PART II

Fundamentals of Extending Eclipse

Having finished Part I, Using Eclipse, you now have an appreciation of what Eclipse offers you in general and specifically as a Java programmer. The goal of Part II is to show you how to make Eclipse your own by explaining the fundamentals of enhancing and extending it with your own ideas. As you will soon learn, Eclipse is more than just an integrated development environment; it is also a platform from which you can create feature-rich applications and tools. Whether it is your goal to create an application or enhance the Eclipse IDE, you'll start with Part II. After you've mastered these base concepts that apply regardless of whether you're building a general application or a tool, Part III, Extending the Eclipse Workbench, teaches you how to use your plug-in development skills to extend the Eclipse user interface.

- Chapter 7, Extending Eclipse for Fun and Profit, is an introductory chapter to help you understand and appreciate what extending Eclipse is all about. Even if you are just curious about what it takes to extend Eclipse, this chapter is the place to start. It may be easier than you think.
- Chapter 8, Overview of the Eclipse Architecture, as the title suggests, provides you with a foundation on Eclipse and its extensibility framework.

- Chapter 9, Getting Started: Plug-in Development, explains how to create a plug-in and the Eclipse tools available to support development and testing of plug-ins.
- Chapter 10, Creating Applications Using the Rich Client Platform, explains the fundamentals required to create a rich client application using Eclipse. This chapter revisits some of the points covered in Chapter 8, so if you're really anxious to learn about the Rich Client Platform, you could skip the previous two chapters and then return afterward to deepen your understanding.
- Chapter 11, Creating Extension Points: How Others Can Extend Your Plug-ins, explains how your plug-ins can be of service to others. It's not hard and is an important item to have in your personal Eclipse toolkit.
- Chapter 12, Advanced Plug-in Development, rounds out the topic of extending Eclipse.
- Chapter 13, Defining Features and Products, teaches you how to package and share your plug-ins.

Remember that this book comes with a CD-ROM that contains working examples that demonstrate the concepts presented in the book. Many chapters refer to contents on the CD-ROM as part of their explanations, extracting bits of code to reinforce your understanding. These examples are well documented with both Javadoc and inline comments. The CD-ROM also includes documentation for the examples and exercises, both of which are integrated with the Eclipse Help system. See the `readme.html` file for installation instructions.

CHAPTER 7

Extending Eclipse for Fun and Profit

The book you hold in your hands is sizable, one might even say imposing. We like to think of it as "impressively comprehensive," yet we appreciate how it may appear daunting as you turn to the first chapter that leads you into this next part. Still, we didn't want to abandon a winning formula that brought the first edition to the Java bestsellers lists for amazon.com, Barnes & Noble, and JavaOne in 2003 and 2004. Instead, we've restructured this unabashedly large book to make it more approachable as both a learning guide and reference.

This opening chapter of Part II has the formidable task of introducing you to the full breadth and depth of Eclipse as a technology platform without being overwhelming, and doing it in such a way that you feel motivated to continue onto more advanced topics. This eliminates any simple "Hello, World" introduction, since that surely won't inspire your imagination of the possibilities Eclipse offers. Instead, we'll present a useful example of how you might extend Eclipse's Java development tools—without writing much code at all. Along the way, we'll play a little loose with details and focus mostly on concepts. Rest assured that after this brief respite, the subsequent chapters will resume a more methodic approach to solving general Eclipse programming problems.

Excited About Extending Eclipse? You Should Be!

Eclipse has received much fanfare and accolades because of its powerful Java development environment. That—coupled with the team environment and other base capabilities—makes Eclipse a compelling integrated development environment, which is great news for Java developers. Moreover, Eclipse is

an open source project. But what makes Eclipse truly exciting are the extension possibilities that it offers you.

A number of open source and commercially available products based on Eclipse show the practical implications of this way of delivering integrated products. Check out the "Community Projects and Plug-ins" page on eclipse.org. You'll see over one hundred projects that demonstrate the impact that Eclipse has already had by lowering development costs and reducing their users' learning curve because of their similar user interfaces and behavioral consistency. Sure, the value of this is self-evident for large software houses, but what's in it for the "little guy?"

That's where the extensibility story of Eclipse gets interesting. Not just integration for those who have large development organizations, but also for anyone willing to invest some time in learning a few Eclipse frameworks. "Oh no," you may be thinking, "not more frameworks. I don't have the time to learn more frameworks." Don't worry; it will be quick and easy. And before that little voice in your head has time to say it, no, we'll keep this chapter's promise of not boring you with a trivial extension of Eclipse. You'll see practical value and a clear demonstration of how you can enhance your use of Eclipse's Java development environment. These same techniques can be applied more generally to enhancing Eclipse to integrate your own homegrown tools and applications. You may even be a little surprised to see that it often takes only a few dozen lines of code to do some amazing things.

This chapter will show you what is possible and where to start, and will give you a firm appreciation for what's involved in getting there. Though extending Eclipse is an advanced topic, you can start with only a passing knowledge of how to use Eclipse's Java development environment.

An Easy and Practical Example

Many Java developers don't worry too much about categorizing method visibility as default (package), private, public, or protected when initially writing code. As they create methods, they often make them all public. Only after they've finalized the organization of packages and finished refactoring methods—whether by extracting new methods from existing code by pulling up or pushing down methods in the hierarchy, or by moving them to another class entirely—do they go back and review method visibility. That's reasonable, since they may not know the final class shapes and have only a little practical usage of the code, so they don't want to declare what their "clients" might need. In other words, before sharing a new framework, you must decide what is implementation detail and what is necessary for others to use.

