Eclipse

By Steve Holzner

Publisher: O'Reilly
Pub Date: April 2004
ISBN: 0-596-00641-1
Pages: 334
Slots: 1.0

O'Reilly's new guide to the technology, Eclipse, provides exactly what you're looking for: a fast-track approach to mastery of Eclipse. This insightful, hands-on book delivers clear and concise coverage, with no fluff, that gets down to business immediately. The book is tightly focused, covering all aspects of Eclipse: the menus, preferences, views, perspectives, editors, team and debugging techniques, and how they're used every day by thousands of developers. Development of practical skills is emphasized with dozens of examples presented throughout the book.

Eclipse

By Steve Holzner

Publisher: O'Reilly
Pub Date: April 2004
ISBN: 0-596-00641-1
Pages: 334
Slots: 1.0

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Preface

Welcome to Eclipse, today's premiere Java Integrated development environment (IDE). Eclipse is an extraordinary tool, and it fills a long-standing need among Java developers—no longer do you have to suffer through pages of errors scrolling off the screen while using command-line Java compilers. Now you've got an IDE that will handle the details for you, letting you get on with writing code. If you've never used Eclipse before, your productivity is about to take a giant jump.

We're going to push the Eclipse envelope in this book, working from the basics up through the advanced. This book has been designed to open up Eclipse and to be more accessible than any other. It's a programmer-to-programmer book, written to bring you up to speed in Eclipse without wasting time.

If you're a programmer, this book is written to give you exactly what you want to see—the good stuff, and only the good stuff. There's as much Eclipse crammed into this book as you need to master the topic, and mastering Eclipse is our goal.

What's Inside

From cover to cover, this book is pure Eclipse, covering hundreds of skills and techniques. We start from the most basic Java development and work up to creating your own plug-in editors for the Eclipse environment. Here are a few of the topics in this book:

- Using Eclipse to develop Java code
- Working with JAR files
- Setting launch configurations
- Selecting Java runtimes
- Creating Javadoc
- Refactoring
- Extracting Interfaces
- Viewing type hierarchies
- Customizing Eclipse
- Testing code with JUnit
- Debugging
- Setting breakpoint hit counts
- Using hot code replacement
- Sharing projects with CVS
- Comparing code with local history
- Using Ant to build Eclipse projects

Conventions Used in This Book

There are some conventions we'll use that you should know about. When we've added a new piece of code and are discussing it, it'll appear in bold face, and when there's more code to come, you'll see three dots. Here's what that looks like:

```
Listener listener = new Listener( ) {

    public void handleEvent(Event event) {

        ToolItem item = (ToolItem)event.widget;

        String string = item.getText( );

            .

            .

            .

    }

};
```

We'll also use the standard convention for selecting menu items in this book; for example, to create a new project in Eclipse, you use the File⟶ New⟶ Project menu item.

The following typographical conventions are used in this book:

Plain text

Indicates menu titles, menu options, menu buttons, and keyboard accelerators.

Italic

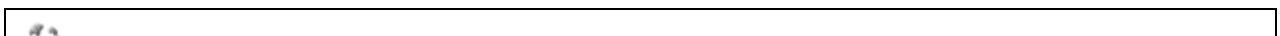
Indicates new terms, example URLs, example email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

Constant width

Indicates commands, options, switches, variables, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, and XML tags.

Constant width italic

Indicates text that should be replaced with user-supplied values.



What You'll Need

All the software you'll need in this book can be downloaded from the Internet for free. You'll need Eclipse—this book was written using Eclipse 2.1.1—and we'll discuss where to get Eclipse in Chapter 1. Other software packages that we'll be downloading throughout the book include the Tomcat web server and various CVS servers (which will allow you to share Eclipse projects with others).

Eclipse is built to be extendible, and hundreds of Eclipse plug-ins are available for free downloading. Plug-ins let you add functionality to Eclipse—new built-in editors, code generators, software launchers, and more—and we'll take a look at a number of the most popular plug-ins in this book. And in Chapter 11 and Chapter 12, we'll develop our own Eclipse plug-ins.

Using Code Examples

All the code in this book is available for download from http://www.oreilly.com/catalog/eclipse. See the file *readme.txt* in the download for installation instructions.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Eclipse, by Steve Holzner. Copyright 2004 O'Reilly Media, Inc., 0-596-00641-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:
O'Reilly & Associates, Inc.1005 Gravenstein Highway NorthSebastopol, CA 95472(800) 998-9938 (in the United States or Canada)(707) 829-0515 (international or local)(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:
http://www.oreilly.com/catalog/eclipse

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

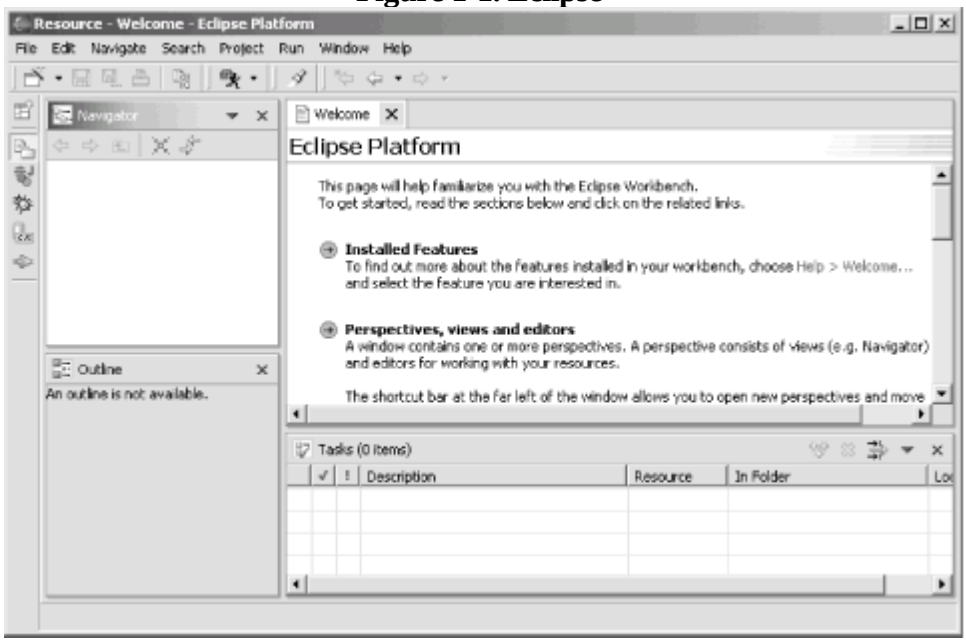
For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:
http://www.oreilly.com

Chapter 1. Essential Eclipse

If you're reading this book, you're most likely a Java programmer, and you know how finicky Java can be at times. Missed import statements, forgotten variable declarations, omitted semicolons, garbled syntax, typos—all these problems will cause the Java command-line compiler, javac, to cough in your face and display pages of annoying error messages. The error messages tell you that javac knows what the error is, so why doesn't it just fix the problem and let you get on developing?

Because javac can't fix the problem; it isn't an editor. That makes long streams of errors scrolling off the page an all-too-common experience for Java developers, and leaves them with the feeling that Java is too prickly about what can go wrong. To change all that, you can use an integrated development environment (IDE), which will not only catch errors before you try to compile, but also suggest solutions. Java is badly in need of a good IDE, and a number of candidates are available, but the premiere Java IDE these days is the one this book is all about: Eclipse. You can see Eclipse in action in Figure 1-1.

Figure 1-1. Eclipse



1.1 Eclipse and Java

Although Eclipse can act as an IDE for many different languages—IDEs are available from C/C++ to Cobol—its great popularity is as a Java IDE, and it comes with Java support built-in. Eclipse refers to itself as a universal tool platform, capable of handling IDEs for many different languages, but the Java IDE that comes with Eclipse is going to be our main focus, as it is for the great majority of Eclipse users.

The whole Eclipse magic is that it will take the rough edges off Java development in the way you've always imagined. The errors that would cause javac to stumble are usually handled before you even try to compile, and if there is an issue, Eclipse will suggest solutions. All you have to do is point and click—no need for serious head-scratching. If you're like most Java developers, you're going to find yourself thinking, This is great!

1.1.1 Some Background

Eclipse is free for the downloading, like a number of other Java IDEs, but Eclipse has a serious advantage behind it: the power of IBM, which reportedly spent $40 million in the development of the IDE. The first version, Version 1.0, appeared in November 2001 and gradually became popular (although—as with any developer tool—there was a great deal of discussion of its faults).

In time, Eclipse has changed and improved, and the current version, 2.1.1, is getting much praise. In fact, it's become so popular that when Version 2.1 first appeared, the servers at http://www.eclipse.org were so busy that it was almost impossible to download a copy for the first few days.

Eclipse was created by IBM in a massive effort that has left Java programmers the winners. It's now an open source project, still largely under IBM's development, but part of a software consortium named eclipse.org. You can see the consortium's page, http://www.eclipse.org, in Figure 1-2.

Figure 1-2. The Eclipse consortium's web page

The Eclipse consortium originally consisted of IBM's subsidiary, Object Technologies International (OTI)—who developed Eclipse in the first place—along with Borland, IBM, MERANT, QNX Software Systems, Rational Software3, Red Hat, SuSE, TogetherSoft3, and Webgain2 in November 2001. Since then, the consortium has grown to more than 45 members, including Sybase, Hitachi, Oracle, Hewlett-Packard, Intel, and others.

1.2 Getting Eclipse

How do you get and install Eclipse? Eclipse is free for the downloading—all you have to do is navigate to http://www.eclipse.org/downloads and select one of the download mirrors available on that page. When you do, you'll be presented with a list of the available downloads, which are of these types:

Release builds

The Eclipse team releases these versions for general use. Usually when you download Eclipse, you'll use one of the release versions. These builds have been thoroughly tested, and the chance of coming across serious bugs is minimal. This is a version of Eclipse comparable to the version that other companies would sell—if Eclipse were for sale.

Stable builds

These are comparable to beta versions. A stable build is a step along the way toward a release version. The Eclipse team considers this build to be relatively stable, but there may be problems. This is where you'll find the new features that are upcoming in Eclipse.

Integration builds

These builds are made up of components that have been fairly well tested, but their operation with other components may still have some issues. If things work out OK and the integration build proves itself, it may be elevated to a stable version.

Nightly builds

These are the most experimental of all publicly available Eclipse builds. They're created nightly by the Eclipse team, and there's really no guarantee that things will work well. Some experience with these builds indicates that they can actually have substantial problems.

Normally, you'll use the most recent release version of Eclipse. To get Eclipse, select the most recent release download for your operating system and click the appropriate link to download it.

Want to learn more about the current and upcoming versions of Eclipse? See http://www.eclipse.org/eclipse/development/main.html.

Installing Eclipse is not difficult—all you've got to do is unzip or untar it, depending on your operating system. Since you've downloaded the version of Eclipse targeted to your operating system, you'll find the executable file ready to run as soon as you uncompress Eclipse.

Windows users will be pleased to learn that Eclipse doesn't use the Windows registry, so

1.3 Understanding Eclipse

So what is Eclipse itself? Most people think of Eclipse as a Java IDE, and when you download Eclipse, you get the Java IDE (this is the Java Development Toolkit, the JDT) and the Plug-in Development Environment (the PDE) with it. If you only want to develop Java, it's easy to think of Eclipse as a Java IDE because that's the main tool you'll be using.

Eclipse itself, however, is a universal tool platform. The JDT is really an addition to Eclipse—it's a plug-in, in fact. Eclipse itself is really the Eclipse platform, which provides support for tools beyond just the Java set you get on download. These tools are implemented as plug-ins, so the platform itself only needs to be a relatively small software package.

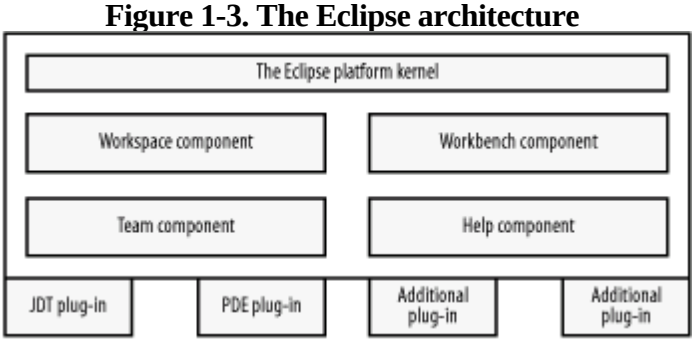
The platform provides the support the plug-ins need to run; if you want to develop Java, you use the JDT plug-in that comes with Eclipse; if you want to develop in other languages, you'll need to get other plug-ins, such as the CDT, which lets you develop C/C++ code. Installing a plug-in is easy, as we're going to see—all you have to do is drop it into the Eclipse plugins directory and restart Eclipse. Eclipse does some checking on each plug-in when it starts, but the plug-ins are not loaded until they're needed in order to save processing time and memory space.

It's also important to realize that although Eclipse is written in Java, it's intended to be language-neutral. To develop in any programming language, all you need is the corresponding plug-in. In fact, Eclipse is also intended to be spoken-language neutral, too—you can easily change the language that Eclipse uses. To change languages, you can use the same plug-in mechanism that supports plug-ins, except that languages are supported with what are called plug-in fragments. OTI has a language pack available that supports a number of languages—Japanese, Korean, German, French, Italian, Portuguese, Spanish, even traditional and simplified Chinese.

1.3.1 The Eclipse Platform

The Eclipse platform is made up of several components: the platform kernel, the workbench, the workspace, the team component, and the help component. You can see an overview of the platform in Figure 1-3.

Figure 1-3. The Eclipse architecture



1.3.2 The Platform Kernel

The kernel's task is to get everything started and to load needed plug-ins. When you start Eclipse, this is the

1.4 Views and Perspectives

When you're working with the workbench, you'll see a number of different internal windows, called views, and the idea is that they give you different "views" into your projects. For example, one view may give you an overview of the Java classes in your project, while another may let you navigate between projects. For example, back in Figure 1-1, you can see the Navigator view at the upper left in Eclipse—this is the view that will display all your projects and let you move from one to another.

Because screen space is always at a premium in GUIs, views are often stacked, one on top of another, and you select the one you want to see using tabs that appear on the edge of the stacked views.

If you ever want to reopen a view you've closed by mistake, select Window⟶ Show View, and select the view you want from the menu that appears.

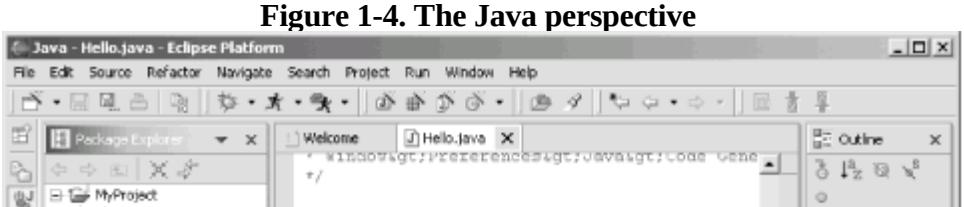
The editor is one special type of window that usually appears in the center of the workbench. When you open up documents, code, or resources, they'll appear in an editor. Eclipse automatically selects the correct type of editor for the item you're opening: the Java editor for a Java source code file, a GUI you're developing using a plug-in with the editor supplied by the plug-in, and so on. You can even open Microsoft Word documents in the Editor (Eclipse displays an MS Word window in the editor space using Windows Object Linking and Embedding, OLE). In Figure 1-1, the space normally reserved for editors is showing the Eclipse Welcome text.

The editor window is where you do most of the work when developing your code; for example, it's where you enter and edit your code. As we're going to see, the JDT has an editor that is lavish with built-in details, such as syntax checking, code highlighting, and much more. You might have several editors open at once, in which case they'll be stacked with tabs showing at the top of the stack, and you can pick out the one you want by clicking the corresponding tab (or with the Window⟶ Switch to Editor menu item, which displays a list of editors you can switch to). You can close an editor simply by clicking the X in its associated tab (or with the Window⟶ Hide Editors menu item, which toggles to Window⟶ Show Editors after hiding an editor). To sum up: views give you overviews of your projects, and editors let you develop code and resources.

There's one more concept to master here as well: perspectives. You don't normally decide what views and editors to display yourself; instead, they're organized into groups called perspectives (although it is easy to customize perspectives yourself). For example, when you create a Java application, you'll use the Java perspective; when you want to debug a Java program, you'll use the Debug perspective.

Perspectives have a predefined set of views and editors built-in; when you select a perspective, that set of views and editors appears automatically. For example, we'll take a look at the Java perspective here: to select a perspective, you use the Window⟶ Open Perspective menu item, which displays a submenu of the installed perspectives. In this case, we'll choose the Window⟶ Open Perspective⟶ Java menu item to open the Java perspective, shown in Figure 1-4.

Figure 1-4. The Java perspective



1.5 Working with Eclipse

You use the Eclipse Java Development Tools (JDT), a series of six seamlessly integrated plug-ins, for Java development in Eclipse. Even if you've written Java for years, you're about to have a whole new experience, one that makes Java development so smooth that when you understand how to use the JDT, you'll wonder what took people so long to make this a reality.

Eclipse is all about code development, and the only way to really understand what's going on is by creating code, so we're going to start by using the JDT to create and run the amazingly useful application you see in Example 1-1. This Java application just displays the message "No worries." on the console.

Example 1-1. The Ch01_01.java example

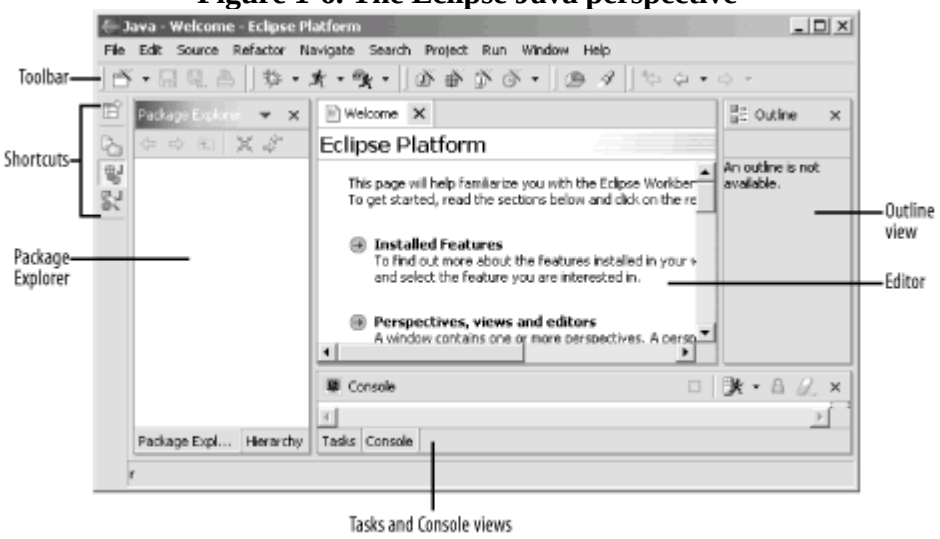
```
public class Ch01_01

{

    public static void main(String[] args)

    {

        System.out.println("No worries.");

    }

}
```

How can you create this application using Eclipse, and how is Eclipse going to make your job easier? The answers are coming up right now. In Eclipse, all Java code must be inside a Java project, so the first step is to create a Java project.

1.5.1 Creating a Java Project

To invoke the Java perspective, and enter the code for our first example, Ch01_01.java, start Eclipse and select the Window⟶ Open Perspective⟶ Java menu item to open the Java perspective using the JDT, as you see in Figure 1-6. This is the perspective you'll use over and over as you start relying on Eclipse for Java development.

Figure 1-6. The Eclipse Java perspective



It's worth getting to know the Java perspective before we start using it. At the top are the standard menu bars and

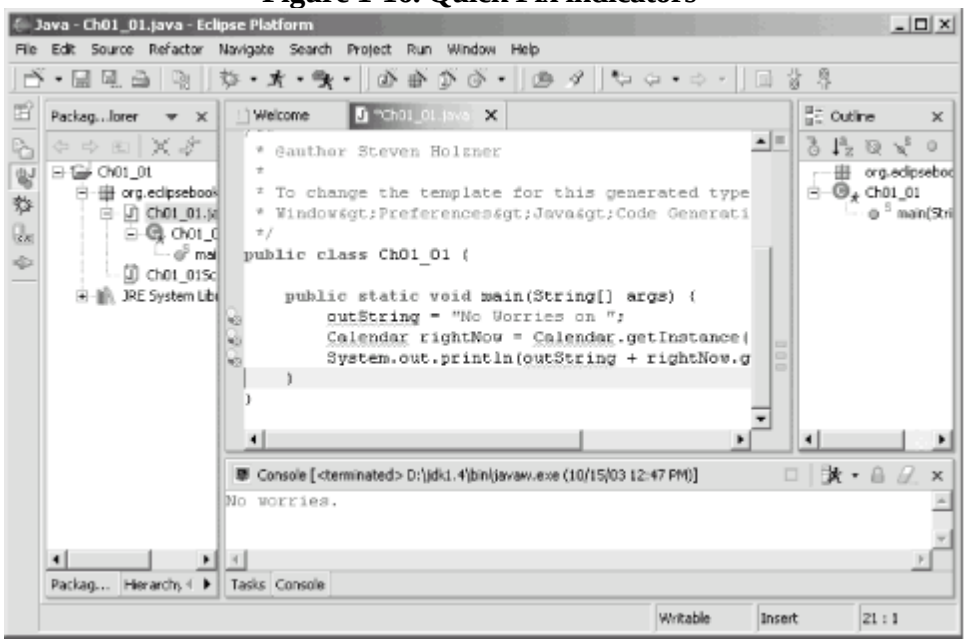
1.6 Using Quick Fix

Quick Fix lets the JDT suggest ways of fixing simple errors, and that's one of the things that Java should have had a long, long time ago. For example, say you change the Ch01_01 code to display the "No worries." message along with today's date:

```
public class Ch01_01 {

    public static void main(String[] args) {

        outString = "No Worries on ";

        Calendar rightNow = Calendar.getInstance( );

        System.out.println(outString + rightNow.getTime( ));

    }

}
```

You can probably spot a few errors here. The variable outString is not declared, which makes the first and last lines of code in main invalid, and the Calendar class has not been imported, making the middle line of code invalid. If you were using javac, you'd have to quit editing and run javac to catch those errors. But the second you enter these lines into Eclipse, they'll be flagged as errors with wavy red underlines, as you see in Figure 1-16.

