going to have the *users* of this class pass it a string that we will use for the window title. This will be done through the constructor, which is implicitly called by the Java **new** operator when a class instance is allocated ("**instantiated**" - we will see this **new** operator shortly).

REMEMBER: A CONSTRUCTOR IS A METHOD THAT HAS THE SAME NAME AS THE CLASS, AND HAS NO RETURN TYPE.

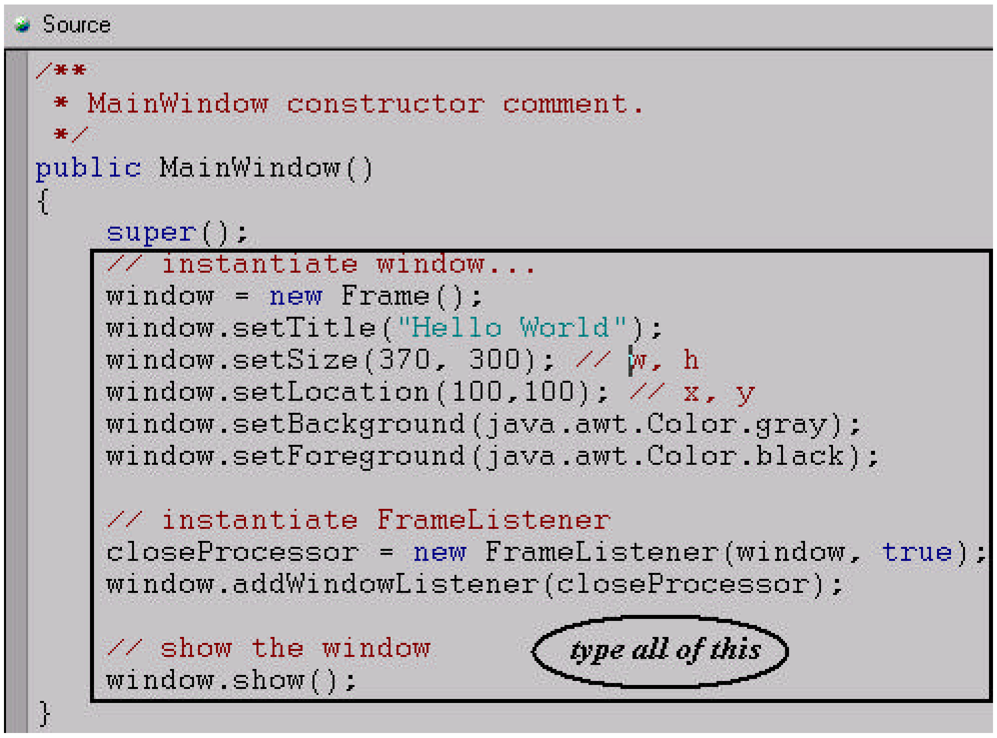
2g. As it turns out, VisualAge for Java automatically created a constructor method for us when we create a class. To see it for our MainWindow class, go to the tree view at the top and click on the plus sign beside the currently selected MainWindow class.
This will expand to show the constructor method MainWindow() underneath the class name. Select this method, so that you see this:


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
CommonLab
Common
Customer
Customer()
main(String [])
FrameListener
MainWindow
MainWindow()
Source

```
/**
 * MainWindow constructor comment.
 */
public MainWindow() {
    super();
}
```

Common.MainWindow() (9/10/98 10:09:14 AM)

__2h.__ In the source window, line up the braces again, and then ***after*** the "super();" line of code type all the following lines of code (and watch those ending semicolons, and the mixed case):



```
/**
 * MainWindow constructor comment.
 */
public MainWindow()
{
     super();
     // instantiate window...
     window = new Frame();
     window.setTitle("Hello World");
     window.setSize(370, 300); // w, h
     window.setLocation(100,100); // x, y
     window.setBackground(java.awt.Color.gray);
     window.setForeground(java.awt.Color.black);

     // instantiate FrameListener
     closeProcessor = new FrameListener(window, true);
     window.addWindowListener(closeProcessor);

     // show the window
     window.show();
}
```

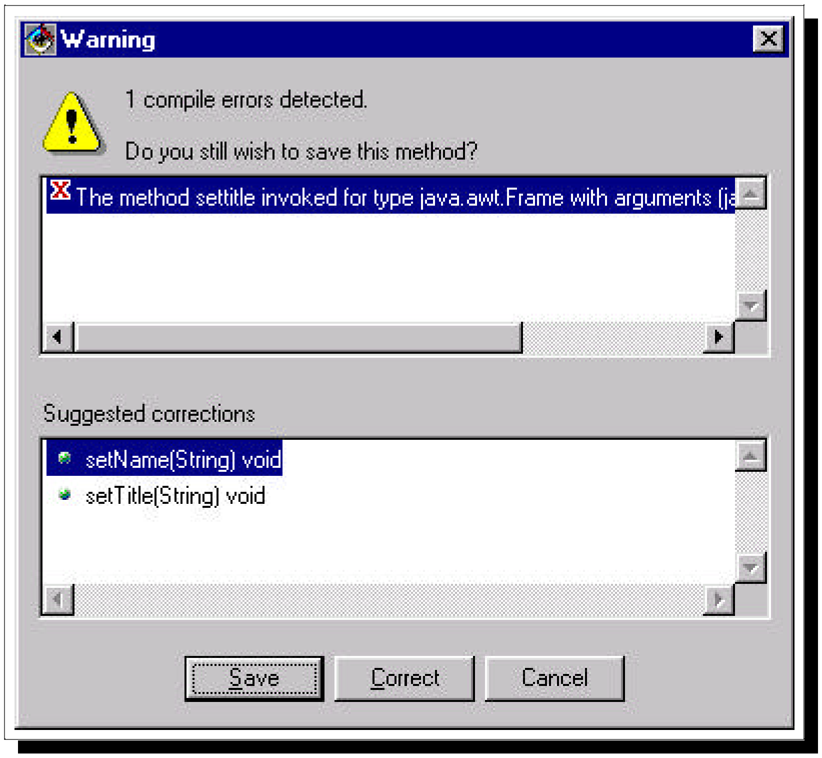
What are all those dots?

*In Java, you refer to methods inside an object by "**qualifying**" the method name with the name of the object:*

`object.method()`

2i. Press **Ctrl+S** to save all this typing!!

DO YOU SEE A WINDOW LIKE THE FOLLOWING? ...


Warning

1 compile errors detected.

Do you still wish to save this method?

