



CHAPTER 3

Using Java Development Tools

Eclipse provides a first-class set of Java Development Tools (JDT) for developing Java code. These tools include a Java perspective, a Java debug perspective, a Java project definition, editors, views, wizards, refactoring tools, a Java builder (compiler), a scrapbook for evaluating Java 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 Java source code and view Javadoc. The Java editor supports syntax highlighting, content assist, error cluing, state of the art refactoring, type-ahead assistance, and code generation among a host of other capabilities. The Java compiler is an incremental compiler; it compiles only what it must based on your changes. It supports JDK 1.3 and 1.4. JDT can be configured with different JREs. You can develop code with one JRE and debug with another.

The Java editor is more than a source code editor. It builds and maintains 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 alert you to potential errors in your code before you save it. In many cases 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're going to look in more detail at the capabilities and use of the Java tools. We'll start with an overview of the JDT user interface and the fundamentals you'll need for creating and navigating Java resources. We'll then look in much more detail at JDT capabilities for coding

Java. Then we'll see how to run Java code. In the final sections of this chapter we'll get into more detail on working with Java elements, tuning the performance of JDT, and specific functions in the JDT views and perspectives.

Getting Started

Let's go. We'll start with a quick overview of the JDT user interface. We'll then cover the fundamentals, like opening a class, navigating to a method or field, and running a Java program. We'll 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 that comes up when you create a new Java project (see Figure 3.1).

The left pane contains the Package Explorer view and Hierarchy view. 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, and open files for editing. The middle pane contains open editors, Java and otherwise. The active editor is the one on top. In Figure 3.1, an editor has `PrimeNumberGenerator.java` open in it. The right pane is the Outline view, which presents a structured, hierarchical view of the contents of the active editor. On the bottom right is the Tasks view, the same one we saw in the section “Working with Tasks” in Chapter 2. As you navigate in the user interface, selecting Java files for editing and selecting Java elements, all the views and the editor stay synchronized with your actions.

- The Java views display Java elements with icons for a package and for public methods. Some of these icons are decorated to provide further information 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” section 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 Source of 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 only show the source for the selected class, method, field, or import statement. This is mostly a matter of personal preference.