Figure 1-16. Quick Fix indicators



Eclipse doesn't let you down by just showing you the errors—it also suggests solutions. Note the yellow light bulb and red X icons in the bar to the left of the code editor, which is called the marker bar. These icons tell you that Quick Fix is available for all the errors. Note also the hollow red rectangles in the bar on the right of the code editor view, called the overview ruler. These hollow red rectangles indicate statements that Quick Fix can fix—solid red rectangles flag compiler errors—and you can use these icons to navigate to problems to fix.

If you let the mouse cursor hover over the first light bulb icon in the marker bar, you'll see a tool tip appear with the description of the error ("outString cannot be resolved"), as in Figure 1-17.

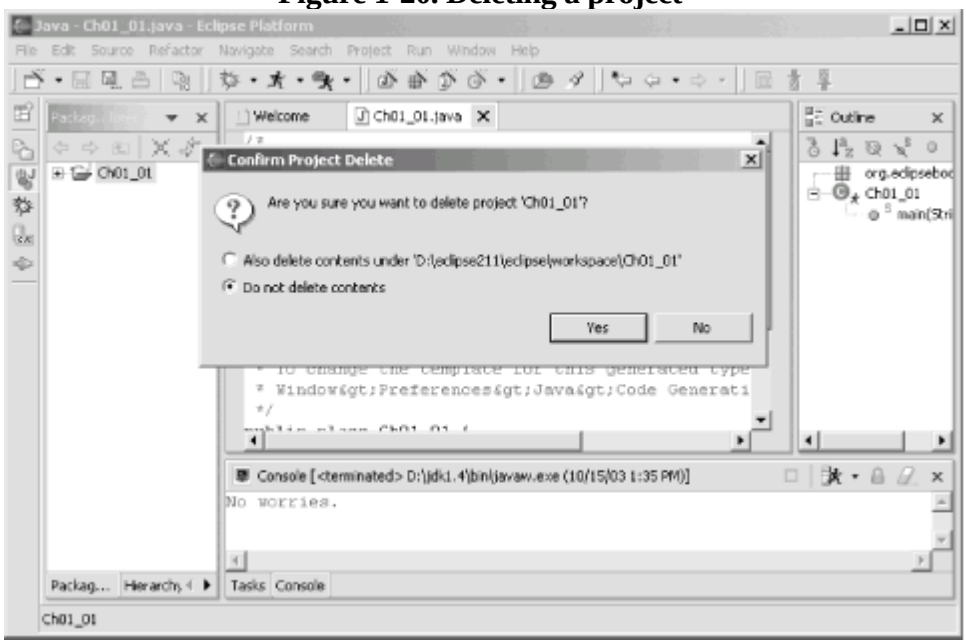
Figure 1-17. Using Quick Fix

1.7 A Word About Project Management

As you create more projects like Ch01_01, you'll find Eclipse getting more and more crowded, since all your projects are displayed in the Java perspective's Package Explorer view, as well as the Navigator view (recall that the Navigator view is there to let you navigate between projects). If you have 30 projects, there will be 30 entries there. There are various ways to deal with this clutter (such as creating working sets, as we'll see in Chapter 2), but we'll take a look at the simplest one here.

To remove a project from the Package Explorer and Navigator views, you can simply delete it. This does not necessarily delete the actual files used for the project, and, whenever you want, you can add the project back to these views. For example, to remove the Ch01_01 project, just right-click its icon and select the Delete item. Eclipse will display the Confirm Project Delete dialog box, as you see in Figure 1-20.

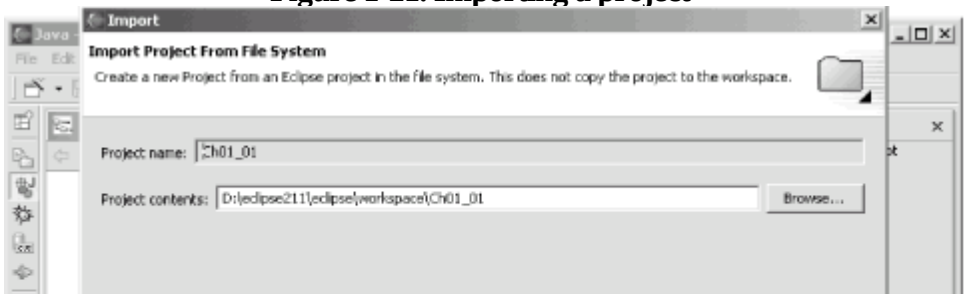
Figure 1-20. Deleting a project



In this case, make sure that the "Do not delete contents" radio button is selected and click Yes to remove the project from Eclipse. The project will disappear from the Package Explorer and Navigator views. Clicking the other radio button will make Eclipse delete all the files and their contents in the project, so don't do that if you want to use the project again later.

When you want to work with the project again, you just import it. To do that, right-click the Package Explorer or Navigator and select the Import context menu item, or select the File → Import menu item. This opens the Import dialog; select the "Existing Project into Workspace" item and click Next. In the next pane, click the Browse button, select the Ch01_01 folder, and click OK, giving you results like those shown in Figure 1-21.

Figure 1-21. Importing a project



Chapter 2. Java Development

This chapter is where we get down to the business of developing Java using Eclipse. We're going to take a look at using Eclipse for Java development as well as project management, going from the basics to the fairly serious. Nearly everything in this chapter is essential knowledge for the Java developer using Eclipse, so let's jump in.

2.1 Developing Java Code

If there's anything that takes more time than it seems to be worth in Java, it's creating code from scratch. While the logic inside a method, interface, or class is unique, the modifiers of a method, the imports for a class, and the syntax involved with new packages is the same over and over again. This often results in a lot of repetitive typing, wasted time, and in many cases, annoying little typo-related bugs. Eclipse can help with all this and more.

2.1.1 Creating New Methods

Eclipse—through code assist—makes it easy to create new methods. As an example, we're going to create and call a new method named printer, which displays the message "No worries.", as you can see in Example 2-1.

Example 2-1. The Ch02_01.java example

```
public class Ch02_01

{

    public static void main(String[] args)

    {

        printer( );

    }


    private static void printer( )

    {

        System.out.println("No worries.");

    }

}
```

How do you create new methods? Start Eclipse now and create a new project named Ch02_01. Then create a new Java class named Ch02_01, making it part of the org.eclipsebook.ch02 package. Leave the checkbox for the creation of a stub for the main method checked when you create this new class. This gives you the code:

```
public class Ch02_01 {


        public static void main(String[] args) {

        }

}
```

You could simply type in the printer method, of course, but Eclipse can also be of assistance here. Move the cursor below the body of the main method and type private to make this new method a private method, and then type Ctrl+Space to open code assist, as you see in Figure 2-1.

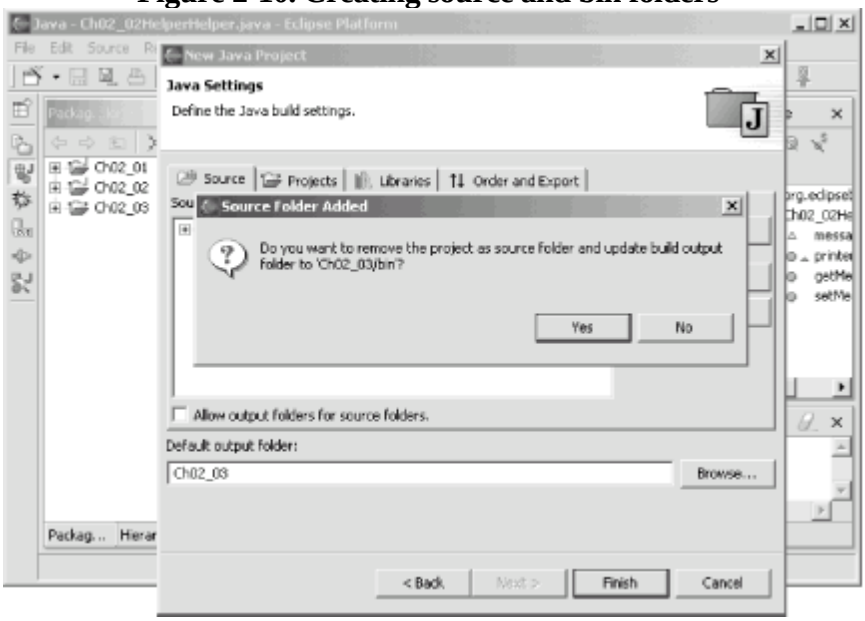
Figure 2-1. Creating a private method

2.2 Building and Running Code

How do you create the Java .class files that are the end result of developing a project? You use the items in the Project menu. The main menu item here is the Project⟶ Build Project menu item. This item will compile the source code files in your project and leave the resulting *.class* files in the same folder as the source code files by default. For example, if you are working with the Ch02_03 project and select Project⟶ Build Project, the *.class* files for this project will appear in the directory workspace/Ch02_03/org/eclipsebook/Ch02 (recall that the classes in this project are in the org.eclipsebook.Ch02 package, which the directory structure reflects). Once created, these *.class* files are ready for use and distribution.

It often makes sense to store all your project's source code in a folder named *src* and the binary output in a folder named *bin*. If you want to set things up this way when you create a new project, open the New Java Project dialog as usual, and, in the second pane, click the Source tab followed by the Add Folder button. Doing so opens the Source Folder Selection dialog; click the Create New Folder button and give the new folder the name *src*. Then click OK twice. Eclipse will recognize that you're creating a source code folder and automatically ask if you want to create a bin folder for the binary output, as you see in Figure 2-10.

Figure 2-10. Creating source and bin folders



Configuring a project this way automatically stores your source code in the folder named *src* and the binary output in a folder named *bin* (*bin* will not appear in the Package Explorer because it doesn't contain any source code).

2.2.1 Using JAR and .class Files

Say that you're writing a Java servlet (for more details on servlets, see Chapter 9), shown in Example 2-4.

Example 2-4. The Ch02_04.java example

```
package org.eclipse.ch02;


import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;
```

2.3 Creating Javadoc

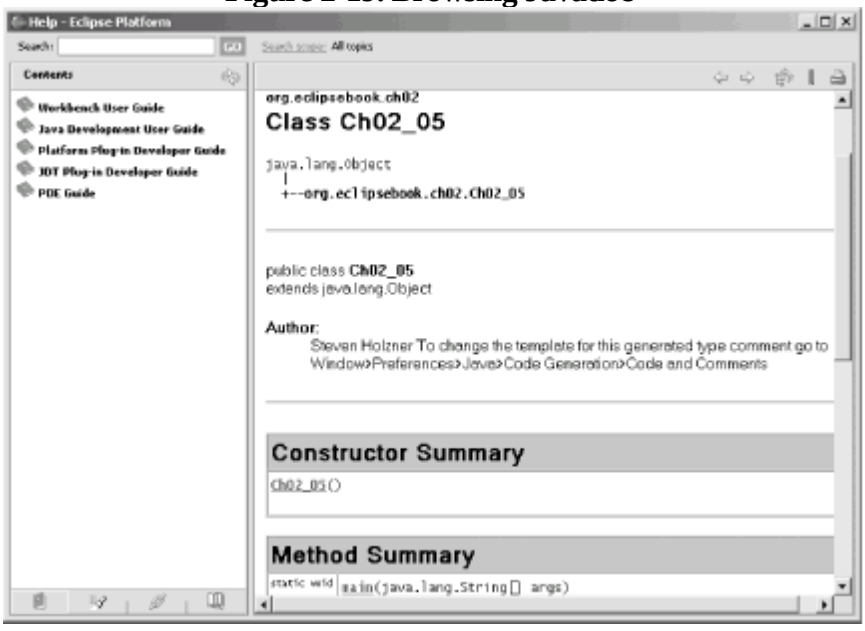
Eclipse also makes it easy to develop Javadoc documentation, the standard Java documentation that accompanies Java programs. You'll notice that in the code it generates, Eclipse inserts some text for Javadoc, as you see in *Ch02_05.java*:

```
package org.eclipsebook.ch02;


/**
 * @author Steven Holzner
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
.
.
.
```

If you want to enter your own Javadoc, code assist helps you here, too; for example, if you enter @param and invoke code assist with Ctrl+Space, code assist will list the parameters a method takes. Typing @exception and using code assist will list the exceptions a method throws, and so on. Typing @ in a comment and pausing will make code assist display the Javadoc possibilities, like @author, @deprecated, and so on.

To generate Javadoc from your code, select the Project⟶ Generate Javadoc item, opening the Generate Javadoc dialog, which lets you select the project for which you want to create Javadocs. To browse a project's Javadocs, select the Navigate⟶ Open External Javadoc menu item. For example, you can see the generated Javadoc for the Ch02_05 project in Figure 2-19.

Figure 2-19. Browsing Javadoc



2.4 Refactoring

One of the major advantages of using a good Java IDE like Eclipse is that it can let you rename and move Java elements around, and it will update all references to those items throughout your code automatically.

2.4.1 Renaming Elements

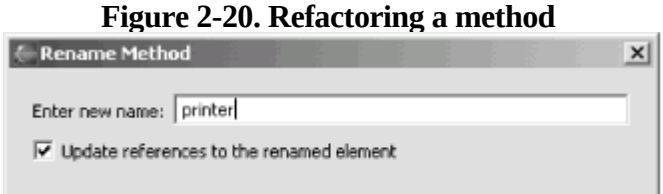
For example, take a look at the code in Example 2-6. Here, we've used code assist to create a new method to display a simple message, but we forgot to change the default name for the method that code assist supplied.

Example 2-6. The Ch02_06.java example

```
package org.eclipse.ch02;


/**
 * @author Steven Holzner
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class Ch0206 {


        public static void main(String[] args) {
                name( );
        }


        public static void name( ) {
                System.out.println("No worries.");
        }
}
```

This default name for the new method, name, is called in the main method, and it could be called from other locations in your code as well. How can you change the name of this method and automatically update all calls to it? Select name in the editor and then select the Refactor⟶ Rename menu item, opening the Rename Method dialog you see in Figure 2-20.

Figure 2-20. Refactoring a method



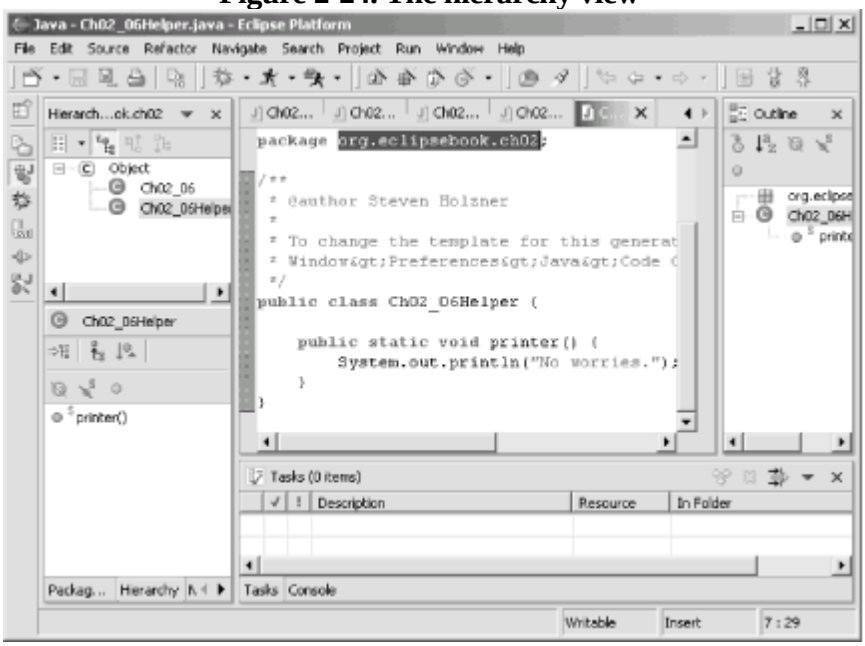
2.5 Some Essential Skills

There are some additional skills that are good to know about. For example, if you highlight an item in the JDT editor, right-click it, and select Open Declaration, the declaration of that item will open. This is great for tracking down where and how methods and fields were created. Several of those are detailed in this section and all are worth adding to your Eclipse toolbox.

2.5.1 Viewing Type Hierarchies

Another menu item in the JDT editor's context menu is the Open Type Hierarchy; when you select an item in the editor and select this menu item, that item's type hierarchy appears in the Java perspective's hierarchy view, as you see at left in Figure 2-24.

Figure 2-24. The hierarchy view



This view acts like an object browser. It lets you explore a type's Java complete hierarchy, and double-clicking an item in this view opens its definition in the editor. That's useful if, for example, you want to see all the members of the System.out class—just highlight System.out in your code and open its hierarchy. You can also open this view by selecting an item in the editor and selecting the Navigate⟶ Open Type Hierarchy item.

The hierarchy view is not dissimilar from the outline view, which you see at right in Figure 2-24. However, the outline view is designed to show an automatic hierarchy of your code rather than the hierarchy of items you specifically select. As you work in the JDT editor, the outline view is updated automatically to show the hierarchy of the current type you're working with.

2.5.2 Browsing Java Code

There's even another entire perspective dedicated to letting you browse through projects in a Java-oriented way: the Java Browsing perspective. To open this perspective, select Window⟶ Open Perspective⟶ Java Browsing; you can see the results in Figure 2-25.

Figure 2-25. The Java browsing perspective

2.6 Customizing the Development Environment

Our last topic in this chapter is all about customizing your development environment. Eclipse is easy to customize, starting from the most basic—you can move any view, editor, or toolbar around simply by dragging it.

If you don't like to work in an environment where things can move around with mouse movements by mistake, you can lock the toolbars with the Window ⟶ Lock the Toolbars menu item. And if a perspective gets all scrambled by inadvertent mouse movements, use Window ⟶ Reset Perspective to restore things.

You can also customize how Eclipse will generate code for you. For example, the default code generation style doesn't place opening curly braces on their own lines:

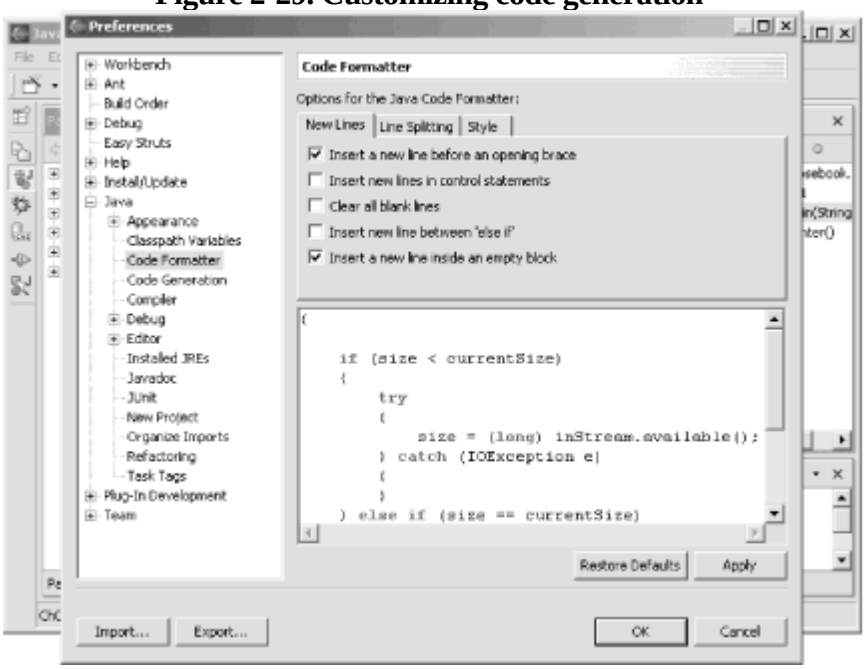
```
public void printer( ) {

        System.out.println("No worries.");

}
```

However, your programming style might be more like this, where each curly brace does get its own line:

```
public void printer( )

{

        System.out.println("No worries.");

}
```

You can customize this with the Windows ⟶ Preferences item, opening the Preferences dialog you see in Figure 2-29. Select the Java ⟶ Code Formatter item, which lets you specify options for code generation. Here, select the "Insert a new line before an opening brace" item, as you see in the figure; the sample code below will change to match.

Figure 2-29. Customizing code generation



Chapter 3. Testing and Debugging

Testing and debugging are a way of life in Java development. Eclipse comes with built-in facilities for testing your code using its JUnit framework and some truly exceptional debugging capabilities.

3.1 Testing with JUnit

JUnit is an open source testing framework that comes with Eclipse. You can create JUnit-based classes in the same project as other classes, and use this JUnit code to test the other classes in your project. Using JUnit in this way, you can construct a set of standard tests for everyone working on an application, and if they change the application's code, all they'll need is a few clicks to verify that the application still passes the standard set of tests.