It would be handy if you could merely select methods in the Outline view, Hierarchy view, or wherever you see methods—and with a click of a menu choice, set one or more methods to the desired visibility. Figure 7.1 shows such an extension to Eclipse's Java development environment in the context of the Java editor's Outline view.

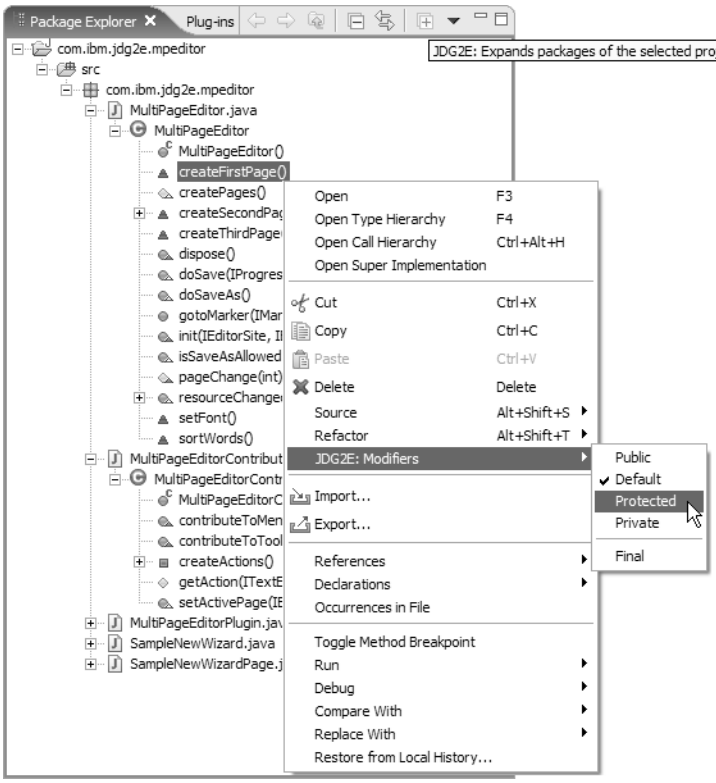


Figure 7.1 Extension of a Member's Context Menu

This is subtle, from a user's perspective, because of the natural way this was introduced into the user interface. There is no inkling that these new menu choices weren't part of Eclipse's original Java Development Tools (JDT). In fact, that's why the menu cascade is prefixed with the abbreviation of this book's title, "JDG2E"—so you can tell it's our extension! What's more, the developer doesn't have to remember that these choices are only available in a particular view or editor because they will be shown anywhere a method is shown. This is the power of Eclipse's extensibility. If you look

carefully at Figure 7.1, you may also notice another addition to the Package Explorer ( ). This "smart expand all" is the converse of the collapse all on the Package Explorer's toolbar. It's also available in the `com.ibm.jdg2e.jdt` project and we'll return to it later in Chapter 27, Extending the Java Development Tools. For now, we'll focus on the workings behind the **JDG2E: Modifiers** extension to Java members.

A Brief Tour of "Hello, World"

"Hey, wait a minute, you promised no 'Hello, World'!" True, but we do need to cover a little about Eclipse's underpinnings before getting to the really interesting stuff. So if you have never written your own extension to Eclipse, please join us in a quick tour of the Eclipse architecture and plug-in development environment. Otherwise, skip to the next section. On with the tour!

In essence, Eclipse is a collection of loosely bound yet interconnected pieces of code. How these pieces of code are "discovered" and how they discover and extend each other captures the fundamental principles of the Eclipse architecture. These functional units are called **plug-ins**. The Platform Runtime, shown in Figure 7.2, is responsible for finding the declarations of these plug-