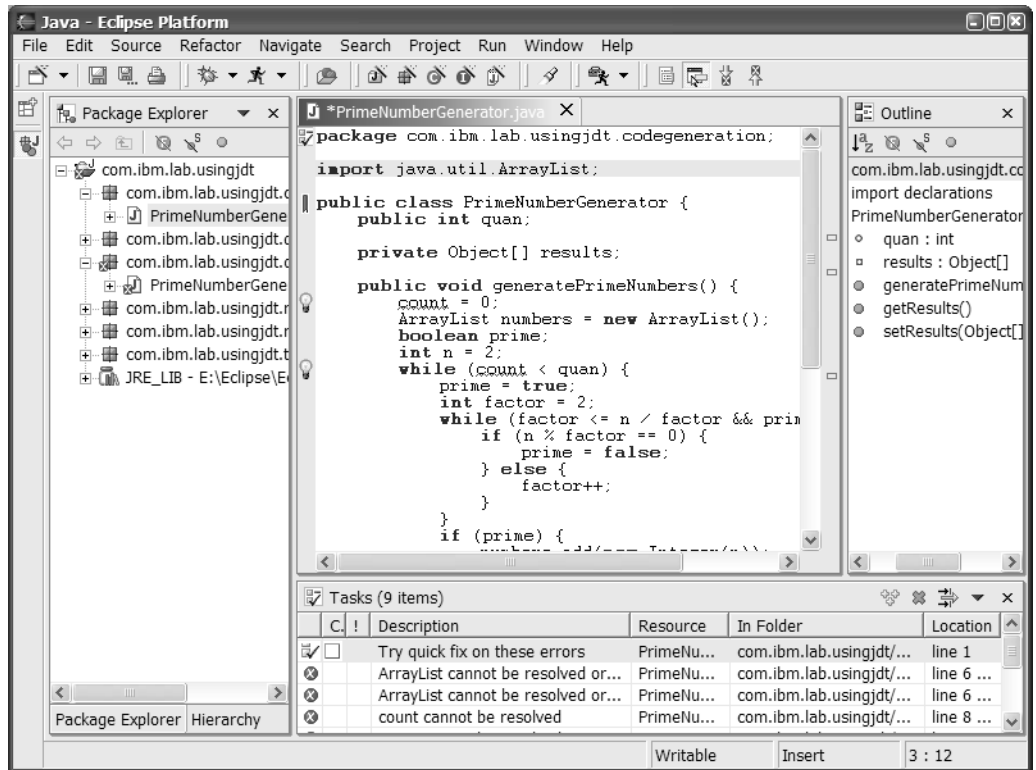


Figure 3.1 Java Perspective

The Fundamentals

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

Creating a Java Project

All Java elements must exist in a Java project. To create a Java project, select **File > New > Project... > Java Project** from the menu or select **Create a 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 will be 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 will be created for you. To create a package, select the containing project and select **File > New > Package** from the menu or select **Create a Java Package** from the toolbar.

Creating a Type

To create a type, select the containing project or package and select **File > New > Class** or **File > New > Interface** from the menu or select **Create a Java Class** or **Create a Java Interface** from the toolbar.

Opening a Type

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

- Double-click on a Java source file, class, or interface, or select one of these in a view and press **F3** or **Enter**. You can do this from any Java view.
- 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.

Opening a Method or Field

To open a method or field definition, 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.

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, and then select **Open Type Hierarchy** from the context menu or press **F4**. 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 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.

Navigating to a Type, Method, or Field

You can navigate to class, interface, method, and field definitions in your Java code simply by selecting one of these elements in a view, such as the Outline or Type Hierarchy view. The editor and open views scroll to your selection.

NOTE If you select an element and you do not see it in an editor, it means the file containing the definition is not open. You must then first open the file using one of the methods described above.

Locating Elements in Projects

As you use the views and editor to navigate in your code, you can locate element definitions in projects by selecting a Java element in one of the views or the editor (or positioning the insertion cursor in the element name) and then selecting **Show in Package Explorer** from the context menu. The Package Explorer scrolls to the selected element. The element must be defined in an open project in your workspace.

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.

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 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 the 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**. Eclipse also has an “incremental find” feature that provides a keystroke-efficient way to do this. Select **Edit > Incremental Find** from the menu or press **Ctrl+J** and note the prompt in the message area to the left on the bottom margin (see Figure 3.2). Start typing the text you’re searching for. The message area displays the text you’re searching for and the first match is selected in the editor. Press **Ctrl+K** to find the next occurrence of the text.

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. 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 3.3. To restrict a search, first select one or more Java elements in the Package Explorer, Outline, or Type Hierarchy view and then select **Search**, or define a Working Set and then specify that working set under **Scope**. (For