JUnit is designed to test your code, and it's made up of assertion methods that can test various conditions. Here they are:

assertEquals(a, b)

Tests if a is equal to b (a and b are either primitive values or must have an equals method for comparison purposes)

assertFalse(a)

Tests if a is false, where a is a Boolean value

assertNotNull(a)

Tests if a is not null, where a is either an object or null

assertNotSame(a, b)

Tests if a and b both do not refer to the identical object

assertNull(a)

Tests if a is null, where a is either an object or null

assertSame(a, b)

Tests if a and b both refer to the identical object

assertTrue(a)

Tests if a is true, where a is a Boolean value

You construct JUnit tests using these methods; when you run a JUnit application, it opens its own view to give you an immediate indication of which tests have passed and which have failed.

3.1.1 Creating a Test Application

We'll use JUnit in an example here to show how it works by creating an application that we can test, Ch03_01. This

3.2 Debugging

Eclipse's debugging capabilities are impressive, even for a fully featured IDE. To get started, we'll take a look at an example with a logic error that we can then track down. After the example, we'll look at more advanced debugging topics, like setting debug launch configurations, using hot code replacement, suspending a running program that isn't terminating (such as when you have an infinite loop), and more. Being able to interactively debug your code is something you should expect to find in a good IDE—and it's a brilliant improvement over trying to debug your code using only the tools that come with Java—but Eclipse has gone above and beyond the call.

3.2.1 A Buggy Program

Our buggy program appears in Example 3-3. The debugger in Eclipse is especially good at working with stack frames, which hold the local variable set for method calls. To get a handle on debugging, we're going to create an example that creates several layers of stack frames: a factorial example (e.g., the factorial of 6, denoted 6!, is equal to 6 * 5 * 4 * 3 * 2 * 1 = 720), which calls itself recursively in order to calculate factorials.

Example 3-3. The Ch03_02.java example

```
package org.eclipsebook.ch03;


public class Ch03_02 {


        public static void main(String[] args) {

                System.out.println(factorial(6));

        }


        public static int factorial(int value) {

                if(value == 0){

                        return value;

                }

                else {

                        return value * factorial(value - 1);

                }

        }

}
```

You calculate factorials like this: n! = n * (n - 1) * (n - 2) ... * 1. In our example, to calculate factorial(n), the factorial method multiplies n by factorial(n - 1), calling itself to determine the factorial of (n-1). To determine factorial(n - 1), the factorial method multiplies (n-1) by factorial(n - 2), a process that continues all the way down until the factorial has been fully calculated. Here's the code we're using to do that:

```
public static int factorial(int value) {

        if(value == 0){
```

Chapter 4. Working in Teams

We've been programming solo with Eclipse up to this point, but it's also designed to be used in team environments, and it supports Concurrent Versions System (CVS) to make teamwork go smoothly. Eclipse can be used by any Java developer, and developers often work in teams. This chapter is all about teamwork using Eclipse and CVS.

4.1 How Source Control Works

When you work in teams, you have to coordinate. That means discussing and planning, of course, but even with the best of intentions, you can still end up with conflicts. You may have made some brilliant changes to the code—only to find them all wiped out by mistake when another programmer uploads his new version of a file.

Source control stops those kinds of problems by controlling access to code and by maintaining a history of the changes that have been made so things aren't destroyed unintentionally. Storing a history of your code is very powerful—not only can you compare a new (buggy) file against an older one, you can also revert to a previous version in case things have gone awry.

Source control also gives you the ability to coordinate the simultaneous development of several different versions of your software—for example, you might want to work on both a release version and a new beta version. You can do that using branches, as we're going to see in this chapter.

4.2 Understanding CVS

CVS is an open source project; it started life as a set of Unix shell scripts in 1986 and came into its own with dedicated software in 1989. Support for CVS is available on many operating systems today—Unix, Linux, Windows, and others. For the full CVS story, take a look at http://www.cvshome.org.

The idea behind CVS, as with any repository software, is to manage and record changes to source code. In fact, there are many types of repository software available, and some are more powerful than CVS, but CVS is in the most widespread use (perhaps because you can get it for free).

In CVS, a module is the basic equivalent of an Eclipse project. Modules are represented by directories in CVS. Standard projects correspond to physical modules, while logical or virtual modules are collections of related resources.

The files you share are stored in the CVS repository. When you retrieve a file from the repository, you check the file out. After you've modified the file, you commit the changes to check it back in and send those changes to the repository. If you want to refresh your own copy of a file, you update it from the repository.

In general, there are two models for source code repositories:

Pessimistic locking

Only one developer can check out a particular file at once—after the file is checked out, the file is locked. It's possible for someone else to check out read-only copies of the file, but not to change it. Access is sequential.

Optimistic locking

Developers can check out and modify files freely. When you commit changed files, the repository software merges your changes automatically. If it can't do that by itself, it'll notify you that you have to resolve the issue yourself. Access is random.

By default, CVS supports optimistic locking—although some CVS software also supports pessimistic locking. We'll be using optimistic locking here, which is what Eclipse supports.

Because each file needs to be kept track of, CVS gives each file a version number automatically. Each time a file is committed, its version number is incremented. When you first commit a file, its version number is 1.1; the next time, it's 1.2, and so on (this can depend on your CVS server; some will start with 1.0). When you commit files to the repository, they'll get a new version number automatically. We'll see these version numbers in the various views you work with when using source control.

When you update a file from the repository, your local version of the file is not overridden. Instead, Eclipse will merge your local file and the one in the CVS repository. If there are conflicts, the conflicting lines will be marked with special CVS markup indicating potential problems. You get the chance to modify your code to handle any conflicts that occurred during the merge operation, after which you can commit your new version of the code. Usually, updating goes smoothly, especially if each developer working on the project has her own set area to work in, but

4.3 Finding a CVS Server

To work with CVS, you need access to a CVS server. If you have access to a CVS server already, you're all set. Otherwise, take a look at the overview that follows.

Most Linux and Unix installations already come with a CVS server built-in. To test if you have a working CVS installation, type cvs --help at the prompt; you should see a list of help items. If you can't find a CVS server, you can download what you need from http://www.cvshome.org.

In Windows, the story is a little more complex. There are a variety of CVS servers for Windows, such as CVSNT, available for free from http://www.cvsnt.org. To install CVSNT, just download the executable file and run it.

4.3.1 Creating a Repository

You'll need to create a repository for your source code using the CVS server. In Linux and Unix, you do that with the command cvs -d *path* init, where *path* gives the location of the directory you want to use as the repository (the permissions and ownership for *path* should be set so all members of your development team can access it).

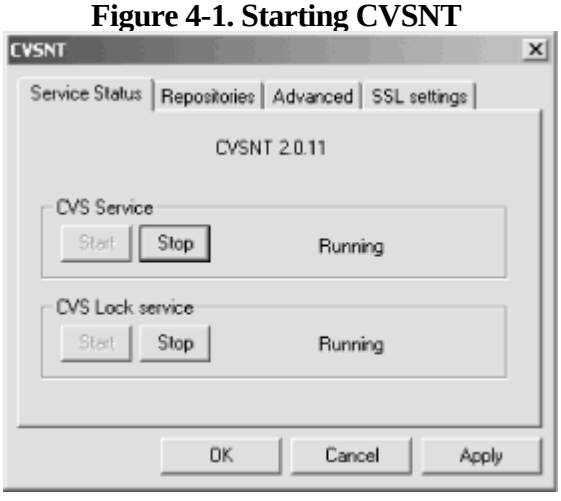
With CVSNT, you click the Repositories tab in the CVSNT control panel, click the Add button, enter the path of the new repository directory, such as c:\repository, and click OK.

4.3.2 Connecting to CVS

In Linux and Unix, you use one of two possible options to reach CVS: SSH (secure shell) or pserver. We'll use pserver here, but you can use either protocol; just make sure that the correct protocol is running on your machine.

In Windows, CVSNT runs as a Windows service, which means it is accessible to Eclipse as soon as you run it. You can start it from the Start menu by selecting the Service control panel item from whatever program group you've added it to, which opens the CVSNT control panel. Click the Start button in both the CVS Service and CVS Lock Service boxes, which will make CVSNT display the message "Running" in both those boxes, as you see in Figure 4-1.

Figure 4-1. Starting CVSNT



4.4 Adding a Project to the CVS Repository

We'll start using CVS by seeing how to add a new project, Ch04_01, to the CVS repository. After this project is in the CVS repository, anyone with access to the repository can check it out and work on it. You can see this sample project in Example 4-1; this sample code does nothing more than display the word "Hello".

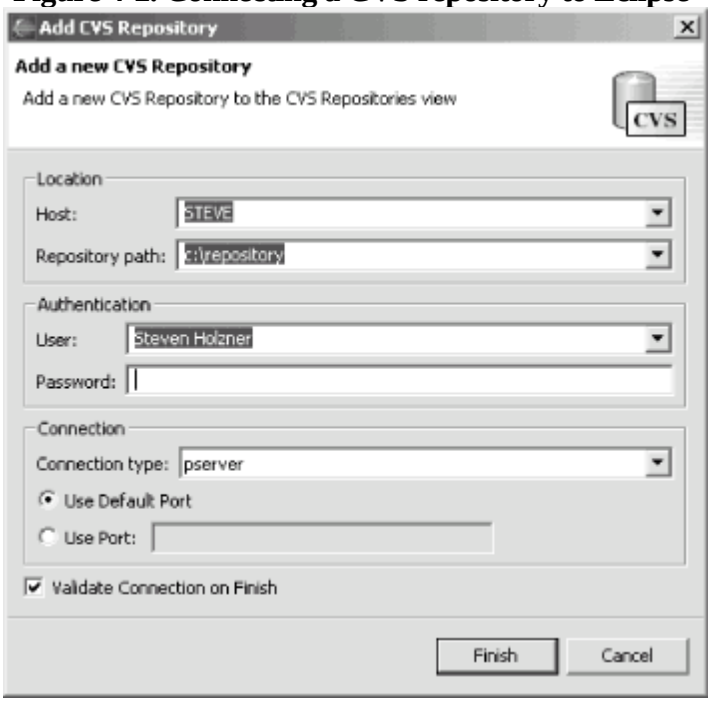
Example 4-1. A sample project

```
package org.eclipsebook.ch04;


public class Ch04_01 {


        public static void main(String[] args) {

                System.out.println("Hello");

        }

}
```

4.4.1 Creating a Repository Location

After you've created a project, how do you add it to the CVS repository? You first have to let Eclipse know about the repository, so select Window⟶ Open Perspective⟶ Other, and select the CVS Repository Exploring perspective (after you do this the first time, Eclipse will add this perspective to the Window⟶ Open Perspective, and will add a shortcut for this perspective to the other perspective shortcuts at the extreme left in Eclipse). When this perspective opens, right-click the blank CVS repositories view that appears at left and select New⟶ Repository Location, opening the Add CVS Repository dialog you see in Figure 4-2.

Figure 4-2. Connecting a CVS repository to Eclipse



Here, enter the name of the CVS server (usually the name of the computer that hosts the server), the repository path, the username and password, and specify the connection type (here, we'll be using the pserver protocol). Then click Finish to add the new repository to the CVS Repositories view, as you see in Figure 4-3.

Chapter 5. Building Eclipse Projects Using Ant

Eclipse is great for building your code. But for more advanced project development, there's still something missing. For example, what if you want not only to compile several files at once, but also to copy files over to other build directories, create JAR files and Javadoc, create new directories, delete previous builds, and create deployment packages all at once?

You can do that with a build tool like Apache's Ant (http://ant.apache.org/). Ant is a Java-based build tool that can perform all these tasks and much more. You can download Ant and run it on the command line, automating your build tasks to not only compile code, but to create JAR files, move and create classes, delete and make directories, and a great deal more.

The good news here is that Ant comes built into Eclipse, ready to use. Ant is the premier build tool for Java development, and we'll get an idea why in this chapter. As your projects become more and more elaborate, Ant can automate dozens of tasks that you'd otherwise need to perform manually. When you have things set up to run with Ant, all you've got to do is point and click to perform a complete build without having to take dozens of separate steps, which can save many steps omitted in error over the development process.

The fact that Ant comes built into Eclipse means that it's easier to use for us than for all those developers who use it on the command line. To see how this works, we'll start with a quick example.

5.1 Working with Ant

To use Ant from Eclipse, create a new project, Ch05_01, and add a new class to it, Ch05_01. In this class's main method, we'll just display the message "No worries.", as you see in Example 5-1.

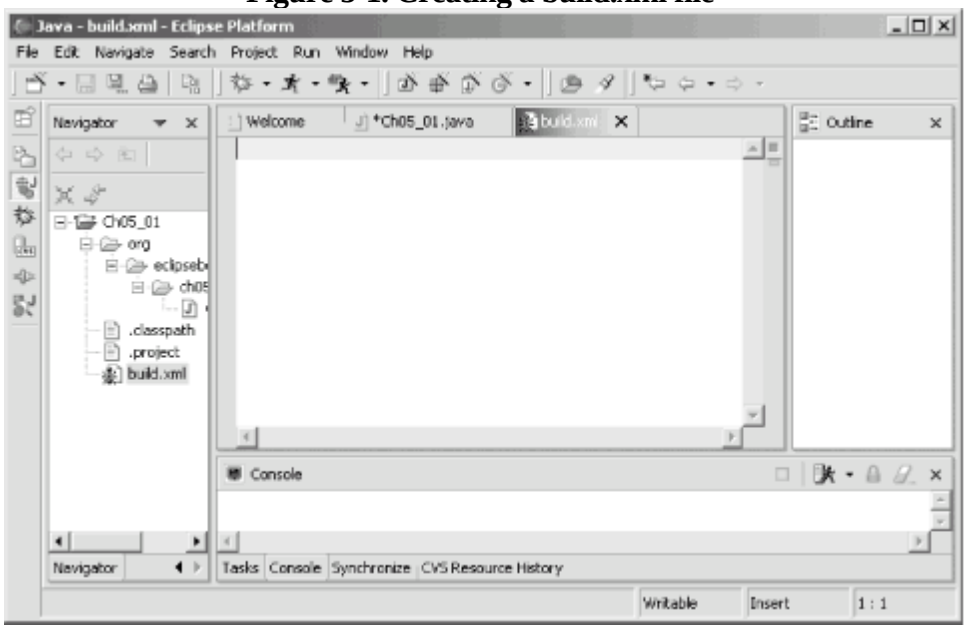
Example 5-1. A sample project

```
package org.eclipsebook.ch05;


public class Ch05_01 {


        public static void main(String[] args) {

                System.out.println("No worries.");

        }

}
```

To work with Ant, we'll need an Ant build file, which is named build.xml by default. To create that file in this project, right-click the project and select New⟶ File. Enter the name of the new file, build.xml, in the File name box and click Finish. You use the XML in this file to tell Ant how to build your project. Although Eclipse can automate the connection to Ant, Ant needs this XML build file to understand what you want it to do, which means that we will have to master the syntax in this file.

Eclipse recognizes that build.xml is an Ant build file and marks it with an ant icon, as you see at left in Figure 5-1.

Figure 5-1. Creating a build.xml file



Enter this simple XML into build.xml—all we're going to do here is have Ant echo a message, "Ant at work!", to the console:

```
<?xml version = "1.0" encoding="UTF-8" ?>

<project name = "Ch05_01" default = "Main Build">

    <target name = "Main Build">
```

5.2 JARing Your Output

Here's another example; in this case, we'll build an Eclipse project and store the resulting class in a JAR file. You won't need to be an Ant professional to follow along because we're interested in looking at Ant from an Eclipse point of view, not in the details of Ant per se. This example is designed to give you the basics of creating a working Ant build file in Eclipse; if you want more details on Ant itself, take a look at the manual at http://ant.apache.org/manual/index.html.

Our goal here is to create a new Java project in Eclipse, use Ant to compile it, and store the resulting .class file in a JAR file. To follow along, create a new Java project, Ch05_02. To emulate a somewhat real-world project, we're going to store the example's source code in a directory named src and its output in a directory named bin. You can set those directories up when you create the project in the third pane of the New Java Project dialog by clicking the Source tab, then clicking the Add Folder button, then the Create New Folder button to open the New Folder dialog. Enter the name src in the Folder name box and click OK twice. Eclipse will ask if you want to remove the project as source folder and update the build output folder to Ch05_02/bin. Click Yes, then click Finish to create the new project, which will be complete with src and bin folders.

Next, add a new class, Ch05_02, in a package named org.eclipsebook.ch05, to the project. Add code to the main method in this example to display the message "This code was built using Ant.", as you can see in Example 5-2.

Example 5-2. A sample project

```
package org.eclipsebook.ch05;


public class Ch05_02 {


    public static void main(String[] args) {

        System.out.println("This code was built using Ant.");

    }

}
```

Finally, add build.xml to the project by right-clicking the project in the Package Explorer and selecting New⟶ File. Type build.xml in the File name box and click Finish, which creates the file and opens it in the Ant editor. We'll start writing build.xml with the standard XML declaration and a <project> element that identifies the Main Build task as the default:

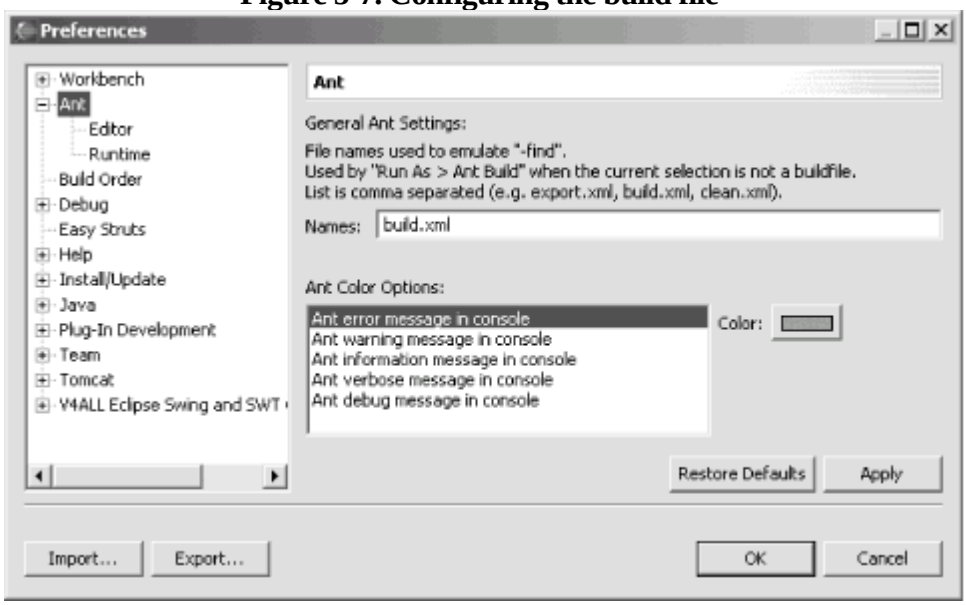
```
<?xml version="1.0" encoding = "UTF-8"?>

<project name="Ch05_01" default="Main Build" basedir=".">

        .

        .

        .

</project>
```

Next we'll create the properties corresponding to the directories we'll use—src, bin, a directory for the JAR file, jardir (we'll create a lib directory under the bin directory to store JAR files), and the JAR file itself, jarfile (we'll call this file Ch05_02.jar). Setting up properties this way lets you access these directory names later in the build file. We'll

5.3 Configuring Ant in Eclipse

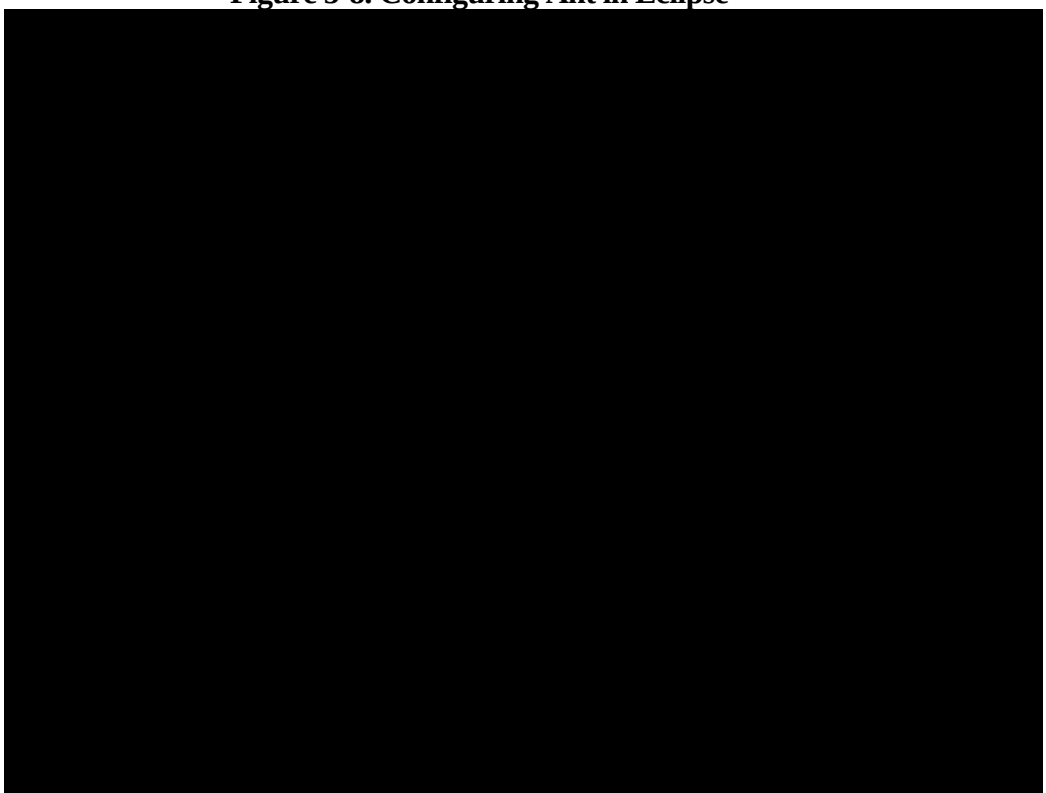
Eclipse also lets you configure its internal version of Ant. To configure how Ant will run, select Window→ Preferences, followed by the Ant item, as shown in Figure 5-7. In Eclipse, you don't need to name your build file build.xml; Ant will try to guess which file is the build file (the build file does need to be an XML file, with a name that has the extension .xml). You can help Eclipse out by giving an alternate name, or a list of names, in this dialog.

