2 Getting started with the Eclipse Workbench

In this chapter...

- Downloading and installing Eclipse
- Essential Eclipse Workbench concepts, including perspectives, views, and editors
- Creating, running, and debugging a Java program
- Customizing Eclipse preferences and settings, including code format style and classpath variables
- Creating and modifying code generation templates

Getting started is often the hardest part of a journey. Mostly this isn't due to any real obstacle, but rather to inertia. It's easy to get set in your ways—even when you know that adventure waits. Eclipse is the new land we'll be exploring here. After downloading Eclipse and getting your bearings, you'll find that you'll soon be on your way, coding and debugging with ease.

2.1 Obtaining Eclipse

The first step toward getting started with Eclipse is to download the software from the Eclipse.org web site's download page at http://www.eclipse.org/downloads. Here you'll find the latest and the greatest versions—which are not usually the same things—as well as older versions of Eclipse. Basically, four types of versions, or *builds*, are available:

- *Release*—A stable build of Eclipse that has been declared a major release by the Eclipse development team. A release build has been thoroughly tested and has a coherent, well-defined set of features. It's equivalent to the shrink-wrapped version of a commercial software product. At the time of this writing, the latest release is 2.1, released March 2003; this is the release we will be using throughout this book.
- *Stable build*—A build leading up to a release that has been tested by the Eclipse development team and found to be relatively stable. New features usually first appear in these intermediate builds. These builds are equivalent to the beta versions of commercial software products.
- *Integration build*—A build in which Eclipse's individual components are judged to be stable by the Eclipse developers. There is no guarantee that the components will work together properly, however. If they do work together well, an integration build may be promoted to stable build status.
- *Nightly build*—A build that is (obviously) produced every night from the latest version of the source code. As you may guess, there are absolutely no guarantees about these builds—in fact, you can depend on their having serious problems.

If you are at all risk-averse (perhaps because you are on tight schedule and can't afford minor mishaps), you'll probably want to stick to release versions. If you are a bit more adventurous, or must have the latest features, you may want to try a stable build; the stable builds immediately before a planned release build usually offer the best feature-to-risk ratio. As long as you are careful to back up your workspace directory, these are a fairly safe bet. You can find out more about the

Eclipse team's development plans and the development schedule at http://www.eclipse.org/eclipse/development/main.html.

After you choose and download the best version for you, Eclipse installation consists of unzipping (or untarring, or whatever the equivalent is on your platform) the downloaded file to a directory on your hard disk. Eclipse, you'll be happy to learn, won't infect your system by changing your registry, altering your environment variables, or requiring you to re-boot. The only drawback is that you'll have to navigate your filesystem searching for the Eclipse executable to start it. If you don't want to do this each time you use Eclipse, you can create a shortcut to it, or put it on your path. For example, in Windows, after you find the Eclipse executable (eclipse.exe) using the Windows Explorer, right-click on it and select Create Shortcut. Doing so will create a shortcut in the Eclipse directory that you can drag to your desktop or system tray. On UNIX and Linux platforms, you can either add the Eclipse directory to your path or create a symbolic link (using `ln -s`) for the executable in a directory already in your path (for instance, /home/<*user*>/bin).

2.2 Eclipse overview

The first time you start Eclipse, it will ask you to wait while it completes the installation. This step (which only takes a moment) creates a workspace directory underneath the Eclipse directory. By default, all your work will be saved in this directory. If you believe in backing up your work on a regular basis (and you should), this is the directory to back up. This is also the directory to take with you when you upgrade to a new version of Eclipse.

You need to check the release notes for the new release to make sure it supports workspaces from prior versions; but barring any incompatibility, after you unzip the new version of Eclipse, simply copy the old workspace subdirectory to the new Eclipse directory. (Note that all your preferences and save perspectives will also be available to you, because they are stored in the workspace directory.)

2.2.1 Projects and folders

It's important to know where your files are located on your hard disk, in case you want to work with them manually, copy them, or see how much space they take up. However, native filesystems vary from operating system to operating system, which presents a problem for programs that must work consistently on different operating systems. Eclipse solves this problem by providing a level of abstraction above the native filesystem. That is, it doesn't use a hierarchy of directories and

subdirectories, each of which contains files; instead, Eclipse uses projects at the highest level, and it uses folders under the projects.

Projects, by default, correspond to subdirectories in the workspace directory, and folders correspond to subdirectories of the project folder; but in general, when you're working within Eclipse, you won't be aware of the filesystem. Unless you perform an operation such as importing a file from the filesystem, you won't be exposed to a traditional file open dialog box, for example. Everything in an Eclipse project exists within a self-contained, platform-neutral hierarchy.

2.2.2 The Eclipse Workbench

Eclipse is made up of components, and the fundamental component is the Eclipse Workbench. This is the main window that appears when you start Eclipse. The Workbench has one simple job to do: to allow you to work with projects. It doesn't know anything about editing, running, or debugging Java programs; it only knows how to navigate projects and resources (such as files and folders). Any tasks it can't handle, it delegates to other components, such as the Java Development Tools (JDT).

Perspectives, views, and editors

The Eclipse Workbench is a single application window that at any given time contains a number of different types of panes called *views* plus one special pane, the *editor*. In some cases, a single pane may contain a group of views in a tabbed notebook. Depending on the perspective, one pane might contain a console window while another might contain an outline of the currently selected project. The primary component of every perspective, however, is the editor.

Just as there are different types of documents, there are different types of editors. When you select (or create) a document in Eclipse, Eclipse does its best to open the document using the most appropriate editor. If it's a simple text document, the document will be opened using Eclipse's built-in text editor. If it's a Java source file, it will be opened using the JDT's Java editor, which has special features such as the ability to check syntax as code is typed. If it's a Microsoft Word document on a Windows computer and Word is installed, the document will be opened using Word inside Eclipse, by means of object linking and embedding (OLE).

You don't directly choose each of the different views in the Workbench or how they are arranged. Instead, Eclipse provides several preselected sets of views arranged in a predetermined way; they are called *perspectives*, and they can be customized to suit your needs.

Every perspective is designed to perform a specific task, such as writing or debugging a Java program, and each of the views in the perspective is chosen to allow you to deal with different aspects of that task. For example, in a perspective for debugging, one view might show the source code, another might show the current values of the program's variables, and yet another might show the program's output.

The first time you start Eclipse, it will be in the Resource perspective (see figure 2.1). You might think of this as the home perspective. It is a general-purpose perspective useful for creating, viewing, and managing all types of resources—whether a resource is a Java project or a set of word-processing documents doesn't matter in this perspective, apart from which editor is used to open specific documents in the editor area.

The panel at upper left is called the Navigator view; it shows a hierarchical representation of your workspace and all the projects in it. At first this view will

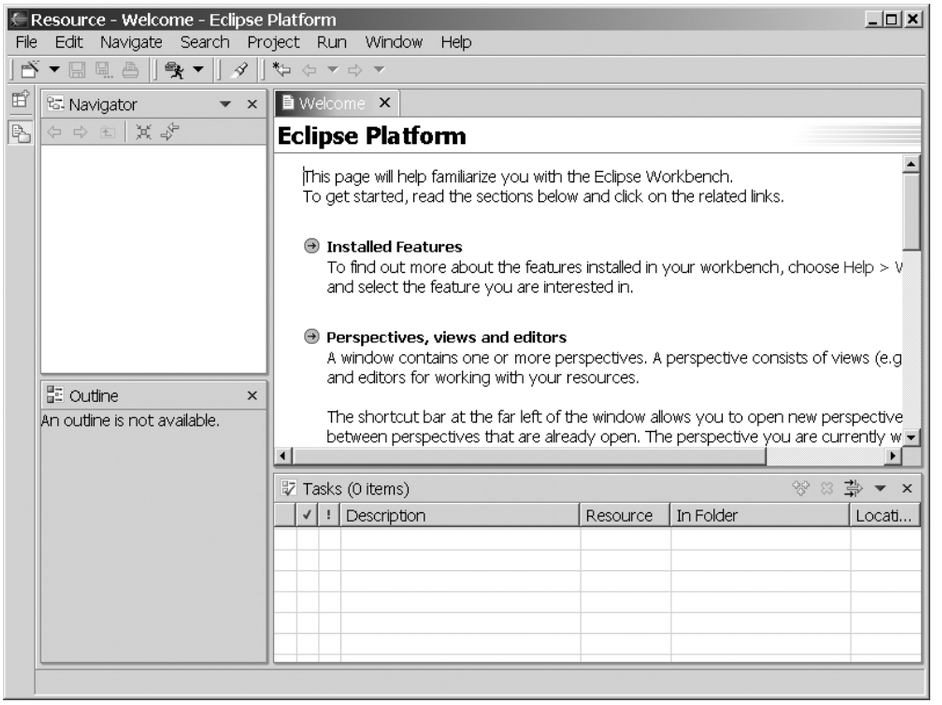

Figure 2.1 The initial view of Eclipse is the Resource perspective—a general-purpose perspective for creating, viewing, and managing all types of resources.

be empty, of course; but, as you'll see, it is the starting point for creating projects and working with Eclipse.

Within the Workbench, as you work, you can choose among the different perspectives by selecting Window→Open Perspective. Eclipse will also change the perspective automatically, when appropriate—such as changing from the Java perspective to the Debug perspective when you choose to debug a program from the Eclipse menu.

Menus and toolbars

In addition to perspective, views, and editors, several other features of the Workbench user interface (UI) are worth mentioning: the main menu, the main toolbar, and the shortcut toolbar. Like the views and editors in a perspective, the Workbench's menu and toolbar can change depending on the tasks and features available in the current perspective.

The Eclipse main menu appears at the top of the Workbench window, below the title bar (unless you are using a Macintosh, in which case the menu appears, Mac style, at the top of the screen). You can invoke most actions in Eclipse from the main menu or its submenus. For example, if the document HelloWorld.java is currently being edited, you can save it by selecting File→Save HelloWorld.java from the main menu.

Below the main menu is a toolbar called the main toolbar, which contains buttons that provide convenient shortcuts for commonly performed actions. One, for example, is an icon representing a floppy disk, which saves the contents of the document that is currently being edited (like the File→Save menu selection). These tool buttons don't display labels to indicate what they do unless you position the mouse pointer over them; doing so causes a short text description to display as a hovering *tool tip*.

Along the left side of the screen is another toolbar called the shortcut toolbar. The buttons here provide a quick way to open a new perspective and switch between perspectives. The top button, Open a Perspective, is an alternative to the Window→Open Perspective selection in the main menu. Below it is a shortcut to the Resource perspective. As you open new perspectives, shortcuts to those perspectives appear here, as well.

You can optionally add another type of shortcut to the shortcut toolbar: a *Fast View* button. Fast Views provide a way to turn a view in a perspective into an icon—similar to the way you can minimize a window in many applications. For example, you may find that in the Resource perspective, you need to look at the Outline view only occasionally. To turn the Outline view into a Fast View icon, click

on the Outline icon in the view's title bar and select Fast View from the menu that appears. The Outline view is closed, and its icon appears in the shortcut toolbar. Clicking on the icon alternately opens and closes the view. To restore the view in its previous place in the perspective, right-click on the Fast View icon and select Fast View.

In addition to the Workbench menu and toolbars, views can also have menus. Every view has a menu you can select by clicking on its icon. This menu lets you perform actions on the view's window, such as maximizing it or closing it. Generally this menu is not used for any other purpose. Views can also have a view-specific menu, which is represented in the view's title bar by a black triangle. In the Resource perspective, the Navigator view has a menu that lets you set sorting and filtering options.

Some views also have a toolbar. In the Resource perspective, the Outline view has tool buttons that let you toggle various display options on or off.

Changing perspectives

As you work in the Eclipse Workbench, you'll occasionally find that the different views aren't quite the right size for the work you're doing—perhaps your source code is too wide for the editor area. The solution is to click on the left or right window border and drag it so the window is the right size.

Sometimes you may want to supersize a view temporarily by double-clicking on the title bar; this will maximize it within the Eclipse Workbench. Double-clicking on the title bar again will reduce it back to its regular size.

You can also move views around by dragging them using their title bars. Dragging one view on top of another will cause them to appear as a single tabbed notebook of views. Selecting a view in a notebook is like selecting a document in the editor pane: Click its tab at the top or bottom of the notebook. Dragging a view below, above, or beside another view will cause the views to dock—the space occupied by the stationary view will be redistributed between the stationary view and the view you are dragging into place. As you drag the window you want to move, the mouse pointer will become a black arrow whenever it is over a window boundary, indicating that docking is allowed. For example, if you want to make the editor area taller in the Resource perspective, drag the Task view below the Outline view so the Navigator, Outline, and Task views share a single column on the left side of the screen.

In addition to moving views around, you can remove a view from a perspective by selecting Close from the view's title bar menu. You can also add a new view to a perspective by selecting Window→Show View from the main Eclipse menu.

Eclipse will save the changes you make to perspectives as you move from perspective to perspective or close and open Eclipse. To restore the perspective to its default appearance, select Window→Reset Perspective.

