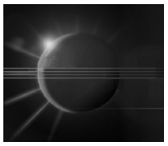

CHAPTER 3

Using Java Development Tools

Eclipse provides a first-class set of Java Development Tools (JDT) for developing, running, and debugging Java code. These tools include perspectives, project definitions, editors, views, wizards, refactoring tools, a Java compiler, a scrapbook for evaluating expressions, search tools, and many others that make writing Java code quick, fun, and productive.

The JDT views and editors provide an efficient and effective way to quickly and intuitively navigate through your source code, view Javadoc, and even view Java class files. The editors support syntax highlighting, code assist, error cluing and correction, state-of-the-art refactoring, type-ahead assistance, and code generation, among a host of other capabilities. The Eclipse compiler is an incremental compiler; it compiles only what it must based on your changes. It can be configured with different JREs. You can develop code with one JRE and debug with another. You might want to do this, for example, if you need to support multiple JREs at runtime. In the spirit of rapid, iterative development, JDT allows you to run and debug incomplete code and code with compile errors.

The editors are more than source code editors. They build and maintain indexed information about your code, enabling you to quickly search for references to classes, methods, and packages. The incremental compiler is constantly running in the background and will discretely alert you to potential errors in your code before you save it. In many cases the JDT will propose several solutions to the problems it detects, such as adding an `import` statement that was omitted, suggesting typographical corrections, or even creating

a new class or interface. This allows you to focus on your code and not be distracted by compiler demands.

In the following sections we look in more detail at the capabilities and use of the Java tools. We start with an overview of the JDT user interface and the fundamental skills you'll need for creating and navigating Java resources, that is, Java projects, input and output folders, and source code files. We then look in much more detail at JDT capabilities for coding Java: writing, searching, refactoring, and generating code. In the final sections of this chapter we peel the onion a bit more and get into further detail on working with Java resources (including more advanced Java project properties, resource local histories, and imported Java resources), tuning the performance of JDT, and using specific functions in the JDT views and perspectives that are useful but not immediately obvious.

In Part VI, we have assembled a number of exercises designed to help you perform many of these tasks and reinforce what we present. Exercise 4, Developing a Simple Web Commerce Application with Eclipse, brings together a number of the concepts and tasks you'll see in this chapter and the next in a robust example. We encourage you to take a look at these exercises.

Getting Started with JDT

To get things going, we start with a quick overview of the JDT user interface. Then we cover the fundamentals, like opening a class and navigating to a method or field. We also provide an introduction to running a Java program. (You'll see much more on running Java programs in the next chapter.) We also discuss how to search Java code.

Overview of the JDT User Interface

The Java perspective is the default perspective for Java development and the one opened when you create a new Java project (see Figure 3.1).

The left pane contains the Package Explorer view and the Hierarchy view (hidden behind the Package Explorer view in Figure 3.1). The Package Explorer view does for the Java perspective what the Navigator view does for the Resource perspective. Use the Package Explorer view to navigate in your Java projects, perform operations on resources, open files for editing, and run your programs. The middle pane contains open editors, Java and otherwise. An editor tab with an asterisk (*) in front of the file name means that the file has unsaved changes. The active editor is the one on top. In Figure 3.1, file `PrimeNumberGenerator.java` is open in the Java editor. The right pane is the

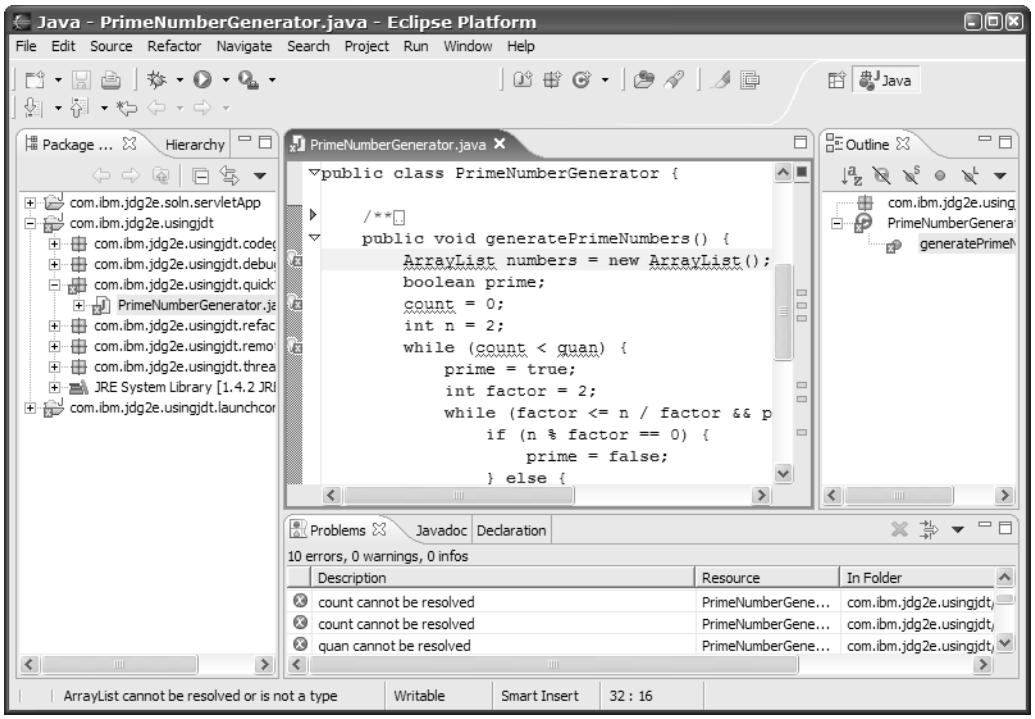

Figure 3.1 Java Perspective

Outline view, which presents a structured, hierarchical view of the contents of the active editor. On the bottom right is the Problems view, which shows code errors. Stacked behind it are the Javadoc and Declaration views. We'll get to these shortly. As you navigate in the user interface, selecting Java files for editing and modifying Java elements, all the views and the editor stay synchronized with your actions.

The Java views display Java elements with icons to help you readily identify the element type and visibility, such as for packages and for public methods. To provide further information, some of these icons are decorated with overlays, such as to indicate a class has a `main` method or to indicate a method overrides a method in a superclass. For a complete list of the JDT icons and decorations, refer to the "Icons" topic in the "Reference" section in the *Java Development User Guide*.

You have two options for how code appears in the editor as you navigate in the user interface. **Show Selected Element Only** on the toolbar controls this. The default is to show the contents of the entire file. If you prefer

to focus on smaller portions of code, toggling this option will show the source only for the selected class, method, field, or import statement. This is mostly a matter of personal preference.

The Fundamentals

Here are the fundamental tasks you need to understand to create and navigate through Java resources. These are available on the toolbar and as wizard shortcuts when you are in one of the Java perspectives.

Creating a Java Project

All Java elements must exist in a Java project for the JDT to correctly recognize them as Java elements. To create a Java project, select **File > New > Project... > Java Project** from the menu or select **New Java Project** from the toolbar.

When creating Java projects and other elements, if you get the names wrong or later decide you want to change them, the JDT refactoring capabilities make it easy to rename elements and update references to them. (Refactoring is discussed in more detail later in this chapter.)

Creating a Package

Java types, that is, classes and interfaces, must exist in a package. If you do not create one, a default package will be created for you when you create a class. To create a package, select the containing project and select **File > New > Package** from the menu or select **New Java Package** from the toolbar.

Creating a Type

To create a type, select the containing project or package. Then you have two choices: (1) select **File > New > Class** or **File > New > Interface** from the menu, or (2) select **New Java Class** or **New Java Interface** from the toolbar.

Opening a Type

To open a Java class or interface for editing, do one of the following.

- Double-click on a Java element, such as a source file, class, or method. Or select one of these in a view and press **F3** or **Enter**.
- From the editor, select the name of a class or interface in the source code (or simply position the insertion cursor in the name), and then select **Open Declaration** from the context menu or press **F3**.

- Select **Ctrl+Shift+T** and enter the name of a class or interface in the **Open Type** dialog.
- In the editor, press and hold the **Ctrl** key, and the names of Java classes and interfaces become hyperlinks you can click on to open the definition.

Opening a Method or Field

To open a method or field definition for editing, do one of the following.

- Double-click on a method or field, or select a method and press **F3** or **Enter** to see its definition. You can do this from any Java view.
- From the editor, select the name of a method or field in the source code (or simply position the insertion cursor in the name), and then select **Open Declaration** from the context menu or press **F3** to see its definition.
- In the editor, press and hold the **Ctrl** key, and the names of Java methods and fields become hyperlinks you can click on to open the definition.

Viewing Supertypes and Subtypes

To view the supertypes or subtypes for a class or interface in the Hierarchy view, do one of the following.

- Select a Java element, such as a Java source file, class, method, or field. Then from within a Java view or the editor, select **Open Type Hierarchy** from the context menu or press **F4**.
- Select **Ctrl+Shift+H** and enter the name of a class or interface in the **Open Type in Hierarchy** dialog.
- Drag a type from one of the views and drop it in the Hierarchy view.
- Select a type in the editor and then press **Ctrl+T**.

Viewing Called Methods or Calling Methods

To view all the methods called by a given method, or all the methods that call a given method, do one of the following.

- Select a method, and then select **Open Call Hierarchy** from the context menu or press **Ctrl+Alt+H**. You can do this from any Java view.
- From the editor, select the name of a Java element in the source code (or simply position the insertion cursor in the name), and then select **Open Call Hierarchy** from the context menu or press **Ctrl+Alt+H**.
- Drag a type from one of the views and drop it in the Call Hierarchy view.

Navigating to a Type, Method, or Field

Ensure **Link with Editor** is enabled in your Java views, either on the toolbar or from an entry on the pull-down menu. Select a class, interface, method, or field definition in one of the views. The editor and open views scroll to your selection. If you select an element and you do not see it in an editor, it means the file containing the definition is not open. This also works in reverse: You can activate a specific editor, and the Java element will be selected in the Package Explorer view.

Locating Elements in Projects

Opening an editor on a class with **F3** shows its source but doesn't update the Package Explorer view to show which package the class is located in. To see where a class is located in the Package Explorer view, select a Java element in the editor (or position the insertion cursor in the element name), and then select **Show in Package Explorer** from the context menu. The Package Explorer view scrolls to the selected element. Note that this works only for elements defined in the file you are currently editing.

Running a Java Program

To run a Java program, select a class with a `main` method, and then select **Run > Run As > Java Application** from the menu. Output is shown in the Console view.

Using the JDT Views

Like the Navigator view in the Resource perspective, the Package Explorer view in the Java perspective is the workhorse for managing your Java code. This view can fill up, and its presentation can get busy. We have two suggestions. First, consider using one or more filters to limit what appears in the view. To add filters, select **Filters...** from the Package Explorer view pull-down menu . A common choice is to select the **Referenced libraries** option. Doing this hides any libraries defined for the project, for example, the JRE libraries. Second, deselect **Link with Editor** . This prevents a lot of scrolling in the Package Explorer view.

We've mentioned the Outline, Hierarchy, and Call Hierarchy views previously. In addition to these views, you will find two others very useful, the Declaration and Javadoc views. They are part of the Java perspective. In the Java editor, if you position the cursor on a variable or method, both the Declaration and Javadoc views show information for the selected Java element. The Declaration view displays the source code, while the Javadoc view shows

any associated Javadoc. This eliminates the need to chase off after the definition. This also works for view selections; when you are navigating through elements in any of the Java views, the Declaration and Javadoc views show content for the element you have selected—no need to open an editor to take a quick look at the code.

Source > Sort Members allows you to organize the order in which initializers, fields, and methods appear in Java views. The order is specified in your **Java > Appearance > Members Sort Order** preferences.

We saw in Chapter 2, Getting Started with Eclipse, how to compare files or different versions of the same file. Some of the same compare and replace operations are available on individual Java elements through the Java views. We'll talk about this more in Local History for Java Elements later in the current chapter.

Eclipse includes a class file editor—a viewer, really, because you can't change the contents of the class file—to allow you a limited view into a class file for which you do not have source. From one of the views, for example, the Navigator view, simply open a class file for editing.

Searching

There are two types of searches: a general Eclipse file search for text strings, and a Java-specific search of your workspace for Java element references. The search capability uses an index of the Java code in your workspace that is kept up-to-date in the background, independent of Java builds. This means that you don't have to have your auto-build preference selected or save your modifications in order to do a search.

Searching a File

To search the file in the active editor for any text, select **Edit > Find/Replace**, or press **Ctrl+F**. Specify the string for which you are searching or a regular expression. Eclipse also has an "incremental find" feature that provides a keystroke-efficient way to do this. Select **Edit > Incremental Find Next** from the menu or press **Ctrl+J** and note the prompt in the message area to the left on the bottom margin. Start typing the text you're searching for. The message area displays that text, and the first match is selected in the editor. Press **Ctrl+J** to find the next occurrence or **Ctrl+Shift+J** to find the previous one.

Searching Your Workspace

Select **Search**  or press **Ctrl+H** and then select the **Java** page to search your entire workspace, or a subset of it, for references to Java elements. This

searches both Java source code and Javadoc. There are three aspects to a Java search: what kind of reference, what kind of Java element, and within what scope. You can specify these by using **Limit To**, **Search For**, and **Scope**, respectively, in the Search dialog, as shown in Figure 3.2. To restrict a search, you can either select one or more Java elements in one of the Java views and then select **Search**, or you can define a working set and then specify that working set under **Scope**. (For more information on working sets, refer to the Working Sets section in Chapter 2, Getting Started with Eclipse.) For example, if you want to search for methods returning `void` in your projects, select **Search**, specify the **Search string** as `* void`, select **Search For Method** and **Limit To Declarations**, select **Workspace**, and then select the **Search** button (or press **Enter**).