Figure 5-7. Configuring the build file



You can also set Ant runtime options by selecting the Runtime node, as shown in Figure 5-8. For example, to use a more recent version of Ant in Eclipse (Eclipse 2.1.1 comes with Ant 1.5.3, but the latest version of Ant, as of this writing, is 1.5.4, and Version 1.6 is out in beta; you can get alternate Ant versions directly from Apache at http://ant.apache.org), select the Runtime item and change the JAR entries you see in Figure 5-8 to the new versions of ant.jar and optional.jar. You can also set Ant variables like ANT_HOME in this dialog.

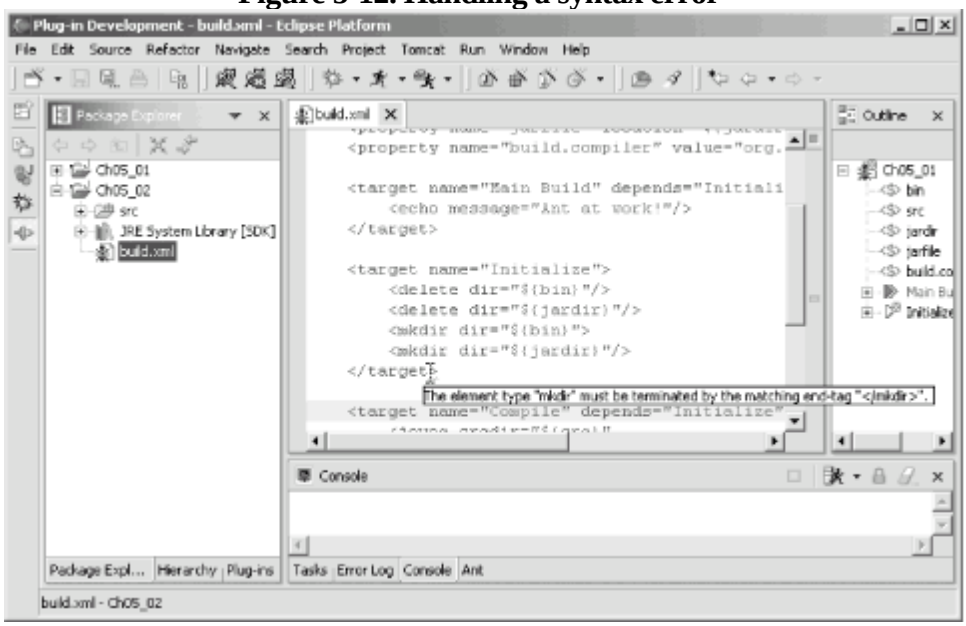
Figure 5-8. Configuring Ant in Eclipse



5.4 Catching Errors in Build Files

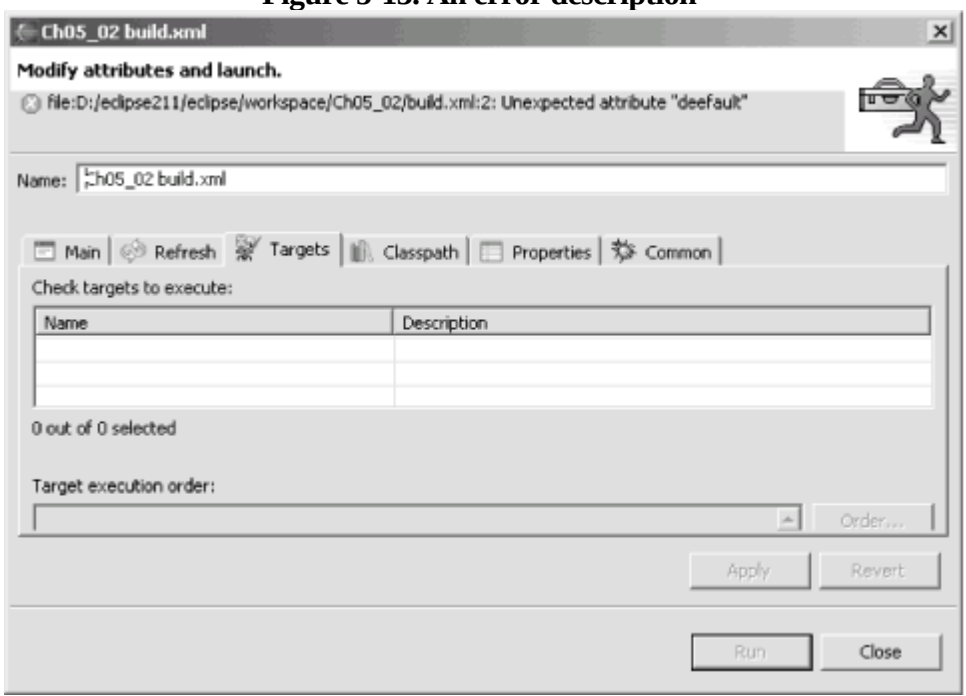
Eclipse gives you some support for catching Ant errors before you run Ant, but not much. The Eclipse Ant editor doesn't handle syntax errors as the JDT editor does for Java—for example, if you misspell the project element's default attribute as deefault, the Ant editor won't have a problem. However, if the XML in your build file has syntax errors, such as missing a closing tag or improper nesting (in XML terms: if your XML is not well-formed), you'll see the same wavy red line and hollow red box you see in the JDT editor in the Ant editor, indicating a syntax error, as shown in Figure 5-12. You can determine what error occurred by looking at the wavy line's tooltip, as shown in the figure, where we haven't closed the mkdir element.

Figure 5-12. Handling a syntax error



If you miss an XML error like this in the editor, you'll see a message reminding you of it as soon as you try to run Ant to build the project. The Ant editor doesn't display non-XML syntax errors (like misspelling the project attribute as deefault), but you'll automatically see any syntax errors listed when you try to run Ant, as shown at the top of Figure 5-13.

Figure 5-13. An error description



Chapter 6. GUI Programming: From Appletsto Swing

This chapter is all about creating GUIs with Java code in Eclipse. So far, our code has just displayed text in the console. In this chapter, we're going to start creating GUIs using applets, the Abstract Windowing Toolkit (AWT), and Swing. We'll also take a look at using an Eclipse plug-in to create Swing code.

Our first topic is all about building applets. Java applets were Java's first foray into the Internet, and they were popular for quite a while. Browsers first started stocking Java in order to support applets. In time, applets have become less popular because they're necessarily limited and have been superceded by glitzy packages like Flash and Java Web Start. Nonetheless, Eclipse has special provisions for developing and testing applets, so we'll take a brief look here at how that support works.

Creating an applet is much like creating any other Java project in Eclipse. In this case, our applet is just going to display the message "Hello from Eclipse!" Create a new Java project named Ch06_01, and give it a new class, Ch06_01, in the org.eclipsebook.ch06 package. To create an applet, we'll need to import java.applet.Applet and java.awt.*:

```
import java.applet.Applet;

import java.awt.*;
```

Then extend the Ch06_01 class from the Applet class:

```
public class Ch06_01 extends Applet {

        .

        .

        .
```

In an applet, the init method lets you initialize the applet, and we'll set the background color to white in that method. The start, stop, and destroy methods let you handle the associated events in the applet's life cycle. To draw our text in the applet's window, we'll use the Graphics object passed to the paint method. Here's the code to add:

```
public class Ch06_01 extends Applet {


    public void init( )

    {

        setBackground(Color.white);

    }


    public void start( )

    {

    }
```

6.1 Creating AWT Applications

The Abstract Windowing Toolkit, or AWT, was Java's early attempt at a GUI toolkit. It's still supported and used, so we'll take a look at an example here that will launch its own window. The original AWT package only took six weeks to write, and the controls, designed for use in applets, were modeled after HTML controls. Since that time, Swing has taken over the standard Java interface—but even Swing is built on top of AWT.

Our AWT example will just launch its own window and display the same message as our applet. To follow along, create a new project, Ch06_02, and give it a new class with a main method, Ch06_02, putting that class into the org.eclipsebook.ch06 package. We need to import java.awt.* for basic AWT support, and java.awt.event.* to handle the window-closing event, so add this code:

```
import java.awt.*;

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

The actual window this example will display will be based on the AWT Frame class and will be called AppFrame, so add this code to *Ch06_02.java*:

```
class AppFrame extends Frame

{

        .

        .

        .

}
```

As in our applet, we'll override the paint method and use the passed Graphics object to display the text we want in the new window:

```
class AppFrame extends Frame

{

    public void paint(Graphics g)

    {

        g.drawString("Hello from Eclipse!", 60, 100);

    }

}
```

Now, in the Ch06_02 class's main method, we'll create a new window of the AppFrame class and set its size:

```
public class Ch06_02 {



    public static void main(String [] args)

    {

       AppFrame f = new AppFrame( );



       f.setSize(300, 300);
```

6.2 Creating Swing Applications

Our first Swing application will be simple, and it will only mimic the applet and AWT applications we've seen by displaying the same message as they did. To put together this application, create a project named Ch06_03 and add the class Ch06_03 with a main method to the org.eclipsebook.ch06 package. We'll need these imports in this example:

```
import java.awt.*;

import javax.swing.*;

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

In this example, we'll derive a new class, Panel, from the Swing JPanel class to display the message in:

```
class Panel extends JPanel

{

        .

        .

        .

}
```

The JPanel class has a method named paintComponent and we'll override that method to display the text message like this:

```
class Panel extends JPanel

{

        Panel( )

        {

                setBackground(Color.white);

        }


        public void paintComponent (Graphics g)

        {

                super.paintComponent(g);

                g.drawString("Hello from Eclipse!", 60, 100);

        }

}
```

The main class in this application, Ch06_03, will extend the Swing JFrame class:

```
public class Ch06_03 extends JFrame {

        .

        .
```

6.3 Using Eclipse Plug-ins

Many Eclipse plug-ins are available, and many are free. You can find over 400 plug-ins at http://www.eclipse-plugins.2y.net/eclipse/, and the plug-in we'll use for Swing is available from that site. On the other hand, Eclipse plug-ins can change very frequently, so we can't base entire chapters on them—the plug-in you download would be different from the one written about in this book.

But we can look at what specific plug-ins have to offer in general terms, and we'll do that here. In this chapter, we'll take a look at the popular, free V4ALL plug-in to help us write some Swing code, available from http://www.eclipse-plugins.2y.net/eclipse/. Navigate to that page, click the plugins link, then the categories link, and then the UI link to find the Eclipse V4ALL Swing & SWT plug-in. Stop Eclipse if it's running and download that compressed plug-in file to Eclipse's eclipse directory (this is the directory that contains the workspace and plugins directories).

When you download plug-ins, you usually download them to the eclipse directory. They'll come zipped or tarred, and you usually uncompress them in the eclipse directory. When you uncompress them, the support files for plug-ins go automatically into the plugins and features directories, which are subdirectories of the eclipse directory.

Plug-ins are not always designed to be uncompressed in the eclipse directory—if possible, check the plug-in's documentation first. When you're about to unzip a plug-in, open it in an unzip utility and see if it'll unzip files to the plugins directory. If so, unzipping it in the eclipse directory is the right thing to do. Some plug-ins should be uncompressed in the plugins directory directly, however.

Unzip or untar the V4ALL plug-in now, which should automatically load its files into the plugins and features directories. That's all the installation you need. Now start Eclipse again. Eclipse may display a dialog indicating that there are pending configuration changes and may ask you if Eclipse should restart. Restart Eclipse if you're asked.

Congratulations—you've just installed the V4ALL plug-in. It's time to put it to work.

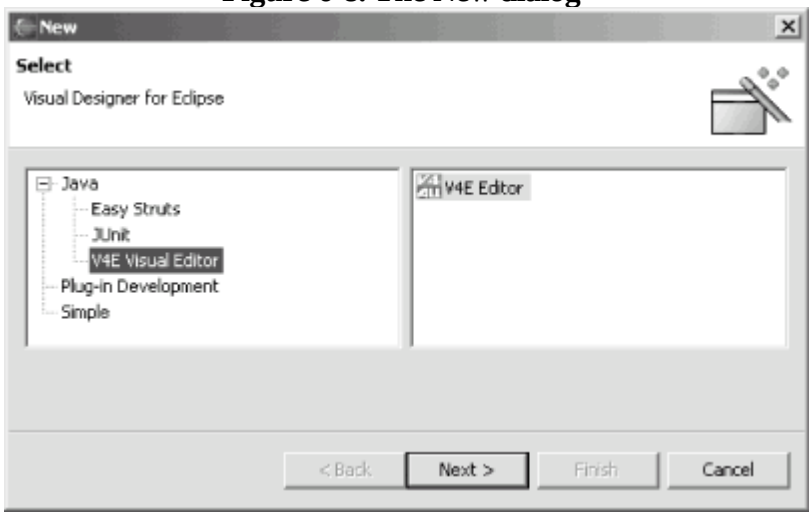
6.4 Using the V4ALL Plug-in

The V4ALL plug-in gives you a palette of Swing components that you can drag to a "whiteboard" to design your application. To see how this works, we'll create a Swing application using this plug-in. Create a new Java project named Ch06_05, and give it a source folder, *src*, in the third pane of the New Project dialog as we've done before. Now create a new class, Ch06_05, in the org.eclipsebook.ch06 package.

6.4.1 Adding a V4ALL Editor to a Project

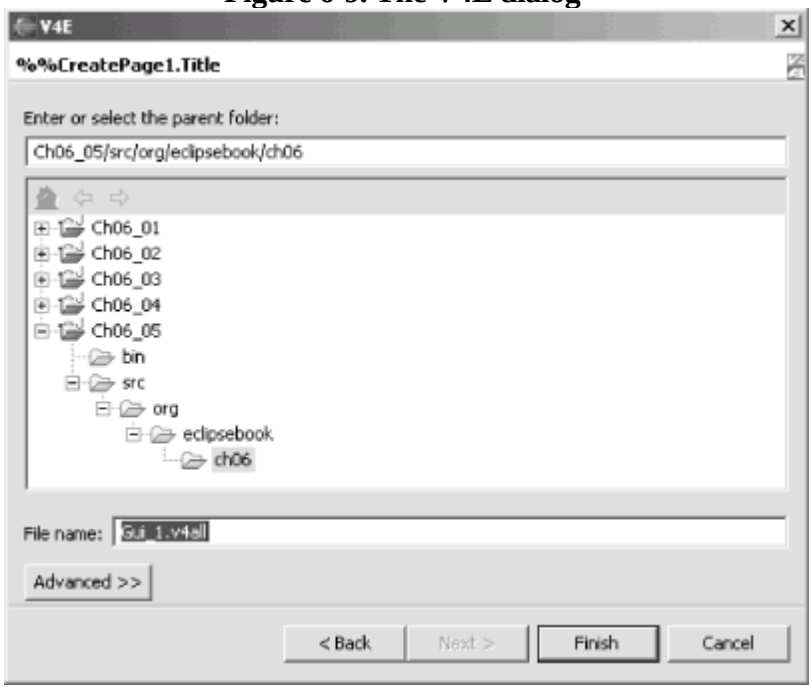
To add a new V4ALL editor to the project, right-click the org.eclipsebook.ch06 package and select New⟶ Other, opening the New dialog you see in Figure 6-8.

Figure 6-8. The New dialog



You should see V4E Visual Editor as one of the options; select that item and click Next to open the V4E dialog you see in Figure 6-9.

Figure 6-9. The V4E dialog



The V4E dialog lets you select the name of the V4ALL file that will contain the design information V4ALL will use

Chapter 7. SWT: Buttons, Text, Labels, Lists, Layouts, and Events

SWT, the Standard Widget Toolkit, was created by IBM as a replacement for both the Java Abstract Windowing Toolkit (AWT) and Swing. It's a big topic, and it comes with Eclipse, ready to use. Here's how the Eclipse documentation describes SWT:

The Standard Widget Toolkit (SWT) is a widget toolkit for Java developers that provides a portable API and tight integration with the underlying native OS GUI platform.

7.1 Java Graphics

Graphics work in Java has a long and glorious past, and, let's hope, a similar future. It started with the very basic AWT, moved on through the powerful Swing package, and now stands on the threshold of the SWT age.

7.1.1 AWT

The AWT, Java's first attempt at a GUI toolkit, was written in a matter of weeks. It lets Java developers display windows with various controls like text boxes and buttons. AWT GUIs were easy to develop, and they used the underlying operating system's controls themselves—for example, in Windows, you'd see a Windows text box. On the Mac, you'd see a Mac text box. Some operating systems had a different control set from other operating systems, which meant that Sun only implemented those controls common to all operating systems Java was targeted to, and that limited AWT to a set of relatively simple controls.

7.1.2 Swing

To address the growing needs of developers, Java introduced Swing, which provides non-native implementations of higher level controls like trees, tables, and text. This provides a great deal of functionality, but it makes applications developed in Swing stand out as being different and very Java-specific.

Sun added a "look-and-feel" emulation (which we saw at work in the previous chapter) to help applications look more like the operating system they're running on, but they couldn't keep up with all the operating system releases (such as Windows ME, 2000, XP, and so on). In addition, because the GUI was implemented in Java and not natively in the operating system, Swing response time was poor compared to native applications.

7.1.3 SWT

SWT addresses many of the issues here by providing a set of widgets that make use of native controls (through the Java Native Interface, JNI) when such controls are available. Otherwise, SWT creates its own controls as needed for those that don't have an operating system counterpart. This does mean that native code is needed to support each operating system, but so far, IBM has been able to provide that and stay up-to-date. Additionally, SWT comes with Eclipse, so it is a fit topic for our discussion.

SWT is substantial and feature-rich, as you'd expect if it's intended to replace AWT and Swing. We're going to get an introduction to SWT in this and the next chapter, and it turns out that you really don't need in-depth SWT knowledge to create powerful and useful SWT applications. All you need to know about are widgets (the user-interface elements you use in SWT), SWT layouts (which let you position those widgets), and SWT events. We're going to see plenty of examples, starting immediately, to see how all these work in SWT.

SWT is being ported to more and more graphical environments all the time. Currently supported are: Windows, Linux GTK, Linux Motif, Solaris Motif, AIX Motif, HPUX Motif, Photon QNX, and Mac OS X.

7.2 An SWT Example

Instead of talking about SWT in the abstract, let's get this show on the road and see some code at work. Coding an example is going to give us the SWT story and what it takes to put together an SWT application. Our first example will be a simple one, and it will just display the message "No worries!" in an SWT window.

To follow along, create a new project, Ch07_01, and add a class, Ch07_01, in the org.eclipsebook.ch07 package. To work with SWT and SWT widgets, you typically start with these two imports:

```
import org.eclipse.swt.widgets.*;

import org.eclipse.swt.*;
```

You'll need to include swt.jar in the build path to make these imports work. Remember that SWT is operating system-dependent, so there's going to be a different swt.jar for different operating systems. To add swt.jar to the Ch07_01 project, select that project in the Package Explorer, right-click it, and select Properties. In the Properties for Ch07_01 dialog that opens, select the Java Build Path item and click the Add External JARs button. Then navigate to swt.jar, which you'll find in one of the following directories, depending on your operating system (note that INSTALLDIR is the Eclipse installation directory; also note that you'll have to update these paths for your version of Eclipse, such as changing 2.1.1 to 2.1.2 or some other value):

Win32

INSTALLDIR\eclipse\plugins\org.eclipse.swt.win32_2.1.1\ws\win32\swt.jar

Linux GTK

INSTALLDIR/eclipse/plugins/org.eclipse.swt.gtk_2.1.1/ws/gtk/swt.jar

Linux Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/motif/swt.jar

Solaris Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/solaris/sparc/swt.jar

AIX Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/aix/ppc/swt.jar

HPUX Motif

INSTALLDIR/eclipse/plugins/org.eclipse.swt.motif_2.1.1/ws/hpux/PA_RISC/swt.jar

Photon QNX

7.3 Working with Buttons

Our next step is going to be adding interactive widgets to SWT applications. In this case, we're going to add a button that, when clicked, will display text in a text control. We start off as before, except this time, we also import org.eclipse.swt.events.* to handle button clicks and org.eclipse.swt.layout.* to set the layout of our controls. And we can also embellish our example a little more by setting the text that should appear in the shell's titlebar, using the setText method:

```
package org.eclipsebook.ch07;


import org.eclipse.swt.widgets.*;

import org.eclipse.swt.SWT;

import org.eclipse.swt.events.*;

import org.eclipse.swt.layout.*;


        public static void main(String [] args) {

           Display display = new Display( );

           Shell shell = new Shell(display);

           shell.setSize(300, 200);

           shell.setLayout(new RowLayout( ));

           shell.setText("Button Example");

        .

        .

        .
```

Now we're dealing with multiple controls, and we're also going to set the SWT layout of our shell to the row layout, which displays controls in rows:

```
public class Ch07_02 {


        public static void main(String [] args) {

           Display display = new Display( );

           Shell shell = new Shell(display);

           shell.setSize(300, 200);

           shell.setText("Button Example");

           shell.setLayout(new RowLayout( ));

        .

        .

        .
```

7.4 Working with Composites and Layouts

Selecting a layout lets you specify how to arrange your controls; there are four layout classes built into SWT:

FillLayout

Lets you fill a shell's client area.