If you find that your customized perspective is particularly useful, you can add it to Eclipse's repertoire of perspectives. From the Eclipse menu, select Window→Save Perspective As; you will be prompted to provide a name for your new perspective.

2.3 The Java quick tour

Eclipse is installed, and you understand how the different views in perspectives work together to allow you to perform a task. Let's take Eclipse out for a spin by writing, running, and debugging a traditional "Hello, world" program.

2.3.1 Creating a Java project

Before you can do anything else in Eclipse, such as creating a Java program, you need to create a project. Eclipse has the potential to support many kinds of projects using plug-ins (such as EJB or C/C++), but it supports these three types of projects as standard:

- *Plug-in Development*—Provides an environment for creating your own plug-ins for Eclipse. This approach is great if you want to extend Eclipse to do new and wonderful things—but we'll get to that later. For now, you'll use Eclipse just the way it is.
- *Simple*—Provides a generic environment, which you might use for documentation.
- *Java*—Obviously, the choice for developing a Java program. Choosing this type of project sets up an environment with various Java-specific settings, including a classpath, source directories, and output directories.

To create a new Java project, follow these steps:

1. Right-click in the Navigator view to bring up a context menu and select New→Project.
2. In the New Project dialog box, Eclipse presents the project options: Java, Plug-in Development, and Simple. Because you want to create a Java program, select Java on the left side of the dialog box.
3. Select Java Project on the right. If you've installed other types of Java development plug-ins, various other types of Java projects may potentially be

listed here (EJBs and servlets, for example). But the JDT that comes standard with Eclipse only offers support for standard Java applications, so you must choose the Java Project option.

4. Click Next to start the New Java Project Wizard. (A *wizard* is a set of dialog boxes that prompts you through a set of well-defined, sequential steps necessary to perform a specific task. This feature is used extensively throughout Eclipse.)
5. The first dialog box prompts you for a project name. This is a simple "Hello, world" example, so enter **Hello**. Clicking Next would take you to a dialog box that lets you change a number of Java build settings, but for this example you don't need to change anything.
6. Click Finish.
7. Eclipse notifies you that this kind of project is associated with the Java perspective and asks whether you want to switch to the Java perspective. Check the Don't Show Me This Message Again box and click Yes.

The perspective changes to a Java perspective (see figure 2.2). Notice that the view in the upper-left corner is no longer the Navigator view; it is now the Package Explorer view, and it displays the new Hello project. The Package Explorer is similar to the Navigator, but it's better suited for Java projects; for one thing, it understands Java packages and displays them as a single entry, rather than as a nested set of directories. Notice also that a new icon has appeared on the left edge of the Workbench: a shortcut for the Java perspective.

At the bottom of the window is a Tasks view. It is useful for keeping track of what needs to be done in a project. Tasks are added to this list automatically as Eclipse encounters errors in your code. You can also add tasks to the Task view by right-clicking in the Tasks view and selecting New Task from the context menu; this is a convenient way to keep a to-do list for your project.

Finally, notice the Outline view on the right side of the screen. The content of this view depends on the type of document selected in the editor. If it's a Java class, you can use the outline to browse class attributes and methods and move easily between them. Depending on whether the Show Source of Selected Element button in the main toolbar is toggled on or off, you can view your source as part of a file (what is sometimes referred to as a *compilation unit*) or as distinct Java elements, such as methods and attributes.

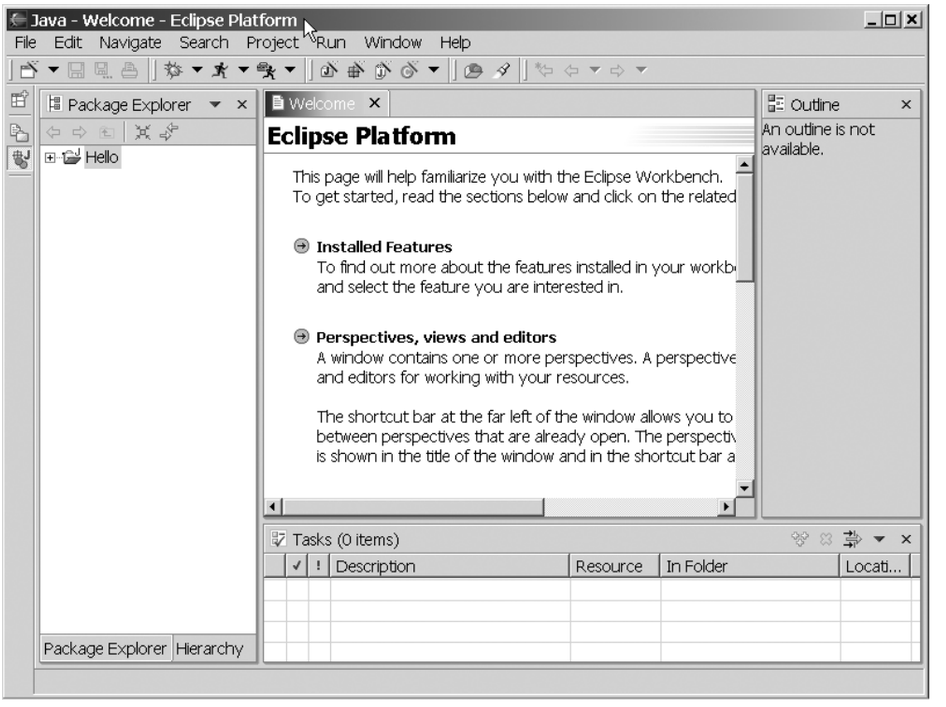

Figure 2.2 The Java perspective includes the Package Explorer view. This perspective is better suited for Java projects because it displays Java packages as a single entry instead of a nested set of directories.

2.3.2 Creating a Java class

Once you've created a project for it to live in, you can create your first Java program. Although doing so is not necessary, it's a good practice to organize your Java classes into packages. We'll put all packages in this book in the hierarchy starting with the Java-style version of the domain name associated with this book, `org.eclipseguide` (which of course is the reverse of the Internet style). Using domain names reduces the likelihood of name collisions—that is, more than one class with exactly the same name. You can use a registered domain name if you have one, but if not, you can use any convenient, unique, ad hoc name, especially for private use. Finally, add a name for this particular project: `hello`. All together, the package name is `org.eclipseguide.hello`.

Follow these steps to create your Java program:

1. Right-click on the project and select New→Class to bring up the New Java Class Wizard.
2. The first field, Source Folder, is by default the project's folder—leave this as it is.
3. Enter **org.eclipseguide.hello** in the Package field.
4. In the class name field, enter **HelloWorld**.
5. In the section Which Method Stubs Would You Like to Create?, check the box for `public static void main(String[] args)`. The completed New Java Class dialog box is shown in figure 2.3.
6. Click Finish, and the New Java Class Wizard will create the appropriate directory structure for the package (represented in the Navigator by the entry `org.eclipseguide.hello` under the Hello project) and the source file HelloWorld.java under this package name.

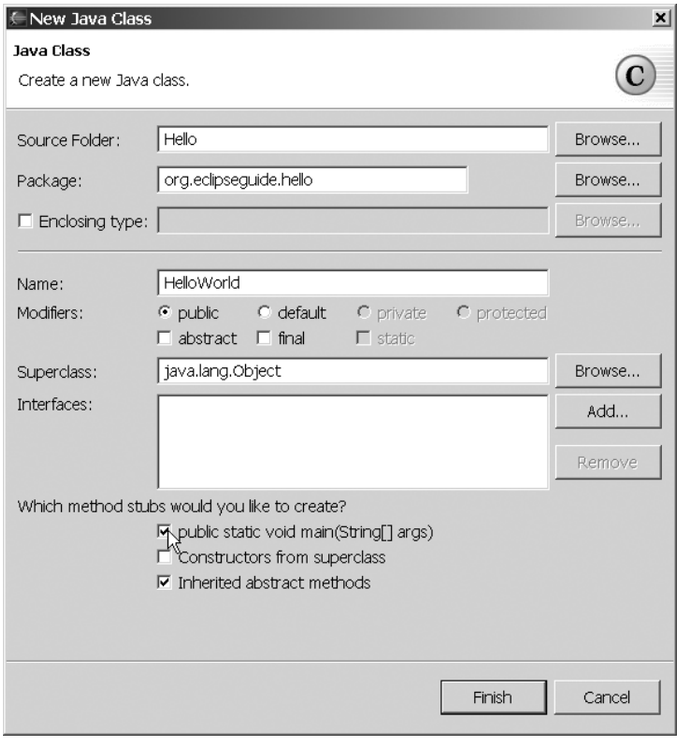

Figure 2.3 Creating the `HelloWorld` class using the New Java Class Wizard

If you examine the workspace directory in the native filesystem, you will find that there is not a single directory named org.eclipseguide.hello, but rather the series of directories that Java expects. If you've installed Eclipse in C:\Eclipse, the full path to your new source file will be C:\Eclipse\workspace\org\eclipseguide\hello\HelloWorld.java. Normally, though, you only need to deal with the visual representation that Eclipse provides in the Package Explorer view.

In the editor area in the middle of the screen, you see the Java code generated by the wizard. Also notice that tabs now appear at the top of the editor area, which allow you to select between the Welcome screen that first appeared and this new HelloWorld.java file. (You don't need the Welcome screen anymore, so you can click on the Welcome tab and click the X in the tab to make it go away.) You may also want to adjust the size of your windows and views to get a more complete view of the source code and the other views.

The code that's automatically generated is just a stub—the class with an empty method. You need to add any functionality, such as printing your "Hello, world!". To do this, alter the code generated by Eclipse by adding a line to `main()` as follows:

```
/*
 * Created on Feb 14, 2003
 *
 * To change this generated comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package org.eclipseguide.hello;

/**
 * @author david
 */
public class HelloWorld {

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

Code completion features

Notice that as you type the opening parenthesis, Eclipse helpfully inserts its partner, the closing parenthesis, immediately after the cursor. The same thing happens when you type the double quote to begin entering **"Hello, world!"**. This is one of Eclipse's code-completion features. You can turn off this feature if you find it as meddlesome as a backseat driver, but like many of Eclipse's other features, if you live with it, you may learn to love it.

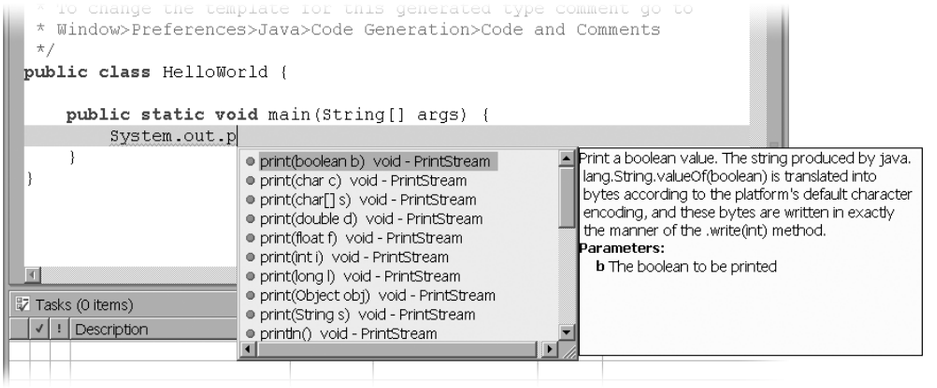

Figure 2.4 The Eclipse code assist feature displays a list of proposed methods and their Javadoc comments. Scroll or type the first letter (or more) to narrow the choice, and then press Enter to complete the code.

Depending on how quickly you type, you may see another code-completion feature called *code assist* as you type **System.out.println**. If you pause after typing a class name and a period, Eclipse presents you with a list of proposals—the methods and attributes available for the class, together with their Javadoc comments. You can find the one you want by either scrolling through the list or typing the first letter (or more) to narrow the choice; pressing Enter completes the code (see figure 2.4). This is most useful when you can't remember the exact name of the method you're looking for or need to be reminded what parameters it takes; otherwise you'll find that it's usually faster to ignore the proposal and continue typing the method name yourself.

You can also invoke code completion manually at any time by pressing Ctrl-Space. The exact effect will depend on the context, and you may wish to experiment a bit with this feature to become familiar with it. It can be useful, for example, after typing the first few letters of a particularly long class name.

Eclipse's code-generation feature is powerful and surprisingly easy to customize, because it is implemented using simple templates. You'll see it in greater depth when we examine Eclipse's settings and preferences.

2.3.3 Running the Java program

You're now ready to run this program. There are several things you might want to consider when running a Java program, including the Java runtime it should

use, whether it will take any command-line parameters, and, if more than one class has a `main()` method, which one to use. The standard way to start a Java program in Eclipse is to select Run→Run from the Eclipse menu. Doing so brings up a dialog box that lets you configure the launch options for the program; before running a program, you need to create a launch configuration or select an existing launch configuration.

For most simple programs, you don't need a special launch configuration, so you can use a much easier method to start the program: First make sure the HelloWorld source is selected in the editor (its tab is highlighted in blue) and then do the following from the Eclipse menu:

1. Select Run→Run As→Java Application.
2. Because you've made changes to the program, Eclipse prompts you to save your changes before it runs the program. Click OK.
3. The Task view changes to a Console view and displays your program output (see figure 2.5).

You may wonder why no separate step is required to compile the .java file into a .class file. This is the case because the Eclipse JDT includes a special incremental compiler and evaluates your source code as you type it. Thus it can highlight things such as syntax errors and unresolved references as you type. (Like Eclipse's other friendly features, this functionality can be turned off if you find it annoying.) If compilation is successful, the compiled .class file is saved at the same time your source file is saved.

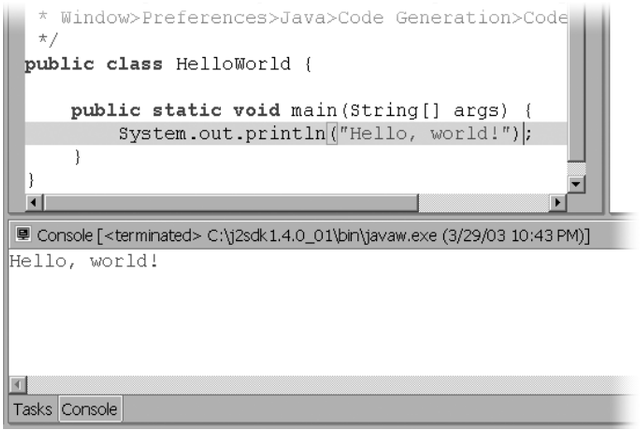

Figure 2.5
The Eclipse Console view displays the output from the HelloWorld program.

2.3.4 Debugging the Java program

If writing, compiling, and running a Java program were all Eclipse had to offer, it probably wouldn't seem worth the bother of setting up a project and using perspectives, with their shifting views, to get around; using a simple text editor and compiling at the command line is at least as attractive. As you learn how to use Eclipse more effectively, it will become increasingly obvious that Eclipse does have much more to offer, largely because it interprets the code in a more comprehensive way than a simple editor can—even an editor that can check syntax.

Eclipse's ability to run the code interactively is one major benefit. Using the JDT debugger, you can execute your Java program line by line and examine the value of variables at different points in the program, for example. This process can be invaluable in locating problems in your code.

Before starting the debugger, you need to add a bit more code to the HelloWorld program to make it more interesting. Add a `say()` method and change the code in the `main()` method to call `say()` instead of calling `System.out.println()` directly, as shown here:

```
public class HelloWorld {

      public static void main(String[] args) {
            say("Hello, world!");
      }
      public static void say(String msg) {
            for (int i = 0; i < 3; i++) {
                  System.out.println(msg);
            }
      }
}
```

To prepare for debugging, you also need to set a breakpoint in your code so the debugger suspends execution and allows you to debug—otherwise, the program will run to completion without letting you do any debugging. To set a breakpoint, double-click in the gray margin on the left side of the editor, next to the call to `say()`. A blue dot will appear, indicating an active breakpoint.

Starting the program under the debugger is similar to running it. Eclipse provides two options: Use the full-service Run→Debug menu selection to use a launch configuration, or use the express Run→Debug As→Java Application selection if the default options are OK. Here, as before, you can use the latter.

Make sure the source for HelloWorld is selected in the editor and select Run→Debug As→Java Application from the main menu. Eclipse will start the program, change to the Debug perspective, and suspend execution at the breakpoint (see figure 2.6).

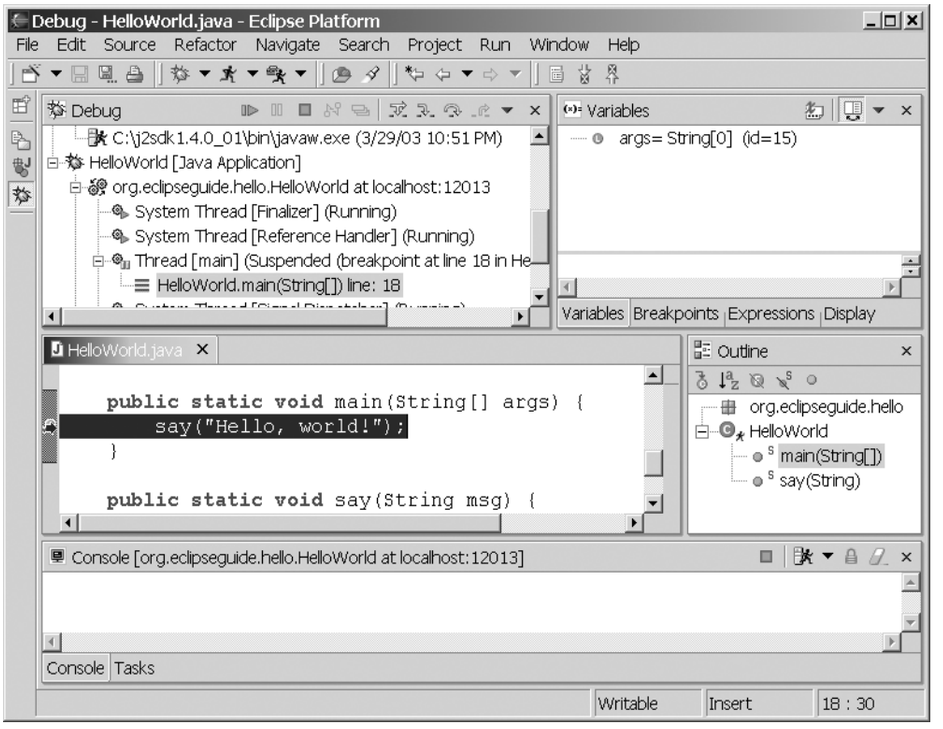

Figure 2.6 Debugging HelloWorld: Execution is suspended at the first breakpoint.

The Debug perspective includes several new views that are, not surprisingly, especially useful for debugging. First, at top left, is the Debug view (not to be confused with the Debug perspective to which it belongs), which shows the call stack and status of all current threads, including any threads that have already run to completion. Your program, which Eclipse started, has hit a breakpoint, and its status is shown as Suspended.

Stepping through code

In the title bar of the Debug view is a toolbar that lets you control the program's execution. The first few tool buttons, which resemble the familiar controls of electronic devices such as CD players, allow you to resume, suspend, or terminate the program. Several buttons incorporate arrows in their design; these allow you to step through a program a line at a time. Holding the mouse over each button in turn will cause tool tips to appear, identifying them as Step With Filters, Step

Into, Step Over, and Step Return. (There are several other buttons that we'll ignore for now; we'll look at them in chapter 3, "The Java Development Cycle: Test, Code, Repeat," when we examine debugging in greater detail.)

For example, click the second step button, Step Into. Doing so executes the line of code that is currently highlighted in the editor area below the Debug view: the call to the `say()` method. Step Into, as the name suggests, takes you into the method that is called: After clicking Step Into, the highlighted line is the first executable line in `say()`—the `for` statement.

The Step With Filters button works the same as Step Into, but it's selective about what methods it will step into. You normally want to step only into methods in your own classes and not into the standard Java packages or third-party packages. You can specify which methods Step Filter will execute and return from immediately by selecting Window→Preferences→Java→Debug→Step Filtering and defining *step filters* by checking the packages and classes listed there. Taking a moment to set these filters is well worth the trouble, because Step With Filters saves you from getting lost deep in unknown code—something that can happen all too often when you use Step Into.

Evaluating variables and expressions

To the right of the Debug view is a tabbed notebook containing views that let you examine and modify variables and breakpoints. Select the Variables tab (if it isn't already selected). This view shows the variables in the current scope and their values; before entering the `for` loop, this view includes only the `say()` method's `msg` parameter and its value, "Hello, world!". Click either Step Over or Step Into to enter the `for` loop. (Both have the same effect here, because you don't call any methods in this line of code.) The Variables view will display the loop index `i` and its current value, 0.

Sometimes a program has many variables, but you're interested in only one or a few. To watch select variables or expressions, you can add them to the watch list in the Expression view. To do this, select a variable—`i`, for instance—by double-clicking on it in the editor, and then right-click on the selection and choose Watch from the context menu. The variable (and its value, if it's in scope) will appear in the Expressions view.

One significant advantage of watching variables in the Variables and Expressions views over using `print` statements for debugging is that you can inspect objects and their fields in detail and change their values—even normally immutable strings. Return to the Variables view and expand the `msg` variable to show its attributes. One of these is a `char` array, `value`, which can be expanded to reveal

the individual characters in the `msg String`. For example, double-click on the character *H*, and you will be prompted to enter a new value, such as **J**.

The Display view is in the same tabbed notebook. It allows you to enter any variables that are in scope, or arbitrary expressions including these variables. Select Display view and enter the following, for example:

```
msg.charAt(i)
```

To immediately evaluate this expression, you must first select it and then click the second Display view tool button (Display Result of Evaluating Selected Text), which displays the results in the Display view. It's usually better to click the first tool button (Inspect Result of Evaluating Selected Text), because it adds the expression to the Expressions view. Either way, the value displayed is not automatically updated as the variables in the expression change; but in the Expressions view, you have the option of converting the expression into a *watch expression*, which is updated as you step through the code. To do this, change to the Expressions view. Notice that the Inspect icon (a magnifying glass) appears next to the expression. Click on the expression and select Convert to Watch Expression from the context menu. The icon next to the expression will change to the Watch icon.

Let's go back to stepping through the code. You previously left the cursor at the call to `System.out.println()`. If you want to see the code for `System.out.println()`, you can click Step Into; otherwise click Step Over to execute the `System.out.println()` method and start the next iteration of the `for` loop.

Below the editor area is another tabbed notebook, which includes a Console view. Program output appears here; if you made the earlier change to the variable `msg`, the line "Jello, world!" will appear. You can either continue to click Step Over until the loop terminates or, if you find this process tedious, click Step Return to immediately finish executing the `say()` method and return to the `main()` method. Or, just click the Resume button to let the program run to the end.

2.3.5 Java scrapbook pages

When you're writing a program, you sometimes have an idea that you're not sure will work and that you want to try before going through the trouble of changing your code. Eclipse provides a simple but slick alternative to starting a new project (or writing a small program using a simple editor for execution at a command prompt): Java *scrapbook pages*. By virtue of its incremental compiler, you can enter arbitrary Java code into a scrapbook page and execute it—it doesn't need to be in a class or a method.

To create a Java scrapbook page, change to the Java perspective, right-click on the HelloWorld project, and select New→Scrapbook Page from the context menu. When you're prompted for a filename, enter **Test**. Enter some Java code, such as the following example:

```
for(int i = 1; i < 10; i++)
{
      HelloWorld.say(Integer.toString(i));
}
```

To execute this code, you first need to import the `org.eclipseguide.hello` package, as follows:

1. Right-click inside the editor pane and select Set Imports from the context menu.
2. In the Java Snippet Imports dialog box that appears, select Add Packages.
3. In the next dialog box, type **org.eclipseguide.hello** in the Select the Packages to Add as Imports field. (You don't have to type the complete name—after you've typed one or more letters you can choose it from the list that Eclipse presents.)
4. Click OK.

Now you can execute the previous code snippet:

1. Highlight the code by clicking and dragging with the mouse.
2. Right-click on the selected code and select Execute from the context menu.
3. As with a regular Java program, the output from this code snippet appears in the console view below the editor.

This code doesn't require any additional imports; but if you used `StringTokenizer`, for example, you could import the appropriate package (`java.util.*`) as described. In such a case, however, it's easier to import the specific type by selecting Add Types in the Java Snippet Imports dialog box and typing in **StringTokenizer**. Eclipse will find the appropriate package and generate the fully qualified type name for you.

2.4 Preferences and other settings

So far, you've been using Eclipse with all its default settings. You can change many things to suit your taste, your working style, or your organization's coding conventions, by selecting Window→Preferences. Using the dialog that appears,

you can change (among numerous other settings) the fonts displayed, whether tabs appear at the top or bottom of views, and the code formatting style; you can also add classpath entries and new templates for generating code or comments. In this section, we'll look at a few of the settings you might want to change.

2.4.1 Javadoc comments

First, let's edit the text that appears when you create a new class. You'll remove the placeholder text, *To change this generated comment…*, and expand the Javadoc comments a bit. You'll also provide a reminder that you need to type in a class summary and a description. Follow these steps:

1. Select Window→Preferences→Java→Code Generation.
2. Click the Code and Comments tab on this page.
3. Select Code→New Java files, and click the Edit button.
4. Change the text to the following:

```
/* ${file_name}
 * Created on ${date}
 */
${package_declaration}

${typecomment}
${type_declaration}
```

5. Click OK in the Edit Template dialog box.

In addition to changing the template used whenever a new Java file is created, you need to change one of the templates it includes: the typecomment template. This is found on the same page, Code and Comments, under Comments:

1. Select Comment→Types and click the Edit button.
2. Change the text to the following:

```
/**
 * Add one sentence class summary here.
 * Add class description here.
 *
 * @author ${user}
 * @version 1.0, ${date}
 */
```

Notice that when you edit the template text, you don't need to type **${date}**—you can select it from the list of available variables by clicking the Insert Variable button. Appropriate values for the two variables in this template (`${user}` and `${date}`) will be inserted when the code is generated.

To see your changes, create a new class called `Test` in the `org.eclipseguide.hello` package. Note how all the variables have been filled out.

2.4.2 Format style

Two general styles are used to format Java code. The most common places an opening brace at the end of the statement that requires it and the closing brace in the same column as the statement, like this:

```
for(i = 0; i < 100; i++) {
    // do something
}
```

This is the default style that Eclipse uses when you right-click on your source code and select Format from the context menu.

The other style places the opening and closing braces in the same column as the statement. For example:

```
for(i = 0; i < 100; i++)
{
    // do something
}
```

To change to this style, do the following:

1. Select Java→Code Formatter in the Preferences dialog.
2. In the Options area, select the first tab, New Lines.
3. Check the first selection, Insert a New Line Before an Opening Brace. When you click to enable this option, the sample code shown in the window below the options is updated to reflect your selection. You may want to experiment with some of the other options to see their effects.

One of this book's authors prefers to enable Insert New Lines in Control Statements and Insert a New Line in an Empty Block, because he finds that doing so makes the structure of the code more obvious. But the important point (beyond one author's personal preference) is that the Eclipse Java editor makes it easy to change styles and reformat your code. If you are working as part of a team with established conventions and your personal preference doesn't conform, this feature lets you work in the style of your choice and reformat according to the coding convention before checking in your code.

2.4.3 Code generation templates

You saw earlier that when editing source code in the Java editor, pressing Ctrl-Space invokes Eclipse's code-generation feature. Depending on the context, this

key combination causes a template to be evaluated and inserted into the source code at that point.

You've already seen one example of a template: the code-generation template the New Class Wizard uses to add comments when it creates a new class file. In addition to this and the other code- and comment-generation templates, another set of templates is used to create boilerplate code such as flow control constructs; these templates are found in preferences under Java→Editor→Templates.

Let's create a template to simplify typing `System.out.println()`:

1. Select Windows→Preferences→Java→Editor→Templates.
2. Click the New button.
3. In the New Template dialog that appears, enter **sop** as the name, ensure that the context is Java, and enter **Shortcut for System.out.println()** as the description.
4. Enter the following pattern for the template:

```
System.out.println("${cursor}");
```

5. Click OK in the New Template dialog box (see figure 2.7).
6. Click OK in the Preference dialog to return to the Workbench.

The `${cursor}` variable here indicates where the cursor will be placed after the template is evaluated and inserted into the text.

To use the new template in the Java editor, type **sop** and press Ctrl-Space (or type **s**, press Ctrl-Space, select *sop* from the list that appears, and press Enter).

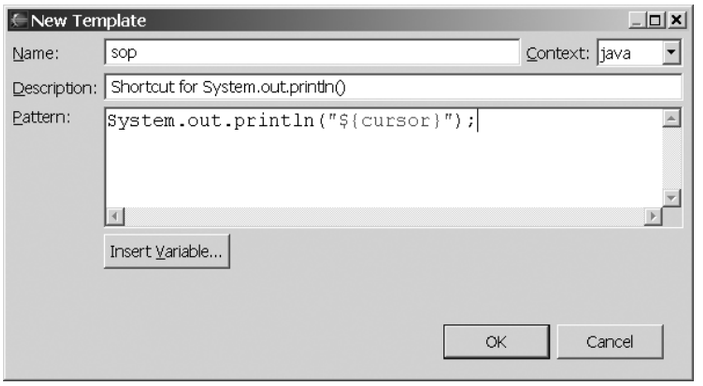

Figure 2.7 Creating a shortcut for `System.out.println()` using a Java editor template

The letters *sop* are replaced with the `System.out.println()` method call, and the cursor is replaced between the quotation marks, ready for you to type the text to be printed.

Let's create one more template to produce a `for` loop. There are already three `for` loop templates; but the template you'll create is simpler than the existing ones, which are designed to iterate over an array or collection:

1. Select Windows→Preferences→Java→Editor→Templates.
2. Click the New button.
3. Enter **for** as the name, **Simple for loop** as the description, and the following pattern:

```
for(int ${index}=0; ${index}< ${cursor}; ${index}++})
{
}
```

4. Click OK in the New Template dialog box.
5. Click OK in the Preference dialog to return to the Workbench.

Notice that this example uses a new variable, `${index}`, which proposes a new index to the user. By default, this index is initially `i`; but the cursor is placed on this index, and anything you type (such as **j** or **foo**) replaces the `${index}` variable everywhere in the template.

Try this new template by typing **for** and pressing Ctrl-Space. From the list that appears, select the entry Simple For Loop. Type a new name for the index variable, such as `loopvar`, and notice that it automatically appears in the test and increment clauses. You might also notice that the index variable has a green underline, indicating a link; pressing Tab will advance the cursor to the next link. In this case, pressing Tab takes you to the `${cursor}` variable. At this point, you can type a constant, variable, or other expression, as appropriate.

2.4.4 Classpaths and classpath variables

There are several ways you can add a directory or a JAR file to a project's classpath: when you create the class using the New Class Wizard, by editing your project options, or by creating a launch configuration for your project. In each case, you can either enter the path to the JAR file or directory you wish to add, or you can use a classpath variable. If you are only adding a JAR file for testing purposes, or if the JAR file is one you'll use only in this project, it's easiest to add the path and filename explicitly. But if it's something you are likely to use in many of your projects (for example, a JDBC driver), you may wish to create a classpath

variable. Besides being easier to type, a classpath variable provides a single location to specify the JAR files used by your projects. This makes it easier to manage your JAR files. When you want to upgrade to a new version of a JAR, a single change will update all your projects.

Suppose you will be using the MySQL database and that the full path and filename of your JDBC driver is c:\mysql\jdbc\lib\mm.mysql-2.0.14-bin.jar. To create a classpath variable for this JAR file, open the Window→Preferences dialog and select Java→Classpath Variables. Click New and enter **MYSQL_JDBC** as the name; either browse for the JAR file by clicking the File button or type the path and filename manually. Click OK twice to save and return to the Workbench.

Now, when you need to add the MySQL JDBC JAR to a project, you don't have to search your hard drive for it; `MYSQL_JDBC` is one of the available classpath variables you can select. To add it to your Hello project, for example, right-click on the project name and select Properties from the context menu. Select Java Build Path on the left side of the dialog box that appears and then select the Libraries tab on the right. You could add the JAR explicitly by selecting Add External Jars, but instead select Add Variable, click MYSQL_JDBC (see figure 2.8), and click OK.

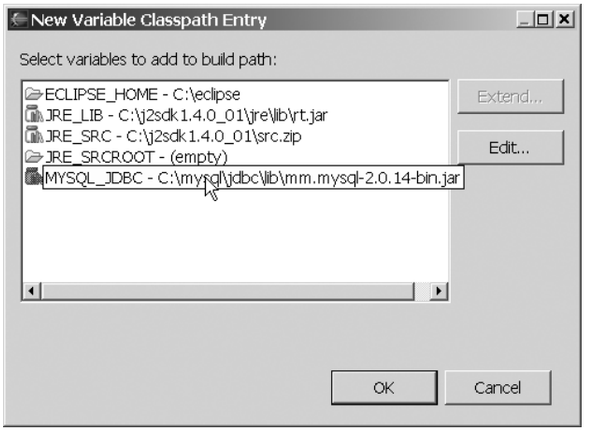

Figure 2.8
Creating a new classpath variable. Classpath variables make it easier to manage your classpath and provide flexibility as well.

2.4.5 Exporting and importing preferences

Eclipse's preferences and settings are numerous, and you can spend a lot of time customizing it to your taste and needs. Fortunately, there is a way to save these settings so you can apply them to another Eclipse installation or share them with

your friends, or, more importantly, so you have a backup in case the file they are stored in (the Eclipse metadata file) gets corrupted.

The Windows→Preferences box has two buttons at the bottom: Import and Export. To save your preferences, click the Export button, type in a filename, and click Save to create an Eclipse preference file. To restore preferences from a preference file, click the Import button, locate the file, and click Open.

2.5 Summary

Many different versions of Eclipse are available—you aren't limited to using only a stable, officially released version. This is one of the most interesting features of open source software. Deciding which one to use requires balancing stability with features. If you need a rock-solid product, you may wish to stick to a release version. If you are a little more daring or you absolutely require a specific new feature, you may wish to try the latest stable release. If you're just curious to see what's new, you can try an integration build. In this book, we're using the official 2.1 release, but most of the material will remain largely applicable to future releases.

The first key to using Eclipse effectively is understanding its organizational concepts of perspectives, views, and editors. The Eclipse Workbench—the window that appears on your screen when you start Eclipse—contains a number of different panes called *views*. The different views that appear at one time on the Workbench are especially selected to enable you to accomplish a specific task, such as working with Java source files. The title bar of each view has a window menu and, optionally, a view-specific menu, a toolbar, or both.

In addition to views, most perspectives have an *editor* as their central component. The specific editor that appears at any given time depends on the resource being edited. A Java source file, for example, will be opened automatically using the JDT Java editor. The Workbench also has a number of other UI elements beside views and an editor: a main menu bar at the top, a main toolbar below that, and a shortcut toolbar along the left side. Because a perspective is a collection of these views, menus, toolbars, and their relative positions, all of these elements can change as the perspective changes. The best way to learn how to use these features is to perform basic tasks, beginning with creating a Java project. (Eclipse is not limited to creating Java projects, but the Java Development Toolkit that is included is powerful, easy to use, and the most popular reason for using Eclipse.) Writing a program, running it, and debugging it provides a good introduction to Eclipse's features.

Eclipse is also highly customizable. You can modify many settings and preferences using the Windows→Preferences selection from the main menu. Preferences can be saved and restored using the Windows→Preferences Import and Export buttons; if you spend a lot of time customizing Eclipse, it's a good idea to export your changes to an Eclipse preference file for backup.

8 Introduction to Eclipse plug-ins

In this chapter...

- Understanding Eclipse's plug-in architecture
- Preparing your Workbench for plug-in development
- Using the Plug-in Development Environment (PDE)
- Creating simple plug-ins using the built-in wizards and templates

Up to now, you've been using Eclipse as it comes out of the box. As you'll discover in this chapter, however, the beauty of Eclipse lies in its extensible architecture. This architecture allows anyone to add features and capabilities the original designers never dreamed of.

Eclipse's loosely connected design is perfect for systems that aren't designed all at once, but instead are built up from components written for particular needs. From its earliest roots, Eclipse was designed as an "open extensible IDE for anything, and nothing in particular." The decentralization that plug-ins provide gives the Eclipse Platform the ability to morph into any application and support any language. Come with us now as we seek out that man behind the curtain and learn the secrets of Eclipse plug-ins.

8.1 Plug-ins and extension points

Imagine a giant jigsaw puzzle. A few pieces are already connected for you—these will form the core around which the rest of the puzzle is built. The boundaries between the pieces are uniquely cut to fit snugly together. If Eclipse is the puzzle, then the pieces are plug-ins. A *plug-in* is the smallest extensible unit in Eclipse. It can contain code, resources, or both.

The Eclipse Platform consists of nearly 100 plug-ins working together. The boundaries between these pieces that let plug-ins connect to one another are called extension points. An *extension point* is the mechanism by which one plug-in can add to the functionality of another.

Appendix C lists the extension points provided by the Platform. Each one can be used to add some new component such as a menu or view to the system, and is usually associated with a Java class that performs the logic for the component.

Unlike most jigsaw puzzles, though, Eclipse has no corners or straight edges. It can be extended forever, with each new plug-in defining its own extension points that other plug-ins can use. Large projects such as WebSphere Studio have hundreds of plug-ins. (Better bring a big table.)

8.1.1 Anatomy of a plug-in

Plug-ins are conceptually simple. If you look at the directory where you installed Eclipse in chapter 2, you will see a subdirectory called plugins. Inside this directory you'll find one directory for every plug-in. The name of each directory is the same as the name of the plug-in, followed by an underscore and a version number. For example:

```
C:\ECLIPSE
|
+---features
+---plugins
|   +---org.eclipse.ant.core_2.1.0
|   |   |    .options
|   |   |    about.html
|   |   |    antsupport.jar
|   |   |    plugin.properties
|   |   |    plugin.xml
|   |   |
|   |   +---lib
|   |
|   +---org.eclipse.compare_2.1.0
|   |
|   ...
+---workspace
```

The `org.eclipse.ant.core` plug-in provides the Eclipse Platform with its integration with the Ant builder (see chapter 5). In every plugins folder, including this one, you will find a *plug-in manifest* file (plugin.xml) together with some optional files. The manifest describes the plug-in—its name, its version number, and so forth. It also lists the required libraries and all the extension points used and defined by the plug-in.

The files and folders typically seen in a plugins folder are as follows:

- *plugin.xml*—Plug-in manifest
- *plugin.properties*—Contains translatable strings referenced by plugin.xml
- *about.html*—Standard location used for licensing information
- **.jar*—Any Java code needed for the plug-in
- *lib*—Directory for more JAR files
- *icons*—Directory for icons, usually in GIF format
- *(other files)*—As needed

8.1.2 The plug-in lifecycle

When you first start the Eclipse Platform, it scans the plugins directory to discover what plug-ins have been defined (this is a slight simplification, but close enough for this discussion). If it finds more than one version of the same plug-in, only one (typically the one with the highest version number) will be used. The list of plug-ins the Platform builds during this scan is called the *plug-in registry*. Although the Platform reads all the plug-in manifests, it doesn't actually load the

plug-ins (that is, run any plug-in code) at this point. Why? To make Eclipse start up faster.

Plug-ins are loaded only when they are first used. For example, if you write a plug-in that defines a menu item, Eclipse can tell by looking at the manifest where the menu should go and what the text of the menu is. Because of the information in the manifest, Eclipse can delay loading your plug-in until it is really needed.

If you select the menu, the plug-in is loaded at that point. This behavior is especially important in large Eclipse-based products with hundreds of plug-ins. Most of the plug-ins will not be needed, because they are in specialized parts of the product that may never be run. So, any time spent loading and initializing those plug-ins would be wasted. This is sometimes referred to as *lazy loading*.

When are plug-ins unloaded? The short answer is, never. However, one of the goals of the Equinox project (http://www.eclipse.org/equinox) is to allow plug-ins to be loaded and unloaded on demand, so this situation may change in the future.

8.1.3 Creating a simple plug-in by hand

Eclipse plug-ins can be created without any special tools. To demonstrate, create a subdirectory in the plugins directory called org.eclipseguide.simpleplugin_1.0.0. Inside this directory, use a text editor like Notepad or vi to create a plugin.xml file containing the following lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.eclipseguide.simpleplugin"
   name="Simple Plug-in"
   version="1.0.0"
   provider-name="Eclipse in Action">
</plugin>
```

Now save the file and restart Eclipse. You won't notice anything different, because this plug-in doesn't do anything. However, you can tell it was registered by selecting Help→About Eclipse Platform and then clicking Plug-in Details. Scroll down to the bottom of this window, and you'll see the plug-in listed as shown in figure 8.1. The More Info button is grayed out because you didn't create an about.html file.