The method settitle invoked for type java.awt.Frame with arguments (ja

Suggested corrections

If so (you may not!), this is an indication you have made a typing error - such as not getting the mixed-case *exactly* as shown (Java variable and method names are very much case sensitive). If you got this message, press **Cancel** and fix the error message (the code in error will be selected, and the error message will display on the very bottom of the Workbench window).

Once you save without any errors, read the following to understand **what you did**:

- ***Instantiate a Frame class.*** We have instantiated (eg, allocated) an instance of the **Frame** class, using the **new** operator, and assigned it to the instance variable **window**.

```
window = new Frame();
```

- ***Tailor the window.*** We then call a number of the **Frame** class methods on our object (referred to by object variable **window**) to specifically change the appearance of the window (eg, change the title, size, location and colors).
-

```
window.setTitle(“Hello World”);
window.setSize(370,300); // w, h
window.setLocation(100,100); // x, y
window.setBackground(java.awt.Color.gray);
window.setForeground(java.awt.Color.black);
```

- ***Instantiate a FrameListener class***. This is the class for which we created an object instance variable (**closeProcessor**) in the previous substep. The constructor for this **FrameListener** class takes two parameters - a **Frame** object reference and a **boolean** value. Any parameters specified on the "**new**" call get passed into the ***constructor*** by Java.

```
closeProcessor = new FrameListener(window, true);
```

- ***Register the FrameListener object for this window***. For the **Frame** window we created, we call its **"addWindowListener()"** method to register the **FrameListener** object we created. Java will call the special **"windowClosing"** method of this class now whenever the user tries to close the window.

```
window.addWindowListener(closeProcessor);
```

- ***Show the window***. Call the **show()** method. By default, newly instantiated **Frame** windows are not visible.

```
window.show();
```

Step 3. Using our new MainWindow Class

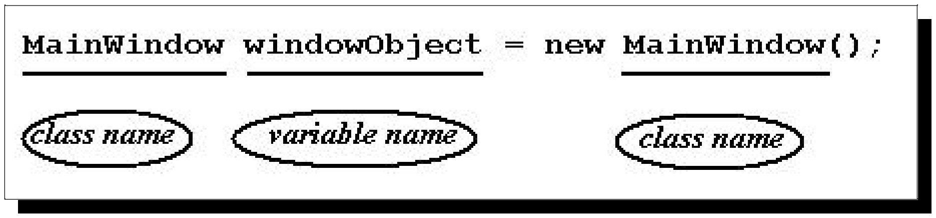
Let's have a look at this last class we just coded. Somewhere, we need to add code to "use" our class. To do this we need to:

- Declare an object variable of type **MainWindow** (our class name).
- Equate the variable to an instance of **MainWindow**, using the **new** operator.
- Pass in the parameter (a String) that **MainWindow**'s constructor expects, on the **new** operation.

The correct place to do this is in the "**main**" method in our initial "**Customer**" class, since that is what gets control initially.

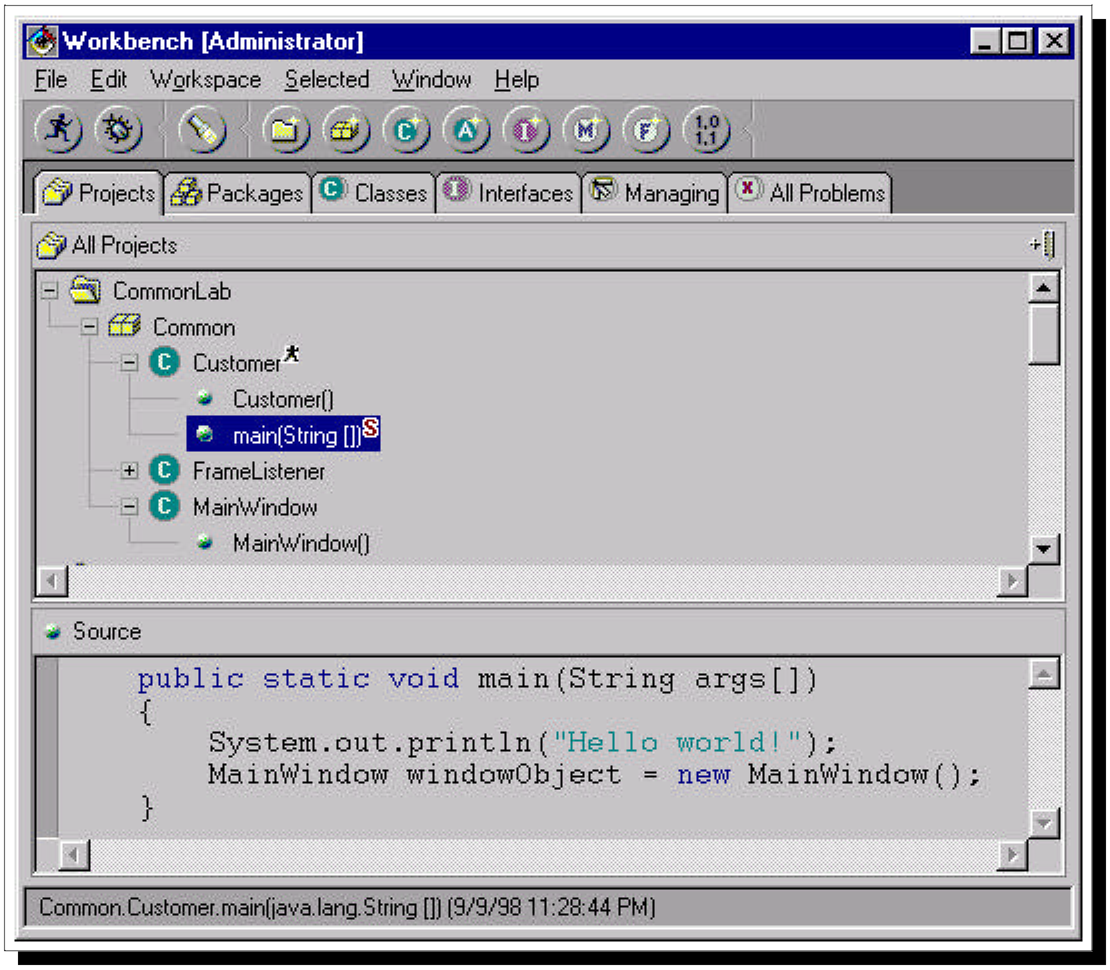
3a. Select the **"main(String[])"** method from the tree view, and then click in the ***Source window*** for it.