GridLayout

Lays out control children in a grid.

RowLayout

Lays out controls in either horizontal rows or vertical columns.

FormLayout

Lets you position controls relative to the parent composite or another control. It is the most precise of all the layouts.

For example, here's how the grid layout works. We're going to create a grid layout with four columns and fill it with buttons. In this case, we'll create a new Composite control, which can contain other controls, and fill the composite with buttons. To start, we create a new shell and use a row layout to display our composite control. Then we create the composite control and a grid layout in it with four columns (the SWT.NONE constant means we're not setting any nondefault styles here):

```
public static void main (String [] args) {

        Display display = new Display ( );

        final Shell shell = new Shell (display);

        shell.setSize(300, 200);

        shell.setLayout(new RowLayout( ));


        final Composite composite = new Composite(shell, SWT.NONE);

        GridLayout gridLayout = new GridLayout( );

        gridLayout.numColumns = 4;

        composite.setLayout(gridLayout);

    .

    .

    .
```

All that's left is to add the buttons to the composite control using a loop and to add the event loop itself, as you see in Example 7-3.

7.5 Working with Lists

Another popular control is the list control, represented by the SWT List class. We'll take a look at an example using this control and how to recover selections made in the control. Here are the styles you can use with lists:

SWT.BORDER

Adds a border

SWT.H_SCROLL

Adds a horizontal scrollbar

SWT.V_SCROLL

Adds a vertical scrollbar

SWT.SINGLE

Allows single selections

SWT.MULTI

Allows multiple selections

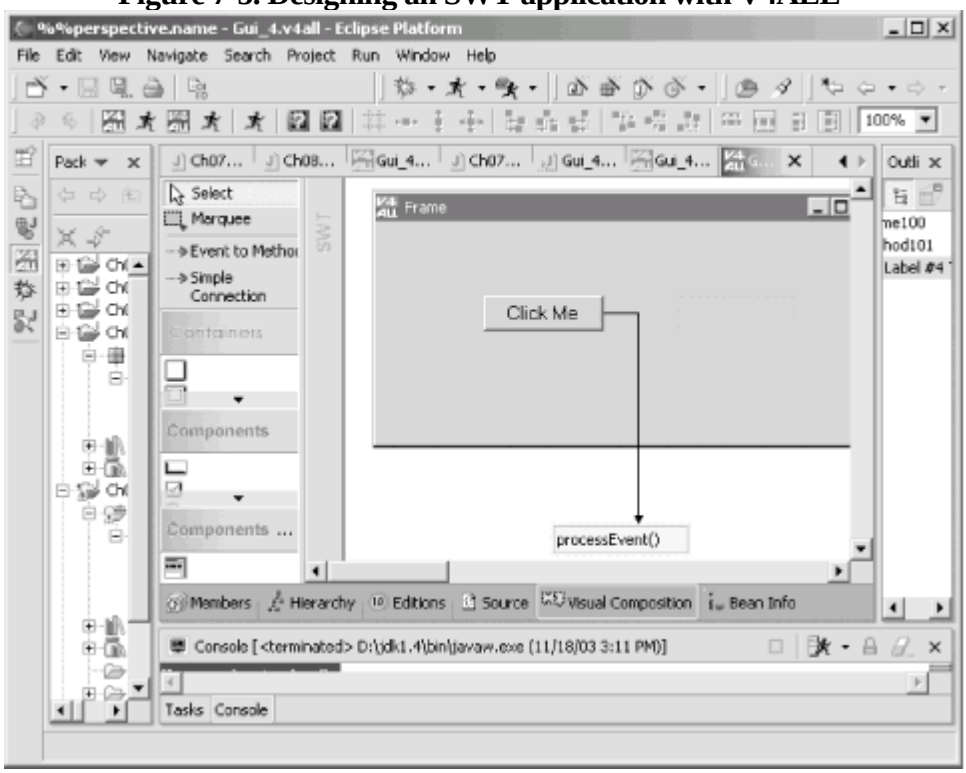
In this case, we're going to fill the entire shell with a list control, using the fill layout. Here's what that looks like—note that we're also using the List class's add method to add items to the list:

```
public class Ch07_04 {


        public static void main (String [] args) {
                Display display = new Display ( );
                Shell shell = new Shell (display);
                shell.setText("List Example");
                shell.setSize(300, 200);
                shell.setLayout(new FillLayout(SWT.VERTICAL));


                final List list = new List (shell, SWT.BORDER | SWT.MULTI |
```

7.6 Using V4ALL with SWT

You can also use plug-ins like V4ALL to create SWT code, which can make things a little easier because you can drag controls where you want them. In Figure 7-5, you can see an example that works much as our Swing example worked in the previous chapter. In this case, we've created a new V4ALL editor in a project named Ch07_05, just as we did in Chapter 6—but here, we're going to generate code for SWT, not Swing.

Figure 7-5. Designing an SWT application with V4ALL



After creating the new V4ALL editor, add a button and a label to the new window, change the caption of the button to Click Me, and clear the caption of the label. Next, drag a method to the whiteboard, and name that method processEvent as you see in Figure 7-5. Connect that method to the button and, using the properties view as we did in Chapter 6, connect the processEvent method to the button's Action Performed event.

To generate the code for an SWT application, start by adding swt.jar to the build path, and select the Code Generation⟶ Generate Code for SWT menu item, creating Gui_1.java. In that file, Eclipse may have marked two imports as never used, so remove those two imports:

```
import org.eclipse.swt.*;

import org.eclipse.swt.graphics.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.custom.*;

import org.eclipse.swt.layout.*;
```

Then add the code to the processEvent method to set the text in the label to "No worries." when the user clicks the button:

```
public void processEvent( ){

// user code begin {1} Swing

// user code end
```

Chapter 8. SWT: Menus, Toolbars, Sliders, Trees, and Dialogs

SWT supports many more widgets than we had a chance to work with in the previous chapter, and we'll get the details on a number of the most central SWT controls here: menus, toolbars, sliders, trees, and custom dialogs.

8.1 Working with Menus

As you'd expect, SWT supports menus, as any GUI builder should. The process of creating and supporting menus in your SWT applications is not necessarily easy, but it's not unduly difficult. To make this work in an example, we're going to create a menu system with a File menu and a Help menu and react to various menu selections, displaying text in a label to match the selected items.

To create a menu system in SWT, you create a Menu object, corresponding to the top level of the menu system. As with all the SWT controls, you can see the allowed styles back in Table 7-1; here are the possible styles for menus:

SWT.BAR

Sets menu bar behavior

SWT.DROP_DOWN

Creates a drop-down menu

SWT.POP_UP

Creates a pop-up menu

SWT.NO_RADIO_GROUP

Prevents the use of radio groups

SWT.LEFT_TO_RIGHT

Sets left-to-right orientation

SWT.RIGHT_TO_LEFT

Sets right-to-left orientation

The File and Help menus are MenuItem objects associated with the Menu object, and here are the possible styles for menu items:

SWT.CHECK

Creates a checkbox

8.2 Working with Toolbars

SWT also supports toolbars, using the Toolbar class. To create a toolbar, you just associate the toolbar with the shell you're working with and set its style:

SWT.BORDER

Creates a toolbar with a border

SWT.FLAT

Creates a flat toolbar

SWT.WRAP

Creates a wrappable toolbar

SWT.RIGHT

Aligns the toolbar on the right

SWT.SHADOW_OUT

Adds a shadow to the toolbar

SWT.HORIZONTAL

Creates a horizontal toolbar

SWT.VERTICAL

Creates a vertical toolbar

Here's how we'll create a new toolbar (the constant SWT.NONE indicates we're not setting a custom style):

```
ToolBar toolbar = new ToolBar(shell, SWT.NONE);
```

Each item in the toolbar is a ToolItem object, and we'll create four push buttons in an example next, using the SWT.PUSH style and the ToolItem class. Here are the possible styles for tool items:

SWT.PUSH

8.3 Working with Sliders

Another handy SWT control is the slider, which lets the user select a value from a continuous numeric range. Sliders are easy to use; this next example will recover a slider's new position when the user moves the slider's thumb (also called the slider's scrollbox). Here are the styles you can use when creating sliders:

SWT.BORDER

Adds a border

SWT.HORIZONTAL

Creates a horizontal slider

SWT.VERTICAL

Creates a vertical slider

We'll add a prompt to the user in a label ("Move the slider"), a horizontal slider using the style SWT.HORIZONTAL (use SWT.VERTICAL to create a vertical slider instead), and a text control to display the new position of the slider:

```
final Label label = new Label(shell, SWT.NONE);

label.setText("Move the slider");

label.setBounds(0, 20, 150, 15);


final Slider slider = new Slider(shell, SWT.HORIZONTAL);

slider.setBounds(0, 40, 200, 20);


final Text text = new Text(shell, SWT.BORDER);

text.setBounds(0, 100, 200, 25);
```

That adds the slider; the next step is to handle user actions. Sliders support a number of events, each of which is given by an SWT constant:

SWT.ARROW_DOWN

The down/right arrow button was clicked.

SWT.ARROW_UP

8.4 Working with Trees

Trees are another standard SWT element; you use the Tree class to create trees and the TreeItem class to add nodes to the tree. Here are the possible SWT styles for trees:

SWT.BORDER

Adds a border

SWT.H_SCROLL

Adds a horizontal scrollbar

SWT.V_SCROLL

Adds a vertical scrollbar

SWT.SINGLE

Supports single selections

SWT.MULTI

Supports multiple selections

SWT.CHECK

Supports checkboxes

In this example, we'll create a new tree with a border:

```
final Tree tree = new Tree(shell, SWT.BORDER);

tree.setSize(290, 290);

shell.setSize(300, 300);
```

To add nodes to the tree, use the TreeItem class, passing the tree object to the TreeItem constructor. In this example, we'll add five top-level items to the tree:

```
final Tree tree = new Tree(shell, SWT.BORDER);

tree.setSize(290, 290);
```

8.5 Working with Dialogs

As our final SWT example, we'll take a look at creating new SWT dialogs and recovering data that the user entered in them. SWT comes with a number of prebuilt dialog classes, such as the FileDialog and DirectoryDialog classes, but in this example we're going to create our own custom dialog. We'll display a custom dialog with the message "OK to proceed?" along with OK and Cancel buttons and determine which button the user clicked.

In the main application shell, we'll display a button named opener that the user can click to display the dialog, and a text control to display the selection the user made:

```
final Button opener = new Button(shell, SWT.PUSH);

opener.setText("Click Me");

opener.setBounds(20, 20, 50, 25);


final Text text = new Text(shell, SWT.SHADOW_IN);

text.setBounds(80, 20, 100, 25);
```

The dialog will also be a Shell object. Here are the styles you can use:

SWT.APPLICATION_MODAL

Makes a dialog box application modal

SWT.BORDER

Adds a border

SWT.H_SCROLL

Adds a horizontal scrollbar

SWT.V_SCROLL

Adds a vertical scrollbar

SWT.CLOSE

Adds a close button

SWT.DIALOG_TRIM

8.6 Opening Internet Explorer in anSWT Window

As you can see, SWT is dedicated to covering the bases when it comes to GUI development, giving you the controls you'd expect to find in standard GUIs. That also means SWT work can be pretty pedestrian, as you stock your GUI with standard controls like sliders, text controls, and so on. So we'll end the SWT discussion with an SWT example that's cooler than most—in this case, we'll see how to open Internet Explorer inside an SWT window and browse to a URL of our selection.

For example, the user may want to search for information on the Internet, in which case you could open a search engine. In our example, we'll assume the user needs some good computer books, so we'll browse to the site http://www.oreilly.com.

In this new project, Ch08_06, we'll use Object Linking and Embedding (OLE) to open Internet Explorer in an SWT window, which means this example will be Windows-only. Unfortunately, there is no counterpart in Linux or Mac OS X—yet. Still, it's worth looking at how this works in Windows because it's impressive.

On the other hand, browser widgets will be coming built-in to SWT in Eclipse 3.0—see the discussion in Chapter 13.

To work with OLE, you import the org.eclipse.swt.ole.win32.* packages and create an SWT OleControlSite object in the main method:

```
package org.eclipsebook.ch08;


import org.eclipse.swt.*;

import org.eclipse.swt.layout.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.ole.win32.*;


public class Ch08_06 {


        public static void main(String[] args) {


                final Display display = new Display( );

                Shell shell = new Shell(display);

                shell.setSize(600, 400);

                shell.setLayout(new FillLayout( ));
```

Chapter 9. Web Development

Java has arrived on the Web with a vengeance in the form of JavaServer Pages (JSP) and servlets, and we'll take a look at how to create these using Eclipse in this chapter. To do that, we're going to use the Tomcat web server, which is the Sun Microsystems reference implementation for both JSP and servlets—and it's free for the downloading.

Even though we're going to use Tomcat in this chapter, the Java code we write and the XML files we edit are not Tomcat-specific. JSP and servlets both must adhere to their respective specifications, which means you can use what we develop here with other JSP/servlet web containers.

9.1 Installing and Testing Tomcat

You can get the Tomcat web server at http://jakarta.apache.org/tomcat/; the current release version as of this writing is 4.1.29. Downloading and installing Tomcat isn't hard—just unzip or untar it, which creates this directory structure:

```
jakarta-tomcat-4.1.29

|_ _bin                         Binary executable files

|_ _common                      Classes available to internal classes and web apps

|   |_ _classes                 Common Java classes

|   |_ _endorsed                Endorsed Java classes

|   |_ _lib                     Common Java classes in Java Archive (JAR) format

|_ _conf                        Configuration files (such as passwords)

|_ _logs                        The server's log files

|_ _server                      Internal Tomcat classes

|_ _shared                      Shared files

|_ _temp                        Temporary files

|_ _webapps                     Directory for Web applications

|_ _work                        Scratch directory for holding temporary files
```

For web developers, the most important directory here is the webapps directory, which is where you store files to make them accessible to client browsers. We're going to see that directory at work throughout this chapter.

As we develop our web applications, we're going to start by running Tomcat outside Eclipse; later in this chapter, we'll take a look at how to launch it while working inside Eclipse. Our first goal is to get Tomcat working. Before running Tomcat from the command line, you must set these two environment variables:

JAVA_HOME

Set to the installation directory of Java, the parent directory of the Java bin directory. For example, this might be C:\jdk1.4 in Windows.

CATALINA_HOME

Set to the installation directory of Tomcat, the parent directory of the Tomcat bin directory. For example, this might be /usr/local/jakarta-tomcat-4.1.29 in Linux.

You can set these environment variables from the command prompt as in this example: set JAVA_HOME=C:\jdk1.4. (In the Unix tcsh shell, use setenv instead.) On most operating systems, you can also set environment variables in a more permanent way using control panel dialogs or by setting configurations.

You can find good instructions on setting environment variables for all operating systems that run Java from the Java download page in the installation notes. Here's the URL with

9.2 Creating a JSP

The web server is running and now it's time to put it to use. At its simplest, you can use Eclipse to write JSP files, which don't require compilation. These files can enclose Java code in scriptlet, declaration, and expression elements. The most general of these are scriptlets, which can enclose multiline Java code. Scriptlets are enclosed with the markup <% and %>, as you can see in Example 9-1. You can use the out object's println method to send text back to the browser; in this case, we're simply sending the text "Using JSP" back to the browser in this JSP.

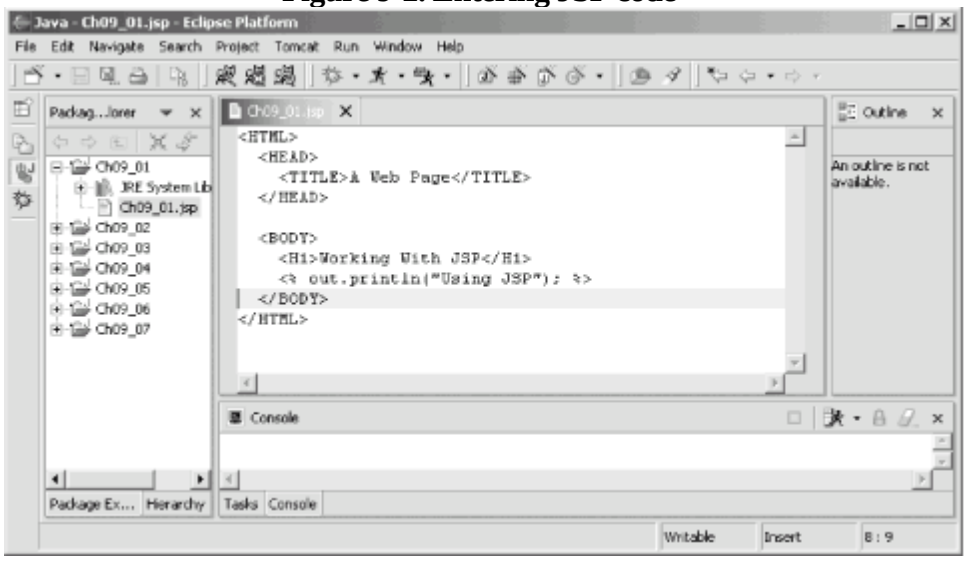
Example 9-1. A sample JSP

```
<HTML>

  <HEAD>

    <TITLE>A Web Page</TITLE>

  </HEAD>


  <BODY>

    <H1>Working With JSP</H1>

    <% out.println("Using JSP"); %>

  </BODY>

</HTML>
```

An easy way to create this JSP file is just to enter it into Eclipse, as you can see in Figure 9-2, where we've created a new project, Ch09_01, and a new file, Ch09_01.jsp, to hold the JSP code. There's no syntax checking going on here; Eclipse is just using its standard default editor.

If you do want to check syntax of JSP documents, give the free XML editor XML Buddy a try (available at http://www.xmlbuddy.com).

Figure 9-2. Entering JSP code



How can you install Ch09_01.jsp so Tomcat can serve it to client browsers? To handle the examples from this

9.3 Creating a Servlet

JSPs were introduced to make Java web programming more appealing for the novice, letting you mix HTML and Java code. JSPs are actually based on Java servlets (they're translated into servlets before being run), which are pure Java code, and which we're going to focus on for the most part in this chapter. You can see an example servlet in Example 9-2.

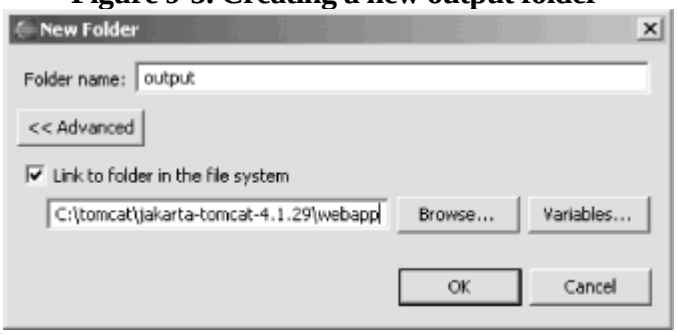
Example 9-2. A sample servlet

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;


public class ch09_02 extends HttpServlet

{

    public void doGet(HttpServletRequest request,

        HttpServletResponse response)

        throws IOException, ServletException

    {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter( );


        out.println("<HTML>");

        out.println("<HEAD>");

        out.println("<TITLE>");

        out.println("A Servlet Example");

        out.println("</TITLE>");

        out.println("</HEAD>");

        out.println("<BODY>");

        out.println("<H1>");

        out.println("Working With Servlets");

        out.println("</H1>");

        out.println("Using servlets");

        out.println("</BODY>");

        out.println("</HTML>");

    }

}
```

9.4 Creating a Servlet in Place

The code we're developing needs to be in the Tomcat directories to run, and storing the compiled code we generate there is no problem with Eclipse. To see how this works, create a new project, Ch09_03. Enter the name of the project in the New Java Project dialog and click Next to bring up the second pane of this dialog. To set the default output folder for compiled code, click the Browse button next to the Default output folder box to open the Folder Selection dialog. Click the Create new folder button, and click the Advanced button in the New Folder dialog. Now select the Link to folder in the filesystem checkbox, and click the Browse button. Browse to the jakarta-tomcat-4.1.29\webapps\Ch09\WEB-INF\classes directory—the root directory for compiled servlet code—and click OK, bringing up the New Folder dialog again. In that folder, give this new folder the name output, as you see in Figure 9-5, and click OK.

Figure 9-5. Creating a new output folder



Setting up the output folder this way means that the code we compile will automatically be placed in the classes folder (or, in our case, in the classes\org\eclipsebook\ch09 folder, following the package name we're using). Now create a new class, Ch09_03, and enter the code for the servlet in Example 9-4 in it.

Example 9-4. A new servlet

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;


public class ch09_03 extends HttpServlet

{

    public void doGet(HttpServletRequest request,

        HttpServletResponse response)

        throws IOException, ServletException

    {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter( );


        out.println("<HTML>");

        out.println("<HEAD>");
```

9.5 Connecting to a JavaBean

JSP files are able to connect to compiled Java code using JavaBeans, and developing those applications are no problem now that you know how to use linked folders. Here's an example, the Ch09_04 project. In this case, we'll use the bean in Example 9-6, which supports a property named msg that returns the message "No worries!"