Table 8.1 describes the purpose of each line in plugin.xml.

Congratulations—you have just created your first plug-in! Next we'll look at the tools Eclipse provides to make this process manageable for more complex projects.

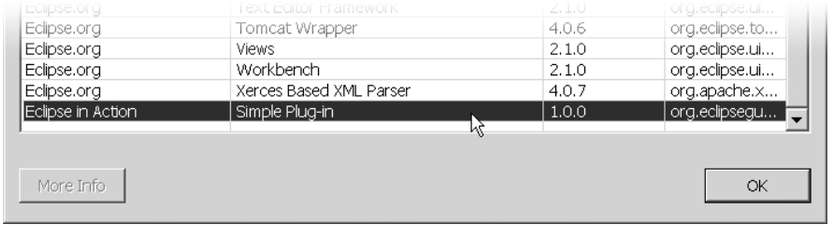

Figure 8.1 In the About page, you can click the Plug-in Details button to see the list of installed plug-ins. The Simple Plug-in shown here was discovered by the Eclipse Platform during startup.

Table 8.1 The plug-in manifest file (plugin.xml) for each plug-in is read when Eclipse starts, in order to build up its plug-in registry. Here is the simplest manifest possible and the meaning of each line.

Line	Purpose
`<?xml version="1.0" encoding="UTF-8"?>`	Required XML prolog; never changes
`<plugin`	Starts defining a new plug-in
`id="org.eclipseguide.simpleplugin"`	Provides the fully qualified id for the plug-in
`name="Simple Plug-in"`	Gives the plug-in a human-readable name
`version="1.0.0"`	Specifies a version number
`provider-name="Eclipse in Action">`	Provides information about the author
`</plugin>`	Finishes defining the plug-in

8.2 The Plug-in Development Environment (PDE)

Creating a plug-in by hand is an interesting exercise, but it would quickly become tedious in practice. Plug-in manifests can grow to be hundreds of lines long, and they need to be coordinated with names and data in various source and property files. Also, plug-ins need a fair amount of boilerplate code in order to run. That's why Eclipse provides a complete Plug-in Development Environment (PDE). The PDE adds a new perspective and several views and wizards to the Eclipse Platform to support creating, maintaining, and publishing plug-ins:

- *Plug-in Project*—A normal plug-in; the most common type
- *Fragment Project*—An addition to a plug-in (for languages, targets, and so on)
- *Feature Project*—An installation unit for one or more plug-ins
- *Update Site Project*—A web site for automatic installs of features

8.2.1 Preparing your Workbench

Before you start using the PDE, you should turn on a few preferences. They are off by default, because Eclipse users who are not building plug-ins don't need them. Bring up the Preferences window (Window→Preferences) and do the following:

1. Select Workbench→Label Decorations and turn on the Binary Plug-in Projects decoration. This is optional, but if you use binary plug-ins (see section 8.2.2) it will help them stand out from the rest of your projects.
2. Select Plug-In Development→Compilers and set all the messages to Warning. Doing so will provide an early indication of any problems in your plug-in manifests.
3. Select Plug-In Development→Java Build Path Control and turn on the Use Classpath Containers for Dependent Plug-ins option. This confusingly named option causes all plug-in JARs that your plug-in uses to appear in a folder of your project called Required Plug-in Entries. The nice thing about this special folder is that Eclipse dynamically manages it based on your plug-in's dependencies.
4. Click OK. A dialog will appear, stating that the compiler options have changed and asking whether you would like to recompile all the projects. Click Yes.

8.2.2 Importing the SDK plug-ins

As mentioned earlier, the Eclipse Platform is made up of dozens of plug-ins. Wouldn't it be nice if you could see the source code for all those plug-ins, and do searches to see how certain classes and interfaces are used internally? The API documentation is not perfect, so this is an important tool for plug-in developers. Of course, you could connect to the Eclipse CVS Repository (host **dev.eclipse.org**, path **/home/eclipse**, user **anonymous**) and download what you need, but there is a better way. It turns out that if you downloaded the Eclipse Platform SDK then all the source code is already installed, just waiting to be used.

The easiest method is to hold down the Ctrl key and hover your mouse over a class or object name in the Java editor, and then click on the name. If the source is available, Eclipse will open it. Or, using the Package Explorer, you can expand just about any JAR file and double-click on one of its class files to open it in the editor.

Sometimes, though, it's more convenient to bring these plug-ins into your workspace just like your regular projects. For example, you can search your

entire workspace for references to a type, but if the type is not currently in the workspace, then it won't be found. The Required Plug-in Entries folder is searched, but it contains only the JAR files from plug-ins you are currently dependent on.

To bring installed plug-ins into your workspace, select File→Import→External Plug-ins and Fragments and then click Next (see figure 8.2). Turn off the option to Copy Plug-in Contents into the Workspace Location and click Next. Then, select the plug-ins you want to import and click Finish. This is called a *binary import*, and projects created this way are *binary plug-ins* because you didn't build them from source.

Later, if you decide you don't want them in your workspace, just delete them—doing so will not affect the Eclipse installation. You can also temporarily

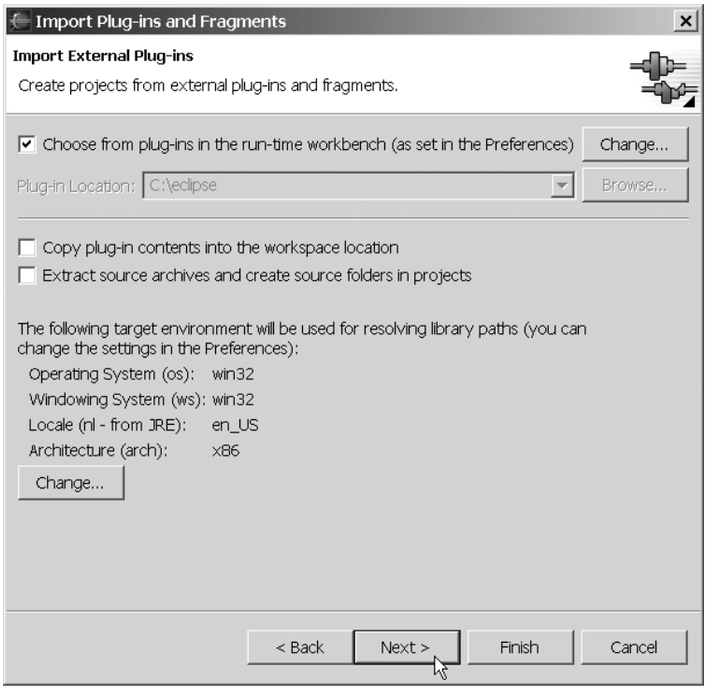

Figure 8.2 You can bring any installed plug-ins into your workspace by importing them. Doing so creates a binary plug-in project for each one and makes them available for searching and browsing. This is a great way to discover how the Eclipse Platform uses the Eclipse SDK classes and interfaces.

hide them from the Package Explorer menu: Select Filters, turn on the option to Exclude Binary Plug-in and Feature Projects, and click OK.

8.2.3 Using the Plug-in Project Wizard

The PDE makes it easy to create a new plug-in by providing wizards that ask a few questions and generate much of the code for you. Let's walk through a simple example:

1. Select File→New→Project to bring up the familiar New Project Wizard shown in figure 8.3.
2. Select Plug-in Development on the left-hand side to bring up the list of plug-in wizards on the right. You can use the Plug-in Project Wizard to create new plug-ins; select it and then click Next to open the first page of the wizard.
3. Enter a name for the plug-in, such as **org.eclipseguide.helloplugin** (see figure 8.4). We recommend using a fully qualified name like this so it can match the plug-in name and not collide with anyone else's name. By default, the PDE creates the plug-in in your normal workspace directory (either the workspace directory where you installed Eclipse or the directory you specified with Eclipse's `-data` option). Click Next to get to the next page.
4. Enter the fully qualified ID of the plug-in (see figure 8.5); by default, the ID is the same as the project name, which is what you want. Select the Create a Java Project option. Some plug-ins consist of only resources,

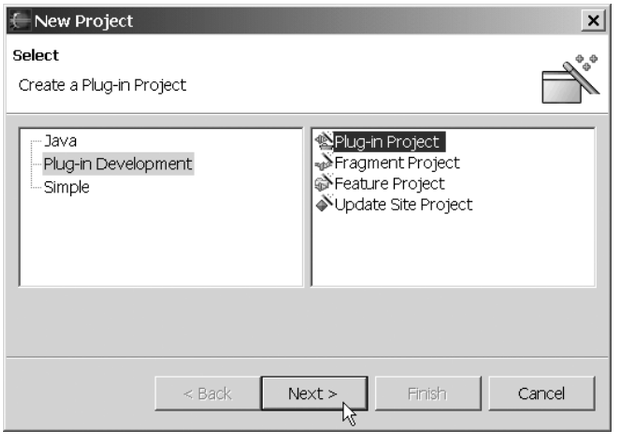

Figure 8.3
The PDE provides several wizards for creating new projects. See section 8.2 for a description of each type of project supported.

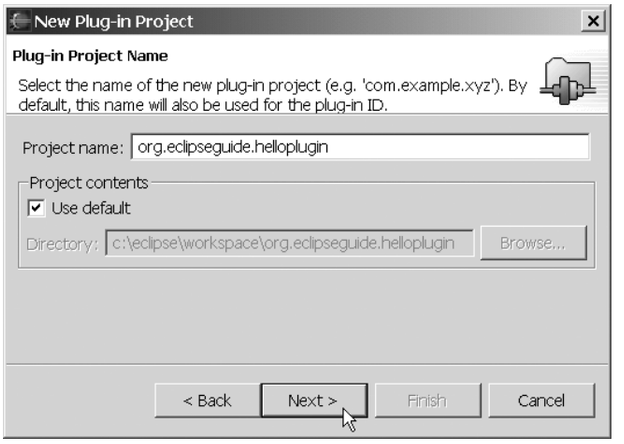

Figure 8.4
Specify the name and location of the project in the first page of the Plug-in Project Wizard.

such as a plug-in that only contains help files, but most of the time plug-ins have some Java code associated with them. The Java Builder Output option controls where the Eclipse compiler places generated .class files. The Plug-in Runtime Library is the JAR file that holds all your Java code, and Source Folder allows you to change the subdirectory that contains your .java files. The defaults for these settings are fine, so click Next to bring up the code generation page.

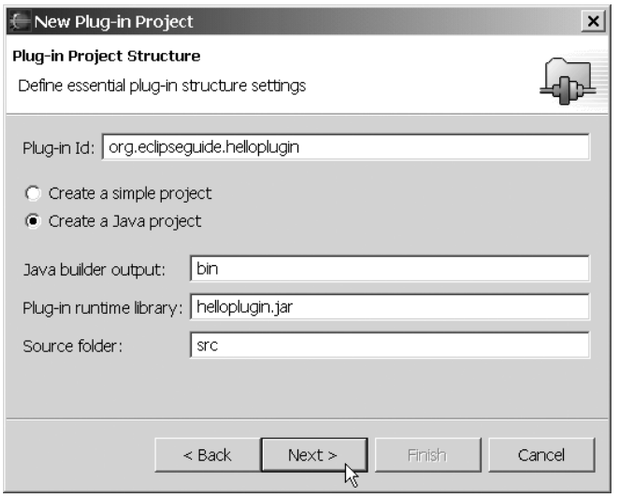

Figure 8.5
Specify the fully qualified ID of the plug-in in the second page of the New Plug-in Project Wizard. You can also control whether this plug-in will contain Java code.