3b. *After* the **System.out...** line, add the following line of code:



```
MainWindow windowObject = new MainWindow();
```

You should see then the following...

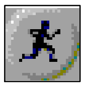

Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
CommonLab
Common
Customer
Customer()
main(String [])
FrameListener
MainWindow
MainWindow()
Source

```
public static void main(String args[])
{
    System.out.println("Hello world!");
    MainWindow windowObject = new MainWindow();
}
```

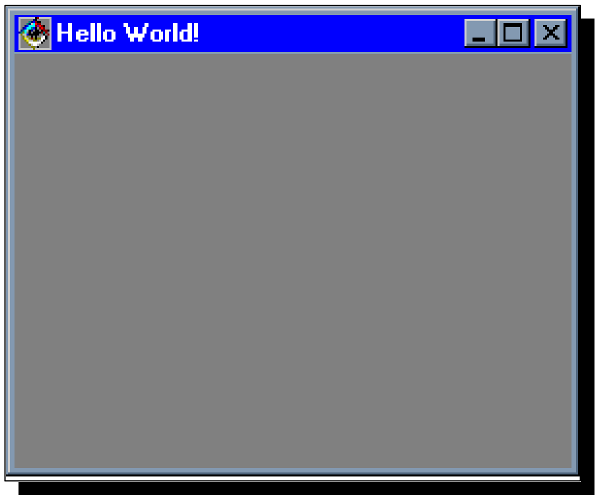
Common.Customer.main(java.lang.String []) (9/9/98 11:28:44 PM)

That is it - that is all the code you need to use that new class. This is because the **new** operator calls the constructor for the class, which as you recall creates and shows the window.

3c. *Save*. (Press **Ctrl+S**).

3d. *Run*. (From the toolbar, select )

When it runs, you should see the console again (you still have the `System.out` statement), and then your new window! (if you do *not* see the console, select ***Window->Console*** from the Workbench window menu).



Notice that:

- The title is the "Hello World" string we passed in the call to `setTitle`.
- The color is what we set it to with: `window.setBackground(java.awt.Color.gray);`
- The window size and location is what we set it to with `window.setSize(370, 300);` and `window.setLocation(100,100);`

<u>3e.</u> *Close it*. (press the '**x**' in the upper right corner).

Step 4. Adding Stuff to the Window

You have seen your first window in Java. Congratulations.

NOTES:

In RPG, you create **DSPF DDS** to define your user interfaces. Java has no such *externally described* user interface language. Your code dynamically creates the whole thing using the supplied classes in the **java.awt** package, as you will see.

VisualAge for Java does have a *What You See Is What You Get* (WYSIWYG) tool for visually creating your user interfaces (like **SDA**). However, our goal here is to get you familiar with the language itself by actually "touching" it, so we will code our window's user interface by hand.

Here are some comparisons of AS/400 display files and Java **AWT** user interfaces:

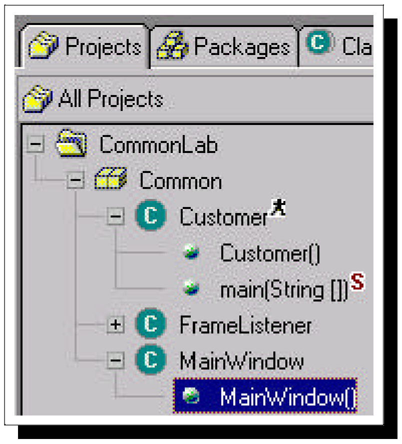
- In display files, you define ***record formats*** that contain ***fields*** of many types (eg named fields, constants).
- In Java AWT, you create ***Panel*** objects that contain ***components*** of many types (eg **TextField**, **Label**) via the Java **Panel** class's **add()** method.

- In display files, you combine multiple record formats to produce one ***screen***, using RPG **WRITE** and **EXFMT** statements.
- In Java AWT, you create a **Frame** or **Dialog** window object and combine multiple panels to form the ***client area*** (via Java window object **add()** method). **Note:** you can also add components (eg, "fields") *directly* to a window object versus a panel.

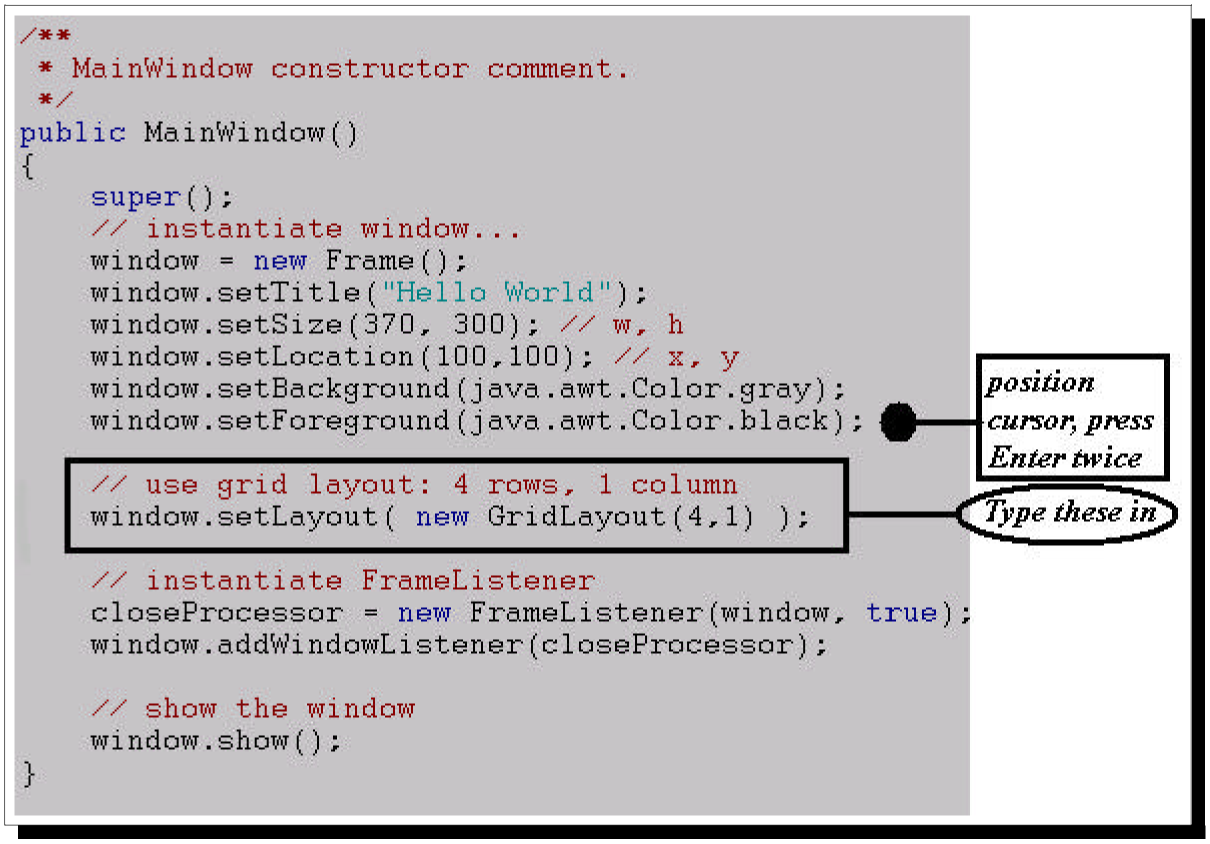
- In display files, all fields are placed using explicit row and column numbers.
- In Java AWT, components can be placed in one of numerous ways, depending on the ***Layout Manager*** you choose (via the window object **setLayout()** method). These include:
  - **BorderLayout**: Divides the window into 5 regions: top ("**North**"), bottom ("**South**"), left ("**East**"), right ("**West**") and center ("**Center**");
  - **FlowedLayout**: Simply "flows" each component added left to right, wrapping when needed.
  - **GridLayout**: Divides the window into equally sized cells, given the number of rows and columns.
  - **GridBagLayout**: Like **GridLayout** but allows different sized cells. More complicated to code but offers the most flexability.