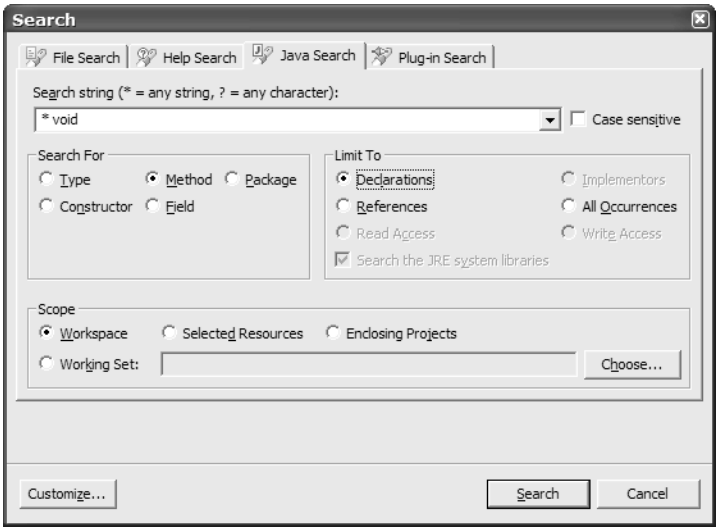

Figure 3.2 Searching for Java Elements

Java search results are shown in the Search view (see Figure 3.3). Matches are indicated in the editor with entries on the marker bar. You can filter the search results by selecting one of the available filters from the Search view pull-down menu ▾. For example, you can filter out Javadoc comments, read accesses, and write accesses this way. Navigate to matches from the Search view by double-clicking an entry or by selecting **Show Next Match** ⇩ and **Show Previous Match** ⇧ from the Search view toolbar. You can also use **Next Annotation** or **Ctrl+.** and **Previous Annotation** or **Ctrl+,** from

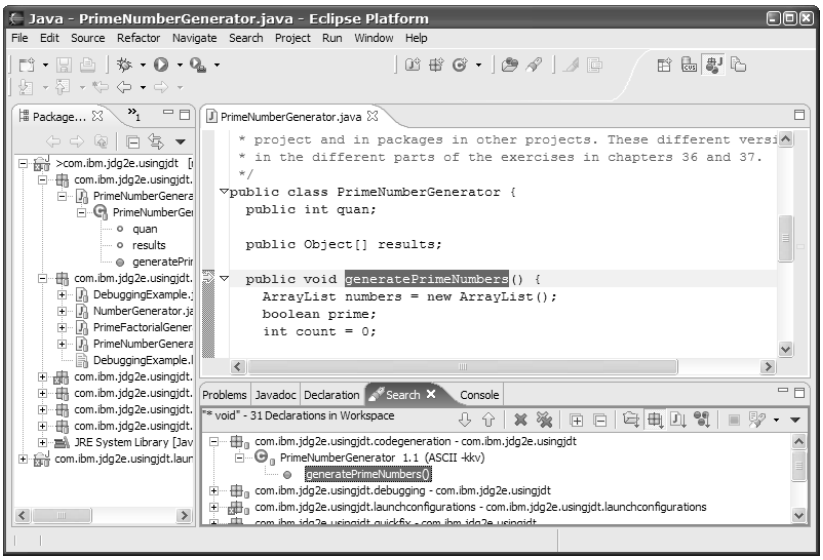

Figure 3.3 Search View

the Eclipse toolbar to navigate matches within the active editor. First select to use the annotation navigation buttons to match **Search Results** with the pull-down menu on either of the buttons.

In all of the JDT views, including the Search view and the editor, you can select a Java element or search result entry and then search on that element from the context menu by selecting one of the **References** or **Declarations** submenu choices. This ability to successively search for Java elements from the views, especially the Type Hierarchy and Search views, provides a simple, efficient way to explore and understand Java code.

Writing Java Code

Now that you've seen how to create and navigate Java resources, let's get to what we get paid for, writing Java. The Java editor provides a wealth of functions to help you write Java code more efficiently with greater productivity and, quite frankly, more fun. Among the capabilities provided by the Java editor are code assist for Java expression completion, code generation, real-time error detection, quick fix remedies for coding errors, Javadoc generation, and the ability to work with different JREs.

Overview of the Java Editor

The organization of the Java editor pane is straightforward and similar to other editors (see Figure 3.4).

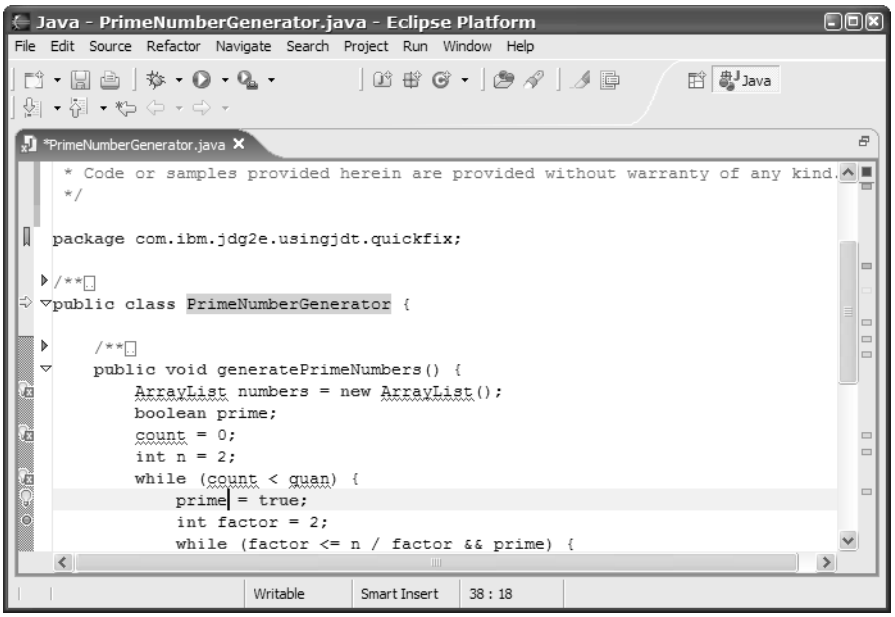

Figure 3.4 Java Editor

On the left border is the **marker bar**, which shows tasks, bookmarks, compiler errors, quick fixes, quick assists, and overridden methods. The marker bar also is shaded to indicate the scope in the file of the type, field, or method you're editing. In Figure 3.4, the `generatePrimeNumbers` method is being edited. Just inside this is the **quick diff ruler** that indicates, through shading, changes to the file you are editing. (We discussed this in Chapter 2, Getting Started with Eclipse, in the section Quick Diff—Real-Time Change Notification.) To the right of the quick diff ruler is a ruler that shows markers for sections of code that can be folded or expanded to allow more efficient viewing. Click on an icon to expand or fold a section of code. Hover over an icon indicating a folded section of code to see a pop-up with the code (see Figure 3.5). Settings for enabling folding and what gets folded are available in your **Java > Editor** preferences on the **Folding** page.

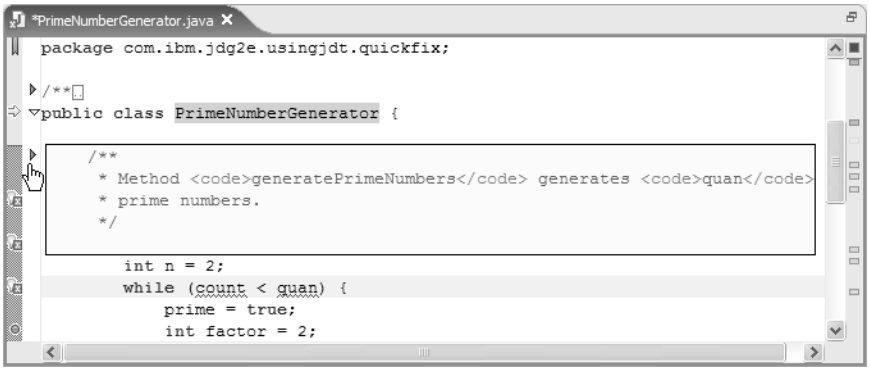

Figure 3.5 Viewing Folded Code

The right margin is the **overview ruler**, which includes color-coded marks that indicate where errors, warnings, bookmarks, tasks, and search match annotations are located in the file. This navigation aid helps you quickly scroll to annotations that are currently not visible. Simply click on one of the marks. At the top of the overview ruler, a red rectangle appears if errors are present in the file. A yellow rectangle indicates warnings. Hover over this to get a count of the errors or warnings in the file.

You can configure the annotations that appear on the marker bar and the overview ruler with your **Workbench > Editors > Annotations** preferences. If there are multiple annotations for a line, so that they overlay each other, hover over the group to see all of them shown next to each other. Navigate through the annotations with **Next Annotation** (**Ctrl+.**) or **Previous Annotation** (**Ctrl+,**). Use the toolbar drop-down menu to specify which annotations you navigate to.

The editor is, as you would expect, completely integrated. This means Java views, such as Type Hierarchy, Outline, and Declaration, update in real time as you type in the editor. Just as you can with a view, double-click on the editor title bar to maximize the editor. The editor allows a great deal of customization. There are many preferences you can use to customize behavior. Be sure to look at these. For example, you can configure which rulers are shown, how annotations (errors, tasks, search occurrences) appear in the code and on the rulers, and how your code should be formatted. The behavior of the editor, for the most part, is specified in your **Java > Editor** preferences. **Workbench > Editors** also has some general editor preferences. You can set fonts in **Workbench > Colors and Fonts**. The JDT provides a more advanced set of color highlighting, which is not enabled by default. Enable it

on the **Syntax** page of your **Java > Editor** preferences. The advanced highlighting adds specific settings for abstract and inherited method invocations, constants and fields, local variable references and declarations, parameter variables, and static fields and method invocations.

The left of the bottom margin, just to the right of the **fast view bar**, is the **message area**. To the right of this are four fields. These are shown only when the editor has focus. From left to right these fields are the current target of incremental find, whether the file is writable or read-only, the toggle status of the **Insert** key, and the line and column number of your position in the editor. The message area and the area for the current target of incremental find will be blank (as shown in Figure 3.4) if there is no information to display. The right of the bottom margin is the **progress area**, for information about background processing. (See Overview of the User Interface in Chapter 2 for more information on the fast view bar, the message area, and the progress area.)

The editor provides virtually unlimited undo/redo capability through **Edit > Undo** and **Edit > Redo**, or **Ctrl+Z** and **Ctrl+Y**, respectively. You can see other keyboard shortcuts by examining the menu items on the menu bars. If there are keyboard shortcuts, they are listed with the menu item. Recall that you can define your own keyboard shortcuts with your **Workbench > Keys** preferences. Format your code by selecting **Source > Format** or by pressing **Ctrl+Shift+F**. You can also select a Java project, source folder, or package and select **Source > Format** to format all the contained Java source files. JDT provides multiple formatting options and the ability for you to customize your own formatter. Refer to your **Java > Code Style > Code Formatter** preferences.

The Java editor provides a good deal of information through hover help. This works for all types of annotations in the editor and on the marker bar and overview ruler. Hover over a Java element and you'll see its Javadoc. If there is more Javadoc than fits in the pop-up window, press **F2** to anchor the pop-up and then scroll. Press **Ctrl** and hover, and you'll see the element's source. When you have **Ctrl** pressed, the element source changes to a hyperlink that you can click on to open the element in an editor. Figure 3.6 shows both the hyperlink display, mouse indication of the link, and the source hover help. Position your cursor in the parameters for a method invocation to see parameter hints. If there are multiple parameters, the editor will highlight the type of the parameter you are editing.

Select a method, type, or package reference and then press **Ctrl+T**. You'll see a quick type hierarchy view in a pop-up. For types, this is a type hierarchy. For methods, this is a hierarchy of all implementers. Successively press-

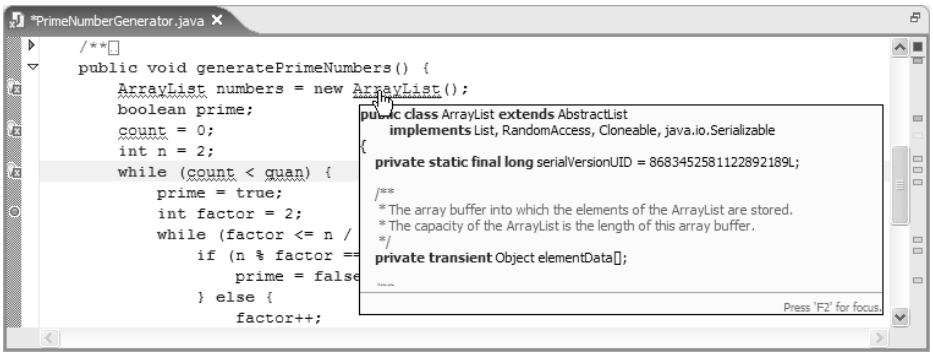

Figure 3.6 Hovering for Element Source

ing **Ctrl+T** toggles between subtype and supertype hierarchies. Select **Ctrl+O** to see a quick outline view in a pop-up. Successively pressing **Ctrl+O** toggles the display of inherited members. Figure 3.7 shows a quick outline view. In these pop-ups, you can navigate through the entries, select one, and press **Enter** to open it in the editor. At the top of the view, begin typing, and only those elements matching the string you typed are shown. One advantage of using these pop-ups to view hierarchies and outlines is that you do not need to keep the respective views open, and hence you can free up real estate in your Eclipse window. Or you can work for a while with the editor in full-screen mode and use the outline pop-up to navigate as required. Try it—this can be very effective depending on the task at hand.