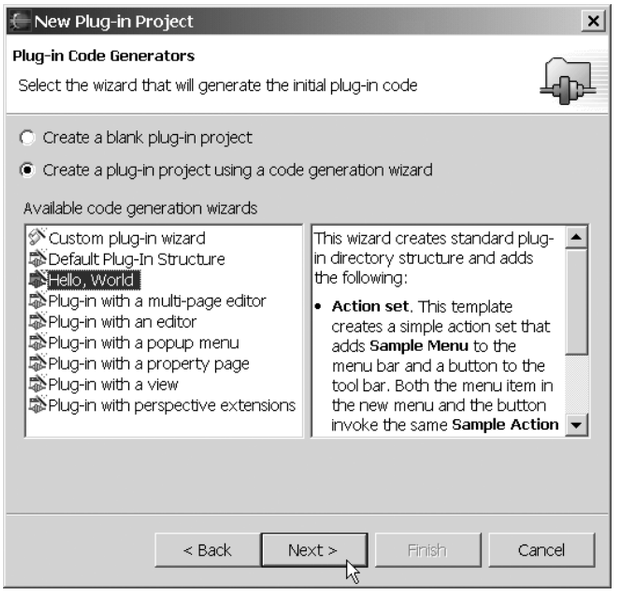

Figure 8.6
Select a code-generation wizard to quickly create a new plug-in from a template. Using the templates lessens the learning curve for Eclipse extensions and is less error-prone than creating the code from scratch. There also is an experimental feature in Eclipse 2.1 for adding your own wizards to this dialog.

5 The PDE provides several standard templates to help you quickly create some common kinds of plug-ins. A few are listed on this page (figure 8.6); you can see the rest by selecting the Custom Plug-in Wizard. The option to Create a Blank Plug-in Project makes a minimal directory with a plugin.xml file but not much else; no code is generated. The Default Plug-in Structure Wizard creates a top-level Java class for you but does not use any extension points. It is possible to add new templates, if necessary.

You're almost done. Now you just have to decide which template to use.

8.3 The "Hello, World" plug-in example

It's time for another "Hello, World" example—this one for plug-ins. In the New Plug-in Project page, select the Hello, World Wizard. It creates the default plug-in structure and also uses two extension points (`org.eclipse.ui.actionSets` and `org.eclipse.ui.perspectiveExtensions`) to add an item to the menu bar and the tool bar. (These extension points and the other wizards will be discussed later). Now, follow these steps:

1 In the code-generation dialog, click Next to start the Hello, World Wizard. Set the Plug-in Name to **Hello Plug-in**, the Class Name to **org.eclipse-**

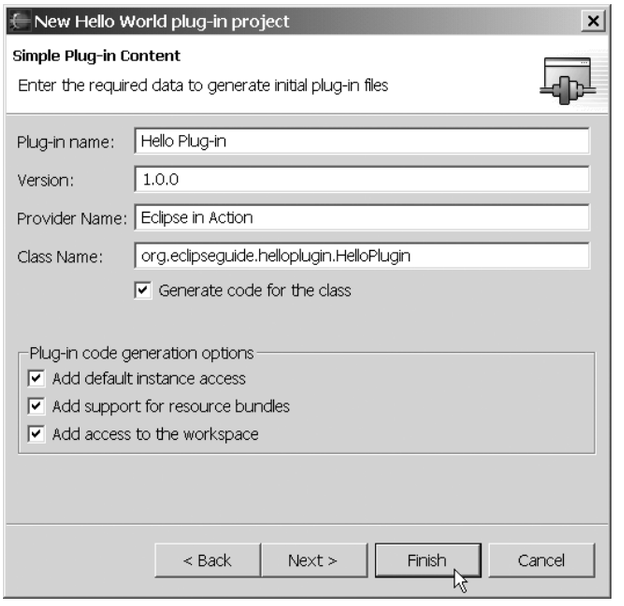

Figure 8.7
This page is common to most plug-in wizards. Use it to fill in the plug-in name and other required data.

guide.helloplugin.HelloPlugin, and the Provider Name to **Eclipse in Action** (see figure 8.7). The next page would let you control the text the example will display; however, the defaults are good, so click Finish to generate the directories, files, and classes necessary for the project.

2. You may get a dialog telling you that the wizard is enabling any needed plug-ins (figure 8.8). This is normal, so click OK.
3. Another dialog asks if you want to switch to the Plug-in perspective (figure 8.9). Click Yes.

That's it! You now have a plug-in project, as shown in figure 8.10.

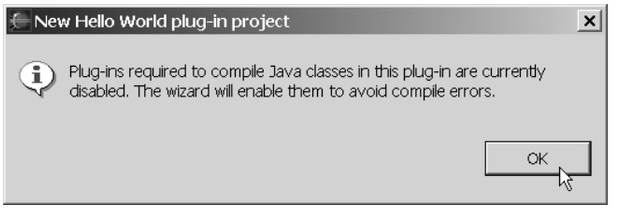

Figure 8.8
The wizard automatically enables plug-ins that this plug-in depends on. You can see the list of enabled plug-ins in the Preferences dialog under Plug-In Development→Target Platform.

Figure 8.9
Plug-in development is best accomplished in the Plug-in perspective. This is optional, however; you can use any perspective you like, as long as it has the views you need.

Figure 8.10
The final result is a plug-in project that prints "Hello, World". Here we have expanded some of the folders so you can see the generated files.

8.3.1 The Plug-in Manifest Editor

The PDE automatically opens the Plug-in Manifest Editor when you first create a plug-in (see figure 8.11). You can bring it up later by double-clicking on plugin.xml. This multipage editor provides convenient access to all the different sections of the plugin.xml file.

Because you'll be spending a great deal of time in the Plug-in Manifest Editor, it's a good idea to familiarize yourself with it now. Its pages are as follows:

- *Welcome*—A quick introduction to the Manifest Editor with links to some of the most important sections. You can turn off this page once you are familiar with the editor.
- *Overview*—Summarizes the plug-in, including the name, version number, extension points consumed and provided, and other information. This is the page you will use most often.
- *Dependencies*—Specifies the plug-ins required for this plug-in.
- *Runtime*—Defines the libraries that need to be included in the plug-in's classpath and whether classes in those libraries should be exported for use by other plug-ins.

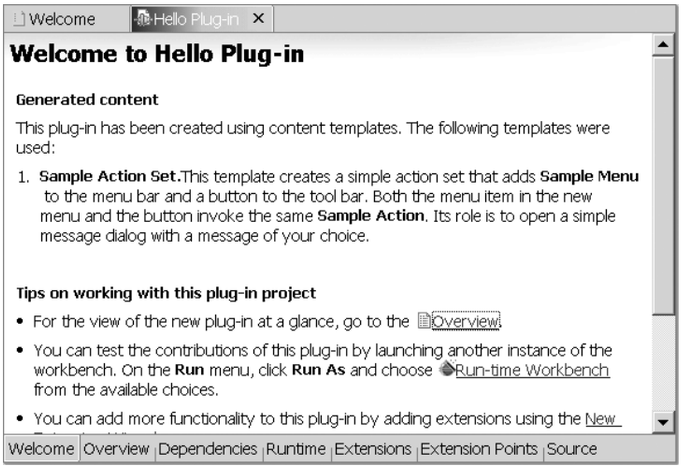

Figure 8.11
When you first open the plugin.xml file, you are greeted with this Welcome screen. It contains hints about how to work with the project, active links to different pages in the Plug-in Manifest Editor, and actions like starting the Run-time Workbench. You can turn off the page when you become more comfortable with the environment.

- *Extensions*—Lists all extension points used by this plug-in.
- *Extension Points*—Lists all extension points defined by this plug-in, along with a cross-reference of who is using those extension points.
- *Source*—A raw XML editor for the plugin.xml file. Often it is useful to make a change in one of the other pages of the editor and then switch to the Source page to see what effect the change had.

8.3.2 The Run-time Workbench

Once you have created a plug-in project, you could compile it, package up a JAR file, copy it to the plugins directory as you did in section 8.1.3, and restart Eclipse. But the PDE provides an easier way in the form of the *Run-time Workbench*, a temporary Eclipse installation created automatically for running and debugging plug-ins.

To run your new plug-in under the Run-time Workbench, select Run→Run As→Run-time Workbench (or use the Run toolbar button). If you haven't done this before, then you will get a notice that Eclipse is completing a new installation, and then a new Eclipse Workbench will appear. This is the Run-time Workbench; it includes all the plug-in projects you are working on in addition to the plug-ins that are part of the standard Eclipse installation. If you look carefully, you will notice a new menu named Sample Menu and a new button in the Eclipse toolbar (see figure 8.12).

Click the button, and Eclipse opens a dialog commemorating your second plug-in (see figure 8.13). We'll look at the Java code behind that button shortly.

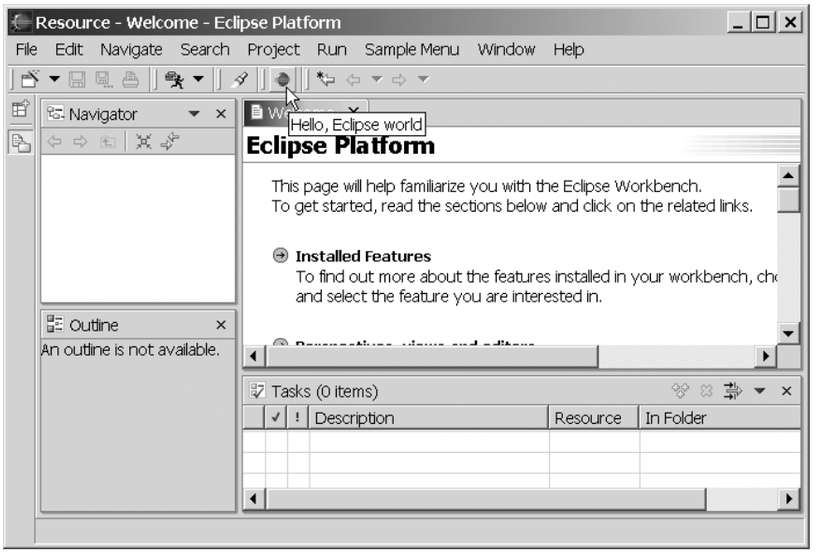

Figure 8.12 Starting the Run-time Workbench opens a new instance of Eclipse with all the plug-ins from your workspace installed. The "Hello, World" plug-in adds a Sample Menu and a toolbar button to the Workbench.

Debugging a plug-in is just as easy. Select Run→Debug As→Run-time Workbench. A second Eclipse Workbench starts up, containing your plug-ins. Debug this as you would any Java application (see chapter 3).

NOTE If you are using JDK 1.4 or higher, you can make small changes to your source code and rebuild, and the changes will be instantly available in the Run-time Workbench. This is called *hot-swapping* or *hot code replace*. Substantial changes (such as adding a new class) may cause a warning to be displayed. If you get this warning, simply close the Run-time Workbench and run it again.

Figure 8.13 This dialog appears when you click the new toolbar button of the "Hello, World" plug-in.

SIDEBAR Eclipse is *self-hosted*, which means it is used to develop itself. The concept of self-hosting got its start in compiler technology. The first version of a compiler is written in a simpler language, such as assembler, or perhaps using a competitor's product. But once the compiler is working, the developer rewrites it in the language being compiled, using the first version to build the second version, the second version to build the third, and so forth. Eclipse developers use Eclipse the same way, and created its plug-in development environment to support this process.

Compilers, being fairly complex programs in their own right, make excellent test cases for compilers. Likewise, by writing Eclipse with Eclipse, the developers can discover and correct any shortcomings in the handling of large projects and optimize the environment for extending the Eclipse Platform.

8.3.3 Plug-in class (AbstractUIPlugin)

The Hello, World Wizard created three files for you: a plug-in manifest (plugin.xml) and two source files (HelloPlugin.java and SampleAction.java). Because all Eclipse plug-ins and extensions follow this pattern (references in the XML file with Java classes to back them up), it's important to understand how it works. Open the Plug-in Manifest Editor and select the Overview page (figure 8.14).

XML

Switch to the Source page. You should see something like this:

```
<plugin
   id="org.eclipseguide.helloplugin"
   name="Hello Plug-in"
   version="1.0.0"
   provider-name="Eclipse in Action"
   class="org.eclipseguide.helloplugin.HelloPlugin">
...
</plugin>
```

Note that the link to the code is provided by the `class` attribute. When the plug-in is activated, this class will be instantiated and its constructor called.

Java

The class that backs up the plug-in definition in the manifest is `HelloPlugin`, contained in the source file HelloPlugin.java (see listing 8.1). In this section, we'll examine this code and explain how all the pieces fit together.