Now, let's add some components into the window, to make it more interesting... we will add code to the constructor to create these.

4a. Select *constructor method* **MainWindow** in *class* **MainWindow** in the tree view:



4b. In the ***source window-pane***, insert the two lines show here...



```
/**
 * MainWindow constructor comment.
 */
public MainWindow()
{
    super();
    // instantiate window...
    window = new Frame();
    window.setTitle("Hello World");
    window.setSize(370, 300); // w, h
    window.setLocation(100,100); // x, y
    window.setBackground(java.awt.Color.gray);
    window.setForeground(java.awt.Color.black);

    // use grid layout: 4 rows, 1 column
    window.setLayout( new GridLayout(4,1) );

    // instantiate FrameListener
    closeProcessor = new FrameListener(window, true);
    window.addWindowListener(closeProcessor);

    // show the window
    window.show();
}
```

What this does is divide the screen into a grid of 4 rows and 1 column. Each "field" we add now will be added to the next available grid cell. Each cell has the same width and height.

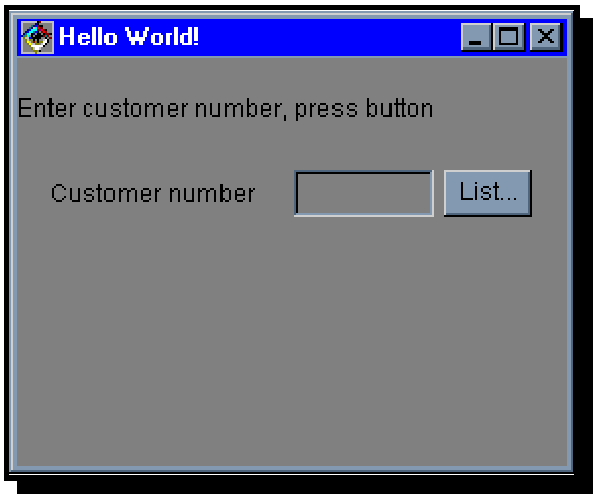
Next, we will add "components" or "parts" into the first 2 grid cells (just read the following two bullets, don't do anything yet!):

- Row 1: A text constant ("**Label**" in Java terms).
- Row 2: A **Panel** containing a text prompt (**Label**), an entry field (**TextField**) and a pushbutton (**Button**). Panels are Java container objects for grouping GUI components together.

4d. Continuing where we inserted the last two lines, now insert the following lines (*comment lines are optional*):

```
// add text constant ("Label") in row 1:
window.add( new Label("Enter customer number, press button") );

// Create sub-panel to hold text and entry field
Panel prompt = new Panel();

// Flow items in sub-panel, left to right...
prompt.setLayout( new FlowLayout() );

// create entry field and list pushbutton...
entry = new TextField(7); // max 7 characters
listButton = new Button("List..."); // Pushbutton

// populate with text-prompt, entry-field, pushbutton
prompt.add( new Label("Customer number") );
prompt.add( entry );
prompt.add( listButton );

// add sub-panel to window grid, row 2:
window.add(prompt);
```

4e. Now, *save* (**Ctrl+S**) and *run* (select class **Customer** and then the toolbar's **Run** icon). Let's see our window now...



Enter customer number, press button

NOTE: if it does not look exactly like this, try stretching it.

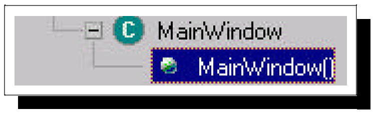
4f. Isn't this fun? ***Close your window***, and let's add more components to it...

(PS: if you want to play with the foreground and background colors, others besides **gray** and **black** are: **lightGray**, **darkGray**, **blue**, **cyan**, **green**, **magenta**, **orange**, **pink**, **red**, **white** and **yellow).**

Step 5. Adding More Stuff to the Window

You have seen your first window in Java. Congratulations. Now, let's add the remaining stuff to make it more interesting...