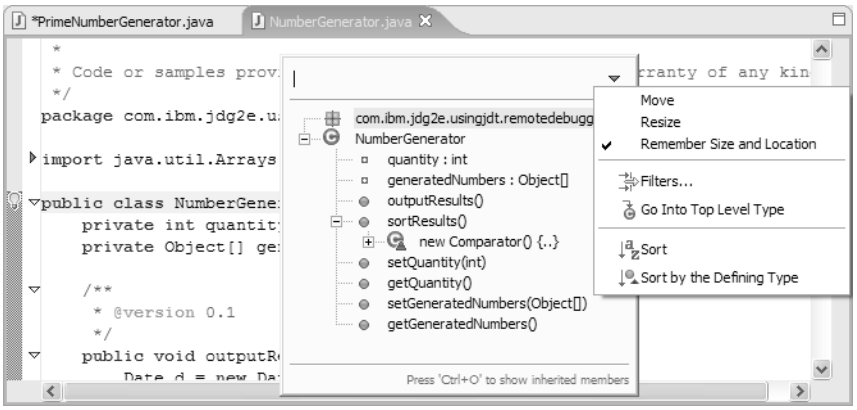

Figure 3.7 Quick Outline View

When you're editing Java code, you'll often find it useful to quickly see all occurrences of references to a field, constant, method, type, local variable, or exception. Position the insertion cursor on one of these items in the Java file you're editing to see other references to the same item highlighted (see Figure 3.8). Enable this capability by toggling **Mark Occurrences** . Position the cursor on the return type of a method to see all the method exit points highlighted. Likewise, position the cursor on an exception to see all the places where that exception is thrown highlighted.

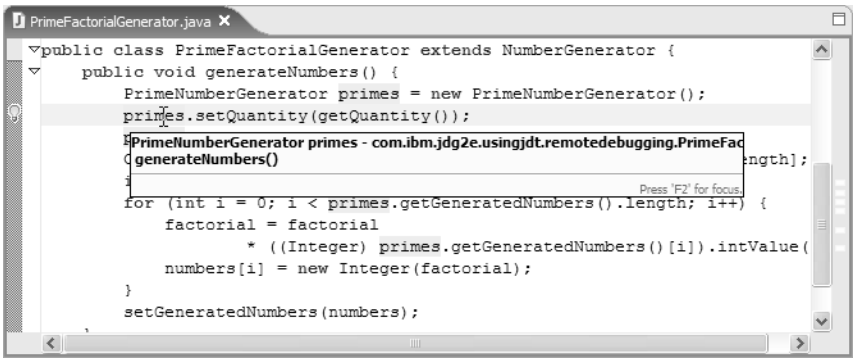

Figure 3.8 Viewing All Occurrences of a Java Item

To see the full Javadoc for an element, select the element in the editor or one of the views and press **Shift+F2**. In order to see Javadoc for your JRE, be sure to set the **Javadoc URL** for your JRE in your **Java > Installed JREs** preferences. To associate Javadoc with a JAR file, select the JAR file on the **Libraries** page of the **Java Build Path** properties on a project. Expand the entry, select **Javadoc location**, and then select **Edit...**. To see full Javadoc in a browser for your code, set the **Javadoc Location** property for your projects. We'll get to generating Javadoc for your Java projects later in this chapter.

Comment and uncomment a line or block of code quickly with **Source > Toggle Comment**, **Add Block Comment**, and **Remove Block Comment**. The keyboard shortcuts are **Ctrl+/**, **Ctrl+Shift+/**, and **Ctrl+Shift+**, respectively. Move the current line, or multiple lines when they have been selected, up or down in the editor with **Alt+Up Arrow** or **Alt+Down Arrow**.

Many of the code generation, block comment, and refactoring operations require a valid Java expression to be selected (we'll discuss refactoring in a moment). Finding a complete logical expression can be at times difficult and/or tedious. Eclipse makes this easy. Place the cursor anywhere in your

code, press **Alt+Shift**, and then press an arrow key (**Up**, **Right**, **Left**, and **Down**) to select logical expressions. These are the shortcuts for **Edit > Expand Selection To**, with the options **Enclosing Element**, **Next Element**, **Previous Element**, and **Restore Last Selection**, respectively.

If you attempt to add an `import` statement and the type or package you want to import does not appear in the dialog, it means you need to add the declaration to your project's **Java Build Path** properties. We'll see how to do this later in this chapter in the Java Projects section.

We saw in Chapter 2, Getting Started with Eclipse, how you can compare different files or different versions of the same file. We'll see in Local History for Java Elements, later in the current chapter, how JDT adds to this for individual Java elements. These are versions of a file that you have saved.

The quick diff ruler indicates changes you've made to the file in your current editing session. This is a lighter-weight, real-time comparison that doesn't rely just on what's in a code repository. It can compare edits that have not been saved yet with those already on disk or those in a code repository. If you don't see this bar, be sure you've enabled it with your **Workbench > Editors > Quick Diff** preferences. Color coding indicates whether the lines are new, changed, or deleted. Hover over an entry on the quick diff ruler to see what has changed. Use the context menu from the quick diff ruler to revert one of more of the changes. In the preferences, you can select to have the reference for the changes be the current file in your workspace or the latest edition in a CVS repository. (We'll discuss using CVS in Chapter 5, Teaming Up with Eclipse.) You can then configure the **Next Annotation**  and **Previous Annotation**  buttons to navigate to the quick diff changes in your file.

In Chapter 2, we presented tasks and bookmarks and how you could use them to mark files and track work. Your Java source is stored in files, so obviously tasks and bookmarks work as we described on your Java files. You can also embed tasks in Java comments through special tags. The default tags configured are `TODO`, `FIXME`, and `XXX`; these are common terms or identifiers used in open source projects. You can customize these tags, create your own tags, and assign priorities in your **Java > Task Tags** preferences. These task tags have many applications. For example, you could define tags of `JOHN` or `DENISE` for team members to record tasks, for example, `//JOHN -- check this`, and `//DENISE -- need doc here`.

Typing Assistance

As you begin to edit Java source, you will notice that the JDT automatically completes some kinds of typing for you. For example, by default, JDT will

close parentheses and braces and put closing quotes on strings. When this occurs, use the **Tab** key to skip past the inserted text to continue typing. The visual clue that you can do this is the green vertical line that indicates where the cursor will jump if you press the **Tab** key. Modify the settings for the kind of assistance you get in your **Java > Editor** preferences on the **Typing** page.

JDT enables you to quickly find a matching brace or parenthesis. Simply position the insertion cursor right after a brace or parenthesis, and its mate will be marked by an enclosing rectangle (see Figure 3.9). To adjust the color of the matching braces indicator, as well as many other things, edit your **Appearance color options** section on the **Appearance** page of the **Java > Editor** preferences.

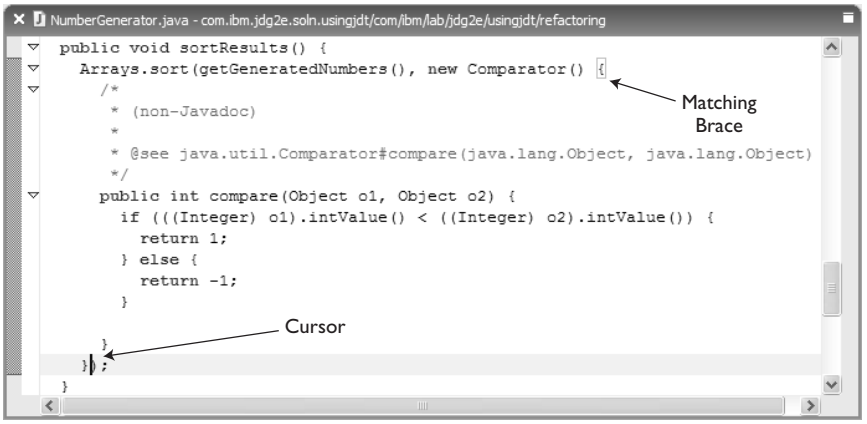

Figure 3.9 Matching Braces and Parentheses

Quick assist is a kind of typing assistance that anticipates your needs based on your editing context and location. As you navigate through your editing session, if the JDT has a proposed quick assist, you'll see a marker on the marker bar. You need to enable this in your **Java > Editor** preferences on the **Appearance** page. Select **Light bulb for quick assists**. Click on the marker or press **Ctrl+1** and you'll see a pop-up with possible assists. Scroll through the proposed assists in the first pane. The corresponding change will display in the second pane (see Figure 3.10). To accept a suggestion, select it and press **Enter**.

The JDT includes a framework for spell-checking your code comments. There are some convenient exclusions, for example, for Internet addresses and words in all caps. Spell-checking is not enabled by default, and you need to configure it with a dictionary. To enable and configure this, edit your **Java > Editor > Spelling** preferences. The preferences are more or less self-

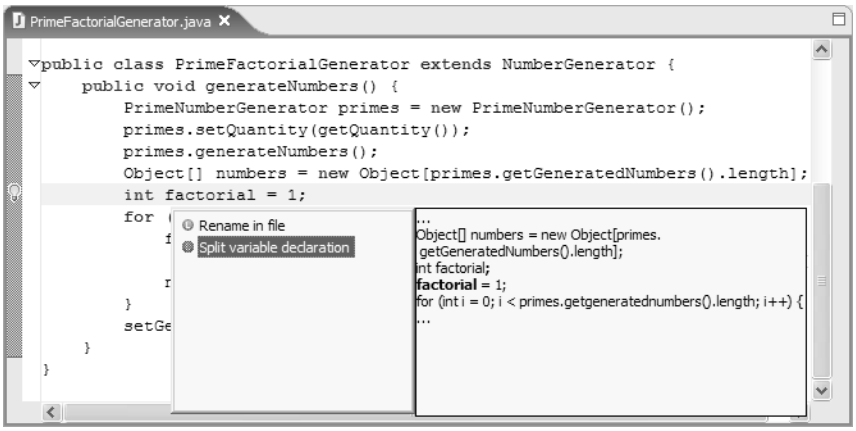

Figure 3.10 Quick Assist Suggestions

explanatory. The dictionary Eclipse needs is simply a flat text file with a list of words. There are several online sources for word lists. Check out http://wordlist.sourceforge.net/.

In the next two sections, we'll see additional assistance the JDT provides for editing Java code—code assist and code generation—including the use of user-customizable templates.

Code Assist

As you edit Java code, you can press **Ctrl+Space** to see phrases that complete what you were typing. This is done based on an ongoing analysis of your Java code as you edit it. You can do this for both Java source and Javadoc. For example, to quickly enter a reference to class `ArrayIndexOutOfBoundsException`, type `arr`, press **Ctrl+Space**, continue typing `ayi` to narrow the list to that class, and press **Enter**. The complete reference is inserted. **Code assist** can also be activated automatically by setting this preference on the **Code Assist** page of the **Java > Editor** preferences. Automatic invocation works based on a character activation trigger and a delay. The default activation trigger for Java is the period (`.`), and for Javadoc it's the "at" sign (`@`). If you type the activation trigger character as you are entering Java source or Javadoc and then pause, code assist will display suggestions (see Figure 3.11).

The code assist pop-up lists matching code generation templates (we'll get to these shortly) and Java references on the left along with their respective icons (class, interface, and method); on the right is the proposed code for the

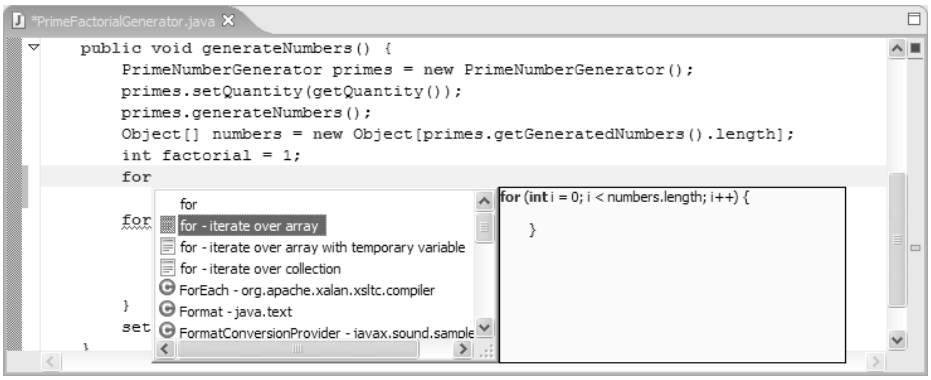

Figure 3.11 Code Assist Suggestions

selected item. These suggestions come from two places: Java references based on an analysis of your code and your defined code templates. Code templates match what you are typing by template name. Pressing **Enter** inserts the selected proposal.

In some cases, you can save typing if you have an idea what you want next or what the suggestion should be. Keep typing with the code assist prompt displayed, and you will see the list narrowed to match your typing. At any point, select an entry and press **Enter**. Or continue typing and when no suggestions apply, the code assist prompt will disappear. This is useful when you have automatic code assist enabled because you can choose to ignore prompts that come up by simply continuing to type what you had been typing.

To reduce the entries that appear as code assist suggestions, consider setting a **type filter**. This is a preference setting, **Java > Type Filters**, that restricts what appears in the Open Type dialog, quick assist suggestions, and quick fix suggestions. For example, if you don't want to see the Abstract Widget Set (AWT) classes, set a type filter for `java.awt.*`.

You can use code assist in a variety of situations, including stubs, `for` loops, `public` methods, and `try/catch` expressions. Take a look at the templates in your **Java > Editor** preferences on the **Templates** page to see what's available. In addition, code assist works for the following.