Example 9-6. A JavaBean

```
package org.eclipsebook.ch09;


public class Ch09_04 {

        private String message = "No worries!";


        public void setMessage(String msg)

        {

                this.message = msg;

        }


        public String getMessage( )

        {

                return this.message;

        }


        public Ch09_04( )

        {

        }

}
```

This code's output, Ch09_04.class (note that you don't need *servlet.jar* in the build path here), goes into the webapps\Ch09\WEB-INF\classes directory, so use that directory as the output directory. After entering the code, compile the project to create *Ch09_04.class*.

In JSP, you can connect to the bean code in a variety of ways, including instantiating an object of the Ch09_04 class using Java in a JSP scriptlet. The recommended way of doing things, however, is to use the JSP jsp:useBean element to create a JavaBean object. Then use the jsp:getProperty element to get the value of a bean property, and use the jsp:setProperty element to set a bean property's value. You can see how this works in Example 9-7, where we're reading the current property of the bean's msg property, setting it to a new value, and reading that new value.

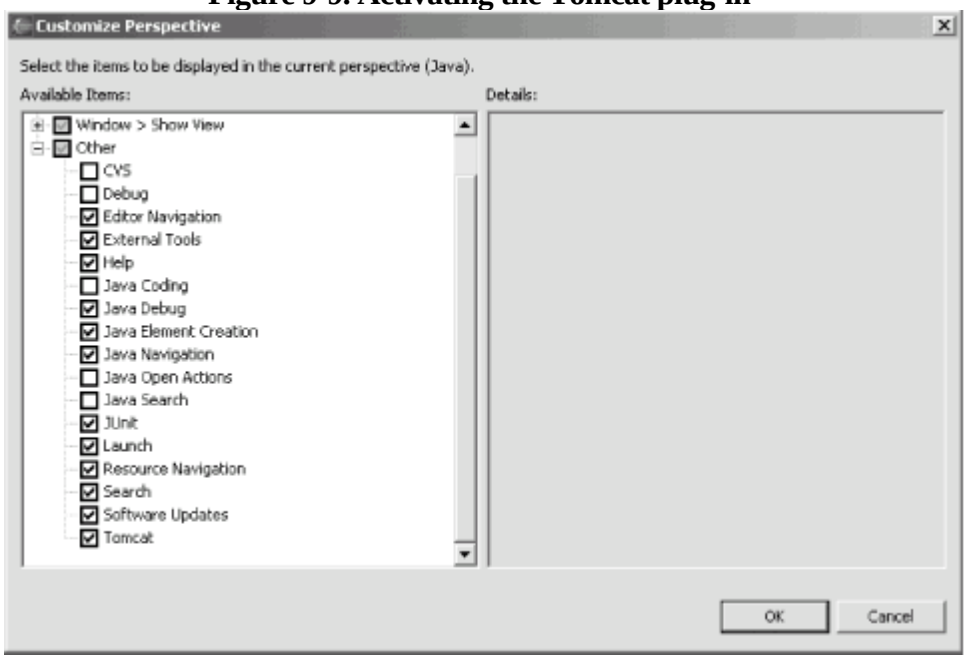
Example 9-7. Connecting to a JavaBean

```
<HTML>

    <HEAD>
```

9.6 Using the Sysdeo Tomcat Plug-in

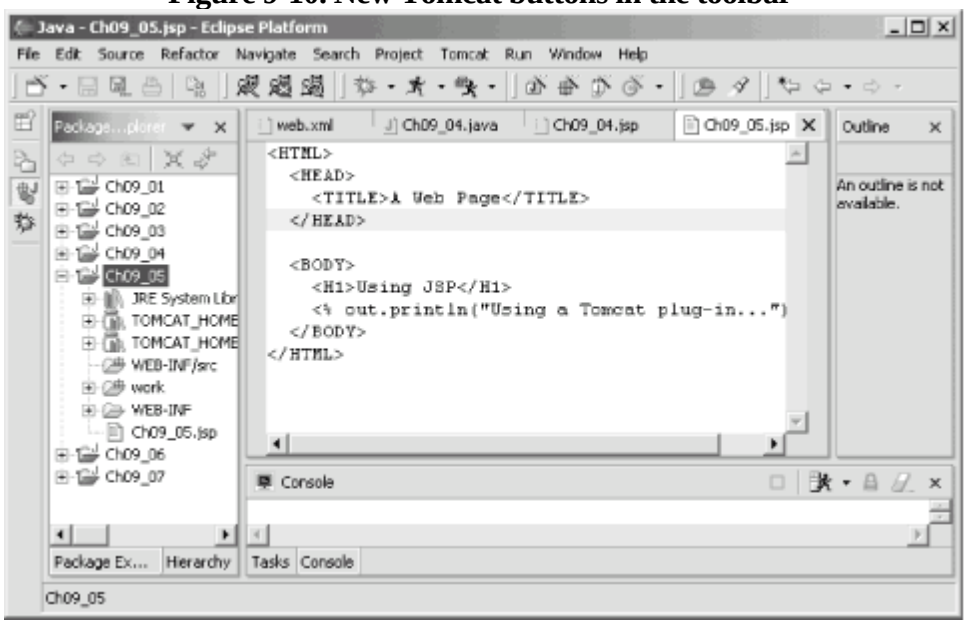
The Sysdeo plug-in lets you start and stop Tomcat from inside Eclipse, and we'll take a look at using that plug-in here. You can download this plug-in for free from http://www.sysdeo.com/eclipse/tomcatPlugin.html. After expanding it in the plugins directory, activate it by selecting Window⟶ Customize Perspective, opening the Other node, and selecting the Tomcat item, as you see in Figure 9-9.

Figure 9-9. Activating the Tomcat plug-in



This adds a Tomcat menu to Eclipse (shown in Figure 9-10) and adds three Tomcat buttons to the Eclipse toolbar that you can see under the Navigate menu; these buttons start, stop, and restart Tomcat.

Figure 9-10. New Tomcat buttons in the toolbar



To connect the Tomcat plug-in to the version of Tomcat you're using, select Window⟶ Preferences and click the Tomcat node, as you see in Figure 9-11.

Figure 9-11. Connecting the Tomcat plug-in to Tomcat

9.7 Deploying Web Applications

The last topic we'll take a look at in this chapter is all about deploying web applications. This process is easy enough—all you've got to do is create a compressed Web Archive (WAR) file of your application and drop it into the Tomcat webapps directory. The next time Tomcat is restarted, it'll expand that WAR file automatically, deploying the application.

As an example, we're going to deploy the servlet you see in Example 9-11. To make the WAR file creation easier, create this new project, Ch09_07, in its own folder, Ch09_07, in the webapps directory. Make this project a standard Java project (not a Tomcat project). Be sure you give the Ch09_07 folder its own WEB-INF directory with the subdirectories classes and lib, and make the classes folder the output folder for the project. Now create Ch09_07.java as we have before, and, after entering the servlet's code into Ch09_07.java, build the project.

Example 9-11. A servlet to deploy

```
package org.eclipsebook.ch09;


import javax.servlet.http.HttpServlet;


import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;


public class Ch09_07 extends HttpServlet {

        public void doGet(HttpServletRequest request,

                HttpServletResponse response)

                throws IOException, ServletException

        {

                response.setContentType("text/html");

                PrintWriter out = response.getWriter( );


                out.println("<HTML>");

                out.println("<HEAD>");

                out.println("<TITLE>");

                out.println("A Web Page");

                out.println("</TITLE>");

                out.println("</HEAD>");

                out.println("<BODY>");

                out.println("<H1>");
```

Chapter 10. Developing Struts Applicationswith Eclipse

In this chapter we're going to take a look at using Eclipse to write Struts-based web applications. This is going to give us experience not only using Struts, but also creating large-scale web applications, including handling build dependencies (where one file needs to be built before another), avoiding deleting files in the output folders when doing a full build (deleting all files in the output folder is called "scrubbing," which Eclipse does by default—and scrubbing a Struts application would delete needed files), organizing your source code files into a folder after the project has already been created, and other issues. We're also going to take a look at a popular Struts plug-in, Easy Struts, to help create Struts applications in Eclipse.

Although we're going to discuss Struts in overview here, we'll assume that if you want to follow the programming in detail, you already have some experience with Struts—in this book, our focus is on Eclipse, not Struts.

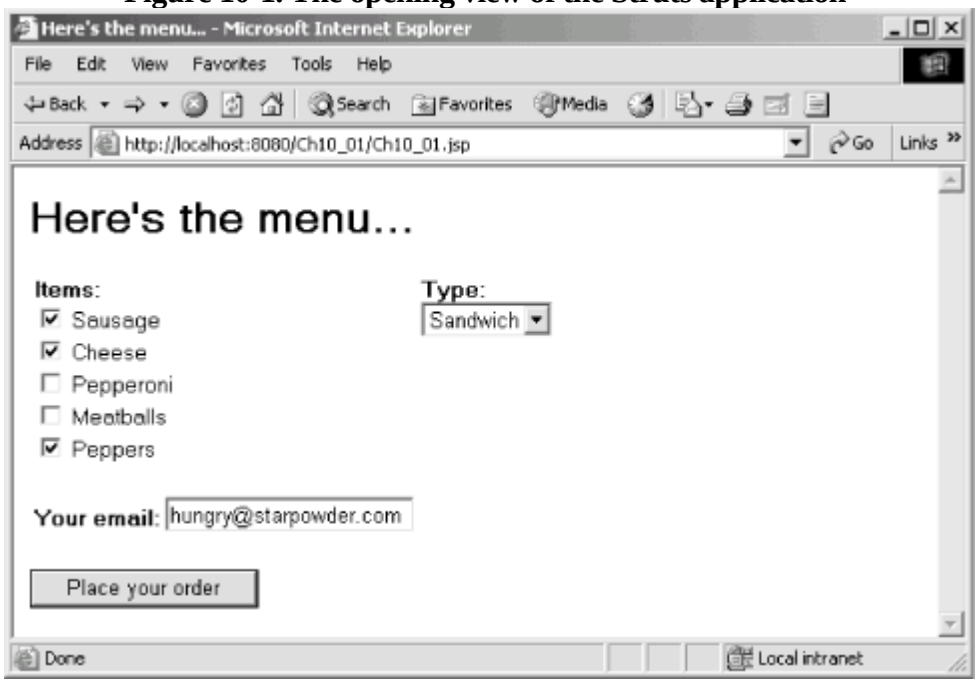
Struts is built on Model-View-Controller (MVC) architecture that has become popular in servlet/JSP programming. The original servlet/JSP programming architecture (sometimes called Model 1) was somewhat ad-hoc, using servlets, JSPs, and beans in a way that was completely up to the programmer. Since that time, web applications have become more large-scale, and the MVC architecture (sometimes called Model 2) has been adopted. In MVC programming, the view (often a JSP) handles the visual interface with the user, the model (often a JavaBean) handles the internal logic of the application, and the controller (often a servlet) handles the overall communication between the view and the model as well as forwarding user reqests as needed to other code.

In Struts terminology, the view is constructed using *forms* and the controller with actions. The model is often implemented with form beans.

10.1 Struts and Eclipse

We'll start by taking a look at the Struts example we're going to create using Eclipse, using the latest version of Tomcat available (Version 4.1.29 as of this writing), and the latest version of Struts (Version 1.1). When you navigate to the opening JSP, http://localhost:8080/Ch10_01/Ch10_01.jsp, you'll see a menu with a number of HTML controls, as appears in Figure 10-1.

Figure 10-1. The opening view of the Struts application



The user can order a pizza or sandwich using the controls here and can also include his email address. Clicking the "Place your order" button sends the data in the form to an underlying bean and to the application's controller servlet, which displays a summary of the order, as you see in Figure 10-2.

Although these results look pretty simple, that's not to say that the implementation of this application is easy. Struts is not a lightweight framework; to create this example, you use these files, arranged in the Tomcat *webapps/Ch10_01* directory:

```
webapps

|

|_ _Ch10_01

   |   Ch10_01.jsp [View: Starting form]

   |   Ch10_05.jsp [View: Summary form]

   |

   |_ _WEB-INF

      |   struts-config.xml [Struts Configuration File]

      |   Ch10.tld  [Custom Tag Definitions]

      |   Struts TLD files [Struts Tag Definitions]

      |   web.xml  [Application Descriptor File]
```

10.2 Creating the View

The first file that the user navigates to is the view file *Ch10_01.jsp*. In this file, we use various custom Struts tags to implement the display you see in Example 10-1. For example, the <html:form> tag creates a Struts-enabled form that can display controls, as you see in Example 10-1; we're setting the form's action attribute to the name we'll give the controller, *Ch10_04.do*, so when the user clicks the Submit button (with the caption "Place your order"), the data in the form will be forwarded to the controller.

Example 10-1. A sample JSP

```
 <%@ page contentType="text/html;charset=UTF-8" language="java" %>

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<%@ taglib uri="/Ch10" prefix="Ch10" %>


<HTML>

    <HEAD>

        <TITLE>Here's the menu...</TITLE>

    </HEAD>


    <BODY>

        <H1>Here's the menu...</H1>

        <html:errors/>

        <Ch10:type/>

        <Ch10:items/>


        <html:form action="Ch10_04.do">

            <TABLE>

                <TR>

                    <TD ALIGN="LEFT" VALIGN="TOP">

                        <bean:message key="items"/>

                        <BR>

                        <logic:iterate id="items1" name="items">

                            <html:multibox property="items">

                                <%= items1 %>

                            </html:multibox>
```

10.3 Creating the Controller

We've set things up in the view so the data in the HTML controls is sent to *Ch10_04.do*. That's the controller, or action servlet, in our application. We'll connect the extension *.do* to the action servlet for the application in *web.xml*. You can see how that's done in *web.xml* in Example 10-5, using the <servlet> and <servlet-mapping> elements, much as we did in the previous chapter.

Example 10-5. web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<!DOCTYPE web-app

  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"

  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">


<web-app>

        <display-name>Struts Example Application</display-name>


  <!-- Action Servlet Configuration -->

  <servlet>

    <servlet-name>action</servlet-name>

    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

    <init-param>

      <param-name>application</param-name>

      <param-value>ApplicationResources</param-value>

    </init-param>

    <load-on-startup>2</load-on-startup>

  </servlet>


  <!-- Action Servlet Mapping -->

  <servlet-mapping>

    <servlet-name>action</servlet-name>

    <url-pattern>*.do</url-pattern>

  </servlet-mapping>


  <taglib>

    <taglib-uri>/Ch10</taglib-uri>
```

10.4 Creating the Model

We're almost done with the code for the project; the final two code files are the summary page that displays the data the user entered, *Ch10_05.jsp*, and the bean that acts as the model and stores the data the user entered, *Ch10_06.java*. In the summary page, *Ch10_05.jsp*, we use the data passed on to us in a bean object and various Struts tags to display that data, as you see in Example 10-9. This file goes into the deployment directory, the same as the original view, *Ch10_01.jsp*.

Example 10-9. The JSP to display order information

```
 <%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>


<HTML>

    <HEAD>

        <TITLE>Here is what you ordered...</TITLE>

    </HEAD>


    <BODY>

        <H1>Here is what you ordered...</H1>

        <bean:message key="type"/>

        <bean:write name="Ch10_06" property="type"/>

        <BR>

        <BR>

        <bean:message key="items"/>

        <BR>

        <logic:iterate id="items1" name="Ch10_06" property="items">

            <%= items1 %>

            <BR>

        </logic:iterate>

        <BR>

        <bean:message key="email"/>

        <bean:write name="Ch10_06" property="email"/>

        <BR>

    </BODY>

</HTML>
```

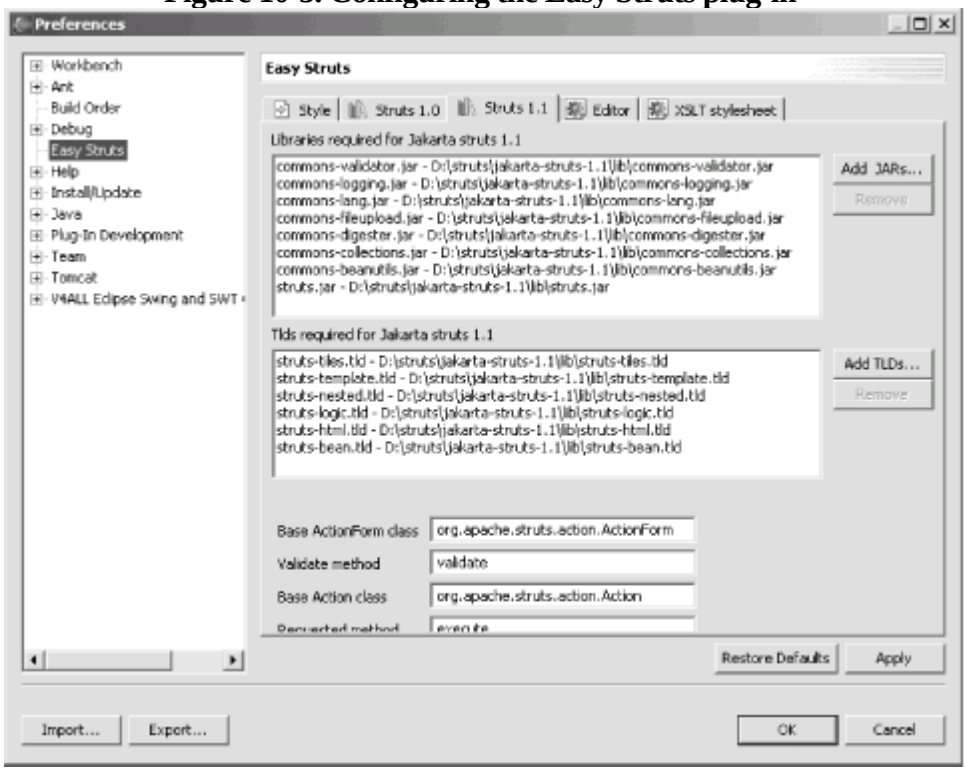
10.5 Using the Easy Struts Plug-in

The Easy Struts plug-in is available for free from http://sourceforge.net/projects/easystruts. This plug-in adds Struts support to a project and has a number of code-generation wizards that let you create actions, forms, and so on.

To use this plug-in, Easy Struts recommends that you start by creating a Tomcat project using the Sysdeo Tomcat plug-in we looked at in Chapter 9 (which means that we'll use an older version of Tomcat in this example, as we did when using the Tomcat plug-in in the previous chapter). Call this Tomcat project Ch10_02.

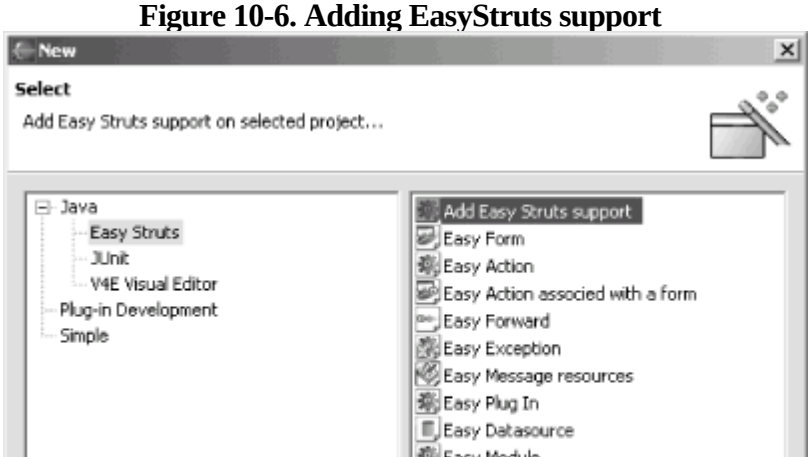
To work with Easy Struts, you start by configuring the plug-in. Select Window⟶ Preferences, then select the Easy Struts item and the Struts 1.1 tab. Add the Struts JAR files and TLD files so the plug-in knows where to find them, as you see in Figure 10-5. Accept the other defaults by clicking OK.

Figure 10-5. Configuring the Easy Struts plug-in



To add Struts support to the Ch10_02 project, right-click that project and select New⟶ Other, then select the Easy Struts item in the left list and Add Easy Struts support in the right list, as shown in Figure 10-6.

Figure 10-6. Adding EasyStruts support



Chapter 11. Developing a Plug-in: The Plug-in Development Environment, Manifests, and Extension Points

In this chapter and the next, we're going to start modifying Eclipse itself. So far, we've been using it as we've downloaded it, adding a few prebuilt plug-ins as needed. Now we're going to start creating our own plug-ins. Eclipse was built as an extensible IDE, and we're going to extend it.

As you know, plug-ins are stored in folders in the plugins directory, and that's where the ones we create will go. These folders typically have names like org.eclipse.swt.win32_2.1.1 or org.junit_3.8.1, where the folder name is the plug-in name, followed with an underscore and a version number. When Eclipse loads a plug-in that has several folders in the plugins directory, it checks the version number in order to load only the most recent version.

You'll typically find the following files in every plug-in's folder:

**.jar*

Java code for the plug-in

about.html

Displayed when the user asks for info about the plug-in

plugin.properties

Holds string data used by *plugin.xml*

plugin.xml

Plug-in manifest that describes the plug-in to Eclipse

lib

Holds additional JAR files

icons

Directory for icons (GIF format is standard)

We'll see these various files as we develop plug-ins, but we'll start with the minimum needed. The *plugin.xml* file is the plug-in manifest, which tells Eclipse about the plug-in. In fact, that's the only file you really need to create a plug-in as far as Eclipse is concerned.

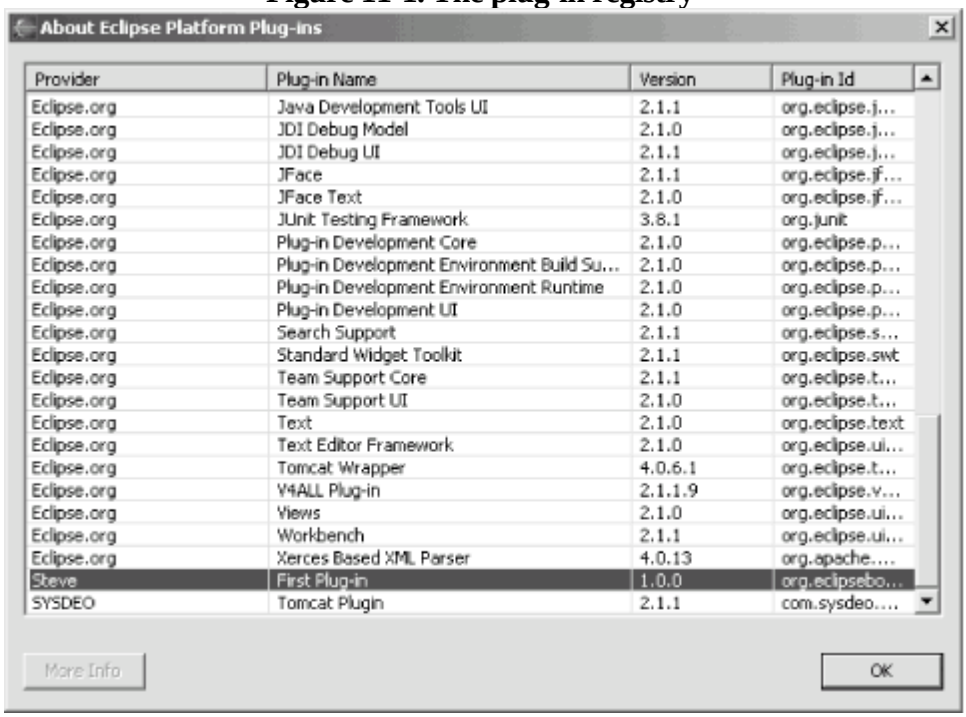
11.1 All You Really Need Is plugin.xml

Creating a very simple plug-in is easy enough—all you need is a working *plugin.xml*. To show how this works, you can use a text editor to create a new plug-in manifest, *plugin.xml*, for a fictional plug-in named org.eclipsebook.first. In this case, we'll set the plug-in's name, ID, version number, and the name of its provider like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<plugin

    id="org.eclipsebook.first"

    name="First Plug-in"

    version="1.0.0"

    provider-name="Steve">

</plugin>
```

Just store *plugin.xml* in plugins\org.eclipsebook.first_1.0.0 and restart Eclipse. You can find the new plug-in in the plug-in registry, which is where Eclipse holds data about all current plug-ins. To see that data, select Help → About Eclipse Platform and click the Plug-in Details button, opening the About Eclipse Platform Plug-ins dialog you see in Figure 11-1.