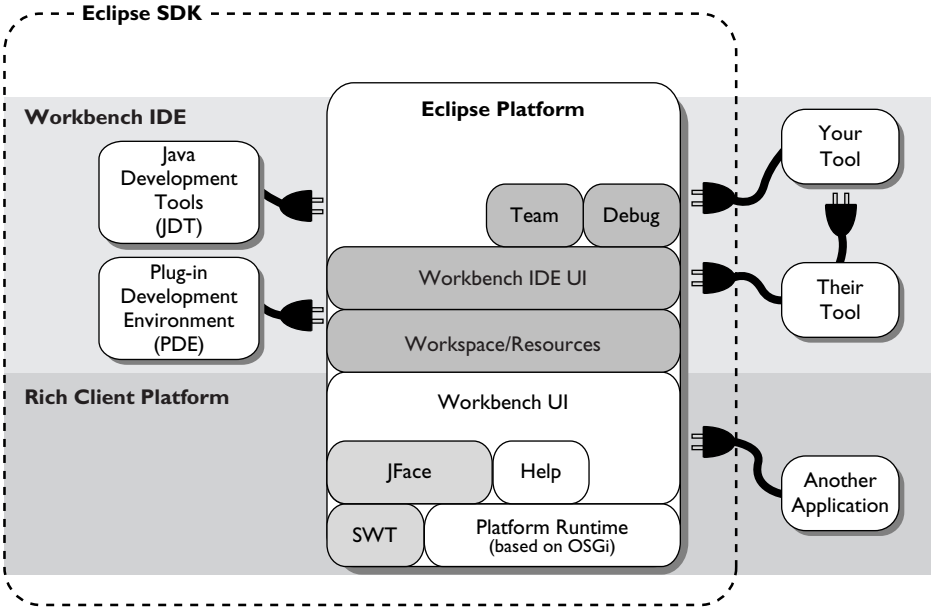


Figure 7.2 Eclipse Platform Architecture

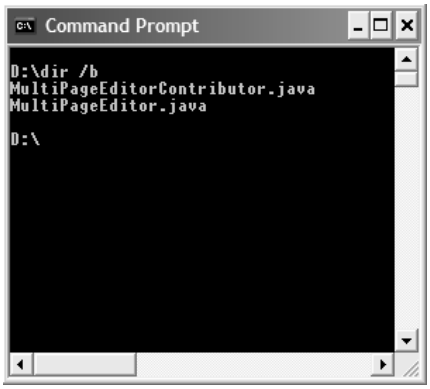
ins, called a **plug-in manifest**, in a file named `plugin.xml`. Each plug-in is located in its own subdirectory below a common directory of Eclipse's installation directory named `plugins` (specifically, `<inst_dir>\eclipse\plugins`).

During startup these files are used to build a global registry, called the **extension registry**. A plug-in refers to this registry to determine what other plug-ins wish to take advantage of its services. A plug-in that wants to allow others to extend it will declare an **extension point**. This is a sort of "power strip" one plug-in provides that others can take advantage of by declaring an **extension** to it.

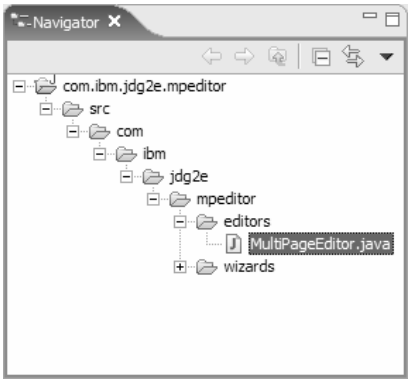
Returning to our example, the mission is to decide where to "plug into" Eclipse by finding the appropriate extension point offering the services we require. Fortunately, once you have used Eclipse as an IDE for a while, you know a surprising amount about what is available, perhaps without realizing it. This is because what you see in the Eclipse user interface and what is modeled by the classes that make up the Eclipse plug-ins often correspond nearly one-for-one to each other. Figure 7.3 makes this point clearer where the different views, that is, different levels of abstraction, are shown.

Here we see a progression of user interfaces showing the same files presented in a different manner, starting from the lowest common denominator on the right, the file system contents shown by a `dir` command in a Command Prompt window, continuing to a highly specialized view, that of the JDT's Package Explorer on the left. That is, the file `MultiPageEditor.java` is the same source in all three views, but its presentation is quite different and the available actions are different too. From a user interface perspective, all these views are visualizing a representation of the same "model," namely some files. As Eclipse users, we naturally expect views to present us different ways of looking at the same thing simultaneously, ways that adapt to the work we are doing. Recognizing how the Eclipse user interface reflects its underlying model and how its models build upon each other gives us an important clue about how we can find the best place to plug in our extension. The Navigator view in Figure 7.3 shows instances of `IFile` and the Package Explorer view shows instances of `ICompilationUnit`. As you can see, the names of these model classes and many like them correspond with what is shown in the user interface; therefore, you already have an intuitive appreciation for what's available programmatically.

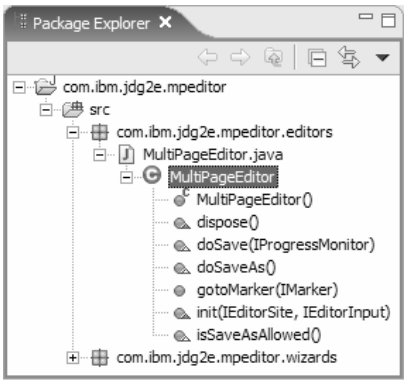
That is the first half of our tour. The second half is a look at developing the solution. Rather than present the solution and explain it piece-by-piece, wouldn't it be more interesting to discover some of it? Let's start with some questions related to the problem at hand: Extending the JDT with our own method visibility refactoring capability.



Files modeled as instances of `java.io.File`



Files modeled as instances of `org.eclipse.resources.core.IFile`



Files modeled as instances of `org.eclipse.jdt.core.ICompilationUnit`

Figure 7.3 Views and Their Models

Asking the Right Question Is More Important Than Knowing the Answer

Our quest begins with some general questions.

- How and where will the extension be shown in the user interface?
- How do you extend the user interface in general?
- How does an extension to the user interface know about events like a user's selection?

Once you have a good handle on the basic Eclipse landscape, we'll turn to some questions more specific to the solution we want to develop.

- How do you extend the user interface of specific objects of the JDT, like members shown in the Outline view? Do you extend the view(s) or their underlying model?
- What is the relationship between objects shown in the Package Explorer and the same objects shown in other views like the Outline view? Does your extension need to be aware of any differences between them?
- How do you change the JDT model programmatically?
- How do you analyze Java source code to apply modifications?

And of course, the final big question:

- Where to go from here?