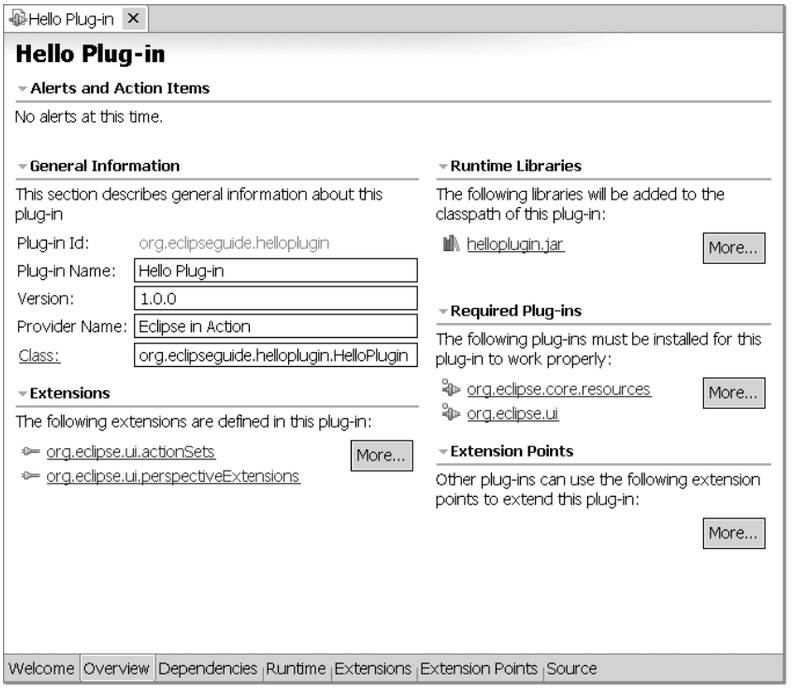

Figure 8.14 The Overview page of the Plug-in Manifest Editor is the central control panel for your plug-in. From here you can get a summary of the plug-in at a glance and access the other pages for more detail.

Listing 8.1 Java class for the "Hello, World" plug-in

```
package org.eclipseguide.helloplugin;                    ❶ Package name

import org.eclipse.ui.plugin.*;
import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;
import java.util.*;

/**
 * The main plugin class to be used in the desktop.
 */
public class HelloPlugin extends AbstractUIPlugin        ❷ Plug-in class
{
   //The shared instance.
   private static HelloPlugin plugin;                    ❸ Singleton instance
   //Resource bundle.
   private ResourceBundle resourceBundle;

   /**
```

```
 * The constructor.
 */
public HelloPlugin(IPluginDescriptor descriptor)
{
   super(descriptor);
   plugin = this;
   try
   {
      resourceBundle =
         ResourceBundle.getBundle(
            "org.eclipseguide.helloplugin.HelloPluginResources");
   }
   catch (MissingResourceException x)
   {
      resourceBundle = null;
   }
}

/**
 * Returns the shared instance.
 */
public static HelloPlugin getDefault()
{
   return plugin;
}

/**
 * Returns the workspace instance.
 */
public static IWorkspace getWorkspace()
{
   return ResourcesPlugin.getWorkspace();
}

/**
 * Returns the string from the plugin's resource bundle,
 * or 'key' if not found.
 */
public static String getResourceString(String key)
{
   ResourceBundle bundle =
      HelloPlugin.getDefault().getResourceBundle();
   try
   {
      return bundle.getString(key);
   }
   catch (MissingResourceException e)
   {
      return key;
   }
}

/**
```

❹ **Plug-in constructor**

❺ **Get a resource bundle**

❻ **Return Workspace handle**

❼ **Look up key in resource bundle**

```
     * Returns the plugin's resource bundle,
     */
    public ResourceBundle getResourceBundle()
    {
       return resourceBundle;
    }
}
```

❶ The package name should be the same as the plug-in name, which should be the same as the project name. Packages in the Eclipse Platform start with `org.eclipse`. Although the convention is not always followed, user interface packages generally have `ui` in their name, and non–user interface packages include `core`. If you see a package with `internal` in the name, it is not intended to be used outside the package itself. Internal packages and interfaces can, and often do, change between releases (and even builds of the same release), so stay clear of them.

❷ This is where the plug-in class is created. There are two types of plug-ins: those with user interfaces and those without. `AbstractUIPlugin` is the base class for all UI type plug-ins, and `Plugin` is the base class for the rest.

❸ A Singleton pattern is used to ensure there will be only one instance of the plug-in's class.

❹ The plug-in's constructor is passed an `IPluginDescriptor` object, which has methods such as `getLabel()` that return information from the plug-in registry. You can get a reference to this descriptor later by using the `getDescriptor()` method. Note that all Eclipse interfaces begin with the letter `I`.

❺ See the name of the bundle in the `getBundle()` call? Once created, the properties file for this bundle goes in your org.eclipseguide.helloplugin project and is named HelloPluginResources.properties. You can manage it by hand or by using the Externalize Strings Wizard (Source→Externalize Strings).

❻ The `IWorkspace` interface is the key to the Eclipse Platform's resource management. It has methods to add, delete, and move resources; most important, it has the `getRoot()` method to return the *workspace root* resource, the parent of all the projects in the workspace. This is a Singleton object (only one in the system).

❼ The wizard has created a standard Java resource bundle for you to look up natural language strings. For example, to get the translated string for a greeting, you could call the method `HelloPlugin.getResourceString("%greeting")`.

Actually, two bundles are at work. The first one is associated with the plug-in externally and may be referenced in the plug-in manifest, plugin.xml. Properties for the external bundle are kept in the file plugin.properties. Generally speak-

ing, you will never use that one in the plug-in code. The second bundle, referenced here, is internal to the plug-in and is kept in the plug-in's JAR file.

8.3.4 Actions, menus, and toolbars (IWorkbenchWindowActionDelegate)

An *action* is the non–user interface part of a command that can be run by a user, usually associated with a UI element like a toolbar button or menu. Actions are referenced in the plug-in manifest and defined as Java classes. Figure 8.15 shows what this extension looks like in the manifest editor's Extensions page.

You will probably find using the Extensions page more convenient and less error-prone than editing the XML in the Source page. However, because XML is a more compact representation than a series of screenshots of property pages, we will show the raw XML for most examples in this chapter and the next. Keep in mind, though, that there is a one-to-one correspondence between the two. Also, you can switch back and forth between the pages of the manifest editor at any time; a change in one is reflected in all the others.

XML

The first extension defined by the plug-in is an action set. An *action set* is a menu, submenu, or draggable group of toolbar buttons that appears in the user interface.

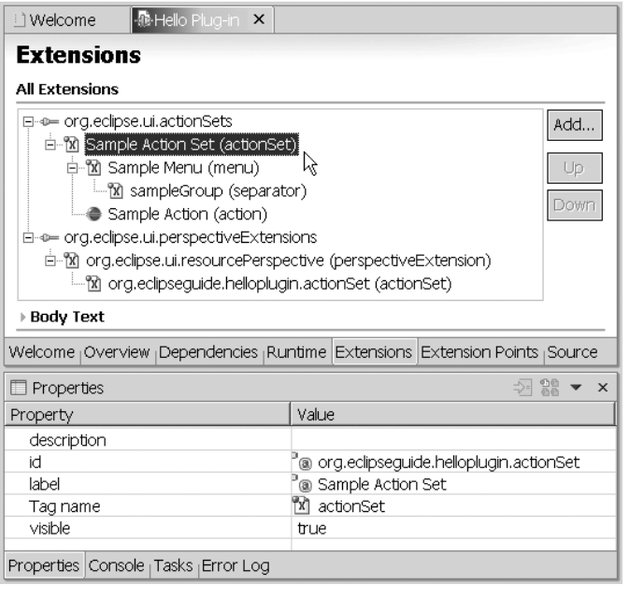

Figure 8.15
You can use the Extensions page of the Plug-in Manifest Editor to add new extensions to your plug-in. Properties and their values are viewed and modified through the Properties view. Required properties such as `id` and `label` are marked with an icon. If you prefer, you can edit the raw XML representation in the Source page.

Listing 8.2 shows the definition in the plug-in manifest (compare this to figure 8.15).

Listing 8.2 The actionSet extension

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet
         label="Sample Action Set"
         visible="true"
         id="org.eclipseguide.helloplugin.actionSet">
      <menu
            label="Sample &amp;Menu"
            id="sampleMenu">
         <separator
               name="sampleGroup">
         </separator>
      </menu>
      <action
          label="&amp;Sample Action"
          icon="icons/sample.gif"
          class="org.eclipseguide.helloplugin.actions.SampleAction"
          tooltip="Hello, Eclipse world"
          menubarPath="sampleMenu/sampleGroup"
          toolbarPath="sampleGroup"
          id="org.eclipseguide.helloplugin.actions.SampleAction">
      </action>
   </actionSet>
</extension>
```

❶ Fully qualified unique ID
❷ Menu ID need not be unique
❸ Placeholder for items/submenus
❹ Points to code
❺ Adds action to menu bar
❻ Adds action to toolbar

❶ Each extension, and indeed just about everything in the plug-in manifest, has a *fully qualified ID* that, by convention, starts with the plug-in ID. It doesn't matter what you call these IDs, as long as you pick unique names.

❷ Of course, there are exceptions, such as menus. They typically use short names like `group1` to achieve some level of consistency between menus. For example, a File menu might have a `group1` section and a Windows menu might also have a `group1` section.

❸ Menus can contain *groups* and *separators*. Separators are simply groups that are drawn with thin lines between them. All menus have a section named `additions`, which is the default place new items are added if you don't specify a location. This particular menu has two levels: `sampleMenu` (the parent menu) and `sampleGroup` (the child group).

❹ As with plug-ins, the `class` property points to the code.

5. The `menubarPath` property indicates the action is being added to a menu bar (in this case, the top-level bar of the Workspace). The paths look like directories, going from higher-level parent menus or groups to lower-level child ones.
6. The `toolbarPath` property indicates that this action is also being added to a toolbar. There is one toolbar called `Normal`, but the name is usually omitted from the path.

Java

Now let's move over to the Java side and dig into the code for the `SampleAction` class, shown in listing 8.3.

Listing 8.3 The SampleAction class

```
package org.eclipseguide.helloplugin.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

/**
 * Our sample action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class SampleAction implements IWorkbenchWindowActionDelegate
{
   private IWorkbenchWindow window;
   /**
    * The constructor.
    */
   public SampleAction()
   {
   }

   /**
    * The action has been activated. The argument of the
    * method represents the 'real' action sitting
    * in the workbench UI.
    * @see IWorkbenchWindowActionDelegate#run
    */
   public void run(IAction action)
   {
      MessageDialog.openInformation(
         window.getShell(),
```

1. JFace and UI imports
2. Code pointed to by manifest
3. Main Workbench window
4. Perform action
5. Open dialog

```
            "Hello Plug-in",
            "Hello, Eclipse world");
    }

    /**
     * Selection in the workbench has been changed. We
     * can change the state of the 'real' action here
     * if we want, but this can only happen after
     * the delegate has been created.
     * @see IWorkbenchWindowActionDelegate#selectionChanged
     */
    public void selectionChanged(
       IAction action,
       ISelection selection)
    {
    }

    /**
     * We can use this method to dispose of any system
     * resources we previously allocated.
     * @see IWorkbenchWindowActionDelegate#dispose
     */
    public void dispose()
    {
    }

