4

Software Engineering for Ajax

Perhaps the greatest advantage of using the Google Web Toolkit to build Ajax applications is having the capability to leverage advanced software engineering. JavaScript was never meant to be used to build large applications. It lacks language features that assist in code organization and compile-time type checking. It also lacks mature tools for building large applications, such as automation, integrated development environments, debugging, and testing. This chapter looks at how to use the Java software engineering tools in GWT to build nontrivial high-quality Ajax applications.

Setting Up the Development Environment

To build applications with GWT, you need the Java Development Kit (JDK) 1.4 or greater. Many other Java tools can also assist with your development, such as an IDE like Eclipse or a build tool like Ant. All of these tools bring a lot of value to the process of building Ajax applications. It is important to note that users don't need any of these tools to use your application. They do not even need to have Java installed on their computer; they only need a reasonably modern web browser like Firefox, Internet Explorer, Safari, or Opera. The GWT compiler compiles your application so it conforms to web-standard technology.

Installing the Java Development Kit

The JDK, a package provided by Sun Microsystems, includes the Java Runtime Environment (JRE), which is required to run Java programs on your computer, and command line developer tools which let you compile Java classes to create code that can run. The JDK is on Sun's web site at http://java.sun.com/javase/downloads.

You can choose from several options to download, but the minimum you need is the JDK without anything else bundled. Some options come with NetBeans or Java EE, but these are not required. There is also a download option for JRE, but this does not include the developer tools that you need.

Once you download the JDK (approximately 50MB), you need to install it. On Windows, the download is an executable file that runs the installation. Install the JDK with all the default options.

Installing the Google Web Toolkit

The GWT complements the JDK by adding the ability to compile your Java code to JavaScript so that it can run in a web browser without the Java Runtime Environment. Think of the GWT as another compiler to run Java on a new platform—your web browser. It also provides a hosted mode browser that lets you take advantage of Java's powerful debugging features, just like you would debug a normal Java application. JavaScript debugging tools are primitive compared to what Java and GWT allow you to do. You can find the Google Web Toolkit SDK at http://code.google.com/webtoolkit/download.html.

On Windows, the GWT zip file is approximately 13MB. After you download it, extract the file to your preferred installation directory. On Mac and Linux you can extract the download using this `tar` command:

```
tar xvzf gwt-mac-1.3.3.tar.gz
```

Let's look inside the distribution. The following list gives you a brief overview of the important files that come with GWT.

- gwt-user.jar

  This is the GWT library. It contains the Java classes that you use to build your application with GWT. Your application uses this file when you run it in hosted mode, but this file is not used when your application is deployed, since your application code and the code used in this file are translated to JavaScript.

- gwt-servlet.jar

  This stripped down version of gwt-user.jar has the classes required for the server side of your application. It is much smaller than gwt-user.jar and better for deployment since it does not contain the GWT classes that are required for hosted mode.

- `applicationCreator`

  This script produces the files required to start a GWT application. The generated files produce a runnable bare-bones GWT application.

- `projectCreator`

  This script generates project files for an Eclipse GWT project.

- `junitCreator`

  This script generates a starter test case along with scripts that start the tests in web mode and hosted mode.

- `i18nCreator`

  This script generates an interface based on a properties file for internationalizing an application.

With only the JDK and GWT installed, you can write, run, and compile web-based applications.

For convenience, you should put the GWT installation directory on your path so that you can call the GWT scripts without specifying the full installation path each time. For example, if you installed GWT to c:\code\gwt (this is a Windows path; for Mac and Linux you would similarly use your install path), you would add this to your PATH variable. Then at a command line you can run the `applicationCreator` script inside your application directory without specifying the script's full path, as shown in Figure 4-1.

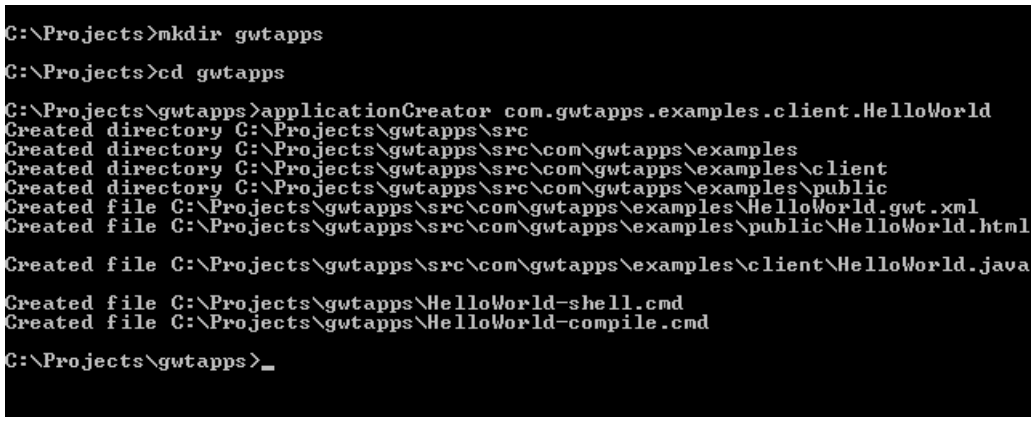

Figure 4–1. Running the `applicationCreator` script for a GWT project

Running this script creates the application named HelloWorld in the current directory. It also generates scripts that let you run the application. You can run this application by just typing the following line:

```
HelloWorld-shell
```

Running this generated script causes GWT to load its hosted browser, which in turn loads the generated application. The hosted browser displays the default generated application, as illustrated in Figure 4-2.

You can also compile the application so that it can be used in a standard browser using the generated HelloWorld-compile script, as seen in Figure 4-3.

The compile script builds the HTML and JavaScript files, which you need to deploy the application, and copies them to the www directory in your application directory, as shown in Figure 4-4.

The generated application can be run in any browser by simply loading the host file. In this HelloWorld application, the host file is named HelloWorld.html. Loading this file in Firefox, as shown in Figure 4-5, results in the same application as in GWT's hosted browser in Figure 4-2, with the major difference being the lack of any Java dependency.

So you can see that the minimum environment for building web applications with GWT is small, only requiring GWT and the JDK to be installed. However, you'll be able to speed up the development process by using

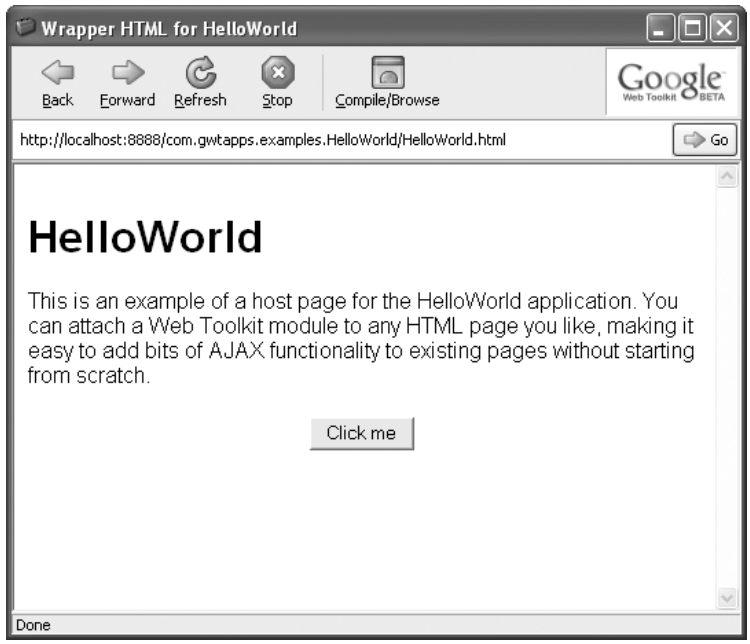

Figure 4–2. Running the default generated project in hosted mode

```
C:\Projects\gwtapps>HelloWorld-compile
Output will be written into C:\Projects\gwtapps\www\com.gwtapps.examples.HelloWo
rld
Copying all files found on public path
Compilation succeeded
```

Figure 4–3. Compiling the project from the command line

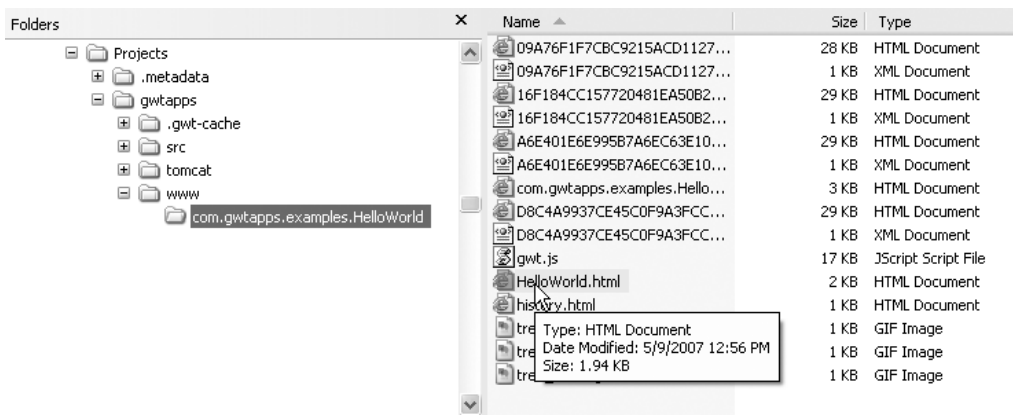

Figure 4–4. The files generated from compiling a GWT project

Figure 4–5. The default project compiled and running in Firefox

many of the available Java tools. For example, an IDE like Eclipse is usually used to speed up Java development.

Installing Eclipse

Eclipse is an open source IDE developed in Java and supported by major technology companies including IBM. An IDE allows you to write, organize, test, and debug software in an efficient way. There are many IDEs for Java, and you can use any of them for GWT development. If you do not have a Java IDE installed, I suggest using Eclipse since it works very well and has support with the GWT scripts to help integration.

Eclipse lets you write, organize, test, and debug your GWT Ajax applications in a productive way. It has great support for the Java language, including refactoring and content assist.[1] You can develop using many languages through plugins with Eclipse by taking advantage of Eclipse's rich plugin framework, but the most widely used language is Java. You can find the Eclipse download at www.eclipse.org/downloads.

1. Content assist is an Eclipse feature that suggests or completes what you are currently typing. It automatically appears, and you can activate it when needed by pressing Ctrl+Spacebar.

Select the Eclipse SDK from this page. After you download the file (approximately 120MB), extract the file to your preferred installation directory. On Windows, the default location for the file eclipse.exe is in the root of the installation directory; you may want to create a shortcut to the file since you will be using it frequently to edit and debug your code.

Adding Projects to Eclipse

When you first load Eclipse, you are prompted by the dialog box shown in Figure 4-6 for the workspace location. This is the location on your computer that will hold your projects.

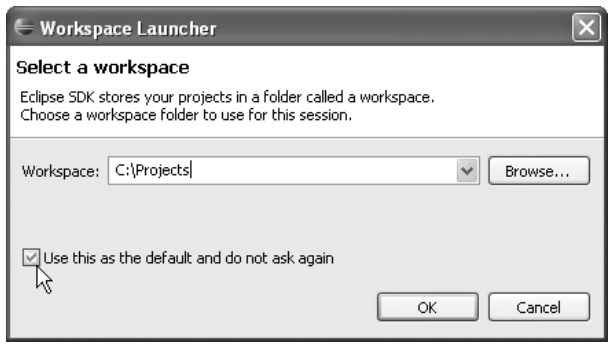

Figure 4–6. Loading a workspace in Eclipse

Figure 4-6 shows setting the workspace to C:\Projects and selecting the check box to save this as the default workspace, so the next time Eclipse opens this workspace is automatically loaded. Since this is a new workspace, when the main Eclipse window loads it will not have any projects listed in its Package Explorer. At this point we could start building a project manually in Eclipse for the HelloWorld application built earlier in this chapter, but GWT gives us a shortcut with the `projectCreator` script shown in Figure 4-7.

This creates an empty project that references GWT and can be easily loaded into Eclipse. To load the GWT project into Eclipse, choose **File > Import** to display the Import dialog box, shown in Figure 4-8.

```
C:\Projects\gwtapps>projectCreator -eclipse GwtApps
Created directory C:\Projects\gwtapps\test
Created file C:\Projects\gwtapps\.project
Created file C:\Projects\gwtapps\.classpath

C:\Projects\gwtapps>_
```

Figure 4–7. Creating a project with the `projectCreator` script and the `-eclipse` flag

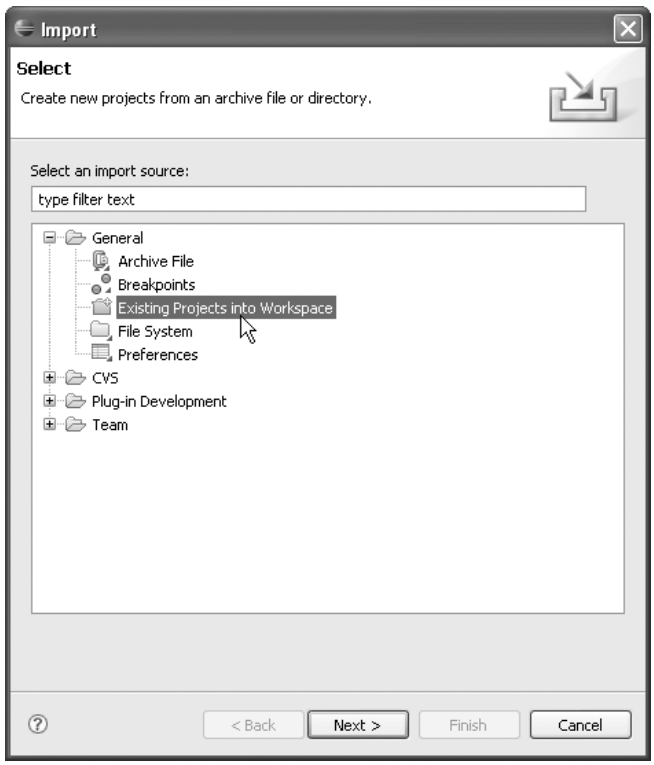

Figure 4–8. Step 1 of importing a generated GWT project into Eclipse

In the Import dialog, select **Exiting Projects into Workspace** and then click **Next**. The next page of the Import dialog, shown in Figure 4-9, lets you select the projects you want to import.