How and Where the Extension Is Shown in the User Interface

As you recall, we decided to show context menu choices for one or more selected methods that allow us to change their visibility with a single action. We prefer that they be available wherever the methods can be displayed, such as the Hierarchy view and Package Explorer. This leads to our next question.

How to Extend the User Interface in General

Learning by example is more fun, and this is where the Plug-in Project wizard can give you a hand. It provides sample code that you can modify to meet your needs. You answer just a few of its questions and the wizard will automatically launch the specialized perspective for plug-in development, known as the **Plug-in Development Environment** (PDE), ready for testing. This wizard includes a number of examples that will get you started. In fact, your old friend "Hello, World" is there. Just for old time's sake, let's generate

it, look at the result to verify that the environment is set up correctly, and then modify it to help answer the current question and lead to the next question: *How does an extension to the user interface know about events like selection?* That will be important, since you want to apply your newly introduced menu choices to the currently selected method(s).

Note that these instructions assume that you're starting from a fresh Eclipse installation. If you have modified the environment or changed preferences, things may not work precisely as described below. You might consider starting Eclipse with a fresh workspace by selecting another one on startup (see Running Multiple Eclipse Windows and Workspaces in Chapter 2 if you need a reminder of how to do this).

Begin by creating a plug-in project using the New Plug-in Project wizard.

1. Select **File > New > Project**.
2. In the New Project dialog, select **Plug-in Development > Plug-in Project** in the list of wizards and then select **Next**.
3. Name the project `com.ibm.jdg2e.helloworld`. The wizard will create a plug-in identifier based on this name, so it must be unique in the system (by convention, the project name and the plug-in identifier are the same). The proposed default workspace location shown under **Project contents** is fine; select **Next**.
4. The next page, shown in Figure 7.4, proposes a plug-in name and plug-in class name. These are based on the last word of the plug-in project, `com.ibm.jdg2e.helloworld`. This example doesn't need a plug-in class because we are accepting the default implementation for controlling the plug-in life cycle, so deselect its code generation option, as shown in Figure 7.4, and select **Next** (not **Finish**; you have two more pages to go).
5. The next page allows you to choose a code template. Select the **Create a plug-in using one of the templates** check box, select **Hello, World**, and then select **Next**. The following page, shown in Figure 7.5, is where you can specify parameters that are unique to the "Hello, World" example, such as the message that will be displayed.
6. To simplify the resulting code, change the target package name for the action from `com.ibm.jdg2e.helloworld.actions` to `com.ibm.jdg2e.helloworld`, the same name as the project. While you might choose to have separate packages for grouping related classes in a real world plug-in, in this case there will be only one class, so there's no need. It also adheres to the convention that the "main" package is named the same as the project.

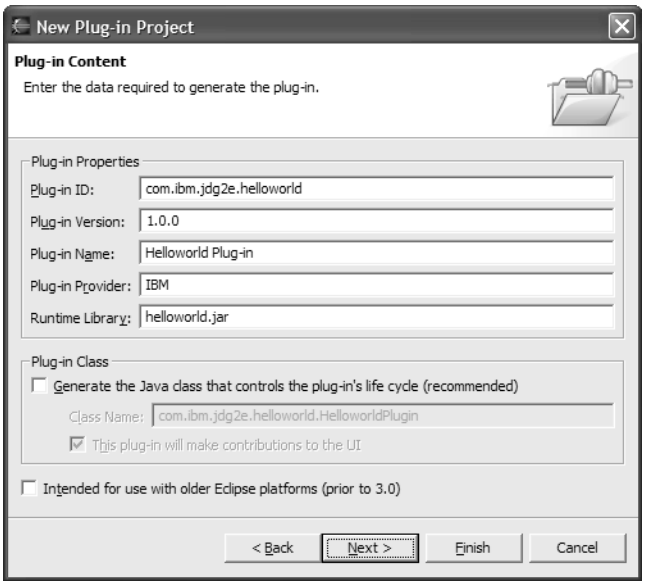


Figure 7.4 Plug-in Content

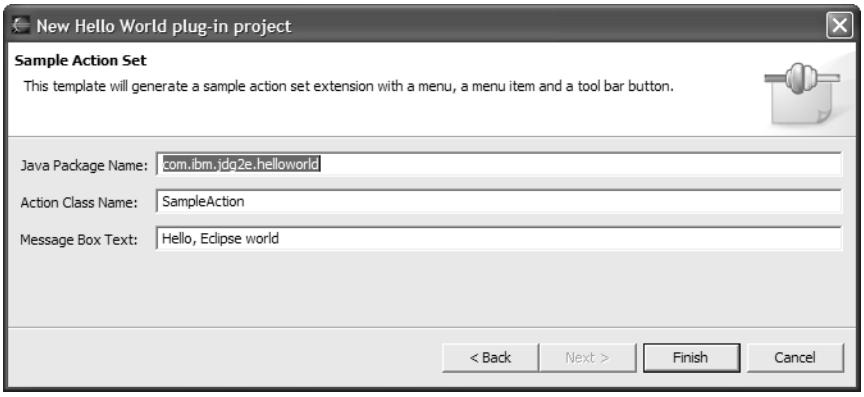


Figure 7.5 Sample Action Set

7. Now select **Finish**.
8. If this is a fresh workspace, you may see the information message "This kind of project is associated with the Plug-in Development Perspective. Do you want to switch to this perspective now?" Select **Yes** to switch as the message suggests.