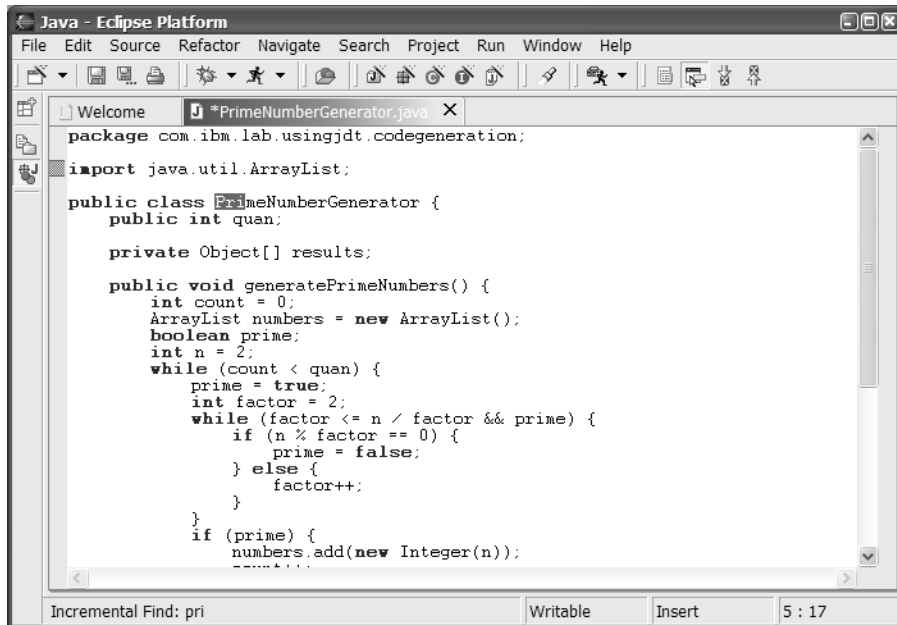


Figure 3.2 Incremental Find

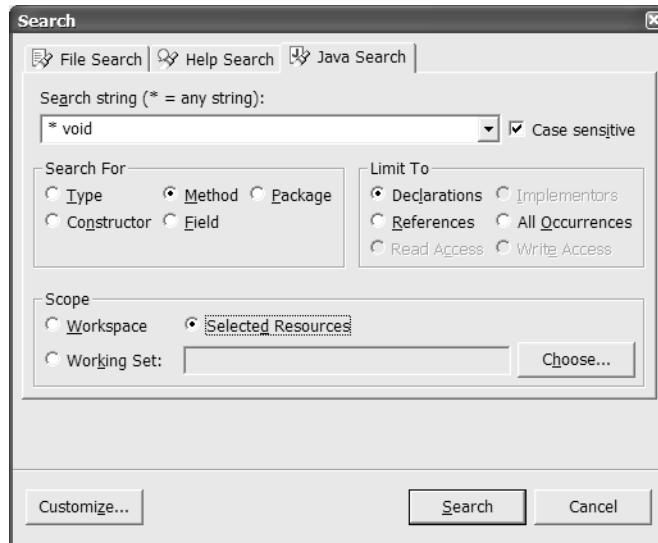


Figure 3.3 Searching for Java Elements

more information on Working Sets, refer to the section “Working Sets” in Chapter 2.) For example, if you want to search for methods returning `void` in your projects, select the projects in the Package Explorer view, select **Search**, specify the **Search string** “* void”, select **Search For Method** and **Limit To Declarations**, select **Selected Resources**, and then select the **Search** button (or press **Enter**).

Java search results are shown in the Search view (see Figure 3.4). Matches are indicated in the editor with entries on the marker bar. Navigate to matches from the Search view by double-clicking an entry or by selecting **Show Next**

↓ ↑ **Match** and **Show Previous Match**.

In all of the JDT views, including the Search view and the editor, you can select an entry or Java element and then search on that element from the context menu by selecting **References >** or **Declarations >**. 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 the real cool stuff, writing Java. The Java editor provides a wealth of functions to help you write Java code more efficiently with greater productivity, and, quite

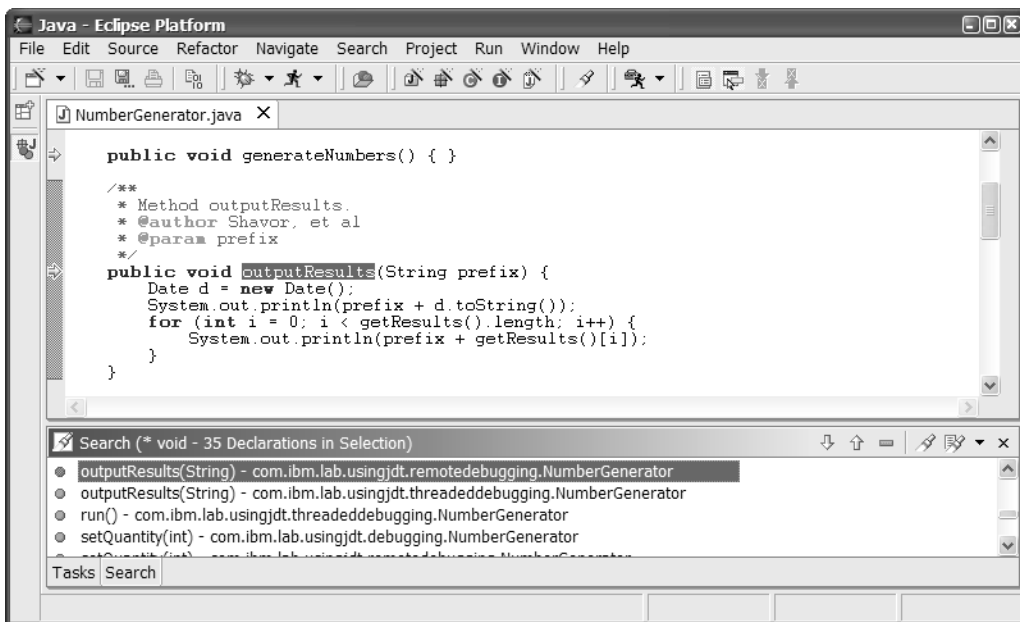


Figure 3.4 Search View

frankly, while having more fun. Among the capabilities provided by the Java editor are content assist for Java expression completion, code generation, real time-error detection, quick-fix remedies for coding errors, the ability to work with different JREs, and Javadoc generation.