In this dialog you first need to select the location of your project files. The dialog then presents the list of possible projects that you can import. Figure 4-9 shows the GwtApps project that we created with the GWT `projectCreator` script. Make sure this project is checked and then click **Finish.**

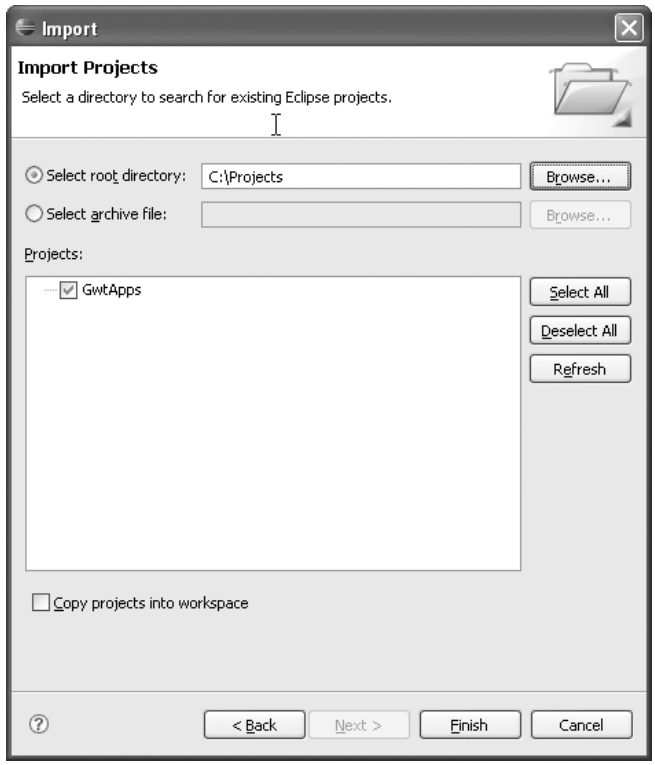

Figure 4–9. Step 2 of importing a generated GWT project into Eclipse

At this point Eclipse loads the project into the Eclipse workspace, and the HelloWorld application is listed under Package Explorer, as shown in Figure 4-10, since it was generated in the Projects directory.

You can add other applications to this project using the `applicationCreator` script, but since we're in Eclipse now we can take advantage of the `-eclipse` option with the script. When the HelloWorld application was run this option was not specified, so we do not have any Eclipse-specific files that allow you to launch the application from Eclipse. So let's run the `applicationCreator` script again, this time specifying the `-eclipse` option, as shown in Figure 4-11.

If you're creating a new application for use in Eclipse, you do not need the `-overwrite` option. This example used this option to overwrite the previously generated application, which did not have Eclipse support. Notice in Figure 4-11 that the new file HelloWorld.launch was created. This launch

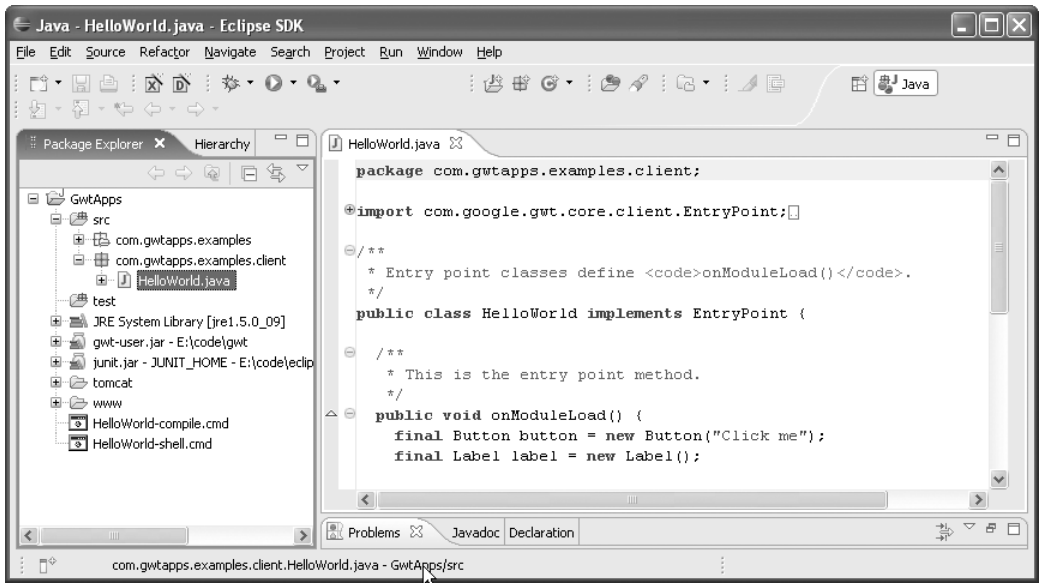

Figure 4–10. The generated GWT project in the Eclipse IDE

```
C:\Projects\gwtapps>applicationCreator -eclipse GwtApps -overwrite com.gwtapps.e
xamples.client.HelloWorld
Overwriting existing file C:\Projects\gwtapps\src\com\gwtapps\examples\HelloWorl
d.gwt.xml
Overwriting existing file C:\Projects\gwtapps\src\com\gwtapps\examples\public\He
lloWorld.html
Overwriting existing file C:\Projects\gwtapps\src\com\gwtapps\examples\client\He
lloWorld.java
Created file C:\Projects\gwtapps\HelloWorld.launch
Overwriting existing file C:\Projects\gwtapps\HelloWorld-shell.cmd
Overwriting existing file C:\Projects\gwtapps\HelloWorld-compile.cmd

C:\Projects\gwtapps>
```

Figure 4–11. Creating an application for use in Eclipse

file allows you to select the **Debug** or **Run** command options for the HelloWorld application inside Eclipse. To see this change in Eclipse, refresh your project (right-click on the project and select **Refresh**), and then run the HelloWorld application in Debug mode by clicking on the Debug icon (see the bug icon on the toolbar in Figure 4-12). If your application isn't listed in the debug drop-down box, which shows a list of recently debugged configurations, you'll need to click **Debug...** in the drop-down menu to load the Debug dialog. You'll find the launch configuration for the HelloWorld application under Java Application.

The application will load in GWT's hosted mode browser, and you can interact with it while still being connected to the Eclipse IDE. This means

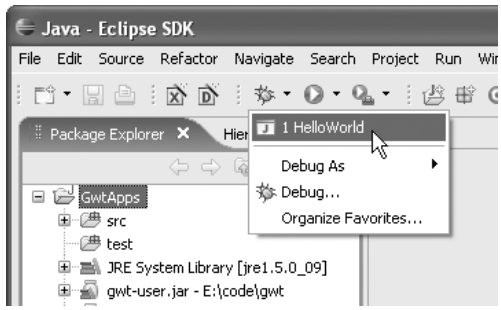

Figure 4–12. Running a GWT application in the Eclipse debugger

you can set breakpoints, change code, and perform other Eclipse functions while your application is running. The ability to do this shortens the code-test cycle dramatically and its ease promotes heavy testing.

Attaching GWT development to Eclipse, or any other Java IDE, is a giant step forward for Ajax application development. Let's look at some of the details of writing code with Eclipse.

Writing Java Code in Eclipse

Eclipse has many tools for writing Java code that provide hints and constraints on what is possible, shortcuts for common tasks, and refactoring functions for large code changes. Of course, you don't have to use these tools to produce Ajax applications with GWT, but they make writing Java code a lot easier.

Creating Classes in Eclipse

First, let's look at Eclipse's tools for creating classes. Eclipse lets you create new classes or interfaces by clicking on the New Class icon on the top toolbar (shown in Figure 4-13). After clicking on the New Class icon, a drop-down menu presents a list of options. For a new class you need to click **Class** in the drop-down menu.

Clicking this icon displays a New Java Class dialog box that prompts you for the information required to create a class. This method is faster than writing a Java class file from scratch and it ensures that everything

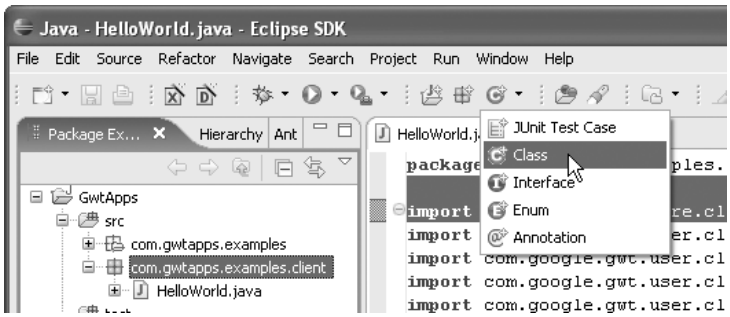

Figure 4–13. Creating a new class

required to be in the file will be there and will be correct. Notice in Figure 4-13 that the `com.gwtapps.examples.client` package is listed. This is where the new class will go. When the New Java Class dialog appears, it displays this package as the default package.

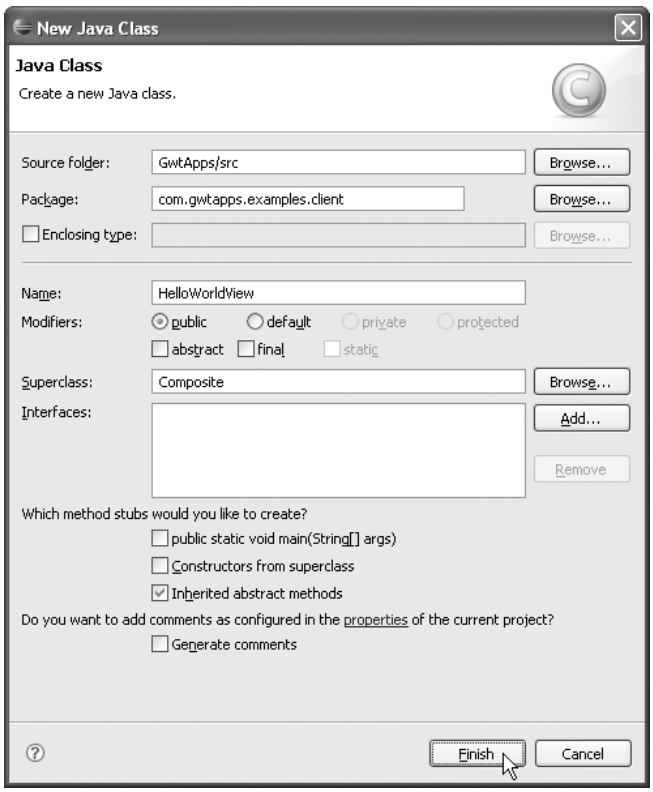

Figure 4–14. The New Java Class dialog in Eclipse

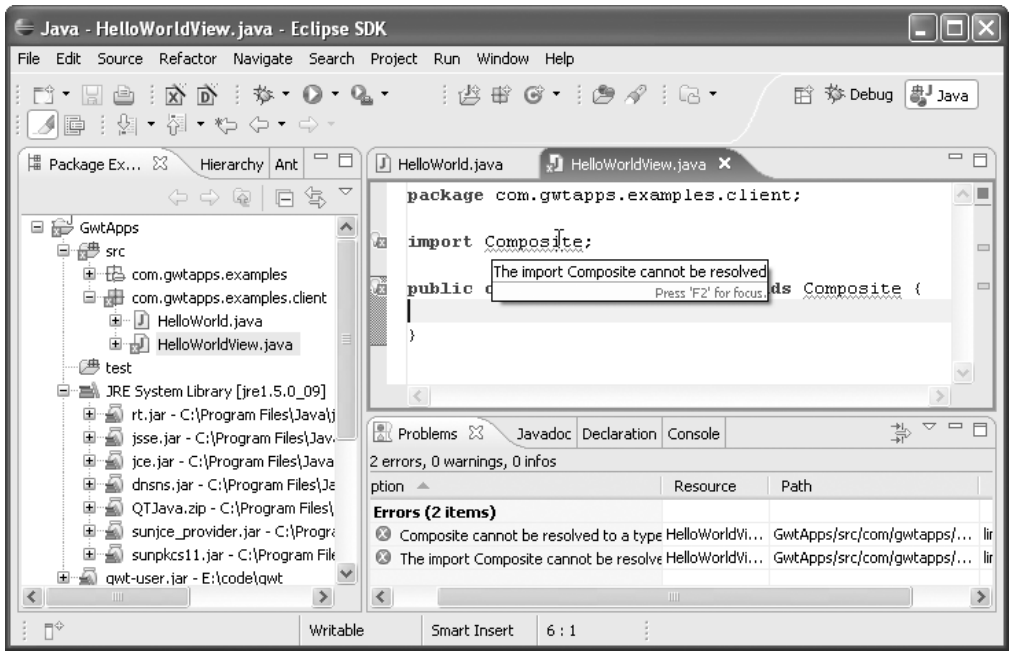

Figure 4–15. The new Java class in the Eclipse IDE

In this dialog, the name `HelloWorldView` is specified as the class name for the new class. The superclass is set to Composite. Clicking **Finish** creates the file and a usable Java class inside, as shown in Figure 4-15.

Actually, the new Java class isn't quite usable yet. We've specified a superclass that doesn't exist. Notice how Eclipse has unobtrusive indicators that let you know something is wrong. The Package Explorer has an X in a red square on the new Java file and on every parent node in the tree up to the project. If we had the project node closed, we would still know that there is an error somewhere in the project. Eclipse also displays a problems list at the bottom that shows a list of errors and warnings in the workspace. It also has the new problems listed. Double-clicking on any of the errors in this list brings you directly to the location of the error in an Eclipse Editor window. In this case there are two errors and the Editor window for the new Java class file is open. Inside the Editor window you can see a further indication of errors. On the right side of the Editor window red marks represent the location of the error within the file.