- *Overriding inherited methods*. Position the insertion cursor within a class definition, but outside any method, and activate code assist. You'll see a list of methods this class can override or must define based on an interface implementation.

- *Import statements*. Type `import` and begin typing the name of a package or type, and then activate code assist to see a list of types and packages.
- *Variable references*. Begin to type the name of a variable or field in an expression, and then activate code assist. It provides a list of possible references, including variables within the scope of the expression.
- *Getters and setters*. Activate code assist within the body of a type, and code assist will suggest getter and setter methods for fields lacking them.
- *Default constructors*. Code assist can present a suggestion for a default constructor for a class.
- *Anonymous inner classes*. Activate code assist within the body of an anonymous inner class and it will present a list of methods to define.
- *Javadoc*. Activate code assist in Javadoc. You can add Javadoc HTML tags and keywords. What's slick about this is that code assist can make smart suggestions based on an understanding of your code. For example, typing `@param` and then activating code assist will display a list of method argument names. Typing `@exception` and then activating code assist will display a list of exceptions a method throws.

When the JDT inserts a code stub from a template, if there were variables to substitute, the JDT makes suggestions based on the template, your Java source, and where in your Java source you are inserting the template. For example, if you use a template to insert a `for` statement to iterate over an array, and your method has an array as a local variable or parameter, the JDT will suggest this local variable or parameter is the array to use and substitute its name for the template variable.

Some assists, when they are inserted in your code, are enabled for further editing. This special editing mode is indicated with outlining and shading. For example, in Figure 3.12, a `for` loop was inserted. Values you can edit (in this case, the first `i` and `numbers`) have a border around them. Multiple instances (in this case, the additional `i`'s) are shaded. When you change one of the multiple instances, all change. There is a placeholder, a vertical line, for the body of the `for` loop. This notation makes it easy to edit the insertion and continue coding. Tab through the editable values, changing those you want, end at the insertion point for the `for` loop body, and continue your coding there. Very efficient. If you want to bypass this selective editing assistance, press **Esc** and you'll be able to edit the entire inserted expression.

Some JDT dialog entry fields are enabled for code assist. The enabled entry fields are indicated by a small light bulb annotation near the field on

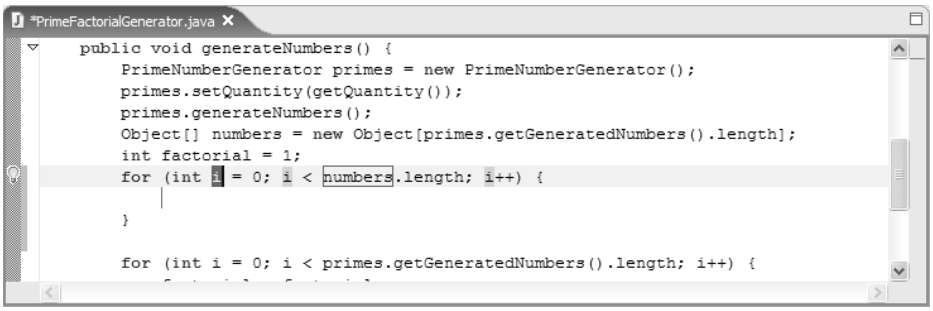

Figure 3.12 Code Assist

the dialog. This light bulb will not appear until you tab to the field. Position the cursor in the entry field, optionally begin typing, and press **Ctrl+Space**. In Figure 3.13, we are using code assist while creating a new Java class to help identify its package.

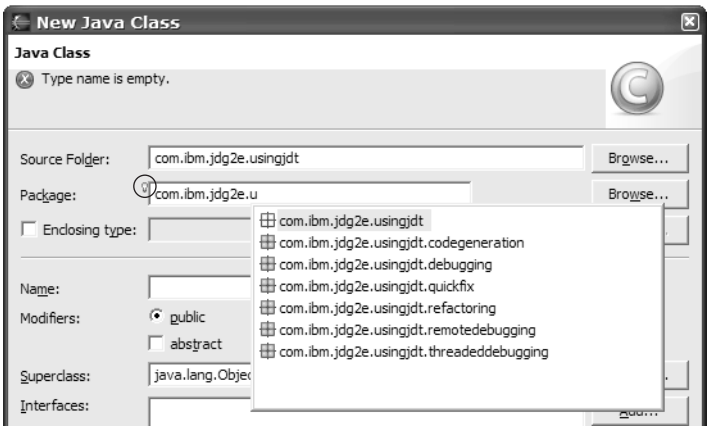

Figure 3.13 Code Assist in a Dialog

Code Generation

In addition to code assist, the JDT provides other code generation capabilities. These are options available under the **Source** menu item, from the editor's context menu, or by pressing **Alt+Shift+S**. Earlier in the Overview of the Java Editor section, we showed how to generate comments and to uncom-

ment code. This is a kind of code generation. In addition, you can generate the following types of code.

- *Import statements*. To clean up unresolved references, select **Source > Organize Imports** to add `import` statements and remove unneeded ones. You can also select an unresolved reference and use **Source > Add Import** to add an `import` statement for that reference only. The keyboard equivalents are **Ctrl+Shift+O** and **Ctrl+Shift+M**, respectively.
- *Method stubs*. Select **Source > Override/Implement Methods...** to see a list of methods to override. Select one or more to have stubs generated.
- *Getters and setters*. A quick way to create getter and setter methods is to select a field and then select **Source > Generate Getters and Setters...**.
- *Delegate methods*. You can generate delegate methods for fields from the editor or any of the Java views. Delegate methods are a form of shortcut for other classes accessing fields in this class. For example, let's say you have a class `Foo` with a field `list` of type `ArrayList`. Further, you know most classes accessing this field are going to be inquiring about its size. Rather than have another class code `aFoo.getList().size()`, you could generate a delegate `size` method for your class for field `list`. To get the size of `list`, you would instead code `aFoo.size()`. Eclipse makes it easy to generate these shortcuts. From a Java view, select a type or a field, or within the editor position your cursor in the body of a type with fields, and then select **Source > Generate Delegate Methods...**.
- *Try/catch statements*. If you select an expression and then **Source > Surround with try/catch block**, the code is analyzed to see if any exceptions are thrown within the scope of the selection, and `try/catch` blocks are inserted for each. This works well with the **Alt+Shift+Up** and **Alt+Shift+Down** expression selection actions to accurately select the code to which you want to apply `try/catch` blocks.
- *Javadoc comments*. You can generate Javadoc comments for classes and methods with **Source > Add Javadoc Comment**. The Javadoc is generated based on template definitions in your **Java > Templates** preferences.
- *Constructors from fields*. Create a constructor for any combination of fields by selecting **Source > Generate Constructor Using Fields...**. Select the fields you want the constructor to accept as input.
- *Superclass constructors*. When you create a class, you have the option to generate constructors from its superclass. If you elect not to do this, you can come back later and add the superclass constructors with **Source > Add Constructor from Superclass...**.

You can view and modify the specifications for the code and commentary the JDT generates. Customizing these preferences is useful, for example, to include a common copyright statement for all your source files. We'll see how to edit these later in this chapter in the section Using Code Templates.

Navigating Java Errors and Warnings

The JDT can flag many different kinds of errors and warnings, including syntax errors, build path errors, unnecessary type checks, indirect access to static members, and superfluous semicolons. Errors and warnings show up in several ways (see Figure 3.14).

Errors are shown as entries in the Problems view, as markers on the marker bar, as label decorations in the views, as error clues (red underlining) in the source, and as small red and yellow rectangles on the overview ruler on the right border. A quick fix icon is displayed for errors for which the JDT can suggest a solution. We'll get to quick fix in the next section.

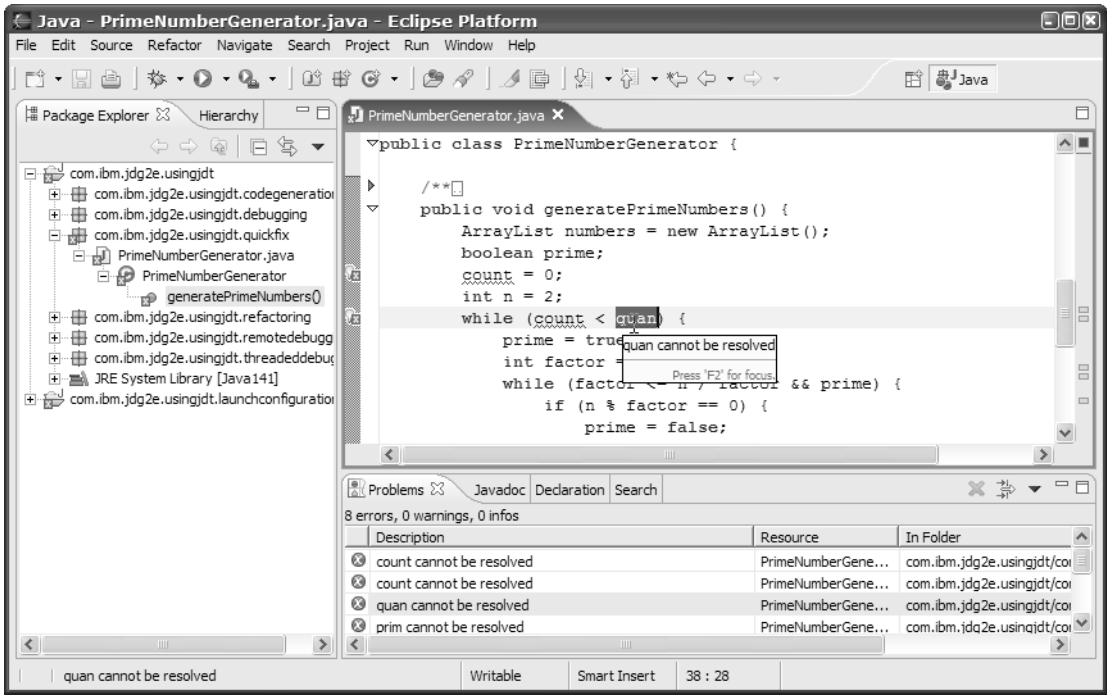

Figure 3.14 JDT Error Indicators

Hovering over an error indicator in the marker bar or overview ruler displays the error message. You can do the same by hovering over error text, underlined in red, within the editor. When you do this, if you want to copy the error message, press **F2** to put the focus on the pop-up, select the error message text, and then use **Ctrl+C** to copy it. Click on an error indicator in the overview ruler to cause the code in error to scroll into view and the error to be selected. Recall that because errors are annotations, you can use **Next Annotation** (**Ctrl+.**) or **Previous Annotation** (**Ctrl+,**) to scroll through and select errors. When you do so, the error message is displayed in the message area.

In the Problems view, double-click on an entry to go to the line in the file with the error. If the file is not open, it is opened in an editor. If the error is in the file you're currently editing, click on an entry in the Problems view to scroll the file, and select the error. If the error message is truncated in the Problems view because of the width of the column, hover over it to display the full message, or select the error to see the full text of the error in the message area.

Fixing Java Errors with Quick Fix

For errors that have suggested corrections, you will see a quick fix icon on the marker bar . You can use quick fix to correct a wide variety of problems, from spelling errors and missing `import` statements to declaring local variables and classes and externalizing strings. Quick fix also works on build path errors such as unresolved `import` statements. Click on the icon to see a list of possible solutions. You can also position the cursor within the red underlined text and then press **Ctrl+1** to display a pop-up with a list of suggestions. Select an item in the pop-up to see the proposed fix or a description. Press **Enter** to make the correction. As we described earlier in the Code Assist section, text in the inserted code is marked for quick customization. Use the **Tab** key to quickly cycle through these. This display, selection, and editing works as it does for code assist.

Refactoring

Refactoring refers to a class of operations you use to reorganize and make other global changes to your code. This includes changes to Javadoc, string literals, launch configurations, and XML files that define plug-ins, called **manifest files**. We'll get to launch configurations in Chapter 4, Running and Debugging Java, and plug-in manifest files in Chapter 7, Extending Eclipse

for Fun and Profit. You may need to refactor code in response to API changes, to improve maintainability, or to make naming more consistent.

To refactor your code, select a Java element in one of the views, position the insertion cursor in an element reference in the editor, or select an element or an expression in the editor. Then select **Refactor** from the menu bar or the Java editor or view context menu, or press **Alt+Shift+T**. You will see a menu with a list of possible refactorings. What appears on this list depends on your code and what you have selected. Some of the refactoring operations require a valid expression to be selected. The **Expand Selection To** menu choices and **Alt+Shift** keyboard shortcuts we discussed earlier in the Overview of the Java Editor section make this easy.

The JDT refactoring capabilities include the following.

- **Move**
  You can move Java elements and, optionally, modify references to the element being moved. This can be used for fields, methods, classes, interfaces, packages, source files, and folders.

- **Rename**
  You can rename Java elements and, optionally, modify references to the element being renamed. This can be used for fields, variables, methods, method parameters, classes, interfaces, packages, source files, and folders.

- **Pull Up**
  This moves a field or method from a class to its superclass, if possible. For example, you can't pull a method up into a class you can't modify.

- **Push Down**
  This moves a field or method to a class's subclass.

- **Extract Method**
  This creates a method from a selected code fragment, complete with parameters based on variables used in the fragment. The JDT looks for other instances of the code fragment and offers to replace them with the method invocation.

- **Change Method Signature**
  This lets you change parameter names, order, and type; exceptions thrown; and access modifiers. You can also update all references to the method being modified.

- **Extract Interface**
  Use this to create an interface based on a class definition.