To verify that everything is set up correctly, let's test your new plug-in.

1. Select **Run > Run As > Run-Time Workbench**. This will launch a second instance of Eclipse that will include your plug-in. This new instance will create a new workspace directory named `runtime-workbench-workspace`, so don't worry—whatever testing you do with that instance will not affect your development setup.
2. You should see something like Figure 7.6, with a new pull-down menu labeled **Sample Menu** having a single choice, **Sample Action**. Selecting it will show the information message below.

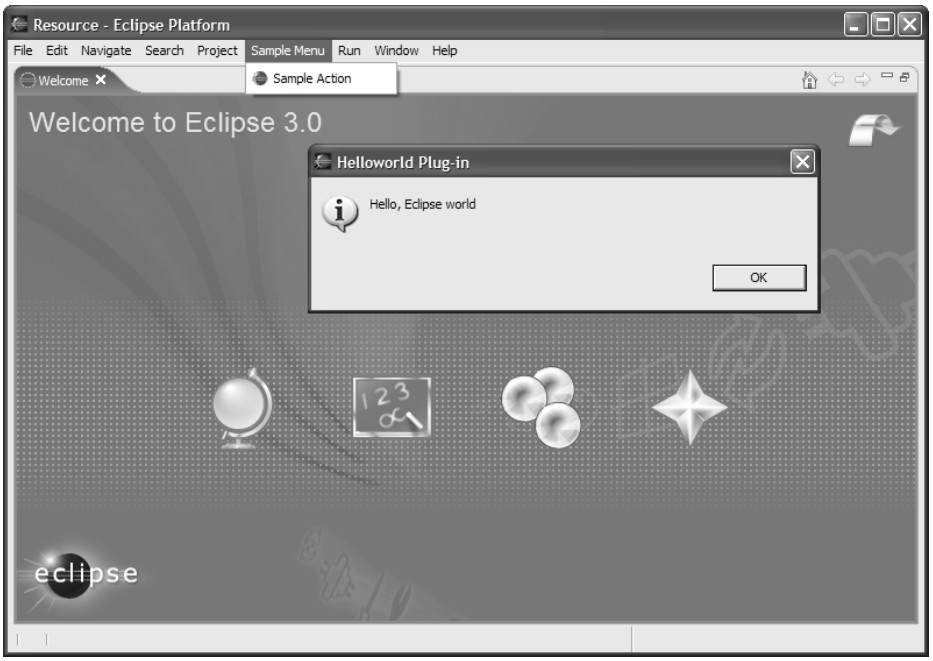


Figure 7.6 Hello, World

3. Close the runtime workbench and return to the PDE.
4. Let's take a quick glance at the plug-in manifest file, `plugin.xml`. Double-click it to open it in the Plug-in Manifest editor. This editor presents several wizard-like pages and a "raw" source page. Turn to it by selecting the **plugin.xml** tab. You'll see something like what's shown on the next page; we're interested in the text highlighted in bold.

```
<extension point="org.eclipse.ui.actionSets">
  <actionSet
      label="Sample Action Set"
      visible="true"
      id="com.ibm.jdg2e.helloworld.actionSet">
    <menu
        label="Sample &Menu"
        id="sampleMenu">
      <separator
        name="sampleGroup">
      </separator>
    </menu>
    <action
      label="&Sample Action"
      icon="icons/sample.gif"
      class="com.ibm.jdg2e.helloworld.SampleAction"
      tooltip="Hello, Eclipse world"
      menubarPath="sampleMenu/sampleGroup"
      toolbarPath="sampleGroup"
      id="com.ibm.jdg2e.helloworld.SampleAction">
    </action>
  </actionSet>
</extension>
```


It isn't necessary to study this too closely. The purpose of the second half of this tour is only to familiarize you with some of the basic mechanisms whereby you can introduce your extensions to the JDT. Here you see a sample of one such technique to add menus and menu choices to Eclipse as an action set. It begins with an extension, declared with the `<extension point="org.eclipse.ui.actionSets">` tag. The Eclipse user interface plug-in defines this extension point, `org.eclipse.ui.actionSets`, and several others like it where other plug-ins can contribute to the various user interface elements.

We still haven't answered how we can add menu choices to the context menu of Java methods. A simple modification can give you some hints. Begin by opening the class that displays the "Hello, World" message, `SampleAction`, and note its `run` method. It isn't particularly interesting; however, you also see another method, `selectionChanged`. Aha! The answer to the next question awaits.

How an Extension to the User Interface Knows About Events Like Selection

Contributed actions, like our contributed menu pull-down choice, are notified when the selection changes. Let's modify this method to tell a bit more about the selection. First, if you haven't already closed the runtime instance of the Workbench, do so now (this won't be necessary if Eclipse is running

the debugger under a JRE that supports hot code replace; in that case you can modify the code directly without restarting and see the result immediately). Add the code below to the `selectionChanged` method.

```
public void selectionChanged
    (IAction action, ISelection selection) {
  System.out.println("==========> selectionChanged");
  System.out.println(selection);
}
```

With this debug code, you'll see what is selected and learn a little more about what makes Eclipse tick. Save the method and relaunch the runtime Workbench.

IMPORTANT Eclipse has a deferred load strategy to avoid activating plug-ins until the user does something that requires its code. So you must *first* select the **Sample Action** menu choice to load your plug-in before your `selectionChanged` method will be called.