5a. Ensure constructor *method* **MainWindow** in *class* **MainWindow** is selected:



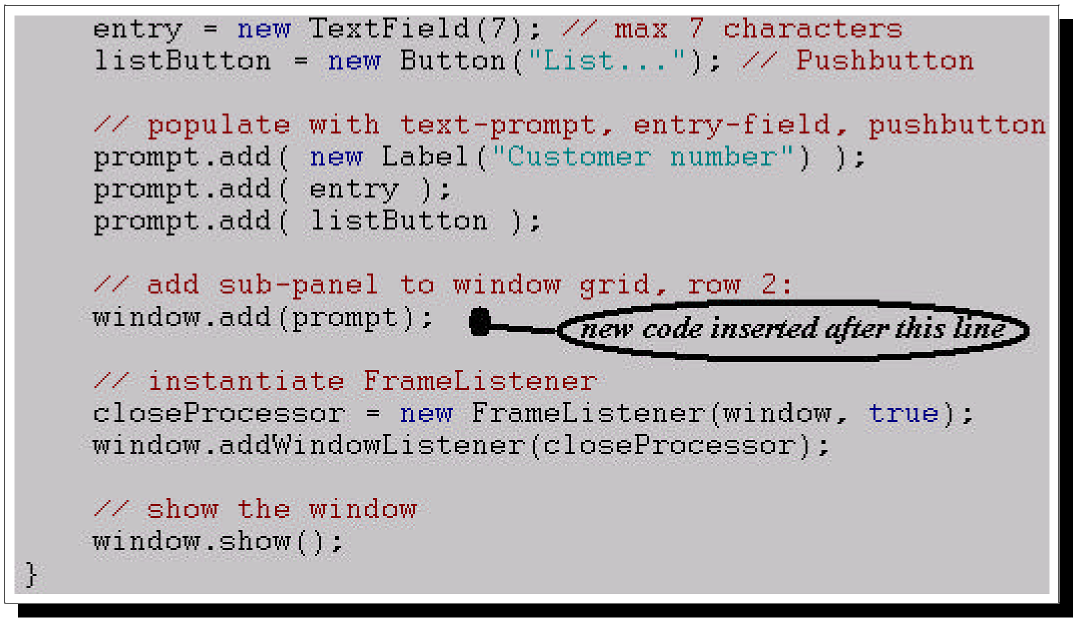
We will now add code to create and show pushbuttons at the bottom ("Display..." and "Close") and below those an information line to display informational and error messages...

FYI: you now will be adding code ***AFTER what you just typed...***

```
     // add sub-panel to window grid, row 2:
     window.add(prompt);
```

and ***BEFORE these lines***

```
     // instantiate FrameListener
     closeProcessor = new FrameListener(window, true);
```

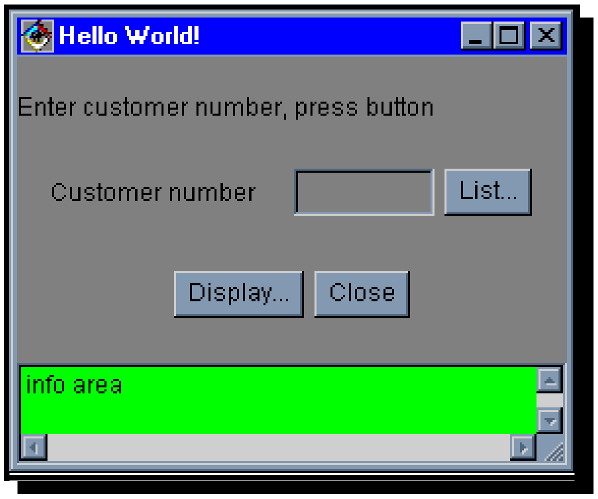


```
    entry = new TextField(7); // max 7 characters
    listButton = new Button("List..."); // Pushbutton

    // populate with text-prompt, entry-field, pushbutton
    prompt.add( new Label("Customer number") );
    prompt.add( entry );
    prompt.add( listButton );

    // add sub-panel to window grid, row 2:
    window.add(prompt);

    // instantiate FrameListener
    closeProcessor = new FrameListener(window, true);
    window.addWindowListener(closeProcessor);

    // show the window
    window.show();
}
```

5b. Insert the following new lines of code (in location described above):

```
// Create subpanel to hold pushbuttons
Panel buttons = new Panel();
buttons.setLayout( new FlowLayout() );
// Create pushbuttons
displayButton = new Button(“Display...”);
closeButton = new Button(“Close”);
// Add buttons to subpanel
buttons.add( displayButton );
buttons.add( closeButton );
// Add button subpanel to Row 3 of window grid
window.add(buttons);

// Create information line
infoArea = new TextArea(“info area“, 2, 1); // 2 rows, 1 column
infoArea.setEditable(false); // read-only
infoArea.setBackground(java.awt.Color.green);

// Add information line to Row 4 of window grid
window.add(infoArea);
```

5c. Now *save* (**Ctrl+S**) and *run* (select the **Customer** class, and use the **Run** icon) again.



Notice that you can:

- Type in information in the “Customer number” entry field (the 7 character limit we gave it only affects initial display width unfortunately - it does not prevent you from typing more than 7 characters. For this you need to “intercept” keystrokes, which is beyond us today!).
- Press the “**`Display...`**” or “**`List...`**” pushbuttons - nothing happens! That’s ok!

Let us see how we make those pushbuttons come alive!

5d. ***Close the window*** (‘**x**’ in the corner).

Step 6. Processing Input

Event Driven Programming in GUI Systems

In RPG you display a screen by writing to one or more record formats, and retrieve data entered by the user by reading a record format. Reading a display file will return data in the fields and indicators (which indicate which key was pressed).

This is *Screen-driven* programming. Your program writes and reads screens of information.

In GUI environments, it is different. Your program gets "*notified*" of every single user action - pressing a key, pushing a button, moving the mouse, etc.. These actions are called ***events***.
Your program can choose to process individual events or let the system do its default action for them (usually nothing). This is called *event-driven* programming.

Rather than one large piece of code that processes user input - you will have multiple small pieces of code that respond to individual events.

Where do you put this code and how does the system know to invoke it?

Event Driven Programming in Java

In Java, "events" are Java objects (instances of Java classes) that are sent to your own class ***if you tell Java to***!

What are these Java event objects?

They are various classes (depending on the event that happened) that inherit from, or "**extend**", the Java class **java.awt.AWTEvent** - the "root" event class. Each unique event - such as **ActionEvent** for pushbutton clicks - sends to your program an object of the appropriate event class. From this object you can query (via method calls) information like the GUI component that triggered the event (**event.getSource**()).

How do I tell Java to send events to my class?

You have to do **three** things (don't do these yet, just read):

1. Indicate your class is capable of responding to these events by including the code "**implements xxxListener**" on the class definition, where xxx indicates the events you want to be informed of. For example, "**implements ActionListener**" will cause the system to inform you of ***action*** events (versus say, ***typing*** events or ***mouse move*** events).

2. Supply a method in your class that will be called for specific events. These methods have to use the *exact* names and parameter types that Java defines for each event. For example, for action events it requires the method **"public void actionPerformed(ActionEvent event)"**.
3. For each GUI component, such as a push button, after creating it you must "register" that it is to send its events to your class. Do this using the **"addActionListener(** *instance-of-your-class* **)"** method that all input-capable Java components support.