The organization of the Java editor pane is straightforward and similar to other editors (see Figure 3.5). On the left border is the marker bar, which shows tasks, bookmarks, compiler errors, and quick fixes. The right margin is the overview ruler, which shows error indicators to help you quickly scroll to errors that are currently not visible. On the left of the bottom margin is the message area; on the right are three fields. From left to right these fields indicate if 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 editor is, as you would expect, completely integrated. This means Java views, such as Type Hierarchy and Outline, update in realtime as you type in the editor. It allows a great deal of customization. The behavior of the editor is specified in your **Java** preferences under the **Editor**, **Code Generation**, **Code Formatter**, and **Refactoring** pages.

The editor provides unlimited Undo/Redo through **Edit > Undo** and **Edit > Redo**, or **Ctrl+Z** and **Ctrl+Y**. View the other keyboard shortcuts by selecting

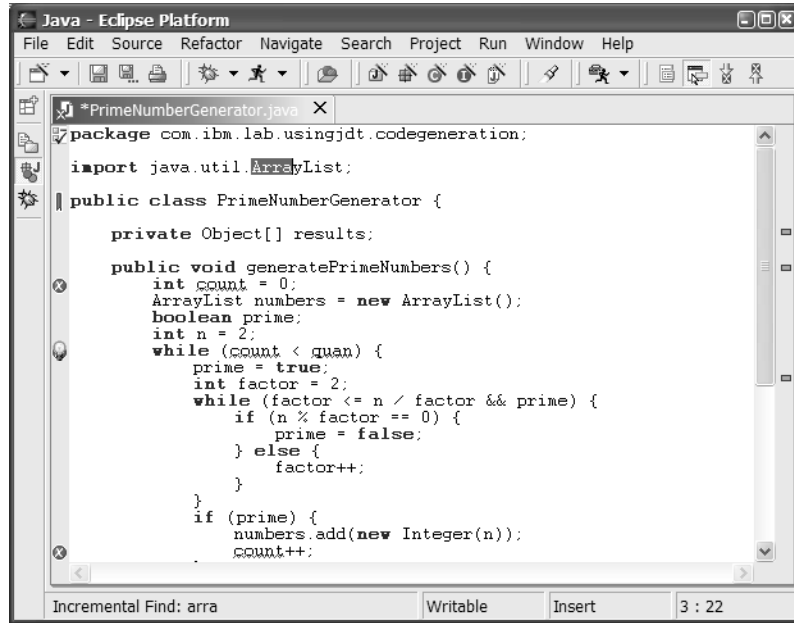


Figure 3.5 Java Editor

Window > Keyboard Shortcuts. Format your code by selecting **Source > Format**, or by pressing **Ctrl+Shift+F**. You can double-click on the editor title bar to maximize it.

Position your cursor in the parameters for a method invocation and then press **Ctrl+Shift+Space** to see parameter hints (see Figure 3.6). If there are multiple parameters, the editor will highlight the type of the parameter you are editing.

Many of the code generation and refactoring operations require a valid Java expression to be selected. To do so, select **Edit > Expand Selection To**, with the options **Enclosing Element**, **Next Element**, **Previous Element**, and **Restore Last Selection**. To select an expression using keyboard shortcuts, press **Alt+Shift** and then an arrow key (**Up**, **Right**, **Left**, and **Down** respectively).

To reference Java elements declared outside your current project, you need to set your project's build path to include their declarations. We'll see how to do this later in this chapter in the section "Creating Java Projects." You will know that you need to do this, for example, when you attempt to add an `import` statement and the type or package you want to import does not appear in the dialog.