The file representation for this vertical space is the same scale as the vertical scroll bar. So if this was a bigger file and there were errors, you could

quickly locate them by moving the scrollbar to the location of one of the red marks to see the error in the Editor window. Inside the Editor window, error icons display on the left side and the actual code producing the error has a jagged red underline. Furthermore, when you hover the mouse over the code with the error, a tooltip displays an error message, in this case "The import Composite cannot be resolved." The problem is that we selected just the simple class name as the superclass in the New Java Class dialog, but Eclipse requires the full class name. Often it's hard to remember the full class name for a class, but Eclipse helps us here as well. We can have Eclipse automatically suggest the full class name by clicking on the error and selecting the **Source > Add Import** command, as shown in Figure 4-16.

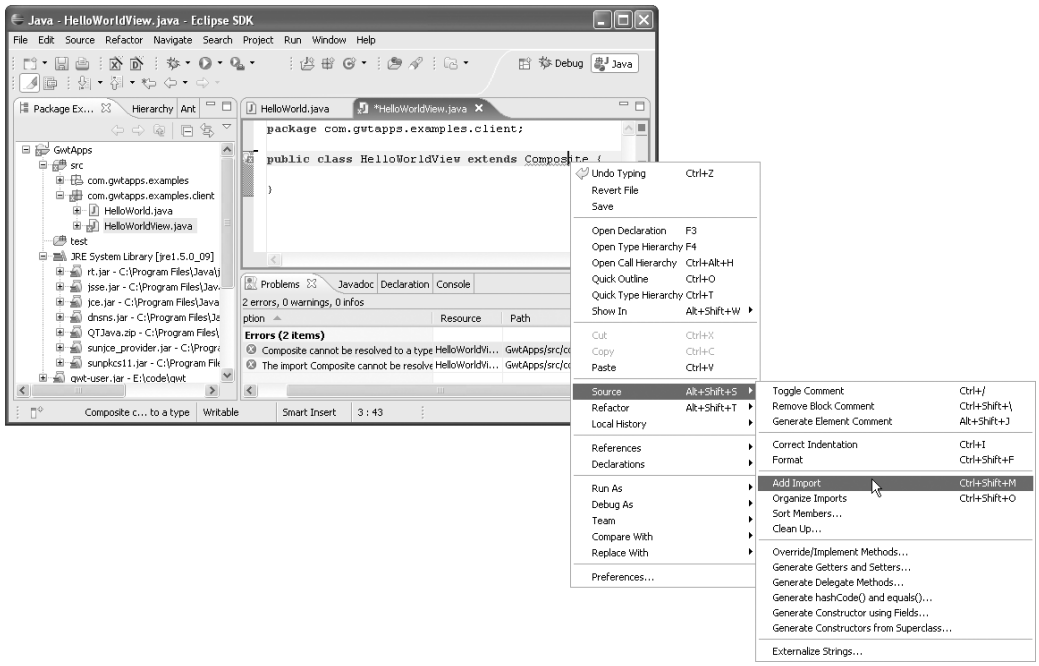

Figure 4–16. Automatically adding a Java import

Alternatively, you could use the keyboard shortcut Ctrl+Shift+M to run the Add Import command. Eclipse automatically adds the required import information. In situations where there is more than one matching import, Eclipse presents you with a choice, as shown in Figure 4-17.

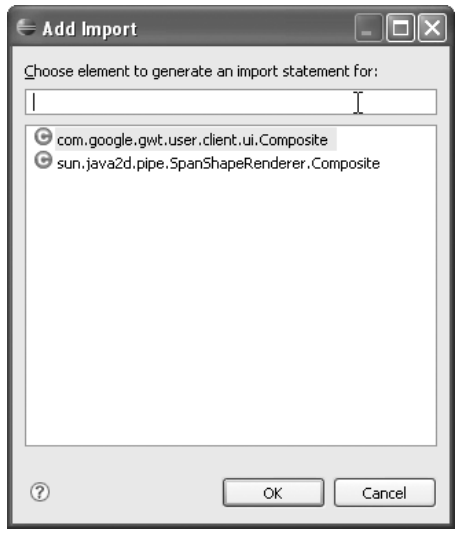

Figure 4–17. Eclipse presents a list of matching import packages

Choosing the GWT Composite class as the import fixes the errors and all of the error indications go away. Eclipse provides this type of early warning of errors for any compile-time errors instantly, instead of having to wait until you compile to get this feedback, as is typical with typed languages. Eclipse updates the IDE with this information as you develop, so you can catch errors immediately after they are created.

Using the Eclipse Java Editor

Now let's look at some of the unique features of the Eclipse Java editor. We'll start by adding some code to the constructor of the `HelloWorldView` class. We can save some typing and generate the constructor by choosing **Source > Generate Constructors from Superclass…**, as shown in Figure 4-18. Eclipse can also automatically suggest items from the Refactor menu if you press Ctrl+L when the cursor is on a section of code. For example, if you implement an interface on a class but have not yet written the methods that must be implemented, you can press Ctrl+L for the suggestions and Eclipse presents a command to automatically implement the required methods.

Syntax may be all the compiler needs to understand code, but adding code syntax coloring in the editor makes it much easier for us to read the Java

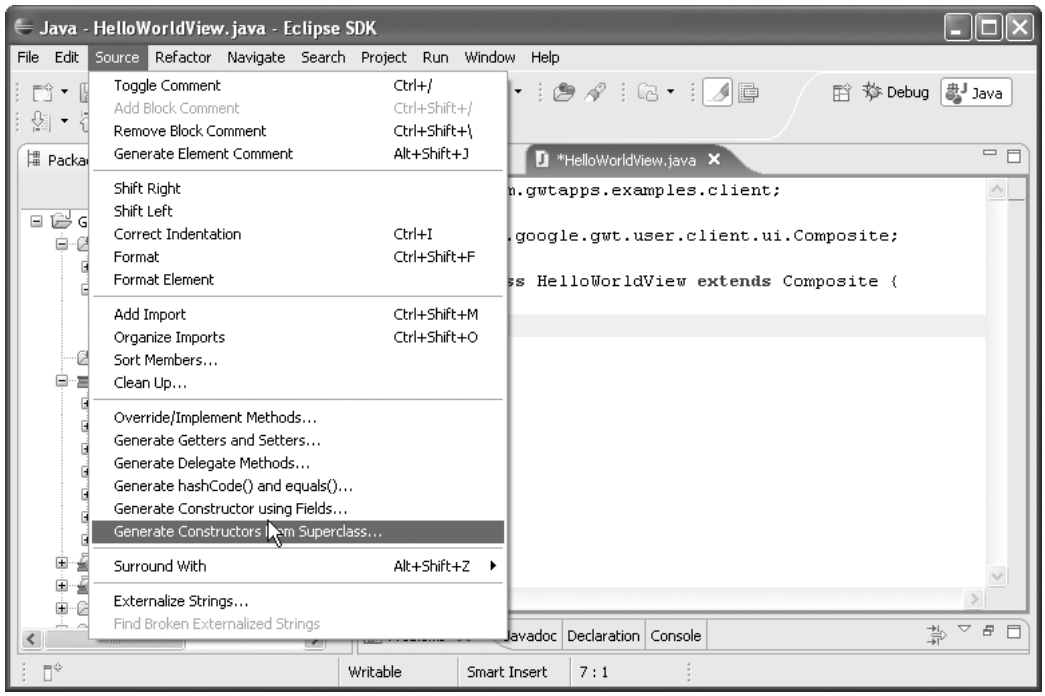

Figure 4–18. Automatically creating a class constructor

code, as illustrated in Figure 4-19. The default syntax coloring in Eclipse uses a bold purple font for Java keywords like class, super, extends, and public, a green font for all comments, a blue font for fields, and a blue italic font for static fields.

Now let's create a `HorizontalPanel` in the constructor and add a couple widgets to it. As you type, Eclipse watches for errors. After you type the word `HorizontalPanel` it will appear as an error, because the class has not been imported. Use the same technique as before to import it (Ctrl+Shift+M or **Source > Add Import**). When you start typing to call a method on the panel, Eclipse's content assist feature displays a list of method suggestions, as shown in Figure 4-20.

```
public class HelloWorldView extends Composite {

    public HelloWorldView() {
        super();
        // TODO Auto-generated constructor stub
    }
}
```

Figure 4–19. An automatically generated constructor

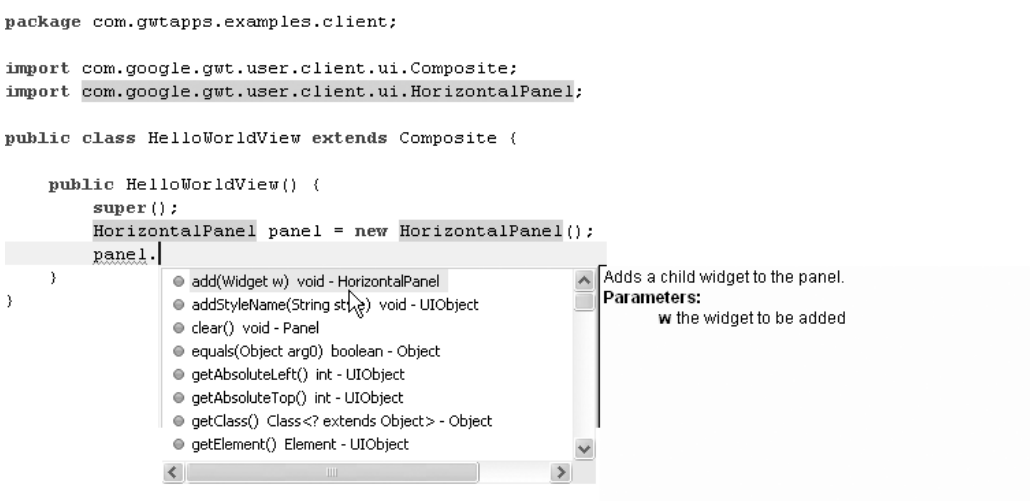

Figure 4–20. Content assist in Eclipse

Eclipse automatically shows the suggestions, or you can force them to display by pressing Ctrl+Spacebar. In this case we want the `add` method, but we can also get an idea of the other methods available. In a way, content assist not only helps speed up typing and locating method names, but it also acts as an educational tool for the class you're using. Instead of leafing through documentation, you can pick up quite a bit of information about a library through this feature.

Another way to educate yourself about a class you're using is to use the editor's Ctrl+Click feature, shown in Figure 4-21. Using Ctrl+Click on a variable, class, or method in the editor takes you to its source in the Eclipse editor. For example, if you click on a variable name, the editor takes you to the variable declaration. If you click on a class name, it takes you to the class' Java file, and if you click on a method, it takes you to the method declaration. This allows you to browse your source code with the same convenience and efficiency as browsing the web. This even works with classes in the GWT library, since the GWT jar file contains the Java source code.

When you can't find what you're looking for while browsing your code, Eclipse provides rich support for searching. First of all, there is a simple single file Find/Replace command which you can access from the Edit menu or by pressing Ctrl+F. This is a standard find and replace feature that you find in most editors. On top of this single file find, Eclipse provides a rich multifile search feature that you can access from the Search menu or by pressing Ctrl+H. Figure 4-22 shows the Search dialog.

```
public class HelloWorldView extends Composite {

    public HelloWorldView() {
        super();
        HorizontalPanel panel = new HorizontalPanel();
        panel.add( new Button("Click Me") );
        panel.add( new Label("Hello World") );
    }
}
```

Figure 4–21. Using Ctrl+Click to browse source code

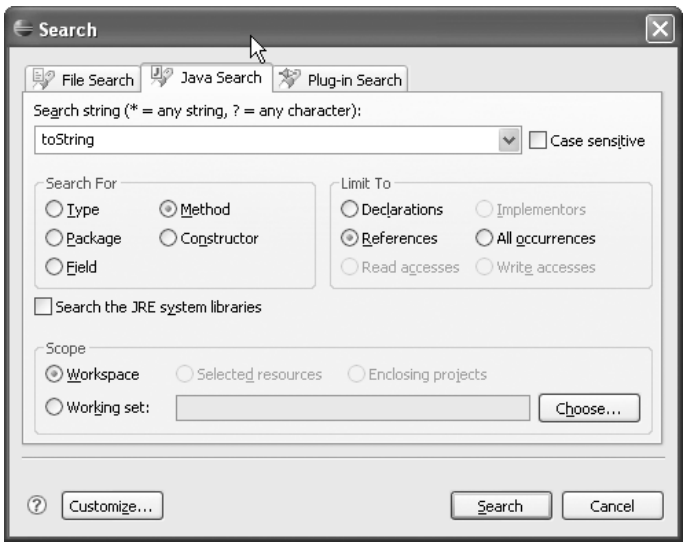

Figure 4–22. Searching in Eclipse

The first tab in the Search dialog, File Search, lets you search for any string within any files in your workspace. The second tab, Java Search, provides a more restrictive search since it has an understanding of the Java language. In this tab you can search for specific instances of a certain string. For example, the dialog in Figure 4-22 shows searching for `toString` when it's being called as a reference. This search would ignore any other occurrence of `toString`, such as `toString` declarations or any comments.

The file search also allows you to replace matching values across files. This is helpful for refactoring code. For example, you could replace all occurrences of `HelloWorld` in our project files with `MyFirstApp`.