- **Generalize Type**
  Select a variable, parameter, field, or method return type. This displays a view up the type hierarchy to allow you to quickly change the type of the selected element.
- **Use Supertype**
  Select a field. This replaces references to a type with references to one of its supertypes, if possible, based on use in your code.
- **Extract Local Variable**
  This creates a new variable from the selected expression and replaces all occurrences of the expression with a reference to the variable within the scope of the enclosing method.
- **Extract Constant**
  Select a literal, such as an integer or a string. This creates a static final variable and replaces all occurrences of the literal.
- **Inline**
  This does the opposite of the **Extract** actions. You can use it for method declarations, invocations, static final fields, and local variables. For example, for a method invocation, the method source is inserted inline where the method invocation had been.
- **Convert Local Variable to Field**
  This does just what it says. Start by selecting a local variable in a method.
- **Introduce Parameter**
  Within the body of a method, select a valid expression that returns something other than `void`. **Introduce Parameter** adds a parameter of the same type as the selected expression to the method, replaces the selected expression with the parameter, and updates invocations to the modified method. Note that you still need to update the changed method invocations to add the parameter—there's no magic here.
- **Introduce Factory**
  This marks a static constructor `private` and replaces it with a static factory method. Calls to the constructor are replaced with calls to the factory method.
- **Encapsulate Field**
  This goes one step beyond what **Source > Generate Getters and Setters…** does. It both generates getters and setters for a field and replaces direct references to the field with getter and setter invocations.

You can refactor by dragging and dropping Java elements in the Java views, for example, dragging a type from one package to another in the Package Explorer view, or dragging a type onto another class to create a nested type.

When you request a refactoring operation, a dialog appears (see Figure 3.15). The first page displays the minimum information needed to complete the operation. Typically this is a new name or destination and options, for example, to update references to the changed element. Specify this information and select to perform the refactoring, or select **Preview** to see what code changes would result.

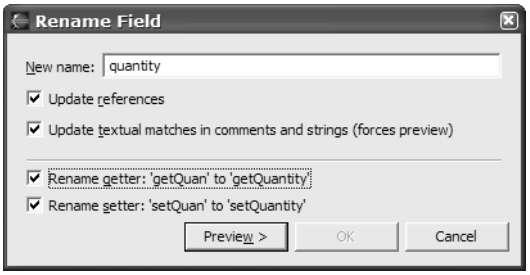

Figure 3.15 Refactoring Wizard

If you select to preview the proposed code changes, you'll see a side-by-side comparison (see Figure 3.16), which shows each of the changes and a before-and-after view of the code. Select the changes you want to apply by checking entries in the top pane. Navigate quickly through changes by clicking on the change entries (rectangles) on the overview ruler to the right.

To undo or redo a refactoring operation, use **Refactor > Undo** or **Refactor > Redo**, respectively. These operations are different from **Edit > Undo** and **Edit > Redo**. The refactoring undo is aware of all of the changes made across all projects; the edit undo is aware of only the changes made as a result of the refactoring operation in the active editor. When you drag and drop elements in the views to affect refactoring, the **Refactor > Undo** and **Refactor > Redo** actions are not available.

Using Code Templates

Templates are outlines, or skeletons, for common Java code or Javadoc patterns. Templates allow you to quickly insert and edit code expressions with a minimum number of keystrokes. They help ensure more consistent code. Templates can contain Java or Javadoc source, variables that are substituted when

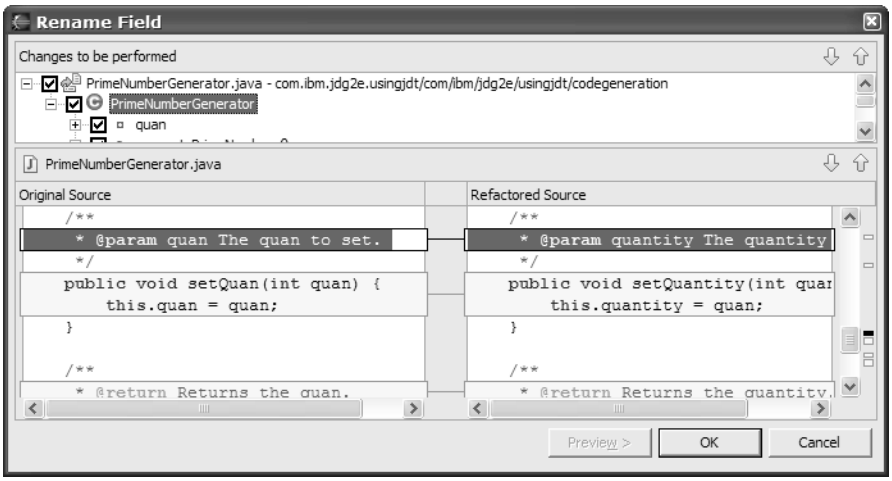

Figure 3.16 Previewing Refactoring Changes

the template is inserted in the Java code, and a specification of where the cursor should be positioned when you've finished editing the inserted code.

It's easy to create your own templates or to modify those provided. For instance, you might add your own template with a `try/catch` block to invoke a special error handling routine. Or you might choose to generate a comment with the user's name and date, or add a task tag to create a task to record additional work. You create templates in your **Java > Editor > Templates** preferences. Select a template to preview it in the bottom pane. Disabling a template will cause it not to appear as a suggestion in the code assist prompt. You create and edit templates in the Edit Template dialog, shown in Figure 3.17. To insert a variable, use the **Insert Variable...** button on the dialog, or press

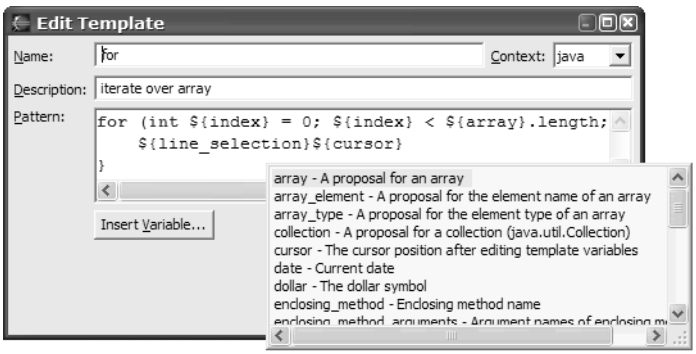

Figure 3.17 Edit Template Dialog

Ctrl+Space in the **Pattern** field to see a list of possible variables. `${cursor}` is the variable that indicates where the cursor is placed after the user has tabbed through the variable fields.

There is another set of templates in your **Java > Code Style > Code Templates** preferences: the specifications for the code and comments inserted when you use one of the **Source** actions to generate code. These include, for example, templates for getter and setter methods, new Java files, and methods. Customize these as you would your **Java > Editor > Templates** preferences.

Templates can play an important role in a team environment to help ensure consistency and adherence to coding conventions and standards. This means you may need to share your templates among team members. One way to do this is to export the templates to XML files. Both sets of templates have **Export...** options to allow you to do this. Then share them among your team. For information on how to do this through CVS, see Chapter 5, Teaming Up with Eclipse. You can also alter the default preference settings for Eclipse. For this, refer to Chapter 6, Managing Your Eclipse Environment.

Externalizing Strings

The JDT allows you to externalize strings in your Java code, that is, to remove string literals from your Java code to separate files and replace them with references. This is most often done in order to have the strings translated into other languages. The Externalize Strings wizard scans your code for string literals and allows you to indicate which string literals are to be externalized and which are not. The JDT also provides support for finding unused and improperly used references to externalized strings. You can use a preference setting to flag strings that have not been externalized as compile errors or warnings. This is on the **Advanced** page of the **Java > Compiler** preferences.

To begin the process, select a project, package, or folder, and then select **Source > Find Strings to Externalize...** from the menu to display the Find Strings to Externalize dialog, which shows Java files containing strings that have not been externalized. Select a file and then select **Externalize Strings...** to go to the Externalize Strings wizard. Alternatively, select a Java project or package and then select **Source > Externalize Strings...** from the context menu.

On the first page of the Externalize Strings wizard, shown in Figure 3.18, you specify which strings are to be externalized (for translation) and keys for accessing the strings. The icon in the left column indicates the status of the string: externalized ☑, ignored ☒, and internalized ↷. Change the status by clicking on the icon to toggle it or by using **Externalize**, **Ignore**, or **Internalize**. If you have your preferences set to flag nonexternalized strings as errors

or warnings, and you select **Externalize** or **Ignore** for the string, it will no longer be flagged as an error or warning. If you select **Internalize**, the error or warning will remain. You can specify a prefix that will be added to the key (when your code is modified, not in the dialog). You can also edit the key to specify a different value by selecting the entry and then clicking on the key.

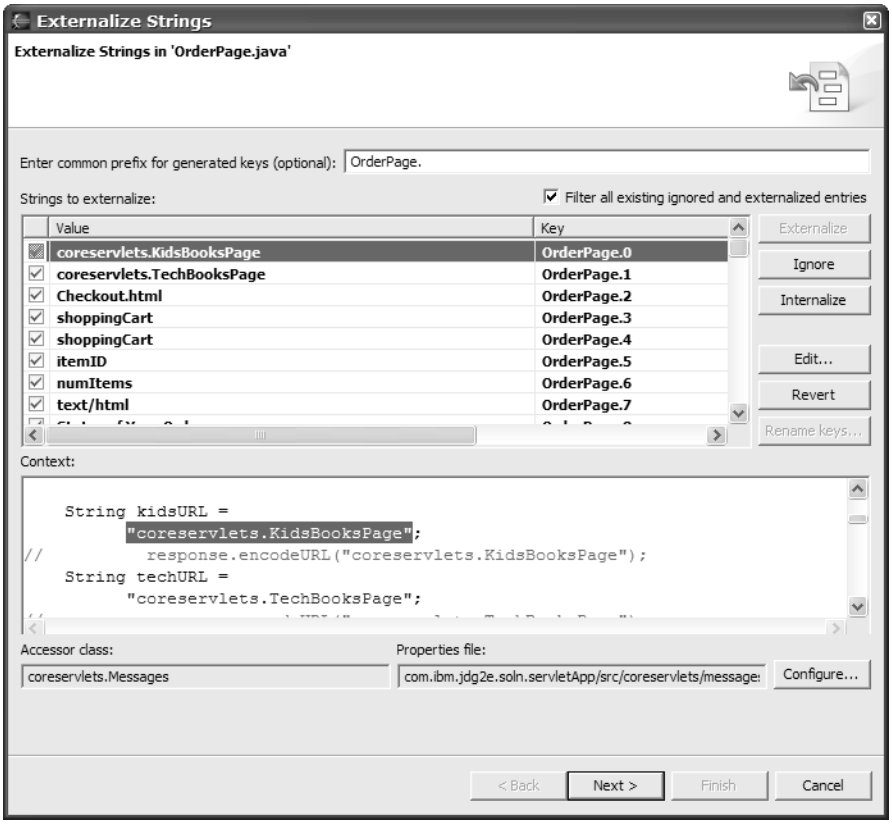

Figure 3.18 Specifying Strings to Externalize

Even if you do not intend to translate a string, it still may be useful to externalize it. For example, if you reference the same string value in several places in your code, perhaps a user interface element, externalize it to get it declared once. Then changing it in the future will be easier.

The JDT generates and adds to your project an accessor class, a method, and a properties file. To change the defaults Eclipse provides, or to specify you're using an alternative implementation, select **Configure...**.

The final page of the wizard is a before-and-after view of your code, similar to the before-and-after view when refactoring. In fact, here you are doing a sort of refactoring. In the top pane, you select to apply or not apply changes for each of the proposed modifications and select **Finish** when ready to make the changes to your code. This replaces the selected strings with references based on the accessor class, generates the accessor class, and creates the file containing the strings. To undo string externalizations, select **Refactor > Undo**. This also removes the generated accessor class.

Generating Javadoc

You generate Javadoc by selecting **Project > Generate Javadoc...** or by exporting it with **File > Export... > Javadoc**. Generating Javadoc requires that the JDT know where to find `javadoc.exe`, the program that actually performs the Javadoc generation. The `javadoc.exe` program is not shipped as part of Eclipse; it comes with a JDK distribution. For example, in Sun's 1.4.2 JDK, `javadoc.exe` is located in the `bin` folder under the main installation folder. If this is not set, do so by selecting **Configure...** from the Generate Javadoc wizard (see Figure 3.19).

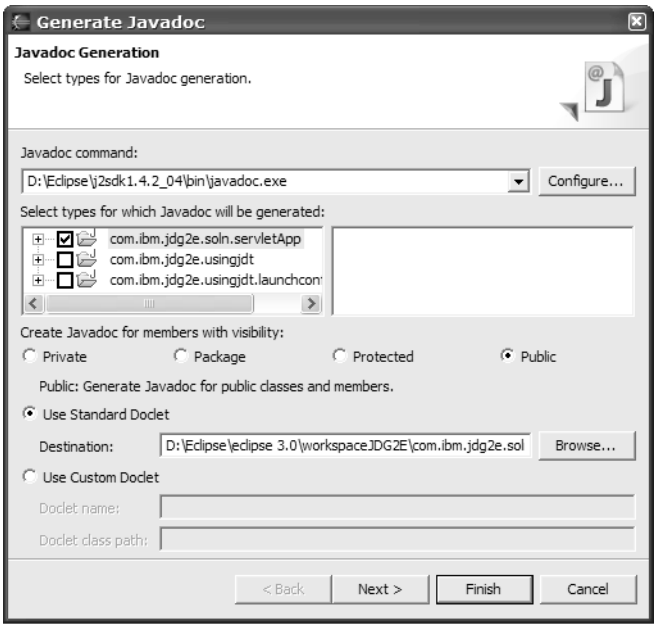

Figure 3.19 Generating Javadoc

You have a number of options for generating Javadoc for one or more of your Java projects. Figure 3.19 shows the first page of options. Select **Finish** to generate the Javadoc or **Next** for more options.