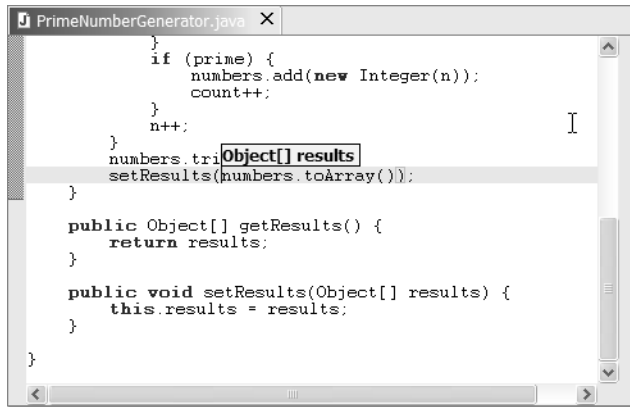


Figure 3.6 Parameter Hints

Content 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**, then continue typing `ayi` to narrow the list to that class and press **Enter**. The complete reference is inserted. Content 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, content assist will display suggestions (see Figure 3.7).

NOTE If you have trouble using content assist, be sure to check your preferences, specifically the **Java Code Generation** page and the **Java Editor Code Assist** page, to verify it isn't disabled.

The content assist pop-up lists matching templates and Java references on the left along with their respective icons (class, interface, and method); on the right is the proposed code. These suggestions come from two places: Java references based on an analysis of your code and your defined code templates. Your defined code templates are part of your **Java** preferences under **Templates**. We'll see how to edit these later in this chapter in the section

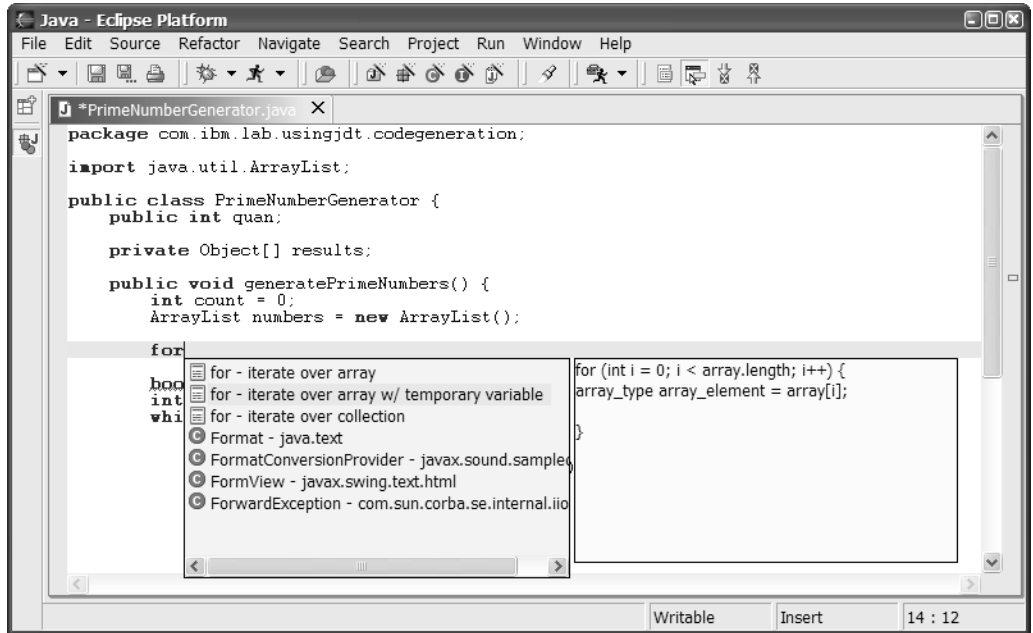


Figure 3.7 Content Assist Prompt

- “Using Code Templates.” Code templates match what you are typing by template name. Pressing **Enter** will insert 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 content 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 content assist prompt will disappear. This is useful when you have auto content assist enabled, since you can choose to ignore prompts that come up by simply continuing to type what you had been typing.

You can use content assist in a variety of situations. Figure 3.7 shows generation of a stub for a for loop from a template. You can also generate public method templates, while loops, and try/catch expressions. Take a look at the templates in your **Java** preferences to see what’s available. In addition, content assist works for the following.

- **Overriding Inherited Methods.** Position the insertion cursor within a class definition, but outside any method, and activate content assist. You’ll see a list of methods this class can override or must define based on an interface implementation (see Figure 3.8).