Eclipse provides refactoring support beyond multiple file search and replace. For example, you can change the name of the `HelloWorld` class

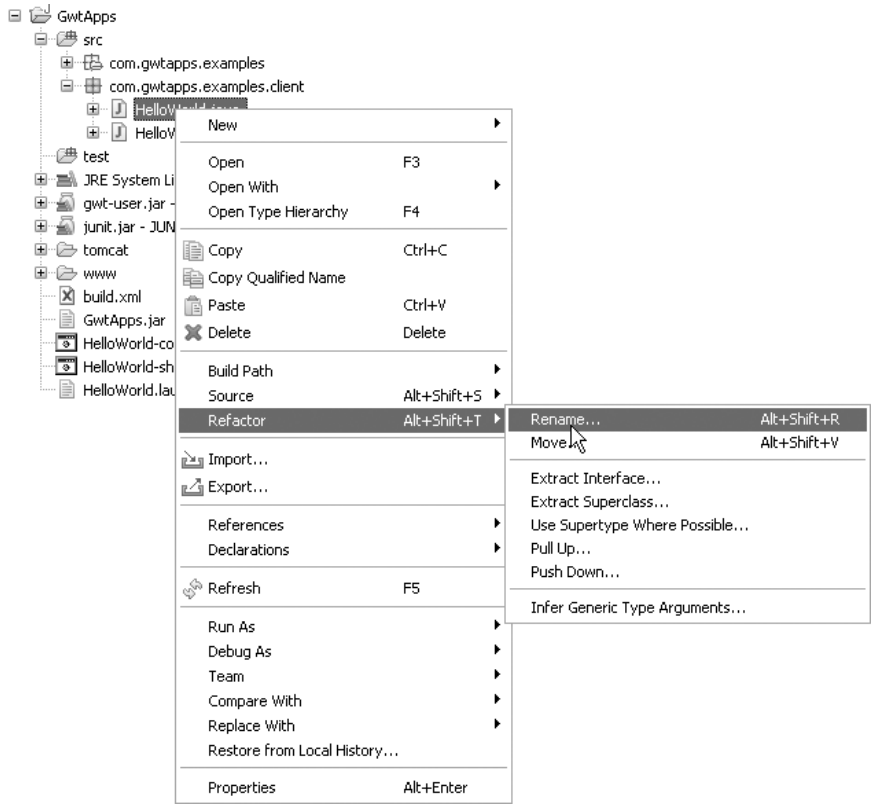

Figure 4–23. Renaming a class

to `MyFirstApp` with the **Refactor > Rename** command, as shown in Figure 4-23.

When you make changes through the Refactor menu, Eclipse ensures that references using the original value are also changed. This method is less error prone than doing a search and replace. Eclipse has many more time-saving refactoring commands, and you can easily find them by checking the Refactor context menu for any item, including highlighted code.

Eclipse also has many more features that can help you write your code. Even though they may not seem like dramatic productivity features, as you start using more of them you'll find yourself writing code faster and with fewer frustrations. Writing code is only one piece of the application development puzzle that Eclipse enhances. The next piece we'll look at is its debugging support.

Debugging in Eclipse

Eclipse provides a nice environment for debugging a running Java application. When you run a GWT application in hosted mode, Eclipse runs it as a Java application and you can debug it within Eclipse. This ability to debug a browser-based web application is a huge advancement for the Ajax development process.

Earlier in this chapter you saw that an Eclipse launch configuration can be automatically created by the GWT `applicationCreator` script by using the `-eclipse` option when creating the application. You can launch the application in hosted mode from Eclipse using either the Run or Debug command. When launched, the application runs in the hosted mode browser. In Debug mode, the hosted mode browser is connected to Eclipse and can use Eclipse's debugging commands.

First, let's look at breakpoints. Breakpoints allow you to set a location within your code where, when reached, the application running would break and pass control to the debugger. This lets you inspect variables or have the application step through the code line by line to analyze the program flow. To see how this works, add a breakpoint to the HelloWorld application on the first line of the button's `ClickListener.onClick` method by right-clicking on the left margin of that line in the editor and selecting **Toggle Breakpoint**, as shown in Figure 4-24.

You'll see the breakpoint added represented by a blue circle in the margin. Alternatively, you can double-click the same spot in the margin to toggle the breakpoint. Now when you debug the application, Eclipse will break into the debugger when it reaches the breakpoint. In this case it will happen when you click on the button. Start the debugger by opening the Debug menu from the Bug icon on the toolbar and selecting HelloWorld, as shown in Figure 4-25.

When the HelloWorld application opens in the hosted mode browser, click on its Click Me button to see Eclipse display the debugger. You should see Eclipse in the Debug perspective, as shown in Figure 4-26.

This is the view you should become familiar with if you are going to be building an Ajax application of any decent size. It provides you with a working view of exactly what is going on in your application. If your appli-

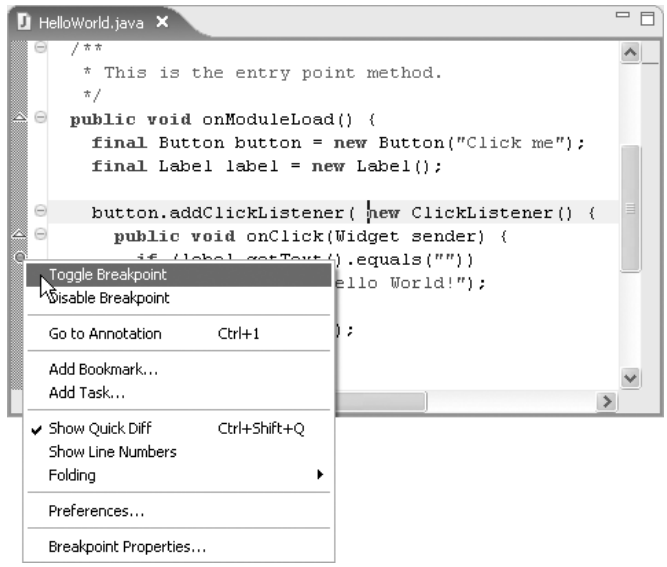

Figure 4–24. Setting breakpoints

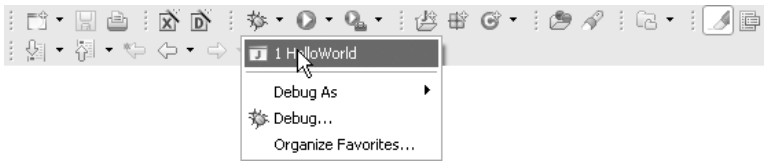

Figure 4–25. Starting the debugger

cation exhibits strange behavior, you can set a breakpoint to see exactly what is happening. If you are a JavaScript developer, this type of debugging tool may be new to you and seem somewhat complex. However, it is definitely worth the effort to learn how to use it properly, since it will save you a lot of time when finding bugs. Instead of printing out and analyzing logs, you can set a breakpoint and step through the program one line at a time, while checking variable values, to determine what the bug is.

Let's briefly look at some of the tools in the Debug perspective. First of all, there are the controls that sit above the stack. The Resume and Terminate buttons are the green triangle and red square, respectively. Resume lets the program continue running. In Figure 4-26 it is stopped on the breakpoint. The Terminate button ends the debug session. You typically end your program by closing the hosted mode browser windows; however,

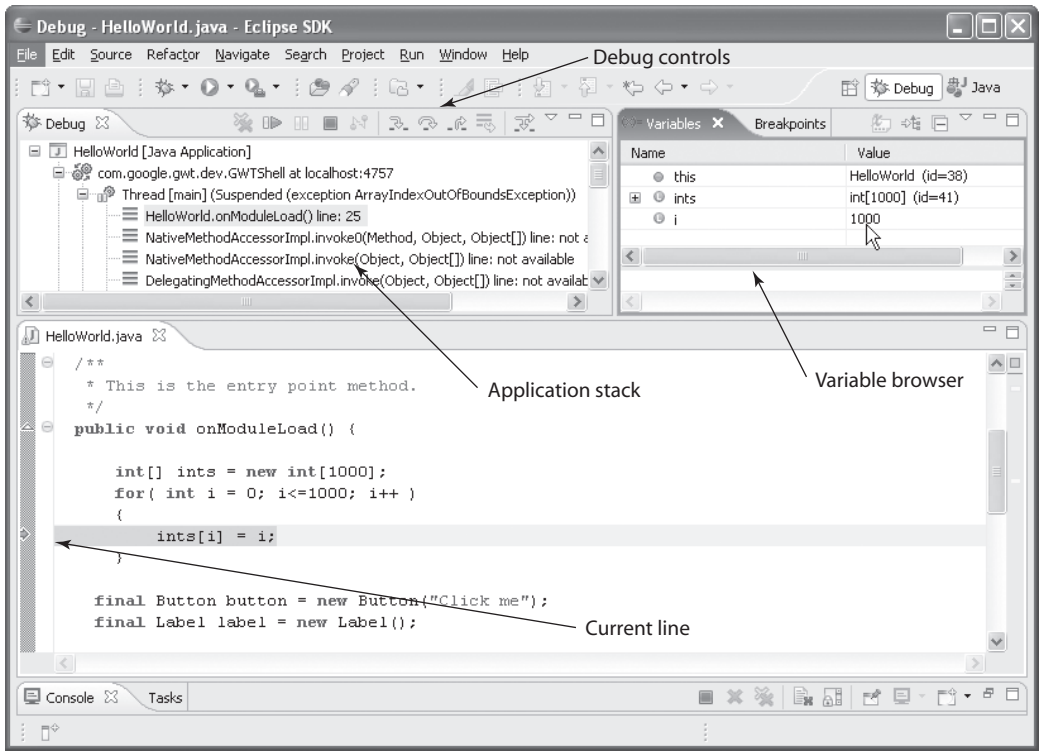

Figure 4–26. The debugging perspective in Eclipse

when you are in a breakpoint, the application has stopped and you cannot access the interface of the hosted mode browser. The only way to end the program in this case is to use the Terminate button. The yellow arrows next to the Resume and Terminate buttons are used for stepping through the application. Taking a step when the application has stopped on a breakpoint executes one step. This allows you to see how one step of code affects any variables. It also lets you inch your way through the program and at a slow pace see how it flows. The first step button, Step Into, takes a step by calling the next method on the current line. Typically this will take you to another method and add a line to the stack. You would use this button when you want to follow the program flow into a method. To avoid stepping into another method, use the next step button, Step Over, which executes the current line, calls any methods, and stops on the next line in the current method. The third yellow arrow button, Step Return, executes the rest of the current method and returns to the calling method, where it stops.

Underneath the debug controls is the calling stack.[2] This is actually a tree that lists threads in the Java application with their stacks as children. The stacks are only visible if the thread is stopped on a breakpoint. Ajax applications are single threaded, so we only need to worry about the one thread and its stack. When we hit the breakpoint in the `onClick` method, the single JavaScript thread displays its method call stack with the current method highlighted. You will find the stack particularly helpful to see when and how a method is called. You can click on other methods in the stack to look at their code in the editor. When you browse the stack like this, the Debug perspective adjusts to the currently selected line on the stack. For example, the editor will show the line in the selected method where the child method was called. It will also adjust the Variables view to show the variables relevant to the currently selected method.

The Variables view lists local and used variables in the current method. The list is a columned tree that lets you browse each variable's contents, and if it is an object, displays its value in the second column. An area on the bottom of the view displays text for the currently selected variable using its `toString` method.

Sometimes stepping through an application with breakpoints isn't enough to find and fix problems. For example, an exception may occur at an unknown time, and placing a breakpoint would cause the debugger to break perhaps thousands of times before you encountered the exception. This is obviously not ideal. Fortunately, Eclipse provides a way to break into the debugger when a specific exception occurs. To add an exception breakpoint you simply need to choose **Run > Add Java Exception Breakpoint**. This displays the dialog shown in Figure 4-27.

In this dialog you select the exception you'd like to break on. The list is a dynamically updating list filtered by the text entered. Figure 4-27 shows breaking on Java's `ArrayIndexOutOfBoundsException`. After clicking on **OK**, you can see that the breakpoint was added by looking at the Breakpoints view in the Debug perspective shown in Figure 4-28.

2. A **calling stack** is a list of methods calls in an application, where each item on the stack is a method preceded by its calling method. So, for example, when a method completes, control returns to the calling method on the top of the stack.

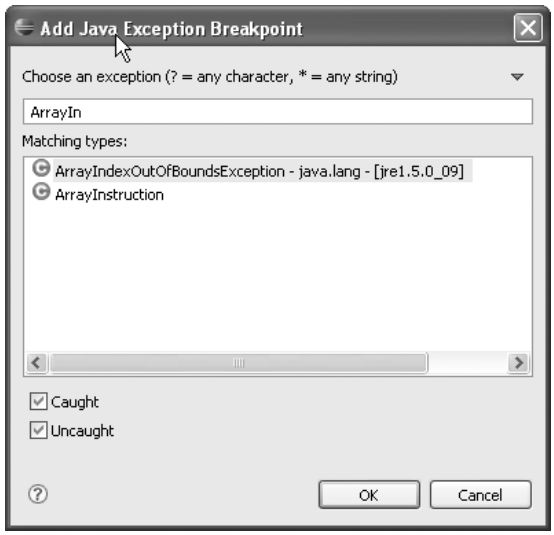

Figure 4–27. Adding an exception breakpoint

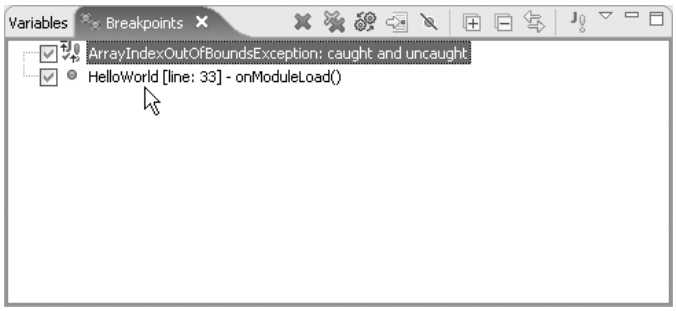

Figure 4–28. The list of breakpoints in Eclipse

To test this, let's write some code that will cause an index to be out of bounds:

```
public void onModuleLoad() {
   int[] ints = new int[1000];
   for( int i = 0; i<=1000; i++ ){
      ints[i] = i;
   }
```

Now when running the application in Debug mode, Eclipse breaks when this code tries to write to the 1,001st `int` in the array (if you bump into another out-of-bounds exception when trying this, press the Resume button). Figure 4-29 shows the Debug perspective stopping on the exception breakpoint.