Figure 11-1. The plug-in registry



You can see the new, fictional plug-in in the registry, near the bottom.

If an Eclipse project has a plug-in manifest, it's considered a plug-in project.

That's a nice exercise but it doesn't go very far in extending Eclipse. Developing a real plug-in involves creating multiple files, including multiple code files. The good news is that Eclipse has built-in wizards to help you out.

11.2 Using the Plug-in Development Environment

The Eclipse platform is already a conglomeration of over a hundred plug-ins, and they build on each other using *extension points*. An extension point lets one plug-in build on what another plug-in exports. In this chapter, we're going to use extension points to add new menus, buttons, and so on to Eclipse in a plug-in.

Plug-ins can only make use of classes exported by other plug-ins, which makes extension points especially important. For example, to let a plug-in make use of prebuilt Java code, you can wrap JAR files inside plug-ins and let other plug-ins depend on it. Much support for custom plug-ins is already built into several standard plug-ins that come with Eclipse.

Using the Eclipse Plug-in Development Environment (PDE), you can build plug-ins that will build on the standard extension points available. Here are the types of plug-in projects that the PDE will create for you:

Plug-in projects

A standard plug-in

Fragment projects

An add-on or addition to a plug-in (sometimes used for internationalization)

Feature projects

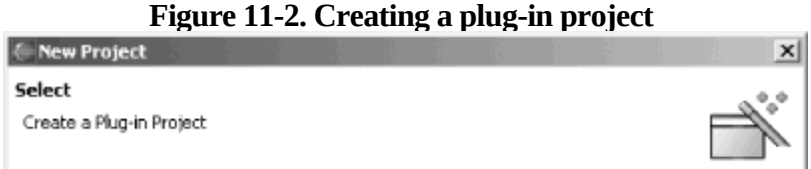
Projects that contain one or more plug-ins

Update site projects

Web site that can automatically install features

The PDE has a number of built-in Wizards, and we're going to make use of that support in our next example. This example will create a simple plug-in that supports both a menu item and a button in the toolbar. To create the plug-in project, select File → New → Project. Select Plug-in Development in the left box of the New Project dialog and Plug-in Project in the right box, as shown in Figure 11-2. Then click Next.

Figure 11-2. Creating a plug-in project

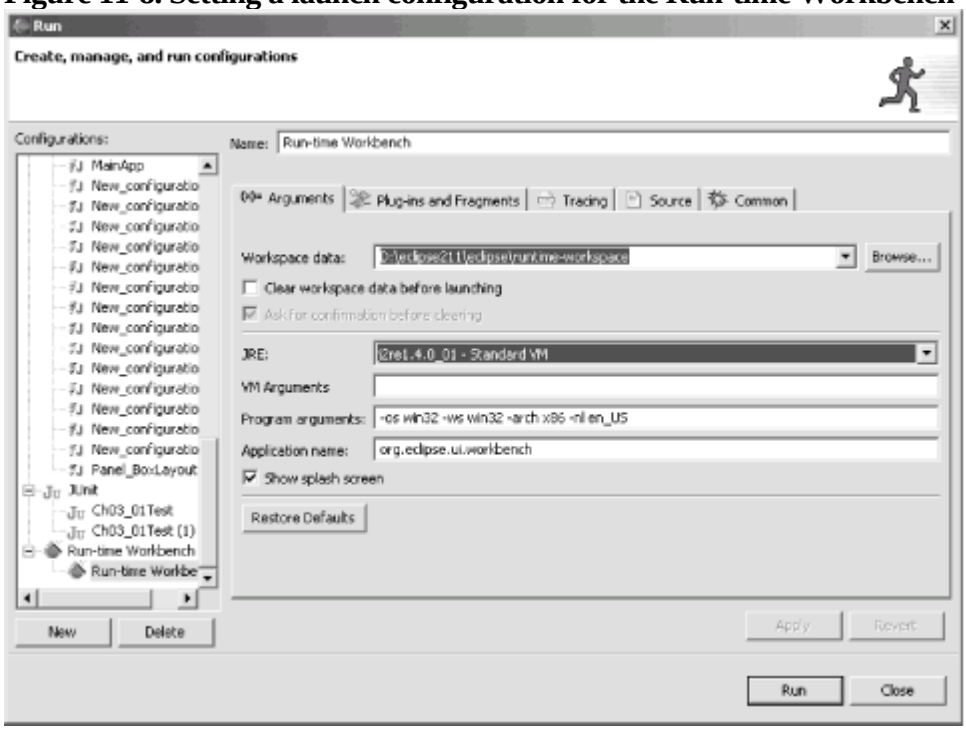


11.3 Using the Run-time Workbench

At this point, you could create *Ch11_01.jar*, and install that JAR file, *plugin.xml*, and so on in the plugins directory and you'd be ready to go—the new plug-in would support a menu item and a toolbar button. But for testing purposes, Eclipse offers a shortcut: the Run-time Workbench. This is a workbench that you can launch from Eclipse and use to test plug-ins.

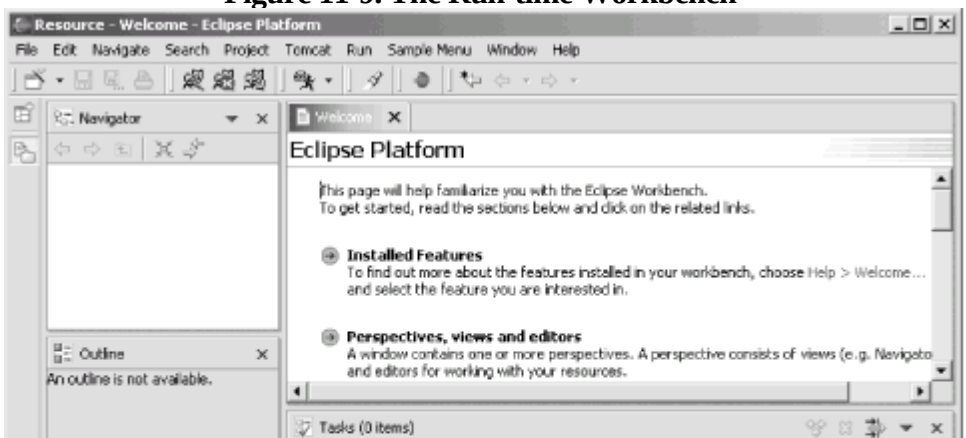
To run the Run-time Workbench, select the plug-in project in the Package Explorer, select Run⟶ Run, open the Run-time Workbench node in the left box, and select the Run-time Workbench item under that node, as you can see in Figure 11-8. Then select the JRE you want to use when working with the Run-time Workbench (you don't need to do this step if you have an installed JRE you've named JRE2, which is what the Run-time Workbench seaches for by default). Then click Run. (The next time, you can simply select Run⟶ Run As⟶ Run-time Workbench.)

Figure 11-8. Setting a launch configuration for the Run-time Workbench



This starts the Run-time Workbench, but we're not done yet. To see the results of this plug-in, select Window⟶ Customize Perspective⟶ Other and select the checkbox for the Sample Action Set item, which is defined by our plug-in, and click OK. This adds the menu defined by the new plug-in, Sample Menu, to the menu bar and a button with the Eclipse icon to the toolbar just under that menu, as you can see in the Run-time Workbench in Figure 11-9.

Figure 11-9. The Run-time Workbench



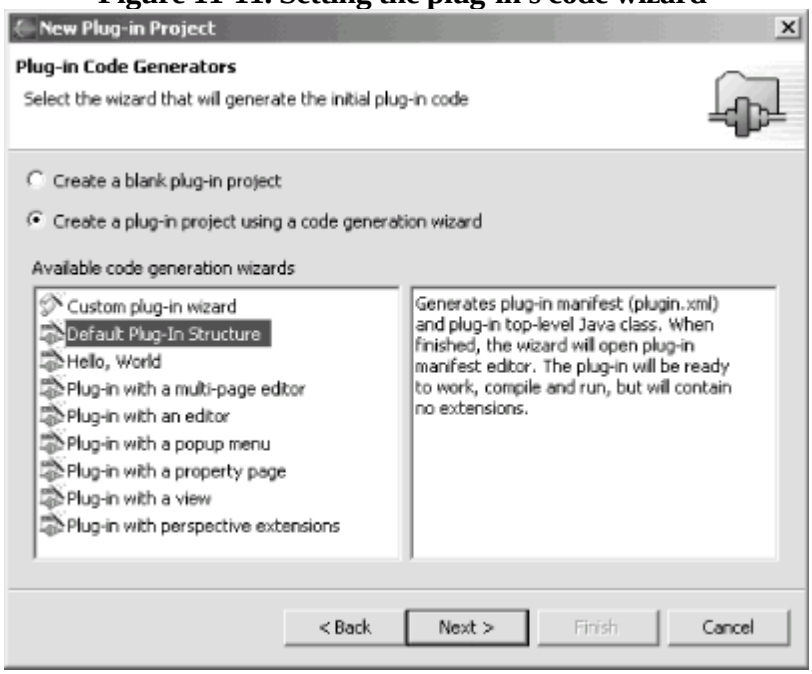
11.4 Creating a Standard Plug-in

In this example, we're going to be responsible for creating our own plug-in. Like the previous example, this plug-in is going to have its own menu item and toolbar button, but this time, we're going to do the legwork ourselves.

Start by selecting File ⟶ New ⟶ Project. In the New Project dialog, select Plug-in Development and Plug-in Project, and click Next. In the next pane, enter the new project's name, org.eclipsebook.ch11.Ch11_02, and click Next to open the Plug-in Project Structure pane. Click Next again to accept the defaults.

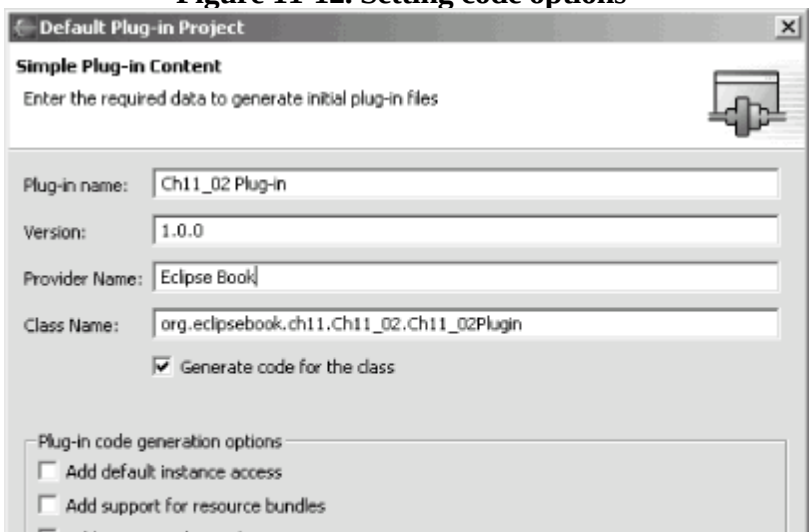
In the Plug-in Code Generators pane, select the Default Plug-In Structure item as shown in Figure 11-11, and click Next.

Figure 11-11. Setting the plug-in's code wizard



In the following pane, the Simple Plug-in Content pane shown in Figure 11-12, set the provider name—we'll use Eclipse Book in this example. We won't need the convenience methods the wizard can generate for us, so deselect the checkboxes in the "Plug-in code generation options" box. Then click Finish.

Figure 11-12. Setting code options



Chapter 12. Developing a Plug-in: Creating Editors and Views

In this chapter, we're going to create custom editors and views in plug-ins. Plug-in development is a huge topic by itself, and it can take dozens of files to create a commercial plug-in. Fortunately, the Eclipse PDE comes with a number of wizards that will write up the plug-in's code framework for you, saving significant time. We'll see how to use those wizards in this chapter, rewriting the code they generate to make the plug-in do what we want.

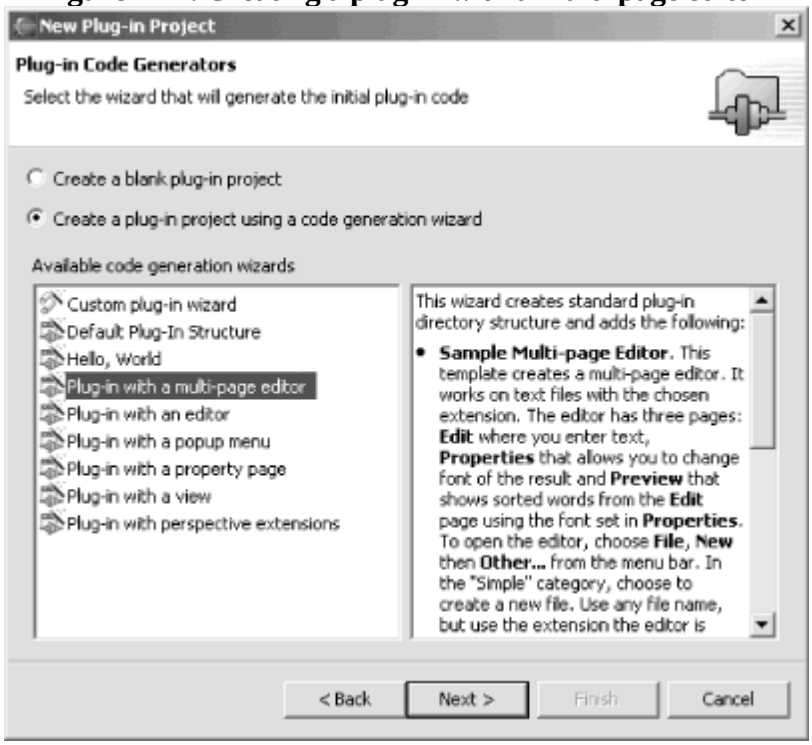
12.1 Creating a Multi-Page Editor

The first example in this chapter is going to create a multi-page editor associated with the file extension we're going to specify: .xyz. When the user double-clicks a file with that extension in the Package Explorer, Eclipse will use our editor to open and edit it. That editor will have two tabs corresponding to its two pages—the default tab will display the text contents of the file, and the Sorted tab will display those contents sorted in alphabetical order. You can create files with the .xyz extension using the New⟶ File menu item, but this plug-in example is also going to have a built-in wizard that will create .xyz files for the user and place default text in them.

12.1.1 Creating the Code

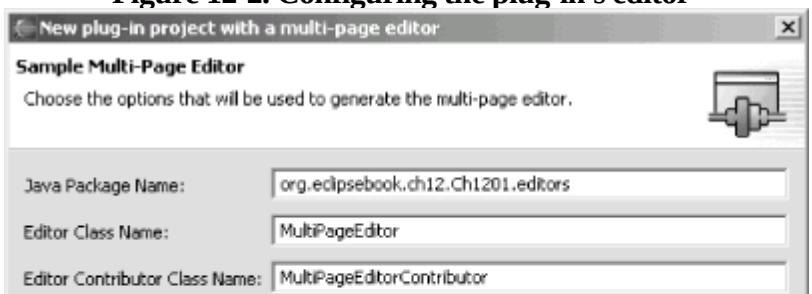
To create the code skeleton for this example, select New⟶ Project, and in the New project dialog, select Plug-in Development in the left box, Plug-in Project in the right box, and click Next. Give the name of the project as org.eclipsebook.ch12.Ch12_01 in the following pane, and click Next. Leave the defaults selected in the Plug-in Project Structure pane to make this a Java project, and click Next again to bring up the Plug-in Code Generators pane shown in Figure 12-1.

Figure 12-1. Creating a plug-in with a multi-page editor



Select the Plug-in with a multi-page editor item, as you see in the figure, and click Next. In the following pane, set the Provider name to Eclipse Book and click Next again. In the next pane that appears, set the File Extensions item to xyz to associate the plug-in with that extension, as you see in Figure 12-2, and click Next again.

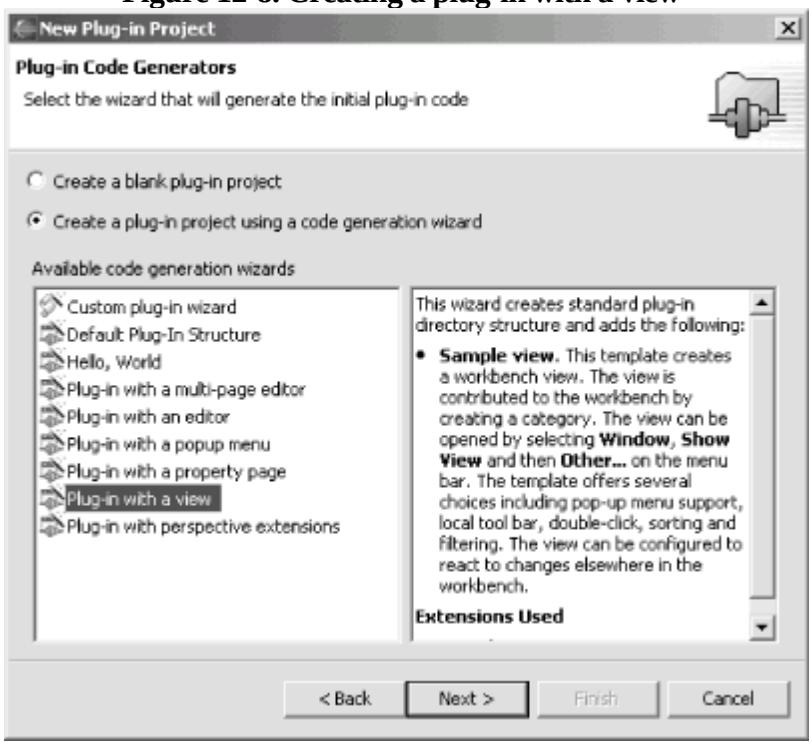
Figure 12-2. Configuring the plug-in's editor



12.2 Creating a View

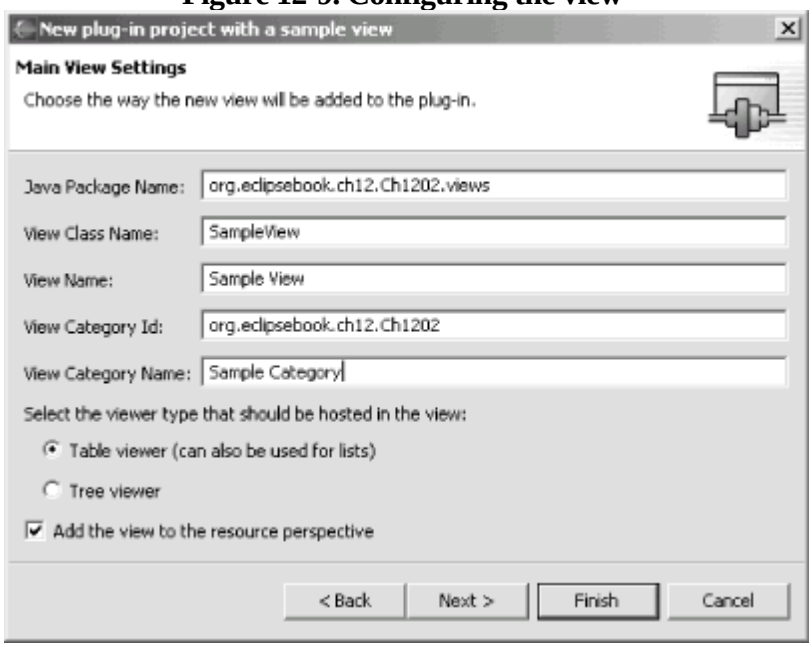
In the next example, we're going to use a plug-in to create a new view. Start by creating a new plug-in project named org.eclipsebook.ch12.Ch12_02. In the Plug-in Code Generators pane, which you see in Figure 12-8, select the "Plug-in with a view" wizard and click Next.