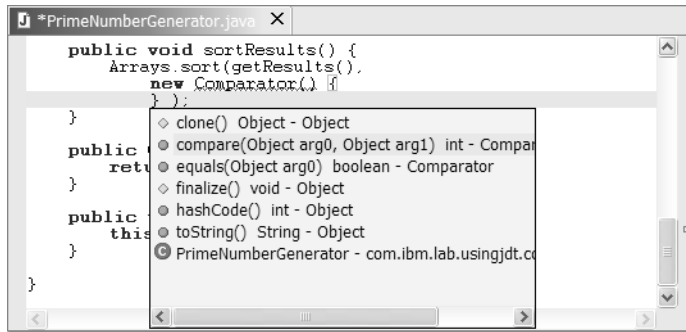


Figure 3.8 Content Assist Override Method

- **Import Statements.** Type in a Java reference that is unresolved and then activate content assist. As shown in Figure 3.9, you'll see a list of applicable types. Select one and it completes the reference and includes the required import statement.

You can also type `import` and begin typing the name of a package or type, and then activate content 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 content assist. It provides a list of possible references, including variables within the scope of the expression (see Figure 3.10).
- **Anonymous Inner Classes.** Activate content assist within the body of an anonymous inner class and it will present a list of methods to define (see Figure 3.11).

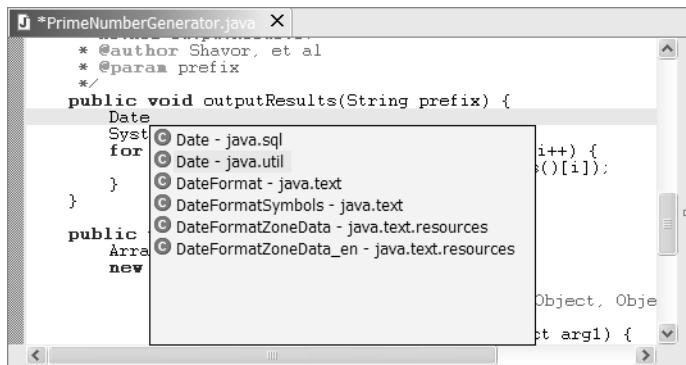


Figure 3.9 Content Assist Import Statement

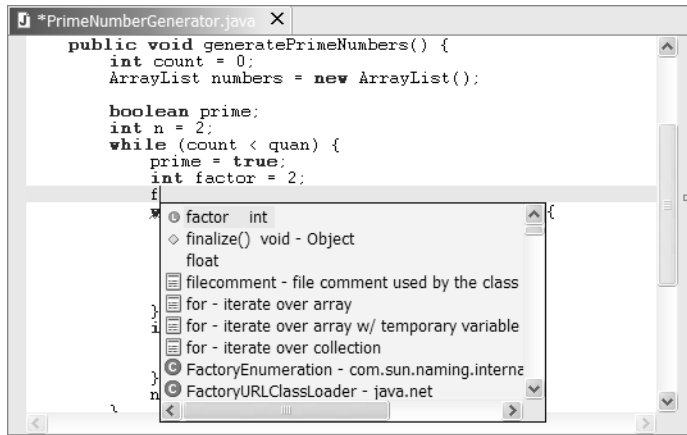


Figure 3.10 Content Assist Variable Reference

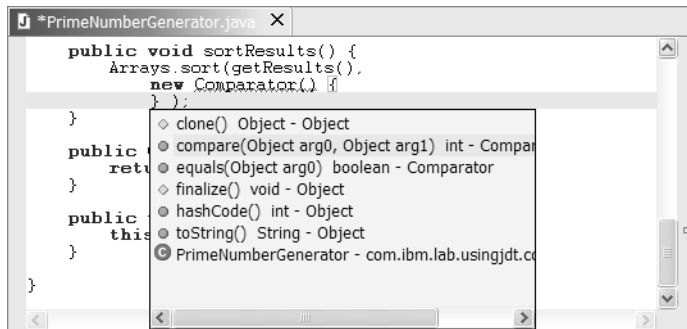


Figure 3.11 Content Assist Anonymous Inner Class

- **Parameter Hints.** Content assist provides another way to see information about parameter types for a method invocation, in addition to **Ctrl+Shift+Space**. Position the insertion cursor in the parameters of a method invocation and activate content assist. It displays the parameter types. As you enter parameters or move your cursor over them, it highlights the type of the parameter (see Figure 3.12).
- **Javadoc.** Activate content assist in Javadoc. You can add Javadoc HTML tags and keywords. What's slick about this is that content assist can make smart suggestions based on an understanding of your code. For example, typing @param and then activating content assist will display a list of method argument names. Typing @exception and