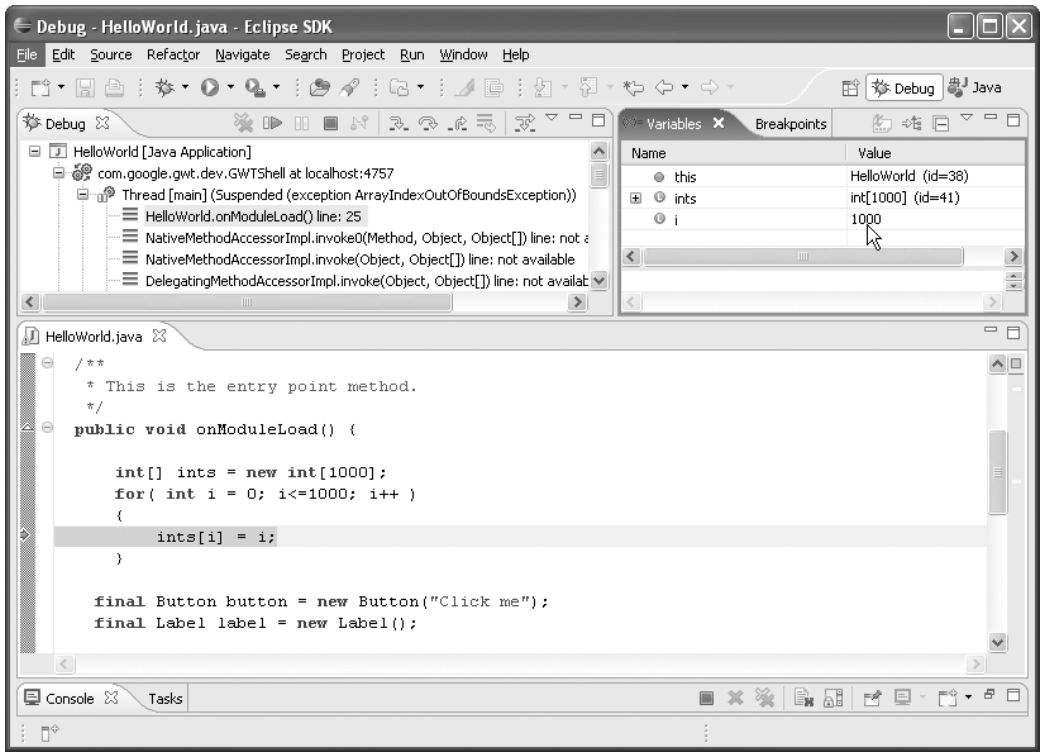

Figure 4–29. Breaking into the debugger on an exception

Notice that the current line is the line where the out of bounds exception occurs. The value of `i` can be seen in the variables window as `1000` (arrays start at 0, so index 1,000 is the 1,001st item and over the bounds which was set at 1,000 items). The benefit of this type of breakpoint is that we did not need to step through 1,000 iterations of the loop to see where the problem is. Of course this is a trivial example, but you can apply this technique to more complex examples that exhibit similar behavior.

Now that we know we have a bug in our HelloWorld code, we can use another great feature of Eclipse that allows us to update the code live and resume the application without restarting. With the application stopped at the exception breakpoint, let's fix the code so that it looks like Figure 4-30.

We've set the comparison operation to less than instead of less than or equals, and removed the 1,000 value to use the `length` property of the array. Save the file, resume the application, and then click the Refresh button on the hosted mode browser. You'll see that the application runs the

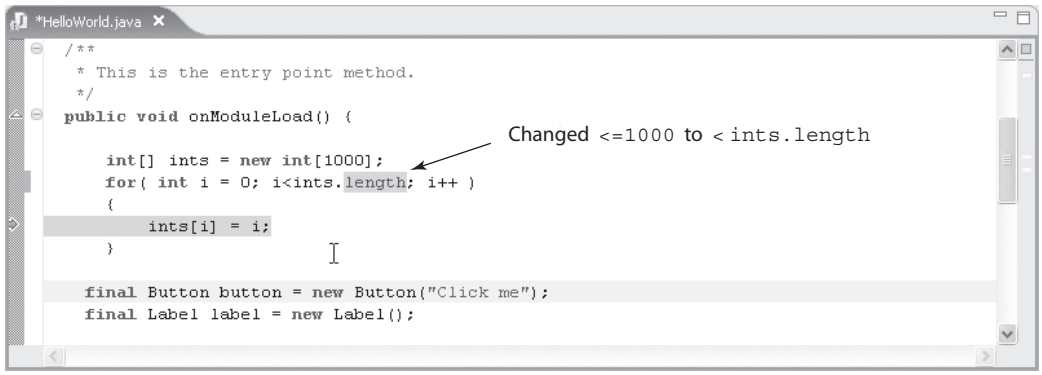

Figure 4–30. Fixing the code while debugging

new fixed code and does not encounter the exception. This technique saves quite a bit of time which would otherwise be spent restarting the hosted mode browser. Also, reducing breaks in your workflow helps keep your mind on the task at hand.

Organizing Your Application Structure

When you generate an application using GWT's `applicationCreator` script, the script creates files and directories that follow a recommended structure. Each application that you create shares your Projects directory. Figure 4-31 shows how the directory looks for the HelloWorld generated application. Figure 4-32 shows the directory result after running the `applicationCreator` again and adding the new application, HelloWorld2, to the same Eclipse project used for HelloWorld. Notice that new scripts were created for the HelloWorld2 application. The `applicationCreator` script creates the application source files in the src directory, and shares this directory with the first HelloWorld application, as shown in Figure 4-33.

The source code is organized in standard Java package structure. Since we created the application as com.gwt.examples.HelloWorld2, the script generates the source files in the `src/com/gwt/`examples directory. This directory structure technique is a nice way of organizing Java modules and applications. It allows you to add packages and give them a unique location in the source tree, avoiding overwriting other classes that may be in a different package but have the same name. It also gives you a unique way to refer to a class from Java code.

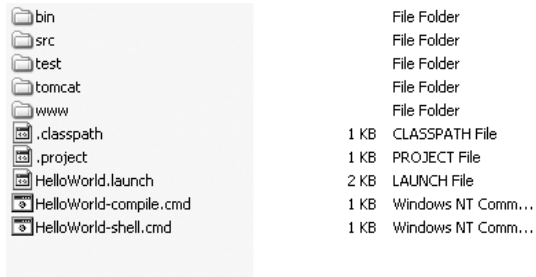

Figure 4–31. Directory structure for the HelloWorld application

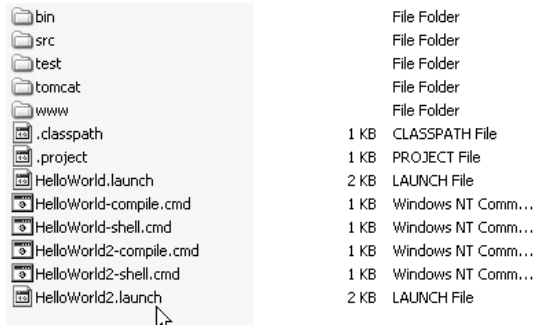

Figure 4–32. Directory structure after adding a new application

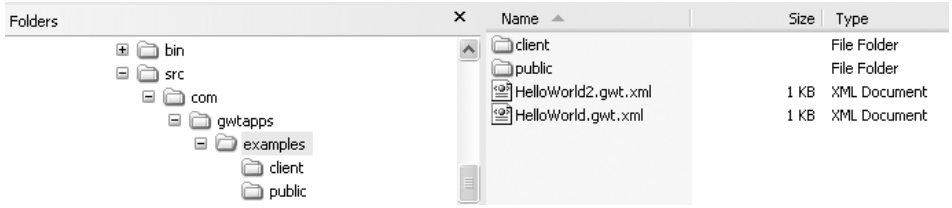

Figure 4–33. Two applications sharing the same source directory

Each generated GWT application has a module file and other source files in the client subdirectory and public subdirectory. Figure 4-33 shows the module file for HelloWorld2, HelloWorld2.gwt.xml. This file specifies the application's configuration options for the GWT compiler. The generated module file looks like this:

```
<module>
   <!-- Inherit the core Web Toolkit stuff.                  -->
   <inherits name='com.google.gwt.user.User'/>
```

```
   <!-- Specify the app entry point class.                   -->
   <entry-point class='com.gwtapps.examples.client.HelloWorld2'/>
</module>
```

This is the minimum specification that the application needs to run. The GWT compiler needs to know the class that acts as the entry point to the application, specified with the `entry-point` tag, and it needs to use the `com.google.gwt.user.User` module for its user interface. When you need to use other modules in your application you specify their location here. The module file has many more configuration options, all of which are outlined on the GWT web site at http://code.google.com/webtoolkit.

Now let's look inside the public folder shown in Figure 4-34. For each generated application, the script creates a new HTML host file in the public directory. The GWT compiler considers files placed in the public directory to be part of the distribution. In other words, when you compile your application, GWT will copy all of the files in this directory to the www output directory. For example, we could move the CSS from inside the HTML host file to a separate CSS file and place it in this directory. Other common files you might place in this directory are images that are used in the application's user interface.

Figure 4–34. The public folder holding the static application files

The generated Java source file for the applications is found in the client directory, as shown in Figure 4-35. When the GWT compiler compiles the Java source to JavaScript, it compiles the Java files in this directory. Any files outside of this directory will not be compiled to JavaScript, and if you use them you will get an exception when compiling or running in hosted mode. However, using the `inherits` tag in your module file tells the GWT compiler to use another module.

Figure 4–35. The client directory holding the files that will be compiled to JavaScript

The GWT compile automatically includes subdirectories and packages in the client directory without inheriting a module. This is useful for organizing subcategories of code within your application. For example, many of the sample applications in this book use a model-view-controller (MVC) architecture and keep the model and view in subpackages. Figure 4-36 shows this type of organization for Chapter 7's Multi-Search sample application. You can use this to organize your client-side code into categories other than model and view.

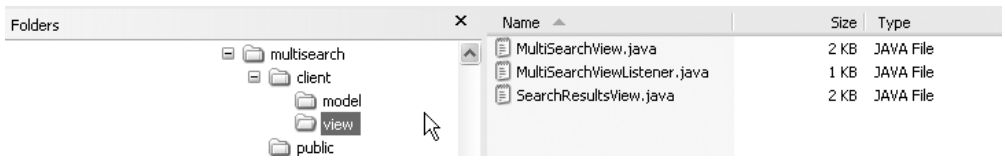

Figure 4–36. MVC organization inside the client directory

There may be situations where you'll have application code that shouldn't be compiled to JavaScript and shouldn't be in the client directory; for example, when writing server-side code in Java, perhaps using a GWT RPC servlet. The common place to put this server-side code is in a server directory. For example, the Instant Messenger application in Chapter 9 places the servlet class in the server subdirectory, as shown in Figure 4-37. Since this is outside of the client directory, the GWT compile ignores the code when compiling the client. Typically the GWT compiler will not be able to compile server-side code since it usually uses packages that aren't emulated by GWT and would not be useful in a browser.

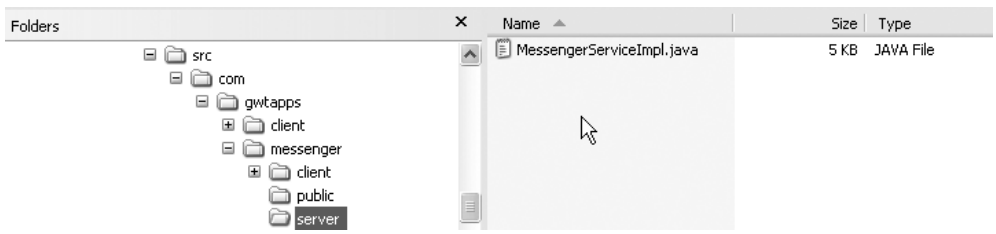

Figure 4–37. Server-side code is placed outside the client directory

The reverse is possible, however. The server classes can use classes in the client directory as long as they don't rely on browser features. The Instant Messenger application does this to share the Java classes that are used to transmit data over RPC between the client and the server.

Finally, when you're ready to deploy your application, you run the generated compile script. GWT copies all of the files used for distribution, including the generated JavaScript files and all of the files in the public directory, to the applications directory inside the www directory. The compiler names the application's directory with the full module name, as shown in Figure 4-38.

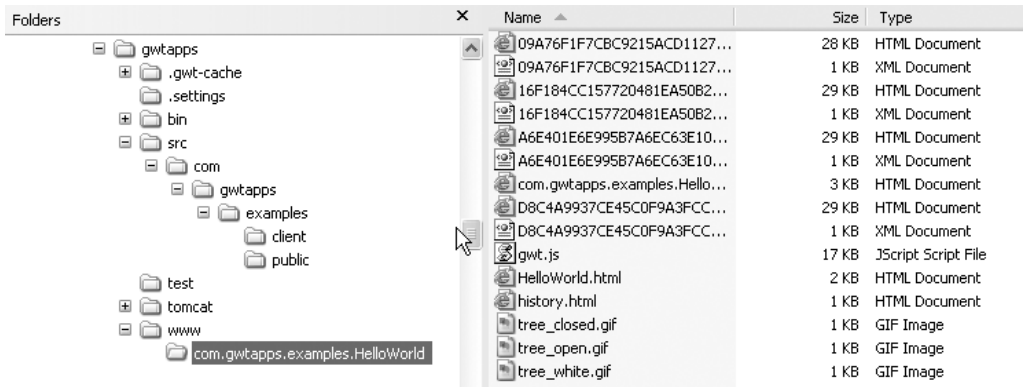

Figure 4–38. The GWT writes the compiled and static files to the www directory

Testing Applications

Having the capability to build Ajax applications with Java gives you many tools that let you maintain larger applications with less work. One very important aspect of maintaining a large application is being able to easily create unit tests for most, if not all, functionality. This need comes from a common problem with software development: the code size grows to a point where small changes can have cascading effects that create bugs.

It has become common practice to incorporate heavy testing into the development cycle. In the traditional waterfall development cycle you would write code to a specification until the specification was complete. Then the application would be passed to testers who would look for bugs. Developers would respond to bug reports by fixing the bugs. Once all the bugs were fixed, the product would be shipped. Figure 4-39 illustrates the steps in traditional software development testing.

The problem encountered with this type of development cycle is that during the bug finding and fixing phase, code changes can easily cause more

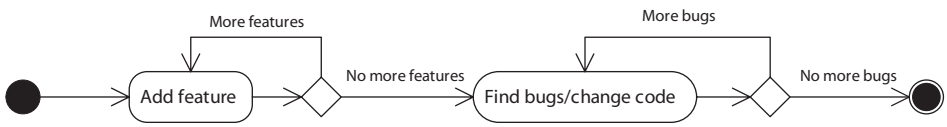

Figure 4–39. Old-style testing = bad

bugs. To fix this problem, testers would need to start testing right from the beginning after every code change to ensure new bugs weren't created and old bugs didn't reappear.