To test your code, begin by creating a few projects, folders, and text files in your runtime Workbench. Now select different things like text in an editor, files in the Navigator, and, of course, members in the Outline view (recall that you'll need to create a Java project and an example Java class to do this, since the runtime instance uses a different workspace). The output below shows some sample messages that are similar to what you will see in the Console of the *development* instance of Eclipse.

```
==========> selectionChanged
[ColorManager.java [in com.ibm.jdg2e.test
    [in src [in com.ibm.jdg2e.test]]]
    package com.ibm.jdg2e.test
    import java.util.HashMap
  //  ...lines omitted...
==========> selectionChanged
<empty selection>
==========> selectionChanged
org.eclipse.jface.text.TextSelection@9fca283
==========> selectionChanged
[Color getColor(RGB) [in ColorManager [in [Working copy]
    ColorManager.java [in com.ibm.jdg2e.test
    [in src [in com.ibm.jdg2e.test]]]]]]
```

Well, that isn't as enlightening as you'd hoped. Clearly the selection isn't something as primitive as an instance of `String`, but it isn't evident what classes are involved either, because these classes have clearly overridden their default `toString` method. We're not yet at the point where you can appreci-

ate what information they are showing without a little more investigation. Returning to the `selectionChanged` method, browse the hierarchy of the interface of the `selection` parameter, `ISelection`. Its hierarchy reveals the general-purpose subtype interfaces, `IStructuredSelection` (for lists) and `ITextSelection`.

NOTE You must import an additional plug-in that wasn't originally specified in the list of "Hello, World" required plug-ins so `ITextSelection` shown in the code below is visible to your plug-in. Add it now by inserting the following to the `plugin.xml` file, beneath the other import statements: `<import plugin="org.eclipse.jface.text"/>`

We'll make the `selectionChanged` method a bit smarter by outputting the class that's selected. The modified `selectionChanged` method is shown below.

```
public void selectionChanged
    (IAction action, ISelection selection) {
  System.out.println("==========> selectionChanged");
  if (selection != null) {
    if (selection instanceof IStructuredSelection) {
      IStructuredSelection ss = (IStructuredSelection) selection;
      if (ss.isEmpty())
        System.out.println("<empty selection>");
      else
        System.out.println("First selected element is " +
          ss.getFirstElement().getClass());
    } else if (selection instanceof ITextSelection) {
      ITextSelection ts = (ITextSelection) selection;
      System.out.println(
          "Selected text is <" + ts.getText() + ">");
    }
  } else {
    System.out.println("<empty selection>");
  }
}
```

Relaunch the Workbench and you'll see different messages in the console, depending if the selection is empty, from a list (`IStructuredSelection`), or an entry field (`ITextSelection`). Again, remember to close the runtime instance, relaunch, and select the **Sample Action** menu choice to load your plug-in. Now when you select various elements of the user interface it is far more revealing. The sample output shown on the next page is similar to what you'll see, depending on what your workspace contains and what you select. The interspersed annotations in bold explain what user action preceded the output.

```
    select a method in the Outline view
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.SourceMethod
==========> selectionChanged
<selection is empty>
    activated the Java editor
==========> selectionChanged
Selected text is <getResourceBundle>
    selected method, class, and
    package in the Package Explorer
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.SourceMethod
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.SourceType
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.PackageFragment
    activated the Navigator view, selected some files,
    folders, and projects
==========> selectionChanged
First selected element is class
  org.eclipse.core.internal.resources.File
==========> selectionChanged
First selected element is class
  org.eclipse.core.internal.resources.Project
==========> selectionChanged
First selected element is class
  org.eclipse.core.internal.resources.Folder
    reactivated the Package Explorer,
    selected some classes and methods in
    JARs of reference libraries
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.JarPackageFragment
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.ClassFile
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.BinaryMethod
```

Specifically, you can confirm that what you see in the user interface corresponds one-for-one with model classes of the JDT. Why you're seeing what appears to be models as selections and not lower-level primitives like strings and images is thanks to another Eclipse framework, called **JFace**. As you'll see in Chapter 15, JFace Viewers, this framework maps between primitives like strings that the widgets close to the operating system expect and the higher-level model objects with which your code prefers to work. This chap-

ter only peripherally touchs on this topic, since our stated goal is extending the JDT. Later chapters covering topics like JFace viewers, views, editors, and JFace Text will broaden your understanding of the JFace framework. This chapter will only cover what's necessary to understand the implementation of our JDT extension.

Returning to the output, two particular selection results draw our attention: those corresponding to the selection of Java members in the user interface. They are repeated below.

```
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.SourceMethod
==========> selectionChanged
First selected element is class
  org.eclipse.jdt.internal.core.BinaryMethod
```

The `internal` in the middle of the package name for these classes is a little disquieting. However, as you'll often find, Eclipse will have a public interface that corresponds to the (internal) implementation class, as is the case here. A quick class lookup reveals that these classes all implement a common set of interfaces that look promising, namely `ISourceReference`, `IJavaElement`, and especially `IMember`. Finally! Now you have what you had hoped to extend, leading to the answer to the next question.