For the visibility settings, **Private** generates Javadoc for all members; **Package** for all default, protected, and public members; **Protected** for all protected and public members; and **Public** for only public members. For more information on all the Javadoc generation options, refer to the "Preferences" topic in the "Reference" section of the *Java Development User Guide*.

The second page of the wizard allows you to specify Javadoc for references to external libraries or other Eclipse projects you select. This is very useful. To do so, the Javadoc location for the library or project must be specified. If it is not, when you select a library or project, you will see a "not configured" warning message at the top of the wizard. Use the **Configure...** button on this page, or go to your library definition or project properties, to define the Javadoc location.

The third page allows you to save your Generate Javadoc wizard settings as an Ant script, among other options. This is also useful. You can then manage this script as part of your project and add it to the Ant view or define it as a builder on the project.

When you generate the Javadoc, you will see a prompt asking whether you want to update the **Javadoc Location** properties of the projects you are generating Javadoc for. You should select to do so. This enables you to browse the generated Javadoc in an external browser. It also allows you to generate Javadoc references to this project from other projects. Output from the Javadoc generation shows up in the Console view. You should review this to ensure there were no errors.

Writing Java for Alternative JREs

Eclipse uses a JRE for two purposes: as a way to run Eclipse itself and as a target execution environment for your Java code. This target execution JRE can be the same one used to run Eclipse or an alternative one. If you want to develop Java code that uses a different JRE as a target from the one that runs Eclipse, you need to configure Eclipse to define an additional JRE and the associated JDK compliance preferences. You can make this configuration change global or specific to a project.

In either case, first you need to specify the JRE you want to use as your target environment. Do this with your **Java > Installed JREs** preferences (see Figure 3.20). Set the **Javadoc URL** field in order to enable browsing of the JRE Javadoc. If you anticipate debugging into the JRE classes, associate

source code with the JRE system library files by selecting **Attach Source...**. Normally, Javadoc and JRE source are not included with JRE distributions; rather, you need to get a corresponding JDK distribution.

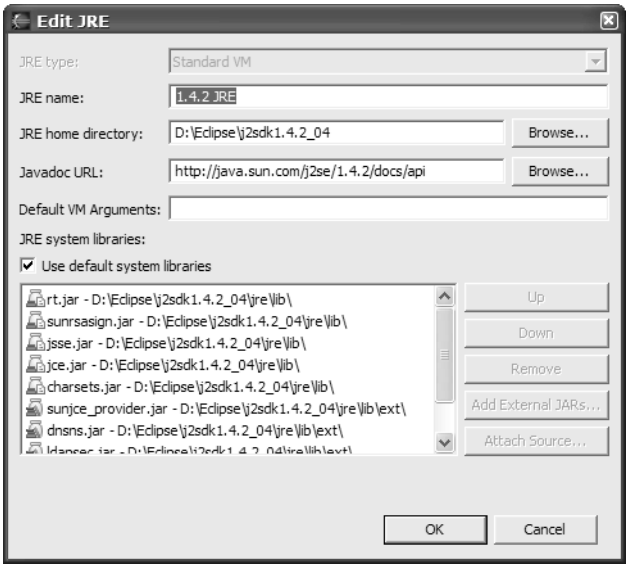

Figure 3.20 Setting Installed JRE Preferences

If you want to make this JRE the global default, set the default JRE in the preference (with the associated check box) to be the one you just installed. This specifies the JRE used as your development target. It does not affect the JRE used to run Eclipse. You'll see in Chapter 4, Running and Debugging Java, how to pass parameters to the JVM when you run or debug your programs. If you use some common JVM parameters, specify them here in **Default VM Arguments** so you don't have to specify them for each project. To specify project-specific JVM parameters, you will use a launch configuration. We'll get to that in Chapter 4.

Second, set your **Compiler compliance level** setting, on the **Compliance and Classfiles** page of the **Java > Compiler** preferences, depending on what your code requires.

If you do not want to make this JRE the global default but rather want to use it in a specific project, you can do this through the project's properties. Once you've defined the JRE in your **Java > Installed JREs** preferences, open the properties for the project by selecting the project in the Package Explorer

view and then selecting **Properties** from the context menu. On the **Libraries** page of the **Java Build Path** properties, select **Add Library...**. In the Add Library wizard, select to add a **JRE System Library**. On the next page of the wizard, under **Alternate JRE**, select the JRE you just installed. Back on the **Libraries** page, remove the default JRE entry. Switch to the **Java Compiler** properties, and select **Use project settings**. Adjust these preferences, which are now project specific, as appropriate. Verify the **Compliance and Class-files** preferences.

Developing for J2SE 1.5

At the time of this writing, J2SE 1.5 was still under development, and the Eclipse JDT team was in the midst of implementing J2SE 1.5 support in Eclipse, code name Cheetah. Instructions on using the interim work on Eclipse 3.0 drivers are available on the Development Resources page of the JDT Core component, JDT subproject. To use the early support, you need to install it from an Eclipse update site. The Cheetah instructions include a pointer to the update site. For more information on installing from update sites, refer to Installing a Feature from an Update Site in Chapter 6, Managing Your Eclipse Environment. This early support shows up first in the compiler and manifests itself in some J2SE 1.5-specific preferences. As the core work stabilizes, you can expect to see enhancements begin to show up in the JDT user interface.

Compiling Your Code

By default, whenever you create, rename, delete, or edit and save a Java element, the JDT compiles (or "builds" in Eclipse terms) the changed element and all other elements impacted by this change. The resulting `*.class` files are put in the project's output folder(s). The output folders can be different folders from the source folders or the same folders. By default, they are the same. We'll see in Java Projects later in this chapter how you can organize your projects into separate source and output folders. When you have this organization and build your project, other non-Java files in your source folder(s) are copied to the output folder(s), subject to inclusion and exclusion patterns.

The JDT is a complete development environment. All it requires is a JRE, not a JDK. This means Eclipse comes with its own Java compiler. It does not use `javac`. If you want to compile the code you develop in Eclipse with `javac`, you can do so through Ant build scripts. We'll get to those in a moment. The JDT provides a great deal of flexibility in allowing you to customize this build process through Java project properties and **Java > Compiler** preferences. In

these preferences, set global settings for how you want various kinds of errors to be handled, how you want your class files to be generated, and how build path errors (such as circular dependencies) should be handled.

In your project properties, you define the inclusion and exclusion patterns for which files in your source folders get processed during builds. The goal here is to provide a mechanism so that only the files needed by your application at runtime are copied to the output folder(s). It allows greater flexibility in how you organize to develop your application. Your project properties include settings for defining additional builders (compilers), overriding the global compiler preferences with project-specific ones, a more general mechanism for excluding files and folders in the source folder from being built, and build path customization. We'll see this in more detail in Working with Java Elements later in this chapter.

Eclipse Ant Integration

Ant is a Java-based `make`-like utility offered by the Apache Software Foundation that you can use to drive your build process. For more information on Ant, see http://ant.apache.org/. Ant runs on any Java 1.1 system or later, which makes it largely portable. As a Java-based open source project, it's easy for people to improve and add to Ant. We'll see an example of this in Exercise 4, Developing a Simple Web Commerce Application with Eclipse, with some specific Ant tasks the Tomcat folks created and delivered with Tomcat.

A Quick Ant Primer

Ant scripts (i.e., build files) are XML files that specify tasks and their dependent tasks. Ant itself is command line driven. However, Eclipse comes with Ant built in and with some pretty slick integration. You may be thinking, "Eclipse compiles Java code, so why do I need another build utility?" The simple answer is that compiling is only one step in a build process that can often get quite complicated when you factor in automated testing, statistics generation, JAR creation, and deployment.

The basic Ant constructs are projects, properties, targets, and tasks. A **project** is a top-level construct, one per build file. Projects contain properties and targets. **Properties** are name/value pairs that you specify inside or outside a build file. For example:

```
<property name="output.home" value="bin"/>
```

You can then reference the property with `${output.home}`. Ant comes with some predefined properties, such as `${basedir}` for the directory in which the

Ant script is running. Eclipse also adds some properties, such as `${eclipse.home}`. These are located on the **Properties** page of your **Ant > Runtime** preferences. Don't confuse these with Eclipse properties on resources. Ant properties are different and apply to Ant only.

Targets are the units of work that make up projects. Targets are comprised of sequences of tasks, and they depend on other targets. **Tasks** are the commands executed inside targets. They do the work. Examples of tasks are `<javac>` to compile and `<copy>` to copy files.

Using Ant in Eclipse

Ant integration in Eclipse shows up in several ways. Eclipse has an Ant script editor, including code assist (**Ctrl+Space**) and formatting (**Ctrl+Shift+F**). When you're using the Ant script editor, the Outline view shows an outline of the script you're editing, broken down into properties, targets, and tasks. Ant build scripts in Eclipse are called, by default, `build.xml`. The Ant editor is associated with this file name.

There are two basic ways to run Ant scripts (i.e., to execute Ant targets) in Eclipse. You can do this through the Ant view, or you can define Ant builders. To run Ant targets from the Ant view, you need to add the Ant script, for example, `build.xml`, to the view (see Figure 3.21).

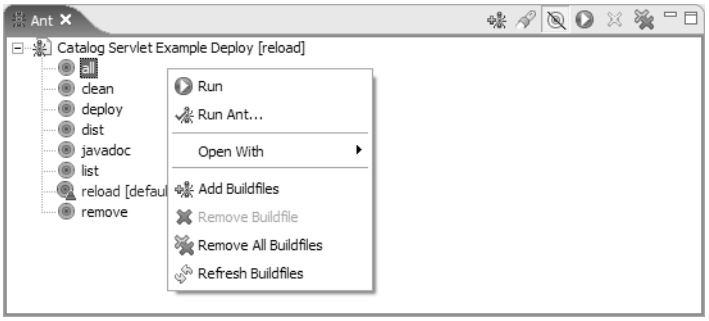

Figure 3.21 Ant View

You can do this by dragging and dropping the file onto the view or by using one of the **Add Buildfiles** buttons, or , on the toolbar. Then double-click on a target to invoke it. Output is shown in the console view. Selected sections of the output are enabled as links, including specific tasks and error messages (see Figure 3.22). Select a link to go to the task in the build script or the failing statement.

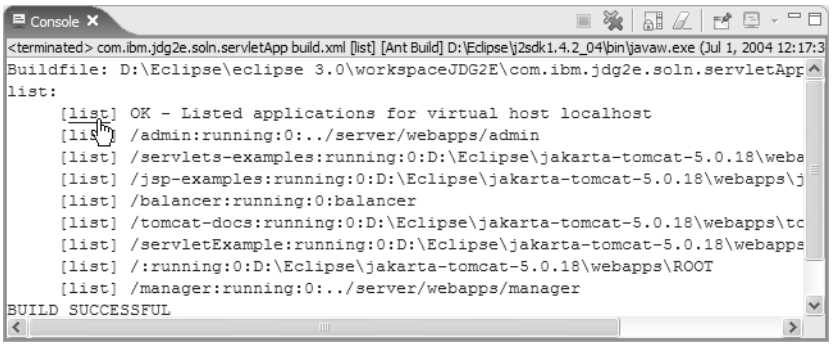

Figure 3.22 Ant Script Output

This is a great way to execute Ant scripts, but it is manual. There is a way to automate this. As you saw in Chapter 2, Getting Started with Eclipse, Eclipse projects have builders. Eclipse allows you to define Ant targets as builders. Do this by opening the properties for a project and editing the **Builders** properties (these are Eclipse properties now, not Ant properties); you want to create a new **Ant Build**. You then select an Ant script and one or more targets in that script.

There are **Ant** preferences you can use to define your Ant runtime, Ant properties, and Eclipse Ant tasks. These are your **Ant > Runtime** preferences. The **Classpath** page lists JAR files that make up the Ant runtime. If you define your own Ant tasks or need to reference Ant tasks in other JAR files, you need to list them here, under **Ant Home Entries**. The **Tasks** page lists the tasks Eclipse defines for your, and its, use. The **Properties** page lists the Ant properties Eclipse defines. You can add your own here. Consider defining properties here if they are global to your Eclipse environment.

Working with Java Elements

In the Fundamentals section earlier in this chapter, we presented an overview of how to create different kinds of Java elements. In this section we'll go into more depth on Java projects, the creation and import of Java elements, and details on local history for Java elements.

Folders

There are three types of folders in Java projects: source folders, output folders, and other nonsource folders. This is an important distinction

because it affects how builds are performed. When you create a source folder, the JDT adds the source folder to the project's build path. Nonsource folders are not added to the project's build path. If you create a folder and later want to make it a source folder, do so from the Java Project Properties dialog, **Java Build Path** properties, on the **Source** page. When in doubt, check the project's `.classpath` file. It is the persistent representation of the properties shown on the project's build settings pages. Source and output folders can exist only in Java projects.

If you create a Java project and select to have your Java source and output files in the same folder, and then decide later to create a source folder, this forces the JDT to change the organization of the project to have separate source and output folders. If you already have Java files in the project, you will have to move these manually to a source folder. It may be easier to create a new project with the folder and package organization you desire and then use the refactoring actions to move the Java elements.

Folders do not have to physically reside in your workspace or within the containing project. You can have linked folders as source and output folders. For example, you might create one or more remote output folders and use the JDT build process to do a simple deploy of your binaries and other output resources. For more information on linked folders, see More on Projects in Chapter 2, Getting Started with Eclipse.