One successful testing methodology has developers write automated unit tests before they write the features. The tests cover every use case of the new feature to be added. The first time the test is run, it will fail for each case. The development process then continues until each test case in the unit test is successful. Then the unit test becomes part of a test suite for the application and is run before committing any source code changes to the source tree. If a new feature causes any part of the application to break, other tests in the automated test suite will identify this problem, since every feature of the application has had tests built. If a bug is found at this point, it is relatively easy to pinpoint the source since only one new feature was added. Finding and fixing bugs early in the development life-cycle like this is much easier and quicker than finding and fixing them at the end. The test suite grows with the application. The initial investment in time to produce the unit tests pays off over the long run since they are run again on every code change, ensuring each feature's health. Figure 4-40 illustrates this process.

In practice, when comparing this approach to the one illustrated in Figure 4-39, there is a large time saving from finding bugs earlier and less of a need for a large testing team since the developer is responsible for much of the testing.

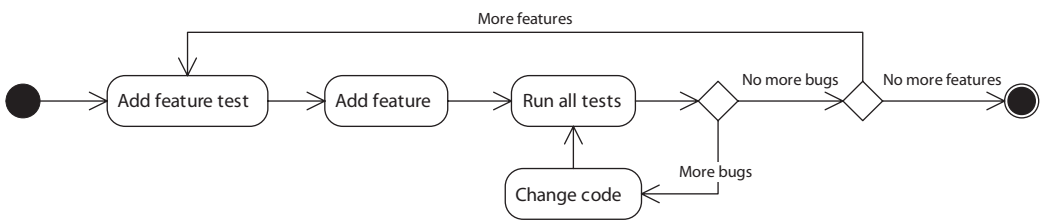

Figure 4–40. Test-first testing = good

This technique is relatively novel for client-side web applications. Testing is reduced to usability testing and making sure that different browsers render pages properly with traditional web applications. This is one of the great things about HTML. It's a declarative language that leaves little room for logical bugs. It's easy to deploy HTML web pages that work (browser-rendering quirks aside). However, using JavaScript introduces the possibility of logic bugs. This wasn't too much of a problem when JavaScript was being used lightly, but for Ajax applications heavily using JavaScript, logical bugs are somewhat of a problem. Since JavaScript is not typed and does not have a compile step, many bugs can only be found by running the application, which makes the creation of unit tests difficult. Furthermore, it is difficult to test an entire application through its interface. Many simple bugs, such as trying to call an undefined function, cannot be caught without running the program and trying to execute the code that has the bug, but by using Java you could catch these bugs immediately in the IDE or at compile time. From a testing perspective, it does not make sense to build large Ajax applications with JavaScript.

Using JUnit

JUnit is another great Java tool that assists in creating an automated testing for your application. It provides classes that assist in building and organizing tests, such as assertions to test expected results, a test-case base class that allows you to set up several tests, and a mechanism to join tests together in a test suite. To create a test case for JUnit you would typically extend the `TestCase` class, but since GWT applications require a special environment, GWT provides a `GWTTestCase` class for you to extend.

Let's walk through the creation of a test case for the Multi-Search application in Chapter 7. The first step is to use the GWT `junitCreator` script to generate a test case class and some scripts that can launch the test case. The `junitCreator` script takes several arguments to run. Table 4-1 outlines each argument.

To run this script for the Multi-Search application we can use the following command:

```
junitCreator -junit E:\code\eclipse\plugins\org.junit_3.8.1\junit.jar -module
com.gwtapps.multisearch.MultiSearch -eclipse GWTApps
com.gwtapps.multisearch.client.MultiSearchTest
```

Table 4–1 `junitCreator` Script Arguments

Argument	Description	Example
`junit`	Lets you define the location of the junit jar file. You can find a copy in the plugin directory of your Eclipse installation.	`-junit E:\code\eclipse\plugins\org.junit_3.8.1\junit.jar`
`module`	Specifies the GWT module that you'll be testing. It is required since the environment needs to run this module for your test.	`-module com.gwtapps.multisearch.MultiSearch`
`eclipse`	Specifies your Eclipse project name if you want to generate Eclipse launch configurations.	`-eclipse GWTApps`
	The last argument should be the class name for the test case. You would typically use the same package as the one being tested.	`com.gwtapps.multisearch.client.MultiSearchTest`

```
C:\gwtapps-ws\gwtapps>junitCreator -junit E:\code\eclipse\plugins\org.junit_3.8.
1\junit.jar -module com.gwtapps.multisearch.MultiSearch -eclipse GWTApps com.gwt
apps.multisearch.client.MultiSearchTest
Created directory C:\gwtapps-ws\gwtapps\test\com\gwtapps\multisearch\client
Created file C:\gwtapps-ws\gwtapps\test\com\gwtapps\multisearch\client\MultiSear
chTest.java
Created file C:\gwtapps-ws\gwtapps\MultiSearchTest-hosted.launch
Created file C:\gwtapps-ws\gwtapps\MultiSearchTest-web.launch
Created file C:\gwtapps-ws\gwtapps\MultiSearchTest-hosted.cmd
Created file C:\gwtapps-ws\gwtapps\MultiSearchTest-web.cmd
```

Figure 4–41. Using `junitCreator` to generate a test case

Figure 4-41 shows the output from this command. The script created two scripts, two launch configurations for launching the test in web mode or hosted mode, and one test case class that is stored in the test directory. In Eclipse the test case class will look like Figure 4-42.

The generated test case has two methods. The first, `getModuleName`, is required by GWT and must specify the module that is being tested. The `junitCreator` script has set this value to the Multi-Search module because it was specified with the module command line argument. The second method, a test case, is implemented as a simple test that just asserts that the value true is true. You can build as many test cases as you like in this one class.

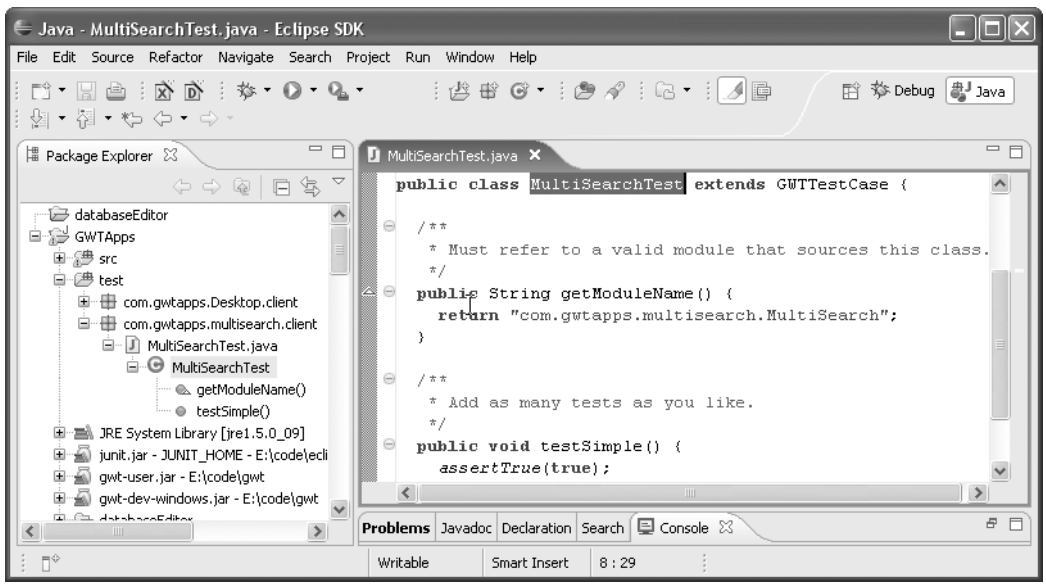

Figure 4–42. A generated test case in Eclipse

You can run the tests by running the scripts generated by `junitCreator`. Alternatively, you can launch JUnit inside Eclipse for a visual representation of the results. Running inside Eclipse also lets you debug the JUnit test case, which can greatly assist in finding bugs when a test case fails. Since `junitCreator` created a launch configuration for Eclipse, we can simply click the Run or Debug icons in the Eclipse toolbar and select the `MultiSearchTest` launch configuration from the drop-down menu. After launching this configuration, the JUnit view automatically displays in Eclipse. When the test has completed, you will see the results in the JUnit view, as shown in Figure 4-43. Notice the familiar check marks, which are displayed in green in Eclipse, next to the test case indicating that the test case was successful.

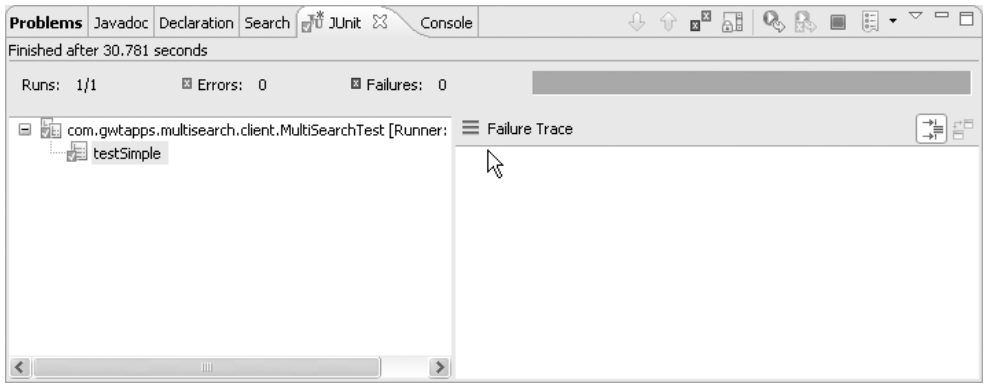

Figure 4–43. Running a JUnit test case from Eclipse

Now let's create a test case for each type of search engine that the application uses. Adding the following code to the test class creates four new tests:

```
protected MultiSearchView getView(){
   MultiSearchView view = new MultiSearchView( new MultiSearchViewListener(){
      public void onSearch( String query ){}
   });
   RootPanel.get().add( view );
   return view;
}

protected void doSearchTest( Searcher searcher ){
   searcher.query( "gwt" );
}
```

```
public void testYahoo() {
   doSearchTest( new YahooSearcher( view ) );
}

public void testFlickr() {
   doSearchTest( new FlickrSearcher( view ) );
}

public void testAmazon() {
   doSearchTest( new AmazonSearcher( view ) );
}

public void testGoogleBase() {
   doSearchTest( new GoogleBaseSearcher( view ) );
}
```

The first two methods, `getView` and `doSearchTest`, are helper methods for each test in this test case. The `getView` method simply creates a view, the `MultiSearchView` defined in the application, and adds it to the `RootPanel` so that it is attached to the document. Then the `doSearchTest` method sends a query to a `Searcher` class implementation. Each test case instantiates a different Searcher implementation and sends it to the `doSearchTest` method. When JUnit runs, each test case runs and submits a query to the respective search engine. Figure 4-44 shows what the result looks like in the Eclipse JUnit view.

If any search failed by an exception being thrown, then the stack trace for the exception would display in the right pane of this view and a red X icon would display over the test case.

The problem with this test case is that it doesn't verify the results. JUnit provides many assertion helper methods that compare actual results to

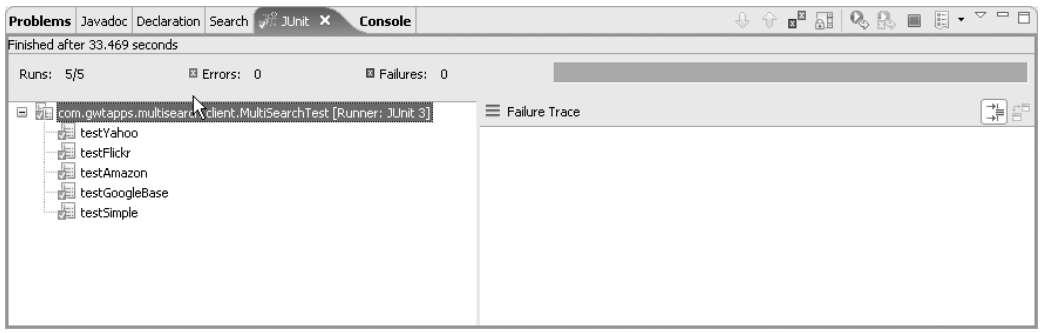

Figure 4–44. Running several tests in one test case

expected results. However, in this case our results are asynchronous; that is, they don't arrive until after the test case completes. GWT provides help with this since much of Ajax development is asynchronous with the `delayTestFinish` method.

To use this method we need to have a way of validating an asynchronous request. When we have validated that an asynchronous request is complete, then we call the `finishTest` method. In the case of the MultiSearch test, we will validate when we receive one search result. To do this we need to hook into the application to intercept the asynchronous event. This requires a bit of knowledge about the application and may seem a little obscure otherwise. We will create a **mock object,** which is an object that pretends to be another object in the application, to simulate the `SearchResultsView` class. By simulating this class we will be able to extend it and override the method that receives search results. The class can be declared as an inner class on the test case like this:

```
private class MockSearchResultsView extends SearchResultsView {
   public MockSearchResultsView( SearchEngine engine ){
      super(engine);
   }

   public void clearResults(){}

   public void addSearchResult( SearchEngineResult result ){
      assertNotNull(result);
      finishTest();
   }
}
```

The class overrides the `addSearchResult` method, which one of the `Searcher` classes calls when a search result has been received from the server. Instead of adding the result to the view, this test case will use one of JUnit's assert methods, `assertNotNull`, to assert that the search engine result object is not null. Then it calls the GWT's `finishTest` method to indicate that the asynchronous test is complete.

To run this test we need to change the `doSearchTest` method on the test case to insert the mock view and tell JUnit to wait for an asynchronous response:

```
protected void doSearchTest( Searcher searcher ){
   searcher.setView(
      new MockSearchResultsView(searcher.getView().getEngine()));
```

```
    searcher.query( "gwt" );
    delayTestFinish(5000);
}
```

In this code we set the view of the searcher to the mock view that we've created, and then call the `delayTestFinish` method with a value of 5,000 milliseconds (5 seconds). If the test does not complete within 5 seconds, it will fail. If the network connection is slow, you may want to consider a longer value here to properly test for errors.