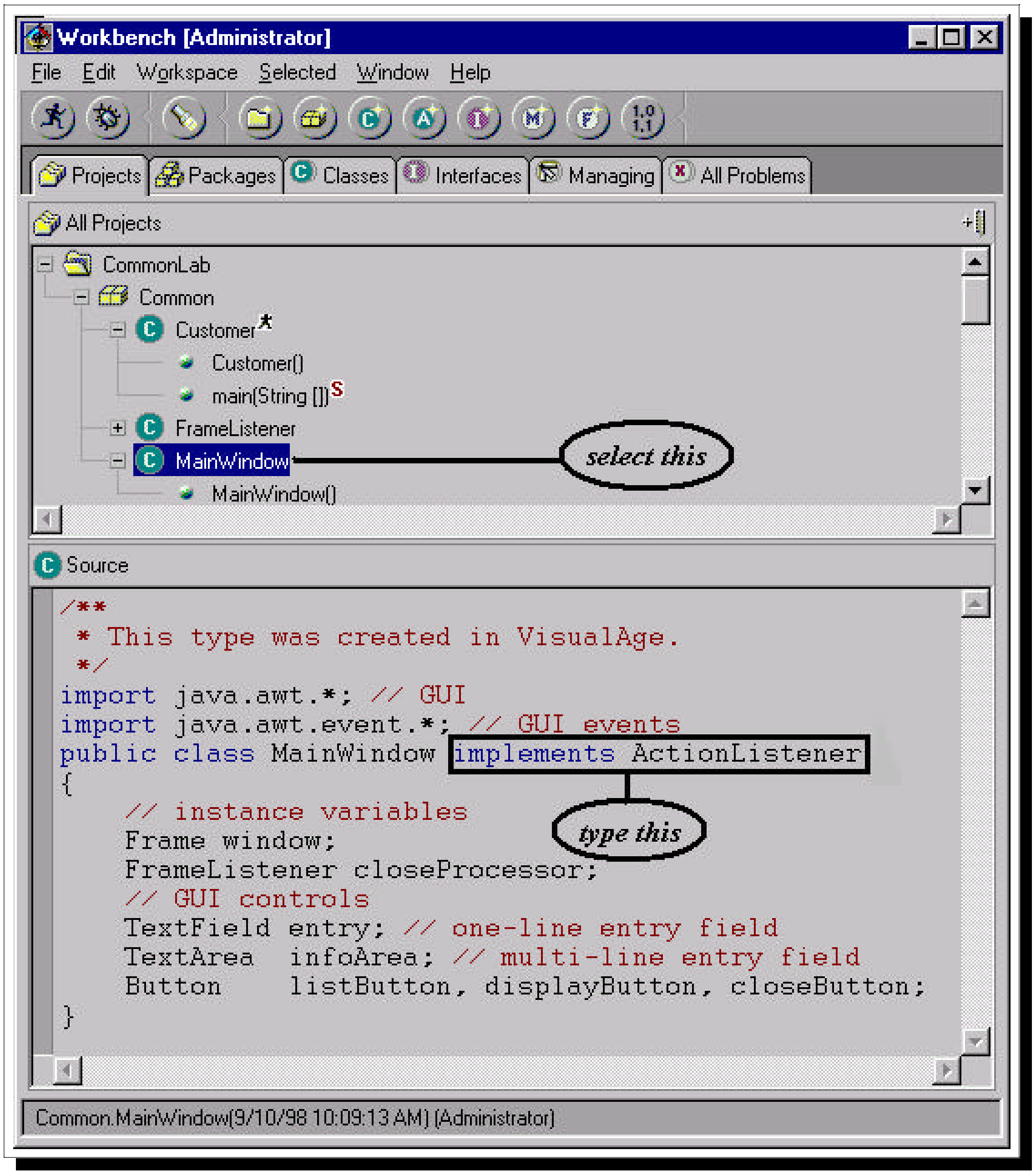
Let us look at an example of this...

Let's go back to our window example now, and add code to process three push buttons (**`List`**, **`Display`** and **`Close`**):

<u>**6a.**</u> Select the class **MainWindow** from the tree view. In the ***source window*** change the "**`class`**" definition line to read:

```
  public class MainWindow implements ActionListener
```

(see next page for picture of this)


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
CommonLab
Common
Customer
Customer()
main(String [])
FrameListener
MainWindow
select this
MainWindow()
Source
type this
Common.MainWindow(9/10/98 10:09:13 AM) (Administrator)

```
/**
 * This type was created in VisualAge.
 */
import java.awt.*; // GUI
import java.awt.event.*; // GUI events
public class MainWindow implements ActionListener
{
     // instance variables
     Frame window;
     FrameListener closeProcessor;
     // GUI controls
     TextField entry; // one-line entry field
     TextArea  infoArea; // multi-line entry field
     Button    listButton, displayButton, closeButton;
}
```

6b. Save this change by pressing **Ctrl+S.** ***** ERROR ***** You get an error message on the bottom of the Workbench window (and an X is shown beside the MainWindow class in the tree view), to the effect:

X Must implement the inherited abstract method void java.awt.event.ActionListener. .....

This is ok! You are getting this because you have not yet supplied the method - **addActionListener** - that Java insists classes which "implement ActionListener" have.

6c. Before the closing brace of the class - **}** - type in that missing method:

```
public void actionPerformed(ActionEvent event)
{
    // read the entry field contents
    String value = entry.getText();
    value = value.trim();
    if (value.length() == 0)
      value = "0";
    // process the pushbutton
    if (event.getSource() == listButton)
      infoArea.setText("List button pressed for '" + value + "'");
    else if (event.getSource() == displayButton)
      infoArea.setText("Display button pressed for '" + value + "'");
    else if (event.getSource() == closeButton)
      {
         window.dispose(); // close the window
         System.exit(0); // end this program
      }
}
```

Note 1: notice the double equals (==) for the **"if (... == ...)"** expressions. Java, unlike RPG, uses a doubled up equals for equality testing versus a single equals for assignment statements.

Note 2: notice how we read the current contents of the entry field - **"`entry.getText()`** ", and how we set the contents of the multiple line entry field - "**`infoArea.setText()`** ". You'll find Java almost always uses **`set/get`** methods as a convention for setting and retrieving values.