How to Extend Objects Like Those Shown in the Outline View

Our simple "Hello, World" example showed that adding a menu choice requires just a few lines of XML in the plug-in manifest file (`<extension point="org.eclipse.ui.actionSet">`) and a class that handles the actual action (`com.ibm.jdg2e.helloworld.SampleAction`). Adding actions to views' pull-down menus, the common editors' toolbars, and pop-up menus is nearly as straightforward. Contributed pop-up menus come in two flavors: Those that are associated with just the view and not selected objects (that is, the default pop-up menu that views display when you right-click on their "whitespace"), and the more common variety, choices that apply to the selected object(s). In this case, we want to target only specific selected objects, so we'll contribute what's called an **action object contribution** to their pop-up menus by defining an extension in the plug-in manifest (some of the identifiers are shortened to format better; they are denoted by "…"), as shown below.

```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution
      objectClass="org.eclipse.jdt.core.IMember"
      id="...imember">
```

```
      <menu
          label="JDG2E: Modifiers"
          path="group.reorganize"
          id="...imember.modifiers">
        <separator name="group1"/>
        <separator name="group2"/>
      </menu>

      <action
        label="Private"
        menubarPath="...imember.modifiers/group1"
        class="...jdt.extras.MakeIMemberPrivateAction"
        id="...imember.makeprivate">
      </action>

      <action
        label="Protected"
        menubarPath="...imember.modifiers/group1"
        class="...jdt.extras.MakeIMemberProtectedAction"
        id="...imember.makeprotected">
      </action>

      //    ...all menu choices not shown...

    </objectContribution>
  </extension>
```

The extension point is named `org.eclipse.ui.popupMenus`, and as the name suggests, it defines contributions to pop-up menus appearing in Eclipse. This particular example will contribute only to specific selected objects, those implementing the `IMember` interface (recall that as defined in the Java language specification, members include both methods and fields). Our investigation has paid off; we have the answer to the current question and we're almost ready to move to the next question.

Before doing so, note at this point that the pattern used for the simple "Hello, World" action example will repeat itself for other menu action contributions. That is, the class named in the `class` attribute will be notified of selection changes (by its `selectionChanged` method) and will be notified when the user selects the menu choice (by its `run` method). The user interface portion of the tour is almost over; the harder part, effecting the desired change, lies ahead. There is only an observation or two to make before continuing, as stated in the next question.

The Relationship Between the Same Objects Shown in Different Views

You may have noticed that when you selected methods in the Outline and Hierarchy views, the class of the selected object was not always the same. For

example, if you expanded the contents of a library (JAR file) in the Package Explorer, then selected a class or method, it was also not the same class as a similar selection in the Java editor's Outline view. What's up with that?

Here you are observing the difference between those parts of the JDT's Java model that are "editable" versus those that are always read-only. Both parts of the Java model will implement a common interface, like `IMember`, but have different implementation classes that understand the underlying restrictions. As another example, there is an implementation class representing a Java compilation unit derived from a `.class` file in a JAR file shown in the Package Explorer and another class representing a compilation unit derived directly from a `.java` file. The latter implementation will allow modifications where the former cannot, yet a shared portion of their API is represented by the interface `ICompilationUnit`. This interface and others from the JDT model are shown in Figure 7.7 along with their corresponding visual representation.

Subsequently, our contributed action that will modify Java members has to be aware of the context in which it is invoked. That is, it will have to recognize that some selected members are modifiable (those in the Java editor's Outline view) while others are not (members from `.class` file stored in a JAR file and shown in the Package Explorer). Keeping this in mind, let's continue on to the next question.

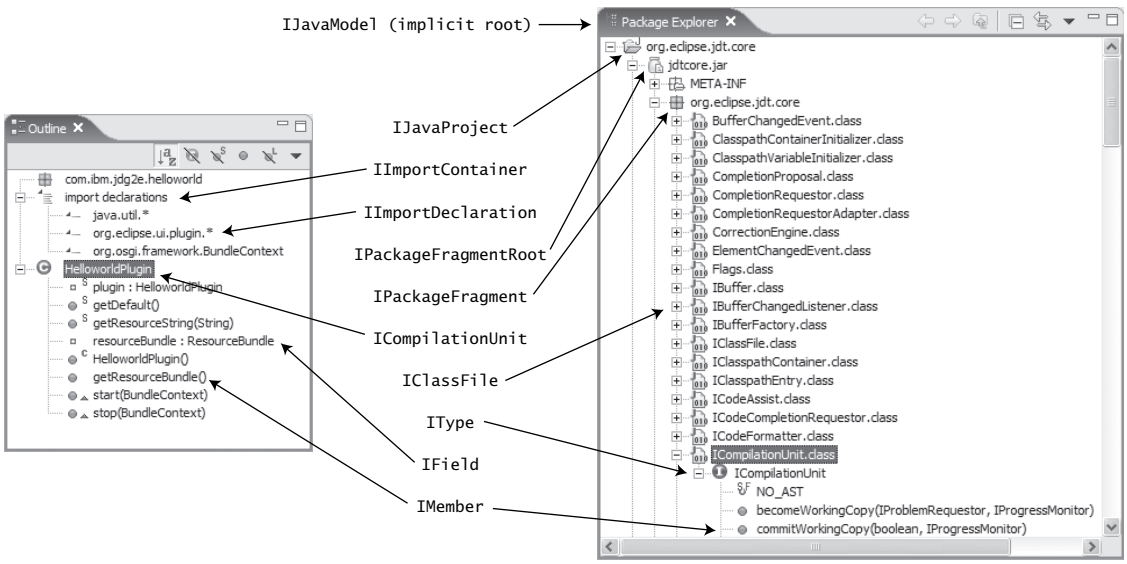


Figure 7.7 JDT Model

How to Change the JDT Model Programmatically

If you explored a bit during the prior tour, you may have noticed that `IMember`, `IJavaElement`, and what appears to be the majority of the interfaces implemented by the selected Java-related items our action saw have no `setXXX` methods. So how do you modify them?