Figure 12-8. Creating a plug-in with a view



In the following pane, give Eclipse Book as the provider's name and click Next again to bring up the Main View Settings dialog you see in Figure 12-9. In this pane, you set the name of the view and its category—we'll stick with the defaults, which will make this a table-based view (i.e., the items in the view will be displayed in a table control) named Sample View in the category Sample Category. Click Next again to bring up the final pane of this wizard.

Figure 12-9. Configuring the view

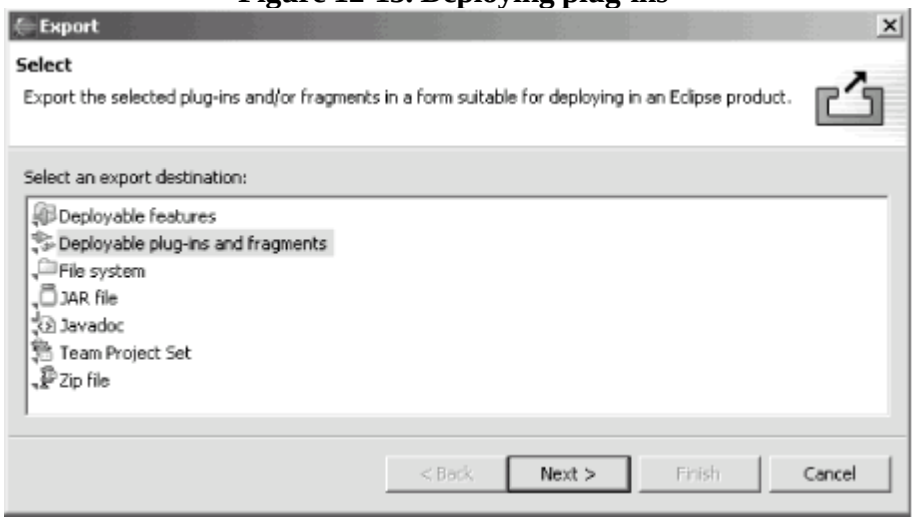


The last pane, shown in Figure 12-10, lets you configure the view's actions, such as responding when the user

12.3 Deploying a Plug-in

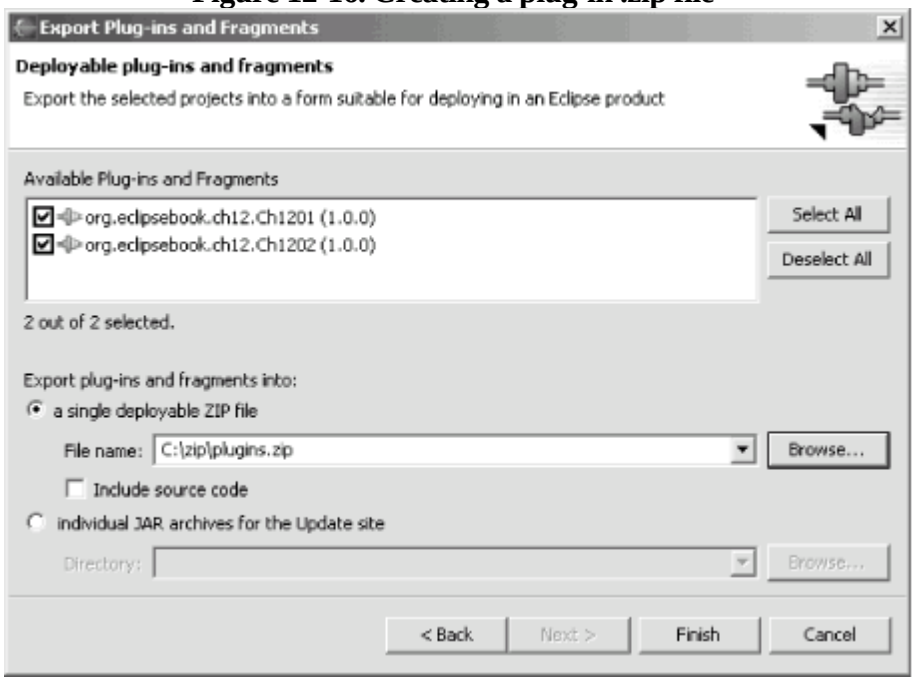
Eclipse makes it relatively simple to create deployment packages for plug-ins. To deploy a plug-in, select File→Export, selecting the "Deployable plug-ins and fragments" item in the Export dialog, as you see in Figure 12-15.

Figure 12-15. Deploying plug-ins



Click Next to open the "Deployable plug-ins and fragments" pane you see in Figure 12-16. You can deploy plug-ins as either *.jar* or *.zip* files; here, we'll deploy both plug-ins in a *.zip* file. Enter the fully qualified name of the *.zip* file to create—*plugins.zip*—in the File name box and select both plug-in projects from this chapter.

Figure 12-16. Creating a plug-in .zip file



This packages both plug-ins (including a *plugin.xml* and a *.jar* file for each) in the plugins.zip file. Unzipping that file in the Eclipse distribution directory installs the plug-ins—the two files for the first plug-in will be installed in eclipse\plugins\org.eclipsebook.ch12.Ch1201_1.0.0, and the files for the second plug-in will be installed in eclipse\plugins\org.eclipsebook.ch12.Ch1202_1.0.0.

Chapter 13. Eclipse 3.0

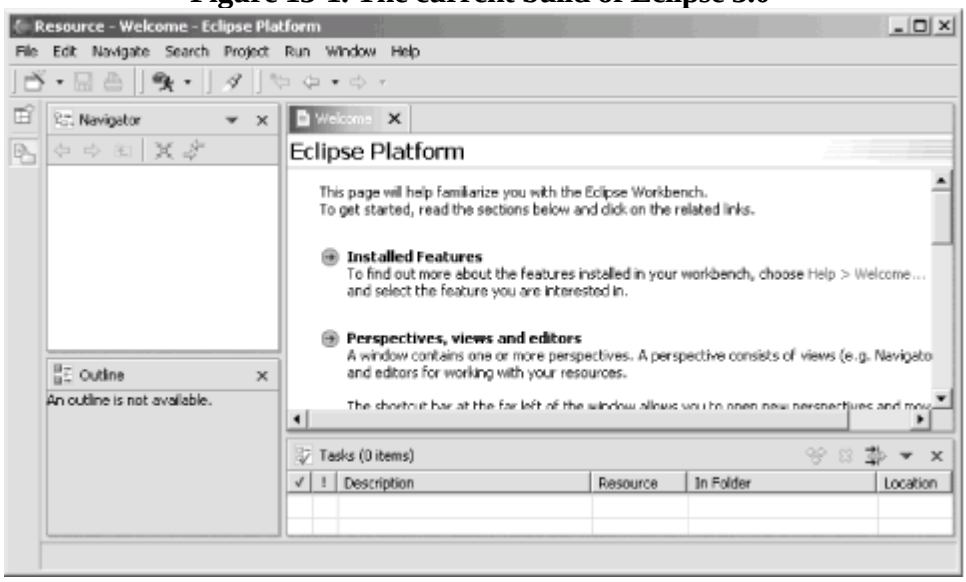
Eclipse 3.0 is on the way, and we'll get a look at what's coming in this chapter. As of this writing, 3.0 is still in beta version.

This chapter is going to use the most recent beta of Eclipse 3.0, milestone build 6. Eclipse 3.0 is being designed to be as compatible as possible with Versions 2.0 and 2.1, and the update team says, "We will provide a comprehensive Eclipse 3.0 Porting Guide that covers all areas of breaking API changes, and describes how to port existing 2.1 plug-ins to 3.0." You'll also be able to import 2.0 and 2.1 projects into 3.0, but you'll get a dialog saying that the project needs to be modified and probably won't work with earlier versions after that, so make sure you copy your projects first.

13.1 A Look at Eclipse 3.0

At this point, Eclipse 3.0 looks a lot like Eclipse 2.1, as you can see in Figure 13-1.

Figure 13-1. The current build of Eclipse 3.0



As we're going to see, some new buttons, menu choices, and views have appeared, but fundamentally, Eclipse 2.0 users will have no difficulty slipping right into this new version of Eclipse.

On the other hand, the Eclipse team has been experimenting with the look of views, editors, and perspectives in Eclipse 3.0, and there's no guarantee that Eclipse 3.0 will keep looking like Figure 13-1.

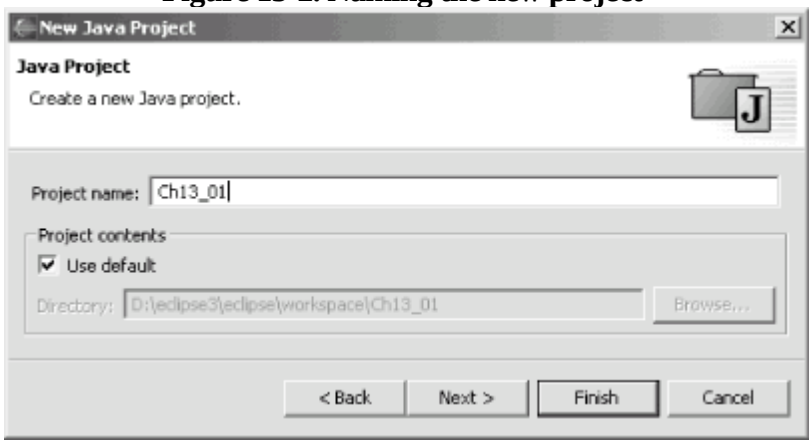
Some of the changes the Eclipse team is contemplating include rounding view tabs instead of the square ones currently in use, only presenting a single editor tab at once instead of stacking them (you can reach other editor tabs with arrow buttons), and displaying icons for perspectives at top right, not at left.

13.2 Creating a Java Project

For Java programmers, the fundamental Eclipse task is creating a Java project. As you're going to see, the process is (so far) virtually identical to working in Eclipse 2.1.

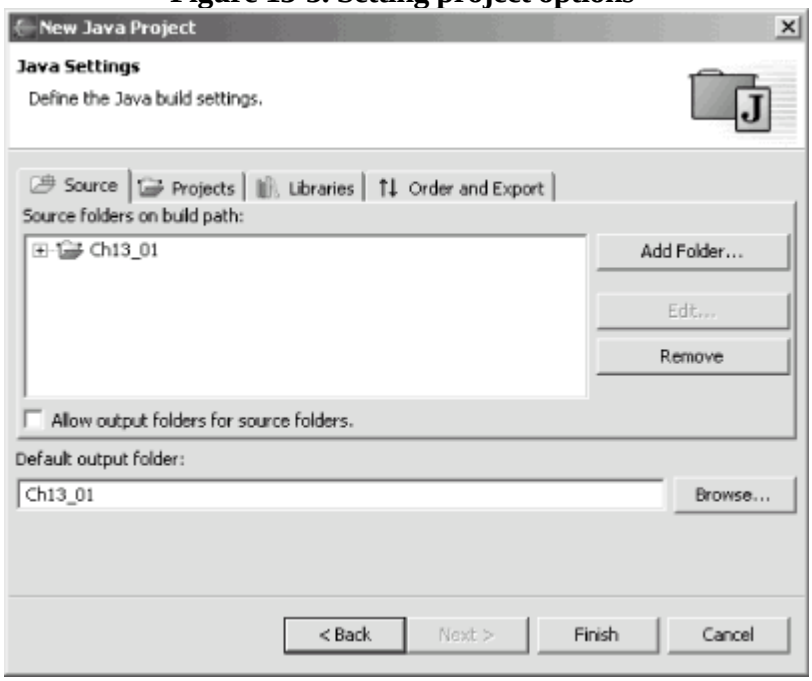
To create a new project, right-click the Navigator view and selectNew → Project, opening the New Project dialog. Select the Java item in the pane on the left and Java Project in the pane on the right and click Next. These dialogs you see in Figure 13-2 are just as we've seen before. Name this new project Ch13_01 and click Next.

Figure 13-2. Naming the new project



This brings up the familiar dialog you see in Figure 13-3, where you can set project settings, such as the source folder and the import and export order—all items we're already familiar with. Click Finish to create the project.

Figure 13-3. Setting project options



Eclipse 3.0 will ask if you want to switch to the Java perspective; click OK. The new Java perspective looks much like what we've seen before, except that the Tasks view has now been renamed the Problems view, as you see in Figure 13-4.

Figure 13-4. The Eclipse 3.0 Java perspective

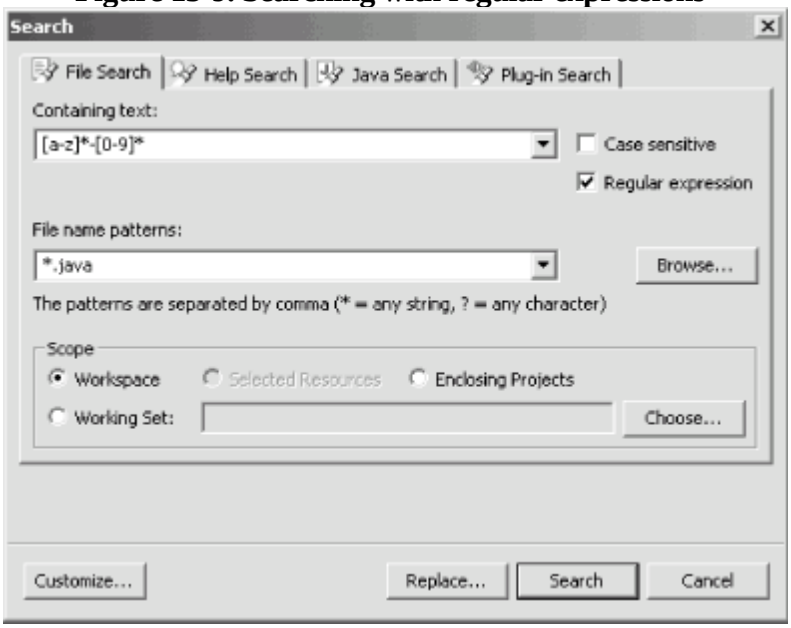
13.3 Changes to the Eclipse Platform

There are a number of changes to the Eclipse platform in Eclipse 3.0. Many of the changes take place behind the scenes—for example, when resources are changed, auto-builds now occur in the background so you don't have to wait, and there's a new Progress view that lets you keep track of those changes. Disabled features can now be uninstalled to free disk space (select Help ⟶ Software Updates ⟶ Manage Configuration, turn off the disabled feature filter, and select Uninstall from the context menu of the disabled feature to uninstall it).

Now you can have Eclipse automatically check for updates—check out the Window ⟶ Preferences ⟶ Install/Update ⟶ Automatic Updates preference page.

Some of the changes are more evident, however. One of the most handy changes is that you can now use regular expressions in the File search page. Select Search ⟶ File to open the dialog you see in Figure 13-9—note the new "Regular expression" checkbox.

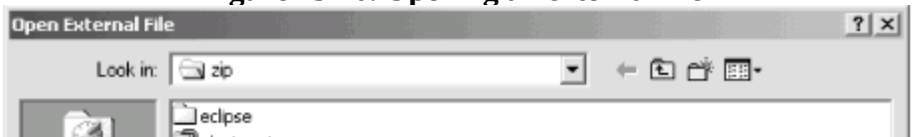
Figure 13-9. Searching with regular expressions



Not familiar with regular expressions? Click the "Regular expression" checkbox and press F1 for an overview of how to use regular expressions to match text.

You can also edit files outside the workspace now, using the new File ⟶ Open External File menu item, which opens the dialog you see in Figure 13-10. Select a file and click Open to open it in an external editor.

Figure 13-10. Opening an external file



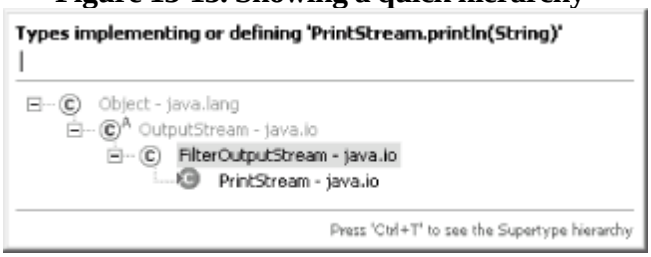
13.4 Changes to the Java Development Tools

The major enhancements in Eclipse 3.0 are to the JDT, as you might expect. The changes so far are mostly for usability and convenience, and they're designed to help you program Java better. It's not clear which ones will be around in the final release of Eclipse 3.0, but we'll take a look at what's available now.

13.4.1 Quick Hierarchy Views

You can now select a type, method, or package reference in the JDT editor and press Ctrl+T to see a quick type hierarchy view, as appears in Figure 13-13.

Figure 13-13. Showing a quick hierarchy



You can also now open a view that shows a method call hierarchy by selecting Navigate⟶ Open Call Hierarchy (or by pressing Ctrl+Alt+H) in the JDT editor (or, for that matter, any of the Java views that show methods).

13.4.2 Creating Constructors from Fields

You can also create a constructor that fills various fields easily in Eclipse 3.0. For example, say you added a String field to a class this way:

```
public class Ch13_01 {


        private String text;


        public static void main(String[] args) {

                System.out.println("Hello from Eclipse 3.0");

        }


}
```

To create a constructor that fills this field in Eclipse 3.0, select Source⟶ Generate Constructor Using Fields, opening the dialog you see in Figure 13-14.

Figure 13-14. Creating a constructor using a field

13.5 Other Changes

There are some other changes to various other subprojects in Eclipse 3.0, including CVS and Ant control as well as the PDE and SWT.

Eclipse now ships with the latest version of Ant, which is Version 1.5.4. The Ant launch configuration is all new, letting you work with Ant in a separate Java VM (you can set the options for choosing the Ant JRE on the JRE tab of the Ant launch configuration).

As far as CVS goes, you can now use the Team → Show Annotation item to open the CVS Annotation view. If you select a line in the JDT editor, the CVS Annotation view will indicate who released the change—which means you can track who released a bug.

There have been a couple of improvements to SWT as well. SWT now lets you embed Swing and AWT widgets inside SWT on other platforms besides Windows. As of this writing, however, this only works in Windows with JDK 1.4 and above, and on GTK and Motif with early versions of Sun JDK 1.5.

SWT shells can now have irregular shapes as well. You can define these shapes with combinations of rectangles and polygons. And SWT has a new browser widget that can display HTML documents. As of this writing, however, this widget is only supported in Windows (and uses Internet Explorer 5.0 or above) and Linux GTK (where it uses Mozilla 1.4 GTK2).

There aren't many changes to the PDE in the current milestone build of Eclipse 3.0. Two significant ones are that the PDE now supports JUnit testing for plug-in development, and that there is also a special build configuration editor that lets you edit a plug-in's build.properties file.

Other changes that are being contemplated include improvements to the PDE editors and debuggers, as well as supporting context-sensitive help for plug-ins.

That's it for our look at Eclipse 3.0. There's little question that there will be other changes before the final version is released, so keep your eyes peeled.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animals on the cover of Eclipse are ornate butterflyfish (Chaetodon ornatissimus). Ornate butterflyfish are easily recognized by their white skin marked with orange diagonal, parallel stripes. A black eye-band runs vertically down the head to conceal the eye-an adaptation that confuses predators as to the direction the fish will flee when attacked. Butterflyfish have laterally compressed bodies that enable them to swim stealthily through coral crevices. These reef-dwelling fish are native to the tropical marine waters of the Indo-Pacific, inhabiting both shallow lagoons and seaward reefs.

Mature butterflyfish are characteristically monogamous and travel in mated pairs. During the day, the home-ranging pairs search for food; at night, they sleep hidden in reef recesses. Adults usually spawn at dusk, rising 30 to 50 feet above their habitats into the water column, where they release a white cloud of gametes before quickly returning to the bottom. The abandoned, tiny, buoyant, fertilized eggs are dispersed by the currents. Once hatched, usually within 30 hours of fertilization, the larvae are protected by bony armor, which is shed during the juvenile stage. Juveniles are solitary until they reach sexual maturity, about a year after birth.

Ornate butterflyfish have short jaws and brush-like teeth for nipping off the coral polyps that sustain their diets. Because they are corallivorous, ornate butterflyfish do not survive well away from the reef. These highly sensitive fish are more susceptible to diseases, bacterial infections, and starvation when kept in a home aquarium.

Marlowe Shaeffer was the production editor and proofreader for Eclipse. Jane Ellin was the copyeditor. Reg Aubry and Mary Anne Weeks Mayo provided quality control. Lucie Haskins wrote the index. Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Julie Hawks to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Marlowe Shaeffer.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

PREV < Day Day Up >