    /**
     * We will cache window object in order to
     * be able to provide parent shell for the message dialog.
     * @see IWorkbenchWindowActionDelegate#init
     */
    public void init(IWorkbenchWindow window)
    {
       this.window = window;
    }
}
```

❻ **Can be used to free system resources**

❶ JFace is a high-level wrapper on top of the Standard Widget Toolkit (SWT) used for the Eclipse UI. JFace deals in concepts like dialogs and viewers, whereas SWT deals in windows, canvases, and buttons. (For more details on SWT and JFace, see appendixes D and E.)

❷ A *proxy* stands in for an object until the real object is available. In this case, there is a proxy for the toolbar button (not seen here) created by the Workbench based solely on the information in the plug-in manifest. Remember that the plug-in (including this code) isn't even loaded until after the button is clicked. When the user finally clicks the button, this *delegate* is created and passed the action to run. It works just like a relay race. The first runner is the proxy, and the baton he car-

ries is the action. The baton is passed to the second runner, the delegate, who finishes the race.

Because the "Hello, World" button is in the Workbench toolbar and the menu item is in the main Workbench menu, this code implements the Workbench window action delegate interface. There are similar interfaces for View, Editor, and Object action delegates, which are all based on `IActionDelegate`. `IActionDelegate` only has two methods—`run()` and `selectionChanged()`—which you will implement a little further down in this class.

❸ `IWorkbenchWindow` is an interface used for the top-level window of the Workbench. It contains a collection of `IWorkbenchPages` that in turn hold all the views, editors, and toolbars. Some common methods you'll use in `IWorkbenchWindow` include `close()` and `getWorkbench()`.

❹ The `run()` method is where the actual work gets done. The `IAction` interface has methods for getting and setting the user interface style of the button or menu it is associated with, and maintains a list of *listeners* that are called when any of its properties change.

❺ `MessageDialog` is one of a group of JFace utility classes that perform common operations. The static method `openInformation()`, as you might guess, opens an information dialog (as opposed to an error, warning, question, or other type of dialog). Its first argument is a *shell*, which is a low-level SWT window. (You'll find that many classes have a `getShell()` method, and you will use it often.) The `openInformation()` method's second argument is the title of the dialog that will be shown, and the final argument is the text that will be displayed on the main area of the dialog.

❻ Because SWT works more closely to the underlying window system than other APIs (notably Swing), it is sometimes necessary to free up system resources in `dispose()` methods that are explicitly called. Garbage collection cannot be relied on for this purpose because it is run at unpredictable times. This is one of the more controversial requirements of SWT, but it is not as painful as you might think.

8.3.5 Plug-ins and classpaths

One "gotcha" that continues to bite plug-in developers (new and old alike) is the way plug-in classpaths work. Plug-ins can only use classes exported by other plug-ins. For security reasons, plug-ins ignore the normal classpath settings at runtime, causing `ClassNotFound` exceptions even when the code compiled just fine.

Because of this restriction, if you want to use an external JAR file (one that is not in your workspace) inside a plug-in, you must bring it into your workspace.

Typically you do so by wrapping the JAR file in its own plug-in and making any other plug-ins that need the library depend on the new plug-in. The Eclipse Platform includes many examples, such as the `org.junit`, `org.apache.ant`, and `org.apache.xerces` plug-ins, which are simple wrappers around the JUnit, Ant, and Xerces libraries, respectively.

Wrapping a JAR file is one of the simplest plug-ins you can create, because no code is involved. The next example walks you through the necessary steps.

8.4 The log4j library plug-in example

As you recall from chapter 3, log4j is a free logging API created for the Apache Jakarta project. When you wanted to use it in a normal Java program from within Eclipse, you created a classpath variable for it and then referenced that variable inside the Java Build Path for the project. But let's say you need to use the library inside a plug-in, so you need to create a wrapper plug-in for it. To create the wrapper for log4j, start with a blank template from the New Plug-in Project Wizard:

1. Select File→New→Project to bring up the New Project Wizard (figure 8.3).
2. Select the Plug-in Project Wizard and click Next to open the New Plug-in Project Wizard.
3. Enter the name for the plug-in, **org.apache.log4j**, and click Next.
4. Leave the fully qualified ID as it is, making sure the option to Create a Java Project is selected, and click Next again.
5. Select the option to Create a Blank Plug-in Project (see figure 8.6). Click Finish to generate the plug-in.

Now, you need to customize the plug-in to contain the log4j library and include the proper export instructions so other plug-ins can use it. To do this, follow these steps:

1. In the new project directory, delete the src directory (right-click on it and select Delete), because there will be no source code in the project itself.
2. Copy the log4j JAR file (for example, log4j-1.2.8.jar) from the place you installed it in chapter 3 into the top level of the project directory. To do this, you can use File→Import→File System or, if you're using Windows, drag the file from your file explorer into the project.
3. Rename the JAR filename to remove the version number by right-clicking on it, selecting Refactor→Rename, and entering the new name, **log4j.jar**. The version number is specified in the plug-in manifest and is

appended to the plug-in directory name, so you don't also have to append it to the JAR filename.

4. Edit the project properties (right-click on the project and select Properties). Select Java Build Path, and then select the Libraries tab. Click Add Jars, navigate into the project, and select log4j.jar. Click OK.
5. While still in the project properties dialog, select the Order and Export tab. Put a check mark next to log4j.jar and click OK to save. This setting lets other plug-ins use this library at compile time.
6. Open the Plug-in Manifest Editor (double-click on plugin.xml) and switch to the Overview page. Set the Plug-in Name to **Apache Log4J**, change the Version to match the version number of the log4j package (for example, **1.2.8**), and fill in the Provider Name with **Eclipse in Action**.
7. Still in the manifest editor, switch to the Runtime page. Verify that log4j.jar is in the library list. Select it and turn on the option to Export the Entire Library. This setting lets other plug-ins use the library at runtime. You can also use this trick to make it look like classes from many other plug-ins come from a single plug-in.
8. Delete the reference to the src/ folder on the Runtime page under Library Content by right-clicking on it and selecting Delete. Again, because you are not building the plug-in from source code, you don't need a source folder.
9. Switch to the Source page of the manifest editor and admire your handiwork. When you are done, the XML in the Source page should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.apache.log4j"
   name="Apache Log4J"
   version="1.2.8"
   provider-name="Eclipse in Action">

   <runtime>
      <library name="log4j.jar">
         <export name="*"/>
      </library>
   </runtime>
</plugin>
```

10. Press Ctrl-S to save, and then close the Plug-in Manifest Editor.

8.4.1 Attaching source

Users of your plug-in will undoubtedly want to view log4j's source code at some point, perhaps while debugging or in order to understand how to use its classes. To make that functionality available, you have to place a zip file containing the source in the top-level directory of the plug-in (zip is used even on UNIX). In order for Eclipse to find the zip file automatically, it must have same name as the JAR file, but with src.zip appended to the end (for example, foosrc.zip goes with foo.jar). In normal plug-ins, the PDE makes this file for you from your own source code. But because you are not building the code for this library, you must make other arrangements:

1. Create a log4jsrc.zip file containing the source. The log4j distribution doesn't include a source zip file, but it does have a directory containing the source. Locate the top of the source tree in the distribution's src/java directory and put the org subdirectory and everything under it into the zip using the `jar` utility (`jar -cvf log4jsrc.zip`) or your favorite zip program, such as WinZip. When you're done, the zip file must have an internal structure like this:

```
org
+---apache
    +---log4j
        +---chainsaw
        +---config
        +---helpers
        +---etc...
```

2. Copy the new log4jsrc.zip file into your project (using File→Import→File System).
3. Associate the source zip to the log4j JAR file by right-clicking on log4j.jar and selecting Properties→Java Source Attachment (see figure 8.16). Click the Workspace button to locate the zip file. Doing so lets you view the source in your own plug-in projects.

8.4.2 Including the source zip in the plug-in package

When the time comes to make your plug-in available for other people to use, you need to package it in a zip file organized exactly as it should be organized under the plugins directory. Eclipse uses the properties in build.properties to tell it which files should be packaged and which ones should be ignored. Obviously, you want the source zip to be included, so follow these steps:

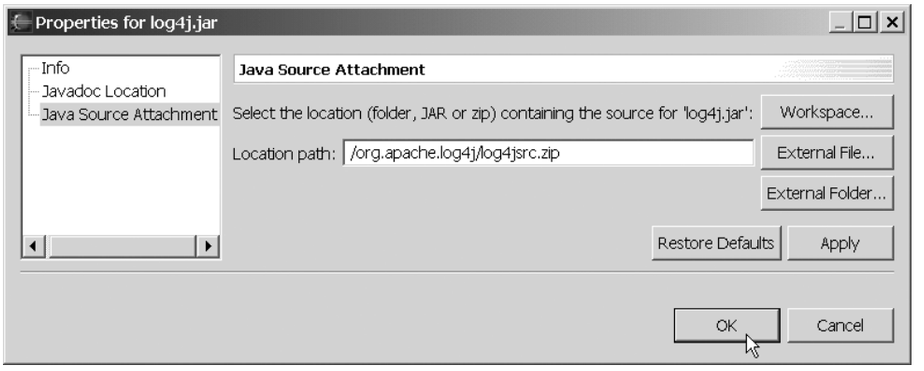

Figure 8.16 Set the Java Source Attachment property on a JAR file to make its source visible in Eclipse. The recommended method is to click the Workspace button to locate a path relative to the workspace, as shown here.

1. Open the Properties Editor on build.properties by double-clicking on it. Note that you can't edit build.properties and plugin.xml at the same time, because the manifest editor needs to write the properties file. So if you get an error that the file is in use or read only, close the manifest editor and try opening build.properties again.
2. Add the zip file to the `bin.includes` property. To do this, select the `bin.includes` property on the left to display the values for that property on the right. Click the Add button under Replacement Values and type **log4jsrc.zip**. Later, when it's time to deploy the plug-in, this property will be used to pick which files from your project are included. Internally, `bin.includes` is a variable name used in an Ant script that does the deployment.

TIP You can use Ant patterns to include many files at once. For example, use a wildcard like ***.jar** to include all files ending with *.jar*. The pattern ****** (for example, ****/*.gif**) matches any number of directory levels, and a trailing slash (for example, **lib/**) matches a whole subtree.

---

3. While you're editing the properties file, remove the `source.log4j.jar` property by right-clicking on it and selecting Delete. Because there is no source code in the project except the zip file you just imported, this property is unnecessary. Press Ctrl-S to save. The build.properties file should now look like figure 8.17.

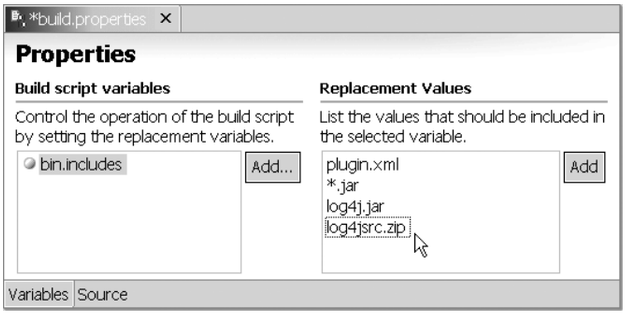

Figure 8.17 Add the zip file containing the source to the `bin.includes` property. Everything listed here will be included in the final plug-in package when the time comes to deploy it.

There you have it—a plug-in that wraps the log4j JAR file and that can be referenced from other plug-ins. You'll use this plug-in for the examples in chapter 9.

8.5 Deploying a plug-in

Once you've created a plug-in in your workspace, you can run and debug it using the Run-time Workbench. But how do you install the plug-in or give it to someone else so they can install it? This process is called *deployment*. You can create deployable zip files with Ant, but the PDE supplies an Export Wizard to make it even easier. To demonstrate this process, let's create the zip file for the log4j library plug-in you just built:

1. Select File→Export to start the Export Wizard, and then select Deployable Plug-ins and Fragments and click Next. The Export dialog shown in figure 8.18 opens.
2. Select the plug-in(s) to export and enter the filename of the zip file you want to create.
3. Click Finish to create the file.

Now you have a plug-in zip file that others can install.

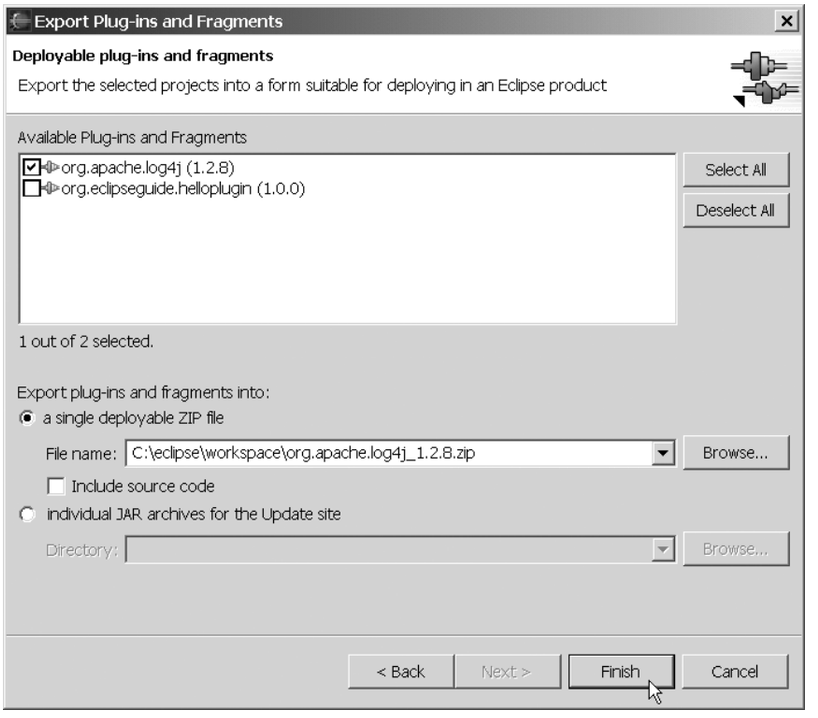

Figure 8.18 You can use the Deployable Plug-ins and Fragments Wizard to create zip files that others can install. Specify the plug-in and the name of the zip file to create and click Finish to create the file.

8.6 Summary

Every component of the Eclipse Workbench—every view, every editor, every menu—is defined in a plug-in. The Eclipse designers took great care to expose a fully functional public API for all plug-in writers to use. Because of this even playing field, high quality plug-ins you provide cannot be distinguished from plug-ins that were originally part of the Platform.

The convergence of an object-oriented polymorphic introspective language (Java), a universal data exchange format (XML), open-source tools (Ant, JUnit), design patterns, and agile programming techniques (such as refactoring) make Eclipse a unique and fun environment in which to program. Wizards and templates greatly lessen the learning curve, and the open source community built around Eclipse provides plenty of examples (and support) for the Eclipse programmer.