Java Projects

The JDT projects permit you to override certain global Eclipse preferences with settings specific to a project. These include the target JRE and preferences for warnings and errors, JCK (Java Constraint Kit) compliance, and class file generation. To set project-specific preferences, open the properties for a project by right-clicking on the project and selecting **Properties** from the context menu. Select **Java Compiler** properties, and then select **Use project settings**. The preference pages are enabled so you can set these project-specific preferences.

Within a Java project, there are two basic ways to organize your code. For small projects, you may choose to have everything in the same package or folder: `*.java` source files, `*.class` output files, and other files required by your program. With this organization, when you save a `.java` file and the file and project are built (compiled), the resulting `.class` files are put with source files in the same folder.

For projects that have lots of modules, you may choose to organize your `*.java` source and `*.class` output files in separate folders in a project. In this

case, each project has one or more source folders and one or more output folders. The build process takes inputs from one or more source folders, or folders under them, builds them if necessary, and puts the results in the respective output folders. This approach allows you to organize and subdivide your Java packages, while the output folders contain all runtime resources.

Two preference settings allow you to customize this source code organization and build processing. **Java > Build Path** preferences allow you to specify the default organization for when you create new Java projects. You can override this in the New Project wizard when you create a new Java project. If you have already created a project, you can change this organization on the **Source** page of the project's **Java Build Path** properties. You can also change the policy for which files do *not* get copied to the output folder when a project is built. This is the **Filtered Resources** preference on the **Build Path** page of the **Java > Compiler** preferences. You can also selectively include and/or exclude files and folders from the build process with a project's **Java Build Path** property, on the **Source** page. Expand a source folder entry and edit the **Included:** and/or **Excluded:** entries. If you specify both an inclusion pattern and an exclusion pattern and there is a conflict, the exclusion pattern takes precedence.

When you organize your code into separate source and output folders, by default, there is one output folder per project. You can configure multiple output folders in a project, one per source folder. On the **Source** page of the **Java Build Path** project properties, select **Allow output folders for source folders**. Then expand each source folder and edit the **Output folder** definition for the selected source folder (see Figure 3.23). This is useful, for example, to separate Java source from, say, HTML and image files in your project. When your project is built, your Java class files are put in one folder, HTML in another, and image files in a third.

Builders

The **Builders** Java project property defines the builders (compilers or other programs) that get run when resources change. With Eclipse it is easy to add Ant builders as well as other programs. For more information on builders, see Chapter 2, Getting Started with Eclipse.

Java Build Path

A project's build path serves two purposes: It specifies the search sequence for Java references in the code in the project, and it defines the initial runtime classpath for the code. When you first create a project, its build path is set to

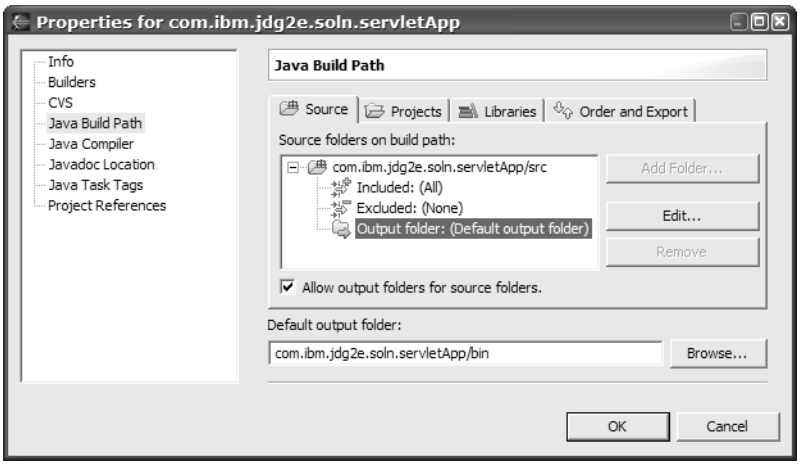

Figure 3.23 Defining an Output Folder for a Source Folder

contain the project itself and the default JRE. The default JRE is defined in your **Java > Installed JREs** preferences. At runtime, you can modify a project's classpath, which is generated from its build path information, by defining a launch configuration (see Chapter 4, Running and Debugging Java). This is useful, for example, if you need to test different runtime environments without modifying your build.

Build path errors show up with other errors in the Problems view. If you have trouble diagnosing build path errors, open the properties on the project and check the **Java Build Path** pages. The icons for the entries indicate whether they cannot be found. For example, a JAR file is represented by . A JAR file that cannot be found is represented by . Preferences for how to handle build path errors are located on the **Build Path** page of your **Java > Compiler** preferences.

Use **Add Folder...** on the project's properties **Source** page to add source folders (see Figure 3.23). You can reference an existing folder or create a new one. The **Projects** page allows you to add other Java projects to the build path of the new project. You need to do this if you are going to make references from the project you are creating to Java declarations in other projects in your workspace.

The **Libraries** page allows you to add other Java libraries to the project build path in order to reference their declarations. For example, if you wanted to develop a Java servlet, you would need to get a J2EE (Java 2 Platform, Enterprise Edition) runtime, extract a JAR file that contained the servlet definitions, and add that JAR file to the project's build path. Select **Add**

Jars... to add JAR or `*.zip` files from other projects, as opposed to adding references to resources in projects not contained in JAR or `*.zip` files, as you do on the **Projects** page. Generally, if you have the resources in a project, you should reference them that way, not within the JAR. This is simply because it's more efficient to modify code in a Java project than code in a JAR file.

To reference a library not defined in a project in your workspace, use one of the following: **Add External JARs...**, **Add Variable...**, or **Add Library...**. Referring to a library as an external JAR is a direct reference to a JAR or `*.zip` file on your file system. This is the quick and easy approach. For more flexibility and especially to enable your work to be more easily shared, you should use either a classpath variable or a user-defined library. Use **Add Variable...** to add a classpath variable to your project's build path. A **classpath variable** is an indirect reference to a JAR or `*.zip` file or folder. To define a classpath variable, use your **Java > Build Path > Classpath Variables** preferences. A **user-defined library** is another indirect reference. A user-defined library allows you to refer indirectly to a series of JAR or `*.zip` files. Do this through your **Java > Build Path > User Libraries** preferences. With an indirect reference, if the location of the target changes, you only have to update the classpath variable or user library definition, not each project. Both classpath variables and user libraries are indirect references.

The advantage of classpath variables is that you can define a single reference to a folder and its contents. However, to reference a library in the folder, you need to select the library file when you add the classpath variable to the project's build path. The advantage of user-defined libraries is that you can define one reference to multiple individual library files, not necessarily in the same folder. User-defined libraries are also easier to share. They can be saved individually as XML files. Classpath variables can only be shared as part of all your exported preferences.

Regardless of whether you refer to a library directly or use a classpath variable or a user-defined library, be sure to set the Javadoc and source location attributes. This will allow you to browse the Javadoc for the library and step into methods in the library while debugging, respectively. For classpath variables, set these attributes in your projects' **Java Build Path** properties on the **Libraries** page. For user-defined libraries, set these attributes when you define the library in your **Java > Build Path > User Libraries** preferences. This illustrates another advantage of user-defined libraries: You specify the associated source code and Javadoc once. With classpath variables, you associate source and Javadoc with each project that references the classpath variable. One final note: While Eclipse supports file-specific encoding, source code read from archives is read using the workspace default encoding.

The **Order and Export** page serves two purposes. First, it allows you to change the order of references to entries in your project's build path. This order becomes important if you have the same declarations in multiple build path entries. Second, by selecting an entry for export, other projects that reference this project will implicitly inherit these exported entries. This means that others can reuse the selected project's referenced libraries without needing to respecify them, reducing the explicit dependencies should the organization of the code later change.

Project References

Project references indicate other projects a given project refers to and dictate how projects in your workspace are built. This property is set based on the projects you include on the **Projects** page of a project's **Java Build Path** properties. For example, if you specify in project A's **Java Build Path** properties that it refers to project B, project A's **Project References** properties are updated. Then, whenever project B is built, project A is (re)built as well, because A referenced B. You affected this when you set the **Java Build Path** properties for project A. You do *not* affect this if you simply set project A's **Project References** properties to refer to project B. Doing so will *not* cause A to be (re)built when B is built. The bottom line is that project references for Java projects are managed by the JDT. You can examine them, but making changes will have no impact.

Classes and Interfaces

When creating a new class or interface, there are several options worth noting. In the New Java Class wizard, **Enclosing Type**, used for nested classes, specifies the class that will contain the class you are creating. As you type the **Name** of the class, watch the message at the top of the wizard page—it will flag invalid class names. If you elect to not generate stubs for superclass constructors or inherited methods, you can later generate the code from the editor by selecting **Source** from the context menu or by using code assist and a code template. If you do not select to have **Inherited abstract methods** generated when you created the class, you will see errors in the Problems view indicating the methods you still need to create. Once the class is created, you can generate stubs for these by selecting the class in one of the Java views and then selecting **Source > Override/Implement Methods...** from the context menu.

Importing Java Elements

In the Resource Management section in Chapter 2, Getting Started with Eclipse, we discussed importing resources with the Import wizard. Recall that you invoke the Import wizard by selecting **Import...** from the Navigator or Package Explorer context menu or by selecting **File > Import...**. With Java projects and Java elements, it is generally easier to deal with projects than individual Java elements because Java projects contain build path and builder information. Note that a Java project import creates a pointer to the project in its original location; if you delete the project from your workspace and say yes to the delete from file system prompt, you have deleted the project for good. Some import wizards copy the target into the workspace; the Java project Import wizard does not.

Local History for Java Elements

The Resource Management section in Chapter 2 also described how you could compare and replace or merge a file with another edition of it from local history, compare two files, compare two projects, and recover files deleted from a project. In addition to this, with the JDT you can compare and replace individual elements, including methods and fields. You can select an element in one of the Java views and then select **Compare With**, **Replace With**, or **Restore from Local History...** from the context menu. Figure 3.24 shows comparing a single method, `generatePrimeNumbers`, with a previous edition.

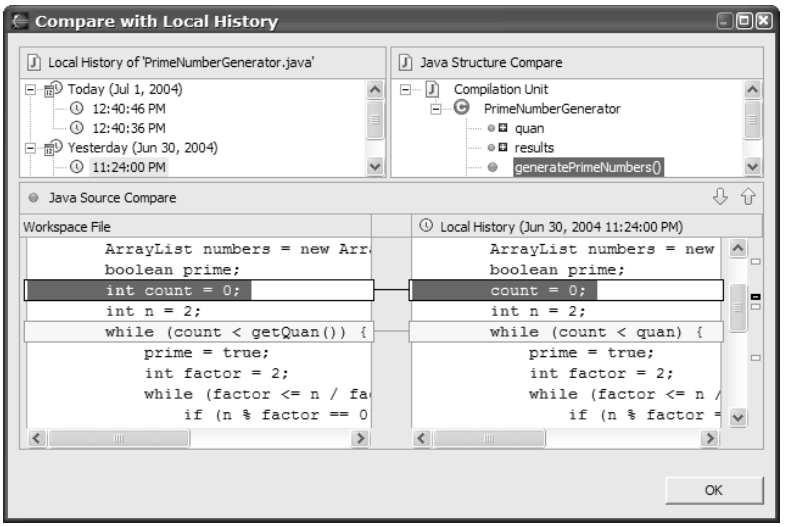

Figure 3.24 Comparing a Java Element from Local History

When you compare Java elements, you can select an element in the **Java Structure Compare** pane to see only its differences. You can restore a deleted method or field by selecting a type in one of the JDT views and then selecting **Restore from Local History…** from the context menu.

Tuning the Performance of the JDT

In the Customizing Eclipse section in Chapter 2, Getting Started with Eclipse, we discussed a number of factors that affect the performance of Eclipse, including startup time and memory usage. These factors include the number of installed tools; how many views, editors, and projects are open; and the number of resources they contained. In addition to these, several other factors can affect the performance of the JDT. These relate to the amount of real-time processing JDT does to analyze and build code you're editing. If the performance of JDT degrades, there are several things you can do.

- Reduce the amount of Java code in your workspace. Do this by partitioning code into multiple workspaces and/or closing projects you're not currently working on. You can also move code to JAR files and define these as user libraries.
- Under **Workbench** preferences, deselect **Build automatically**. If you do this, you will need to perform builds manually by selecting **Project > Build Project** or **Build All** (**Ctrl+B**) from the menu.
- If you are defining your own builder(s), consider defining a working set of relevant resources that will cause the builder to run. This is on the **Build Options** page of the builder's properties.
- Under **Java > Editor** preferences, on the **Code Assist** page, deselect **Enable auto activation**. On the **Mark Occurrences** page, deselect **Mark occurrences in file**.
- Turn off quick assist and code folding. Under your **Java > Editor** preferences are, respectively, **Light bulb for quick assists** on the **Appearance** page and **Enable folding when opening a new editor** on the **Folding** page.
- Turn off quick diff. This is your **Workbench > Editors > Quick Diff** preference.
- On the **Typing** page of your **Java > Editors** preferences, disable **Analyze annotations while typing**. This means, for example, that you will not get real-time error cluing and annotations in the editor. You'll need to save the file you're editing to have the annotations updated, including any errors flagged.

More on the JDT Views and Preferences

We'll end this chapter with a little more detail on some of the views and perspectives that comprise the JDT and some cool stuff that may not be immediately obvious.

Filtering View Contents

The JDT views allow you to filter the elements that are displayed. You can select to hide static members, fields, nonpublic members, and local types. These filtering criteria are useful if your workspace contains a significant amount of code, and navigating (especially in the Package Explorer view) becomes cumbersome. From the pull-down menu on the title bar, several of the views allow you to specify filters and/or working sets.