Running these tests at this point tests the application code in the proper GWT environment and with asynchronous events occurring. You should use these testing methods as you build your application so you have a solid regression testing library.

Benchmarking

When using GWT to create Ajax applications, taking user experience into consideration almost always comes first. Part of creating a good user experience with an application is making it perform well. Fortunately, since GWT has a compile step, each new GWT version can create faster code, an advantage that you don't have with regular JavaScript development. However, you probably shouldn't always rely on the GWT team to improve performance and should aim at improving your code to perform better. Starting with release 1.4, GWT includes a benchmarking subsystem that assists in making smart performance-based decisions when developing Ajax applications.

The benchmark subsystem works with JUnit. You can benchmark code through JUnit by using GWT's `Benchmark` test case class instead of `GWTTestCase`. Using this class causes the benchmarking subsystem to kick in and measure the length of each test. After the tests have completed, the benchmark system writes the results to disk as an XML file. You can open the XML file to read the results, but you can view them easier in the benchmarkViewer application that comes with GWT.

Let's look at a simple example of benchmarking. We can create a benchmark test case by using the `junitCreator` script in the same way we would for a regular test case:

```
junitCreator -junit E:\code\eclipse\plugins\org.junit_3.8.1\junit.jar -module
com.gwtapps.desktop.Desktop -eclipse GWTApps com.gwtapps.desktop.client.
CookieStorageTest
```

In this code we're creating a test case for the cookie storage feature in Chapter 6's Gadget Desktop application. The application uses the `CookieStorage` class to easily save large cookies while taking into account browser cookie limits. In this test we're going to measure the cookie performance. First, we extend the `Benchmark` class instead of `GWTTestCase`:

```
public class CookieStorageTest extends Benchmark {

   public String getModuleName() {
      return "com.gwtapps.desktop.Desktop";
   }

   public void testSimpleString(){
      try {
         CookieStorage storage = new CookieStorage();
         storage.setValue("test", "this is a test string");
         assertEquals( storage.getValue("test"), "this is a test string" );
         storage.save();
         storage.load();
         assertEquals( storage.getValue("test"), "this is a test string");

      } catch (StorageException e) { fail(); }
   }
}
```

You can run this benchmark from the Eclipse JUnit integration or the launch configuration generated by the `junitCreator` script. The test simply creates a cookie, saves it, loads it, and then verifies that it hasn't changed. The generated XML file will contain a measurement of the time it took to run this method. At this point the benchmark is not very interesting. We can add more complex benchmarking by testing with ranges.

Using ranges in the benchmark subsystem gives you the capability to run a single test case multiple times with different parameter values. Each run will have its duration measured, which you can later compare in the benchmark report. The following code adds a range to the cookie test to test writing an increasing number of cookies:

```
public class CookieStorageTest extends Benchmark {

   final IntRange smallStringRange =
      new IntRange(1, 64, Operator.MULTIPLY, 2);

   public String getModuleName() {
      return "com.gwtapps.desktop.Desktop";
   }
```

```
   /**
   * @gwt.benchmark.param cookies -limit = smallStringRange
   */
   public void testSimpleString( Integer cookies ){
      try {
         CookieStorage storage = new CookieStorage();
         for( int i=0; i< cookies.intValue(); i++){
            storage.setValue("test"+i, "this is a test string"+i);
            assertEquals( storage.getValue("test"+i),
               "this is a test string"+i );
         }
         storage.save();
         storage.load();
         for( int i=0; i< cookies.intValue(); i++){
            assertEquals( storage.getValue("test"+i),
               "this is a test string"+i );
         }
      } catch (StorageException e) { fail(); }
   }
   public void testSimpleString(){
   }
}
```

This code creates an `IntRange`. The parameters in the `IntRange` constructor create a range that starts at one and doubles until it reaches the value 64 (1, 2, 4, 8, 16, 32, 64). GWT passes each value in the range into separate runs of the `testSimpleString` method. GWT knows to do this by the annotation before the method, which identifies the parameter and the range to apply.

Notice that there is also a version of the `testSimpleString` method without any parameters. You need to provide a version of this method with no arguments to run in JUnit since it does not support tests without parameters. The benchmark subsystem is aware of this and is able to choose the correct method.

After running this code we can launch the benchmarkViewer application from the command line in the directory that the reports were generated in (this defaults to the Projects directory):

```
benchmarkViewer
```

The benchmarkViewer application shows a list of reports that are in the current directory. You can load a report by clicking on it in the list. Each

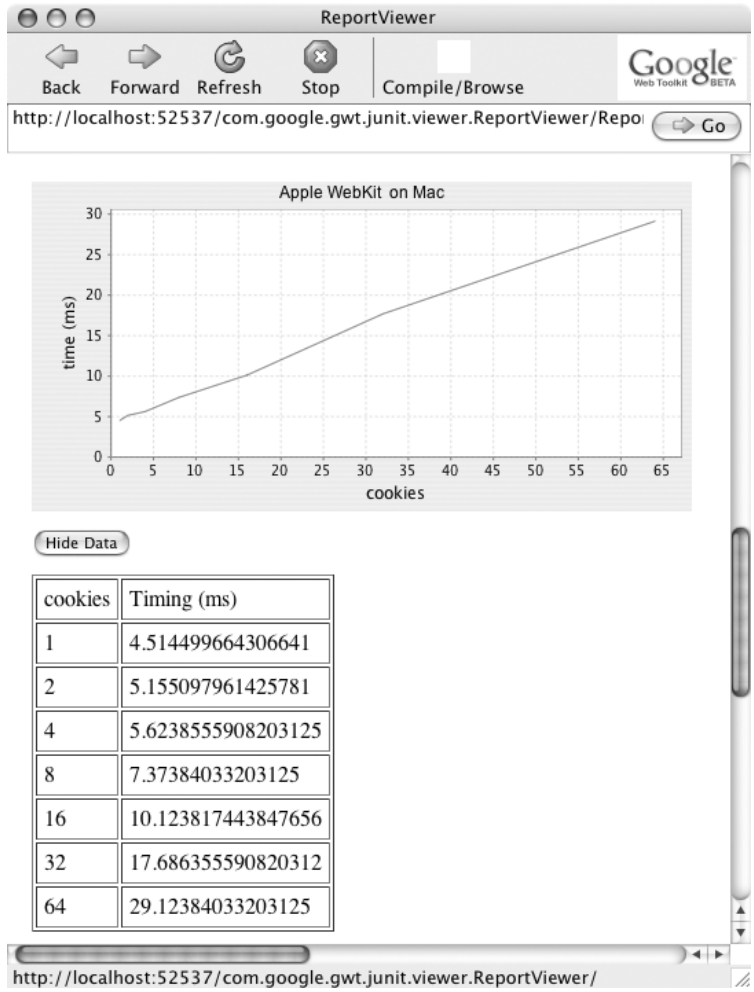

Figure 4–45. Benchmark results for the cookie test

report contains the source code for each test along with the results as a table and a graph. Figure 4-45 shows the result of the testSimpleString test.

The benchmark system also recognizes beginning and ending methods. Using methods like these allows you to separate set up and take down code for each test that you don't want measured. For example, to define a setup method for the testSimpleString test, you would write the following code:

```
public void beginSimpleString( Integer cookies ){
   /* do some initialization */
}
```

Building and Sharing Modules

Each GWT module is not necessarily a full application, but it can be used as a reusable library for other applications instead. The GWT module structure, also used for applications, gives you the tools necessary to package your module and share it with other applications. In fact, GWT itself is divided into several modules, as you've seen with the user interface, XML, JSON, HTTP, and RPC modules, so you've already used the process of importing other libraries.

Using Modules

GWT modules are distributed as jar files that you can include in your application by adding them to your project's classpath and inheriting their project name in your application's module file. This is the same process that you use to include the GWT library classes in your application. In this case GWT automatically adds the module jar file, gwt-user.jar, to your project's classpath when you generate the project using the GWT `createProject` script. The `createApplication` script then generates a module XML file for your application and automatically adds the `com.google.gwt.user.User` module to it. When we generate the module XML file for the Gadget Desktop application in Chapter 6, we get the following XML:

```
<module>
   <inherits name='com.google.gwt.user.User'/>
   <entry-point class='com.gwtapps.desktop.client.Desktop'/>
</module>
```

This module file tells the GWT compiler how to compile the application to JavaScript. The `inherits` element tells the compiler that we are using classes from the `name` module, which will also need to be compiled to JavaScript. We can continue to add modules from gwt-user.jar since the file is are already on the classpath. For new modules in other jar files, we first need to add the jar to the classpath. In Eclipse, you can do this by going to the project's Properties dialog and selecting the Libraries tab from Java Build Path, as shown in Figure 4-46.

From here you can add and remove jar files. Notice that gwt-user.jar is already in the list. For the Gadget Desktop application we add the `gwt-google-apis` library to the project to use the Gears module from it. First,

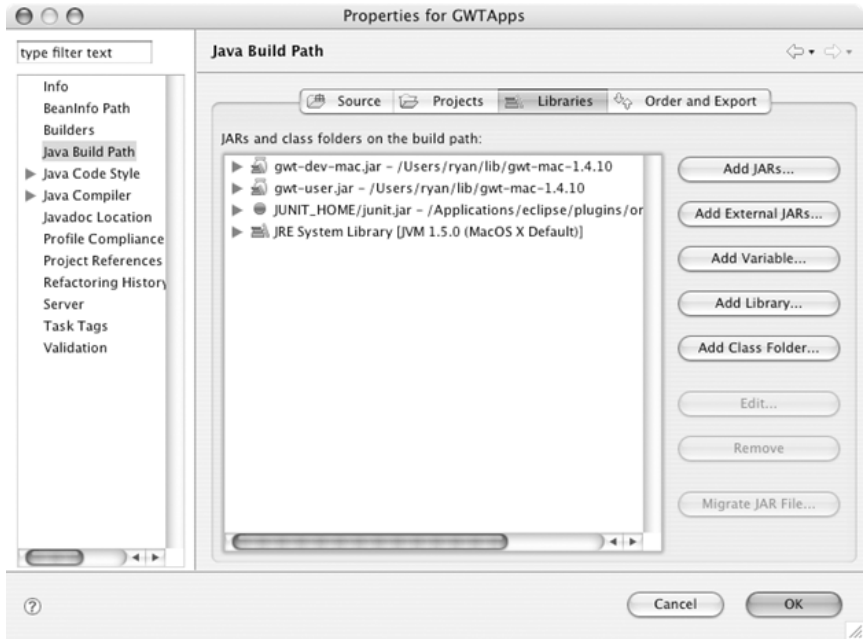

Figure 4–46. Editing the build path in Eclipse

we add the `gwt-google-apis` jar to this list, and then the application's module XML file inherits the Gears module like this:

```
<module>
   <inherits name='com.google.gwt.user.User'/>
   <inherits name='com.google.gwt.json.JSON'/>
   <inherits name='com.google.gwt.xml.XML'/>
   <inherits name='com.google.gwt.gears.Gears'/>
   <entry-point class='com.gwtapps.desktop.client.Desktop'/>
</module>
```

Notice also that this project imports the JSON and XML modules which are already in the gwt-user.jar file. If you miss this step—adding the `inherits` tag to your application's module file—you will get an error from the GWT compiler that it can't find the module that you're using.

Creating a Reusable Module

If you've built a GWT application, you've already built a reusable module. The only difference is that your application has specified an entry point and can be turned into a GWT application loaded on its own in the GWT

hosted mode browser or web browser. You could reference the application's module file from another application to reuse its components.

You create a module the same way you create an application, using GWT's `applicationCreator` script. You may want to use the `ant` flag with this script to build an ant file that will automatically package your module in a jar file for distribution.

The module structure of GWT is hierarchical using inheritance. For example, if you write a module that inherits GWT's User module, then any module or application that uses your module also automatically inherits GWT's User module. This is an important feature, since it allows the users of your module to automatically get all the requirements to run. GWT takes this concept further and lets you also inject resources into modules to ensure CSS or other JavaScript libraries are automatically included.

For example, if you were creating a module of widgets that required default CSS to be included, you could reference this in the module XML like this:

```
<module>
   <inherits name='com.google.gwt.user.User'/>
   <stylesheet src="widgets.css"/>
</module>
```

The widgets file would need to be included in your module's public files, and when other modules inherit your module they would automatically get the widgets.css file without directly including it.

You can similarly include JavaScript in your module using the `script` tag like this:

```
<module>
   <inherits name='com.google.gwt.user.User'/>

   <!-- Include google maps -->
   <script src="http://maps.google.com/
maps?file=api&amp;v=2&amp;key=ABQIAAAACeDba0As0X6mwbIbUYWv-RTb-
vLQlFZmc2N8bgWI8YDPp5FEVBQUnvmfInJbOoyS2v-qkssc36Z5MA"></script>
</module>
```

This tag is similar to the `script` tag that you would use in your HTML file to include a JavaScript library, except that this file would be automatically included with every module that includes this module.

As of GWT 1.4 you can use image bundles to include resources with your reusable modules. Image bundles allow you to package several images together into a single image for deployment. If you use an image bundle within your module, applications that use your module will automatically generate the single image. In Chapter 6, images bundles are used to build the Gadget toolbar in the Gadget Desktop application.

Sharing a Compiled Application (Mashups)

Sometimes you'd like to share your compiled JavaScript application for use on other web sites. As of GWT 1.4, other web sites can easily load your application using the cross-site version of the generated JavaScript. The cross-site version has `-xs` appended to the package name for the JavaScript file name, and you can find it in the www directory after compiling an application. For example, to include the Hangman application developed in Chapter 1, you would use the following `script` tag:

```
<script language='javascript' src='http://gwtapps.com/hangman/
com.gwtapps.tutorial.Hangman-xs.nocache.js'></script>
```

Notice that this line differs from the line found in the original host HTML page for the application; it has the addition of `-xs` in the filename and is loading the script from the `gwtapps.com` domain.