You'll find it is surprisingly easy, yet perhaps not immediately obvious. The JDT's Java model is in most practical respects "read only." With the integrated cooperation of the Java compiler, changes to the underlying Java source of a given element are synchronized with the rest of the Java model. In effect, all you have to do is update the Java source, and the rest of the necessary model changes are propagated to whoever is dependent on them.

That's a relief! This point is why the Java model is key to plug-in integration: It provides a common shared in-memory model of the entire Java environment, its scope beginning from a project and continuing to all its referenced libraries, all without you having to worry about manipulating `.java` files, `.class` files, and `.jar` files in the file system. You can focus on the high-level model and let the JDT deal with many of those messy details.

Not yet convinced it is that easy? The listing below contains the diminutive snippet of code that is at the heart of this solution, extracted from the contributed action's `run` method and simplified slightly for readability.

```
public void run(IAction action) {
  IMember member = (IMember)
      ((IStructuredSelection) currentSelection).
      getFirstElement();
  ICompilationUnit cu = member.getCompilationUnit();
  cu.becomeWorkingCopy(...);
  IBuffer buffer = cu.getBuffer();
  buffer.replace(...); // update member's source code
  cu.reconcile(...);
  cu.commitWorkingCopy(...);
  cu.discardWorkingCopy();
}
```

Seems a bit anticlimactic, doesn't it? Your contributed action is given the selected member. You ask it for its parent container (the model of the Java `.class` or `.java` file, collectively referred to as a **compilation unit** in JDT parlance) because that's who manages the underlying source. Next, update the source code returned using a buffer. The `IBuffer` interface is similar to `StringBuffer`, the principal difference being that changing the buffer associated with a compilation unit updates the corresponding elements of the Java model. The call to `reconcile` tells the JDT to notify other interested parties like the Package Explorer view that your model updates are ready for public consumption.

You no doubt noticed the ellipsis in the code "`buffer.replace(...)`" shown above. That's where you have to analyze the source code itself to apply your modifications, which in this case will modify the method visibility from its current value to `public`, `private`, or `protected`. Again, the JDT comes to your aid and helps you finish the task. We'll not dwell on those details just yet, saving them instead for later in Chapter 27, Extending the Java Development Tools. You'll find the source code on the CD-ROM in the `com.ibm.jdg2e.jdt` project, if you want to peek ahead.

Leaving the implementation details aside for a moment, we hope you've gained an appreciation that you can introduce other worthwhile extensions to Eclipse using a similar discovery-oriented approach presented here. This isn't to suggest that you won't have lots of guidance—the weight of this book alone confirms there's a lot to learn! We encourage you to approach the task of learning about extending Eclipse by combining your experience from the user's point of view and the concepts and exercises of this book.

We covered quite a few new terms. Before closing, let's pause for just a moment and briefly review them as a reminder to you.

- Plug-in: a structured bundle of code
- Plug-in manifest: a formal declaration of a plug-in specified in a file named `plugin.xml`
- Extension points: a portion of the plug-in manifest defining where others can contribute
- Extensions: a contribution to a plug-in's extension point
- Action contributions: one of the Workbench extension points where the contributor defines a new action; actions can appear in several places within the Workbench user interface, such as our example, in an object's context menu

If some of the ideas presented in this chapter aren't yet solidified in your mind, don't worry. The next chapter will return to the terms and concepts underlying Eclipse, and in doing so, cover them more thoroughly and using more developed examples to assure your firm understanding.

Where to Go from Here?

The stated goal of this chapter was to give you a nontrivial extension to Eclipse's Java development tools that enhances your productivity and paves the way for further study. In all honesty, more than once we skipped over some details for reasons of brevity. Nonetheless, we hope that you got a good taste of what's possible and you're convinced that it isn't all too difficult.

What we've covered in this chapter will be covered in more detail starting with Chapter 8, Overview of the Eclipse Architecture. Chapter 21, Action Contributions: The Integration Fast Track, and more specifically in Chapter 27, Extending the Java Development Tools, will reprise their respective topics that we briefly introduced here. Like this chapter, they include a documented working example that reinforces what you've learned, much in the same style as what you've seen here (though perhaps not covered at such a breakneck pace!).

As a final reminder, note this chapter only covered one aspect of Eclipse plug-in development, namely extending the Eclipse IDE itself. There is an exciting and much broader aspect to Eclipse development, one to which this book dedicates the remainder of Part II. We'll pick up this theme in the next chapter, and then really address the subject in earnest starting with Chapter 10, Creating Applications Using the Rich Client Platform. After you've understood the concepts presented in these early chapters, you may choose to skip around to specific topics that apply to the task before you. The Guide to Reading This Book suggests pathways you might consider based on your experience and interests. The close of each chapter will summarize what you've learned, and if there are additional pathways to other chapters to consider, the closing will give recommendations of the directions in which you might choose to continue.