Package Explorer View

If the package names in the Package Explorer view are long and/or you have sized the width of the view small enough that you often find yourself scrolling the view horizontally to find the right package, you can abbreviate the package names in the view. In your **Java > Appearance** preferences, select **Compress all package name segments, except the final segment**, and enter a pattern to represent the compression scheme. The pattern is a number followed by one or more optional characters specifying how each segment of the package name is to be compressed. The pattern number specifies how many characters of the name segment to include, and the pattern characters become the segment separators. The compression is applied to all package name segments, except the last one. For example, for the package name `org.eclipse.jface`:

"`1.`" causes it to be compressed to `o.e.jface`,
"`0.`" results in `jface`, and
"`2.`" results in `or.ec.jface`.

If you have compressed names, you can quickly determine the full name of a resource by selecting it in the Package Explorer view. The full name is displayed in the message area.

In addition to filters, you can use **Go Into** from the context menu to focus the contents on the selected element, thereby reducing the content in the view. If you do this often, select **Go into the selected element** in your **Java** preferences to make this the double-click default, instead of expanding the element in the view.

Finally, if you work with JAR files in your **Workbench > File Associations** preferences, add a file compression program that understands `*.jar` files to make it easy to view their contents.

Hierarchy View

The Hierarchy view is an exceptionally useful one, especially for exploring Java code (see Figure 3.25). There are three presentations available from the toolbar: the **type hierarchy**, the **supertype hierarchy**, and the **subtype hierarchy**. These three presentations revolve around a focal point type in the view, indicated by the label decoration. This is the type on which you opened the view. Change this with **Focus On...** from the context menu. For classes, the type presentation shows a single slice of the class hierarchy from **Object** through to all of the subclasses in a class. For interfaces, it shows all classes that implement the interface and their subtypes. The supertype presentation shows classes that are extended and interfaces that are implemented. The subtype presentation shows a class and its subclasses or an interface and the classes that implement it.

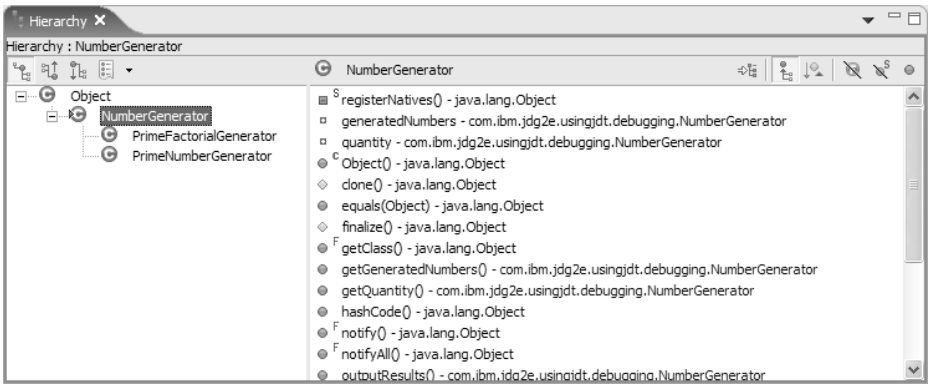

Figure 3.25 Hierarchy View

Lock View and Show Members in Hierarchy locks the contents of the bottom pane and prevents it from changing as you select elements in the top pane. With this and **Show All Inherited Members**, it's easy to get a handle on where fields are defined and what classes and interfaces implement which methods.

You can drag any Java element from one of the Java views and drop it on the empty area of the Hierarchy view to open it there. Do not drop an element

on an existing element in the view; that may be interpreted as a refactoring move request.

The Hierarchy view also maintains a list of the elements you have opened. Rather than opening an element again, you can select **Previous Type Hierarchies** to see a list of those available. Change the orientation of the view from vertical to horizontal by selecting **Layout > Horizontal View Orientation** from the pull-down menu on the Hierarchy view toolbar.

One last Hierarchy view tidbit: You can use a package, a source folder, or even a project as input to the Hierarchy view by dragging it to the view or using the pop-up menu option. The end result is an overview of all the types contained in or implemented by the input, which can sometimes be useful when naming patterns alone are not sufficient to organize your code.

Call Hierarchy View

The Call Hierarchy view is a useful one that shows all invocations of a given method or all methods invoked by a given method (see Figure 3.26). This view does not appear by default on the **Window > Show View** list. You need to select **Window > Show View > Other...** and then select it from the **Java** category. The Call Hierarchy view behaves a lot like the Hierarchy view. One of the methods listed in the view has the focus. This is the method for which callers or callees are shown. Change this method by selecting another method and then **Focus On** from the context menu. The pull-down menu allows you to switch between caller and callee views, to orient vertically or horizontally, and to set the scope of the search for callers and callees.

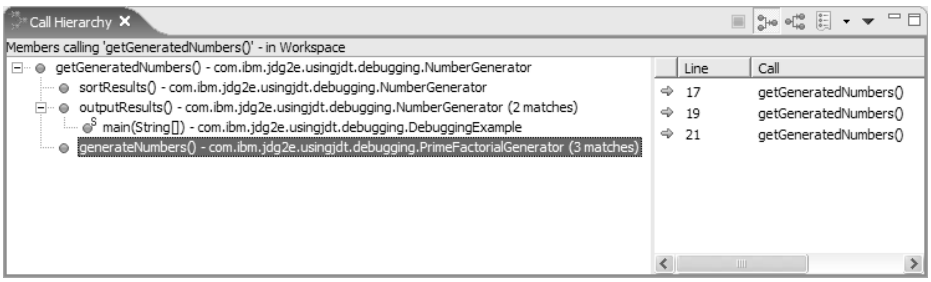

Figure 3.26 Call Hierarchy View

Outline View

The Outline view provides a filtering mechanism you might use to reduce unnecessary content in the view. Select **Filters...** from the view's drop-

down menu. You can define name pattern filters or select from predefined filters that can suppress import and package declarations.

Problems View

If you end up with a lot of errors and warnings, say, after you've imported a chunk of code, the errors can be easier to digest if you filter them. Select **Filter...** on the title bar and then select **On selected resource only**, **On selected resource and its children**, or even **On any resource in same project**.

Search View

In addition to displaying the results of a search, you can use the Search view to quickly navigate through Java references and declarations. Once you have a set of search results, select a declaration, reference, and implementer, then select **References** or **Declarations** from the context menu to execute another search. The view also maintains a history of your search results. Rather than executing a search again, select **Show Previous Searches** and then select from the list.

Java Type Hierarchy Perspective

The Java Type Hierarchy perspective is a perspective optimized for use with the Hierarchy view (see Figure 3.27). When you select **Open Type Hierarchy**, the JDT replaces the contents of the Hierarchy view if it is open or opens a new one in the current perspective. Preferences provide a useful alternative to this default behavior. In the **Java** preferences, select **Open a New Type Hierarchy Perspective**. This causes the Java Type Hierarchy perspective to open in a new window.

Java Browsing Perspective

The Java Browsing perspective, shown in Figure 3.28, provides a slightly more structured approach to navigating through Java resources than the Java perspective. In addition to an editor pane, it contains Projects, Packages, Types, and Members views. The organization of this view will be familiar to users of VisualAge for Java. In fact, it's intended to mimic the VisualAge for Java user interface.

The navigation model is to select an element (by single-clicking it) in a pane to see its contents in the next pane. **Open** and **Open Type Hierarchy**

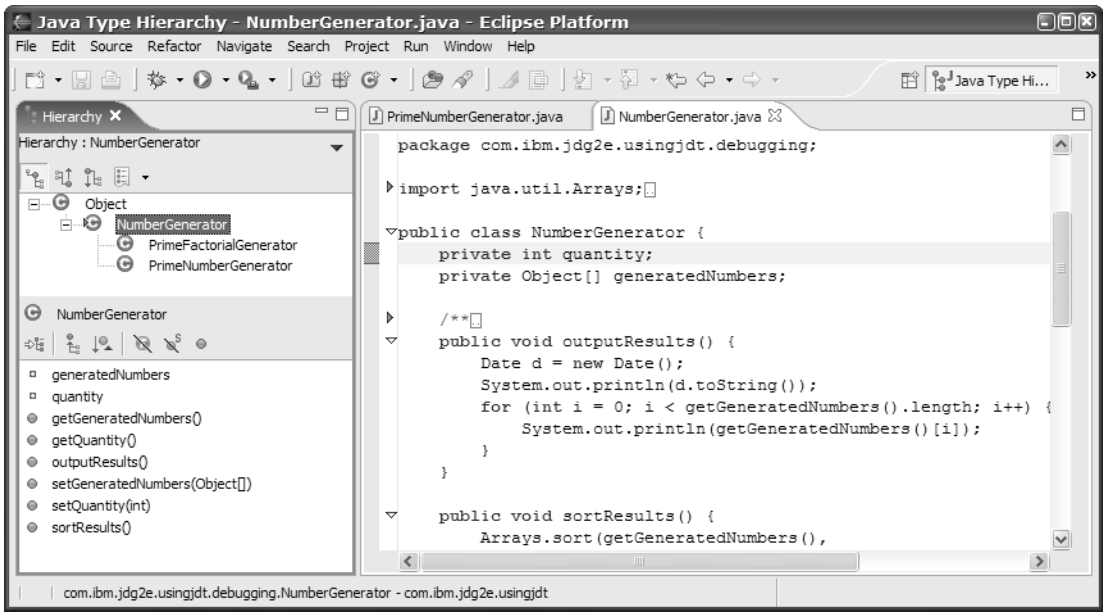

Figure 3.27 Java Type Hierarchy Perspective

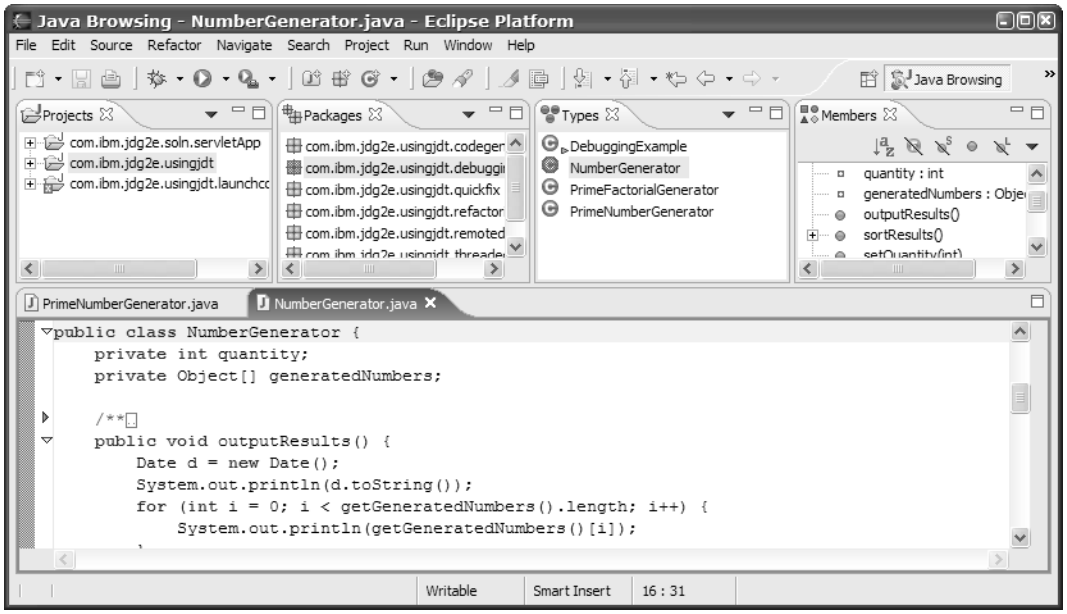

Figure 3.28 Java Browsing Perspective

actions are available from all the views and the editor. You can specify different filters for the contents of each of the views. The default orientation of the Java Browsing perspective is shown in Figure 3.28. You can also orient the views vertically on the left with the editor on the right by changing your **Java > Appearance** preferences to **Stack views vertically in the Java Browsing perspective**.

Exercise Summary

Exercise 2, Using the Java Development Tools (found in Part VI), accompanies this chapter. The exercise is broken down into a number of parts, each demonstrating different concepts and building on the previous part.

1. Hello World

   You'll create the ubiquitous "Hello World" program, and you will see how to run it, including how to use a scrapbook page.

2. Quick Fix

   You'll use the JDT's quick fix capabilities to fix a series of errors in a Java program.

3. Code Generation

   You'll see how to significantly improve your productivity by generating Java code to complete expressions, including the use of code templates.

4. Refactoring

   In this part, you'll really start to see the power of the JDT when using refactoring to clean up, reorganize, and extend a program.

In addition, Exercise 4, Developing a Simple Web Commerce Application with Eclipse, provides a more robust programming example, developing, deploying, and debugging a servlet application in Apache's Tomcat. This exercise brings together a number of individual concepts presented in Part I.

Chapter Summary

This chapter provided a comprehensive overview of using the JDT to explore, write, and run Java. We described how to create different kinds of Java elements, how to navigate and search them with the different views, and how to write Java code using code assist, code generation, and refactoring. We covered how different kinds of errors are marked and how to fix errors with quick fix. We also discussed how to set Javadoc locations and

view Javadoc. Next we looked in some detail at Java projects and their properties, how these properties impact your projects, and how to use different JREs and reference declarations in other projects and JAR files. The chapter also described editing code templates used in code generation and externalizing strings.

Reference

Arthorne, John, and Chris Laffra. 2004. *Official Eclipse 3.0 FAQs*. Boston, MA, Addison-Wesley. http://eclipsefaq.org. (See Chapter 3 and FAQs 313, 315–318.)