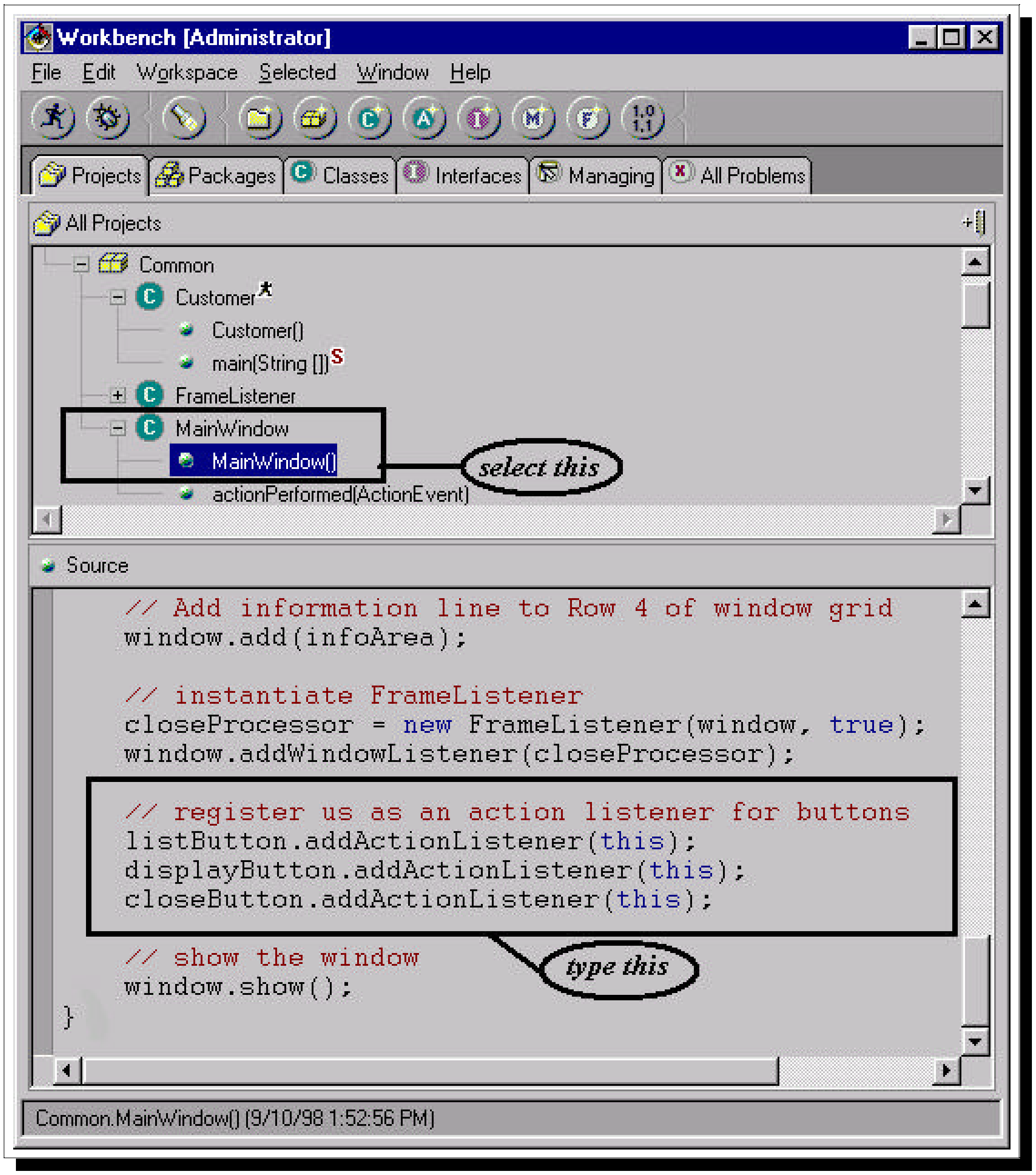
6d. Save your changes - and as usual your new method is added to the tree view, under the class, and is automatically selected. Note your error message and "X" disappear now too.

We are almost done - we have done 2 of 3 things we indicated were necessary to get and process events from Java. What is missing is to "register" with the GUI components that we want our class to be a target for their events.

6e. Select the *method* **MainWindow** in the *class* **MainWindow**. Add the following lines of code at the bottom of the method, *before* the "**window.show()**" line at the bottom:

```
// register us as an action listener for buttons
listButton.addActionListener(this);
displayButton.addActionListener(this);
closeButton.addActionListener(this);
```

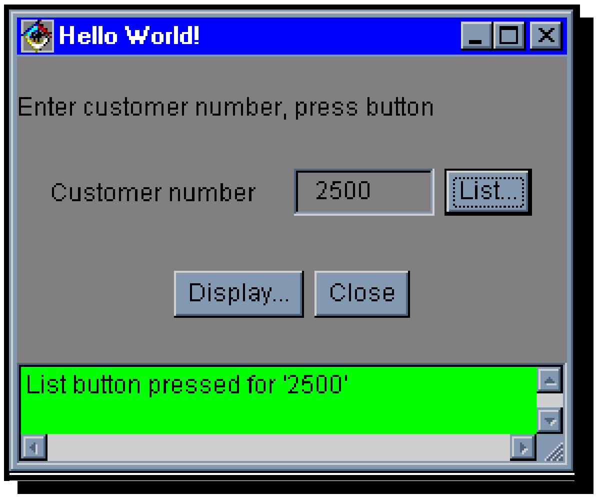
So you should have this...


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Classes Interfaces Managing All Problems
All Projects
Common
Customer
Customer()
main(String [])
FrameListener
MainWindow
MainWindow()
actionPerformed(ActionEvent)
select this
Source
type this
Common.MainWindow() (9/10/98 1:52:56 PM)

```
    // Add information line to Row 4 of window grid
    window.add(infoArea);

    // instantiate FrameListener
    closeProcessor = new FrameListener(window, true);
    window.addWindowListener(closeProcessor);

    // register us as an action listener for buttons
    listButton.addActionListener(this);
    displayButton.addActionListener(this);
    closeButton.addActionListener(this);

    // show the window
    window.show();
}
```

Note: "**this**" is a special Java built-in keyword that represents the current instance of the current class. So, for example, a reference to an instance variable, as in **`x=10`** is equivalent to **`this.x=10`**

6f. All right - **save** and **run**! When your window comes up, press the ***List*** and ***Display*** buttons to see if your new "**`actionPerformed`**" code gets control.



List button pressed for '2500'

Type something in the entry field, then press the buttons. Fun, no?

6g. Press the **Close** pushbutton to close your cool little window.