Each application that you share may have additional requirements for integration on another site. In the Hangman example, the application looks for an HTML element with the ID `hangman`, so anyone including this on their site would need to also have the following HTML in the location where they'd like the Hangman application to show up:

```
<div id="hangman"></div>
```

Deploying Applications

Deploying a GWT application can be as easy as deploying a regular web page. A simple GWT client is made up of HTML and JavaScript files that can be copied to a directory on a web server and loaded into a browser. For example, the Gadget Desktop application in Chapter 6 does not use any server-side code, so its files for deployment are simply its JavaScript files,

several image files, and the host HTML file. You can install this application on any web server simply by copying the files.

Deploying to a Web Server

You've seen how to set up development environments with Java tools, run the GWT scripts, and use the GWT jar files, but for a client-side application these files are left on the development machine. You simply need to run the GWT compile script, or click the Compile button in the GWT hosted mode browser, to generate the files needed for deployment. For example, compiling the Gadget Desktop application can be done from the command line. Or it can be compiled from the hosted mode browser, as shown in Figure 4-47. GWT places these files in a directory named after your application inside the www directory for your project, as you can see in Figure 4-48. This is the file list that you would copy to a directory on your web server.

Figure 4–47. Compiling your application from the GWT hosted mode browser

Deploying a Servlet to a Servlet Container

If you are using GWT-RPC, you will need to deploy your service implementation to a servlet container. Although the GWT hosted mode browser runs

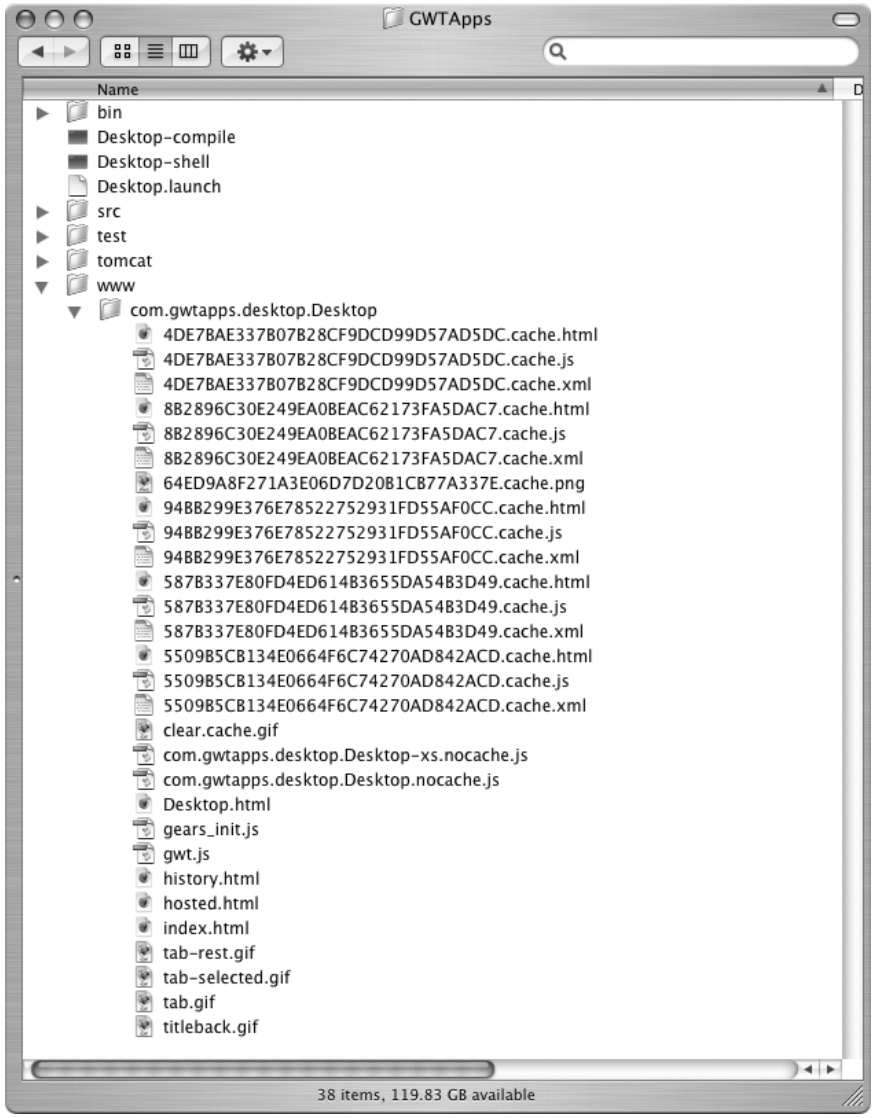

Figure 4–48. The GWT compiler places the files to deploy in www

an embedded version of Tomcat, deploying to a regular Tomcat instance is somewhat different. If you are deploying to Tomcat, you'll need to add your application to its webapps directory. Figure 4-49 outlines the steps to add your application to Tomcat's directory structure.

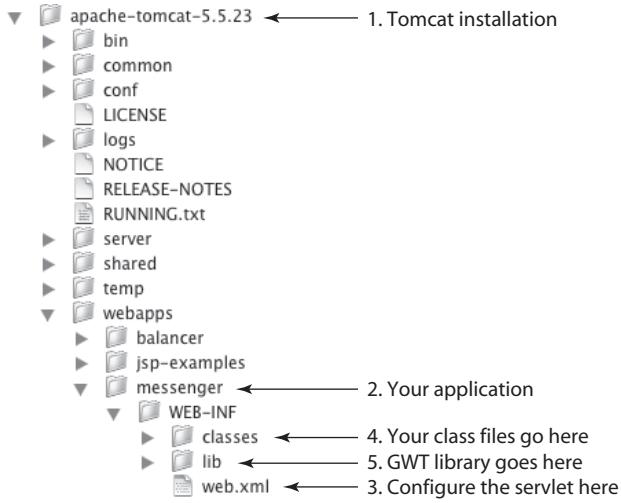

Figure 4–49. Steps to deploy your application to Tomcat

Let's look at the five steps shown in Figure 4-49. First you need to locate the installation directory for Tomcat. Second, you need to create your application directory under the webapps directory. Third, you need to set up your web.xml file in the WEB-INF directory for your application. For the Instant Messenger application, the file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <servlet>
      <servlet-name>messenger</servlet-name>
   <servlet-class>com.gwtapps.messenger.server.MessengerServiceImpl</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>messenger</servlet-name>
      <url-pattern>/messenger</url-pattern>
   </servlet-mapping>
</web-app>
```

Fourth, copy your servlet class to the class' directory in the WEB-INF directory. Finally, fifth, copy the gwt-servlet.jar file to the lib directory in the WEB-INF directory. The gwt-servlet.jar file has the GWT classes required to support the server-side RPC. You could use gwt-user.jar instead, but gwt-servlet.jar is smaller and therefore preferred. Deployment can be automated by using a build tool such as Ant.

Automating Deployment with Ant

As you can see from the previous section, deployment to a server container often involves many steps of compiling code, copying files, and creating directories. When a task involves many steps like this, it is best to automate the process. Ant is the ideal Java tool for automating build tasks like this. With it you can accomplish all of the previous steps of deploying a GWT web application with one Ant step.

Ant is a command line tool that accepts an XML build file. The build file contains a list of build targets with steps to accomplish build tasks. There is rich support for different types of steps, including copying files, creating directories, and compiling code. The Ant system is also extensible, so you can develop new steps or add new steps from other developers.

Let's run through an example of how to build a GWT application for use on a servlet container with Ant. First, verify that you have Ant installed and in your path. You should be able to type `ant -version` at the command line, as shown in Figure 4-50.

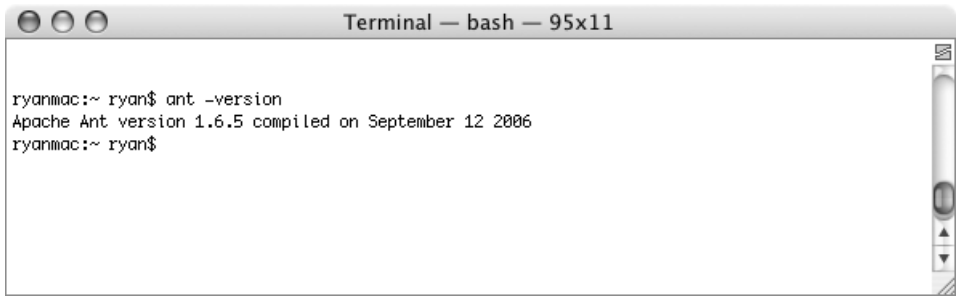

Figure 4–50. Verifying Ant is on your system

If you don't have Ant installed, you can download it from http://ant.apache.org. After ensuring that Ant is installed on your development machine, you can write a build.xml file for a project. The following is the build.xml file we will use:

```
<project default="deploy">
   <property name="gwtpath" value="/Users/ryan/lib/gwt-mac-1.4.10"/>
   <property name="gwtapipath" value="/Users/ryan/lib/gwt-google-apis-1.0.0"/>
   <property name="targetdir" value="${basedir}/www/${app}"/>
```

```
    <property name="wwwdir" value="${basedir}/www"/>
    <property name="srcdir" value="${basedir}/src"/>
    <property name="bindir" value="${basedir}/bin"/>

    <path id="classpath">
       <pathelement location="${gwtapipath}/gwt-google-apis.jar"/>
       <pathelement location="${gwtpath}/gwt-user.jar"/>
       <pathelement location="${gwtpath}/gwt-dev-mac.jar"/>
       <pathelement location="${srcdir}"/>
       <pathelement location="${bindir}"/>
    </path>

    <target name="compile-gwt">
       <java classname="com.google.gwt.dev.GWTCompiler" fork="true">
          <classpath refid="classpath"/>
          <jvmarg value="-XstartOnFirstThread"/>
          <arg value="-out"/>
          <arg value="${wwwdir}"/>
          <arg value="${app}"/>
       </java>
    </target>

    <target name="compile" depends="compile-gwt">
       <mkdir dir="${targetdir}/WEB-INF/classes"/>
       <javac srcdir="${srcdir}"
          destdir="${targetdir}/WEB-INF/classes"
          excludes="**/client/*.java">
          <classpath refid="classpath"/>
       </javac>
    </target>

    <target name="deploy" depends="compile">
       <mkdir dir="${targetdir}/WEB-INF/lib"/>
       <copy todir="${targetdir}/WEB-INF/lib" file="${gwtpath}/gwt-servlet.jar"/>
       <copy tofile="${targetdir}/WEB-INF/web.xml"
          file="${basedir}/${app}.web.xml"/>
    </target>
</project>
```

The file begins by defining a project element with a default target. This target is run when one is not specified on the command line. The first few elements inside the `project` tag are property definition elements. You can place variables in these elements that will be reused throughout the build file. For example, in this file we have the source directories and jar directories set for use later. Inside the attributes you can see how the properties can be referenced with the `${name}` format. Before the targets are defined in the file, we set a path element. This element lists the jar files

and directories that are on the classpath. We use this classpath later and can refer to it by its ID.

The first target, `compile-gwt`, runs the GWT compiler on our GWT module. The module is not specified in this target. Instead the `${app}` placeholder is used. We have not defined this as a property, but we can pass in this variable as a command line argument. This gives the build file the flexibility of being used for more than one application. Running this target generates the compiled JavaScript files for the application and copies all of the public files used for the project to the www directory.

The second target, `compile`, uses the regular javac compiler to compile all of the other Java class files. These are class files that will be needed on the server and will include the GWT-RPC service servlet if one is used. The Ant script copies these class files to the www directory under WEB-INF/classes. This is the standard location for class files for a servlet container web application.

The final target, `deploy`, copies the required GWT library, gwt-servlet.jar, to the WEB-INF/lib directory. This is the standard location for jar files for a servlet container web application. The target also copies a predefined web.xml file to the www directory. The web.xml file is required to describe the servlets in the web application.

Running the task for the Instant Messenger application in Chapter 9 results in the output shown in Figure 4-51. Once this is complete, we should have a www directory that is ready to be used in a servlet container, and which follows the conventions for servlet containers for file names and locations, as illustrated in Figure 4-52.

Figure 4–51. Compiling and deploying with Ant

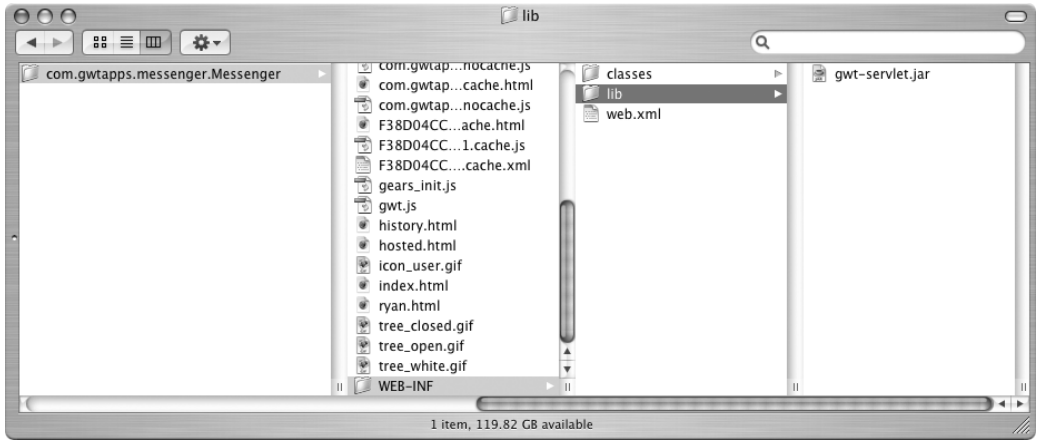

Figure 4–52. The output from an Ant script

Summary

GWT simplifies real software engineering for Ajax applications. This was really lacking when attempting to build substantial applications based on JavaScript. Using Eclipse to write and debug applications can substantially increase development productivity. Java organization and modularization help you decouple application parts and leverage existing code. Testing and benchmarking using JUnit helps ensure that your applications are of high quality and perform well. When it's time to deploy your application, Ant can automate any tedious tasks. Overall, the ability to leverage the vast range of mature Java software engineering tools is a significant part of creating great Ajax applications with GWT.