FYI: More on "Implements"

In this section you used the Java keyword "**`implements`**" for the class definition. This is used to tell Java that your class will "implement" the methods in an **interface**. An interface is just like a class, except that it:

- has only method prototypes, not method bodies (no code).
- has no instance variables.

Here is an example of an interface definition:

```
public interface Printable
{
    void print();
}
```

This defines a new interface called **Printable**. It contains one method signature - **print** - which takes no parameters and returns nothing (remember, that is what **void** means). Here is an example of a class that implements this interface:

```
public class MyClass implements Printable
{
    void print()
    {
        System.out.println("Here I am");
    }
}
```

*** Done the Lab! ****

For a two hour lab, you have done very well! This brief introduction to Java may have left you bewildered, but you got this far - congratulations. Now you can read a book on Java and perhaps you'll think - "hey, I remember that term!".

If you want to see all of the code for the finished application, including the AS/400 database access, with your web browser surf over to:

`http://www.software.ibm.com/ad/as400/vajava`

See you around the water cooler!

Appendix - Some Helper Methods

This section is not part of the two hour core lab. This is simply some helper code you may find useful as you start into Java....

We want our programs to have robust error checking code of the entry fields, as we are used to on the AS/400. Unfortunately, in Java this means we need to write some "helper" functions to:

- *Display an error message string*. This simply displays the string in the information area at the bottom of the window, positions the cursor at the entry field, and changes the entry field color to red. This is to mimic our beloved **DSPATR(PC RI)** DSPF keyword!
- *Display an error message string, with substitution*. The Java String class, while reasonably robust, does not have a method to replace one substring with another. We thus have to write our own message substitution methods - one each for 1 and 2 substitution strings.
- *Display a status message*. Similar to above, but does not position cursor or change the field's color.
- *Verify a given String is a valid positive integer within a given range*. If it is, returns that integer value.
- *Pad a positive integer with zeros*. Return the padded String.

All of these helper methods are coded so as to take as parameters everything they need to do their job. Thus, we pull them out into their own class and make each method **"static"** so you don't have to instantiate the class to use it - you just code "`UIHelpers.methodname()`".

1a. Create a new class called **UIHelpers**:

```
imports java.awt.*; // import Abstract Windowing Toolkit
public class UIHelpers
{

}
```

1b. Create all the following methods in this new class, stir in some of your own Java code that needs this error checking, and a pinch of blind faith, and cook over an open flame...

Helper Methods - Part 1 of 2

```
public static void displayError(TextComponent infoArea, Component field,
                                String msg)
{
   infoArea.setText(msg);
   field.requestFocus();
   setFieldColorsError(field);
}
public static void displayError(TextComponent infoArea, Component field,
                                String msg, String sub)
{
   displayError(infoArea, field, stringSubstitute(msg, "&1", sub));
}
public static void displayError(TextComponent infoArea, Component field,
                                String msg, String sub, String sub2)
{
   String newmsg = stringSubstitute(msg,"&1",sub);
   newmsg = stringSubstitute(newmsg,"&2",sub2);
   displayError(infoArea, field, newmsg);
}

public static void displayStatus(TextComponent infoArea, String msg)
{
   infoArea.setText(msg);
}

public static void displayStatus(TextComponent infoArea, String msg, String
                                 sub)
{
   displayStatus(infoArea, stringSubstitute(msg, "&1", sub));
}

public static void displayStatus(TextComponent infoArea, String msg,
                                 String sub, String sub2)
{
   String newmsg = stringSubstitute(msg,"&1",sub);
   newmsg = stringSubstitute(newmsg,"&2",sub2);
   displayStatus(infoArea, newmsg);
}

public static void setFieldColorsOK(Component field)
{
   field.setBackground(java.awt.Color.white);
   field.setForeground(java.awt.Color.black);
}

public static void setFieldColorsError(Component field)
{
   field.setBackground(java.awt.Color.red);
   field.setForeground(java.awt.Color.white);
}
```

Helper Methods - Part 2 of 2

```
public static int getValidPositiveInteger(String stringInt, int min, int max)
{
   int returnVal = -1;
   try
   {
    Integer intObj = Integer.valueOf(stringInt); // exc if invalid format
     returnVal = intObj.intValue();
     if ((returnVal < min) || (returnVal > max))
       {
        System.out.println("Out of range: " + min + ", " + max);
        returnVal = -2;
       }
   }
   catch (NumberFormatException e) {}
   return returnVal;
}

public static String padPositiveNumeric(int intnum, int finalLen)
{
   String paddedString;
   paddedString = Integer.toString(intnum);
   int curLen = paddedString.length();
   if (curLen < finalLen)
     {
      StringBuffer temp = new StringBuffer(finalLen);
      int padAmount = finalLen - curLen;
      int i,j;
      for (i=0; i < padAmount; i++)
         temp.append('0');
      for (j=0; i < finalLen; i++,j++)
         temp.append(paddedString.charAt(j));
      paddedString = temp.toString();
     }
   return paddedString;
}

public static String stringSubstitute(String msg, String subOld, String
                                      subNew)
{
   StringBuffer temp = new StringBuffer();
   int lastHit = 0;
   int newHit = 0;
   for (newHit = msg.indexOf(subOld,lastHit); newHit != -1;
        lastHit = newHit, newHit = msg.indexOf(subOld,lastHit))
      {
        if (newHit >= 0)
          temp.append(msg.substring(lastHit,newHit));
        temp.append(subNew);
        newHit += subOld.length();
      }
   if (lastHit > 0)
     temp.append(msg.substring(lastHit));
   return temp.toString();
}
```

FrameListener Source

Also, if you are curious, here is the source for the `FrameListener` class we copied and used in our project. It handles the “close” event for specificied windows by closing the window, and for if this is the primary window will also close the application:

```
import java.awt.*;
import java.awt.event.*;

public class FrameListener extends WindowAdapter
                           implements ActionListener
{
   Window owner;         // instance variable
   boolean main = true; // instance variable
   // constructor 1
   public FrameListener(Window owner)
   {
       this.owner = owner;
   }
   // constructor 2: use in non-main windows when you
   // don’t want close to exit the whole application...
   public FrameListener(Window owner, boolean main)
   {
       this(owner);
       this.main = main;
   }
   // assumption is following only listens on Close button
   public void actionPerformed(ActionEvent e)
   {
       closeWindow();
       return;
   }
   public void closeWindow()
   {
       owner.dispose();
       if (main)
          System.exit(0);
       return;
   }
   public void windowClosing(WindowEvent e)
   {
       closeWindow();
       return;
   }
}
```