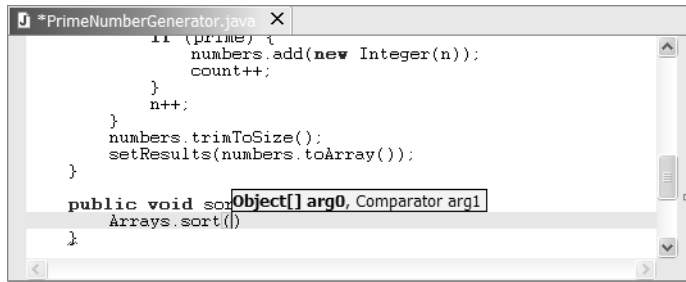


Figure 3.12 Content Assist Parameter Hints

then activating content assist will display a list of exceptions a method throws.

Code Generation

In addition to content assist, JDT provides other code generation capabilities. These are options available under the **Source** menu item or from the editor's context menu.

- **Code Comments.** You can quickly add and remove comments in a Java expression with **Source > Comment** (or **Ctrl+/**) and **Source > Uncomment** (**Ctrl+)**. Adding comments generates line comments, which are prefaced by two backslashes (`//`) on the lines with the selected expression.
- **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.** You can create a stub for an existing method by dragging it from one class to another. You can also select **Source > Override 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 Getter and Setter...** Figure 3.13 shows the Generate Getter and Setter dialog.

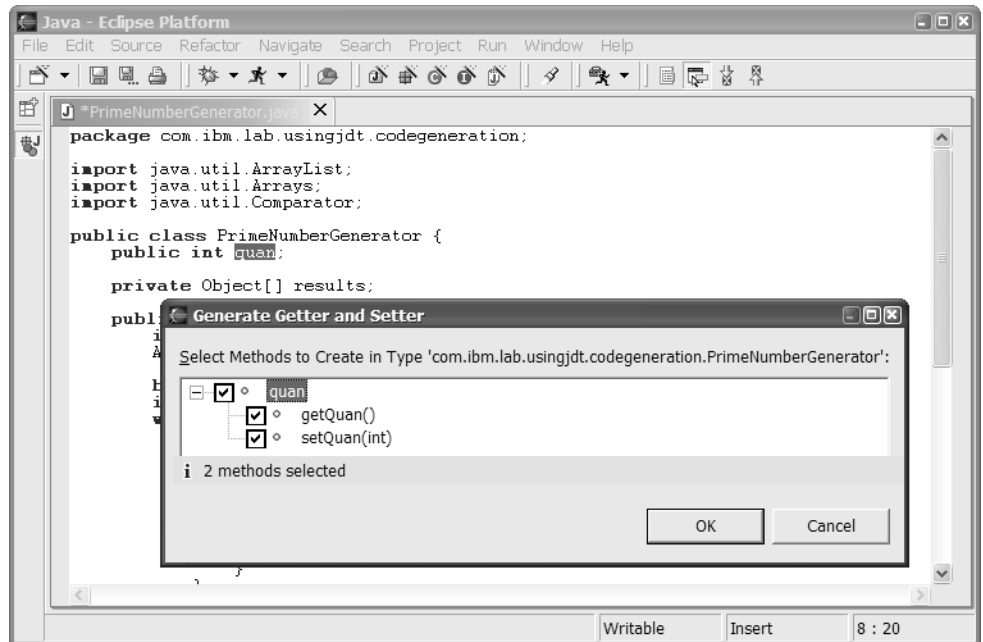


Figure 3.13 Generating Getter and Setter Methods

- **try/catch statements.** If you select an expression and then **Source > Surround with try/catch**, 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. Customize these by modifying the `typecomment` or `filecomment` template. For more information on this, refer to the section “Using Code Templates” later in this chapter.
- **Superclass constructor.** 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**.

Navigating Java Errors and Warnings

- JDT displays two types of errors in your Java code. The first are errors and warnings resulting from compiling your code. These are shown as entries in the Tasks view, as markers on the marker bar, and as label decorations in the views (refer back to Figure 3.5). The second are errors detected as you edit Java code, before it's compiled. These are shown as error clues (red underlining) in the source and on the overview ruler on the right border (small red rectangles). A quick fix icon is displayed for errors for which JDT can suggest a solution. We'll get to quick fix in the section "Fixing Java Errors with Quick Fix" later in this chapter.

Hovering over an error indicator in the marker bar or over text underlined in red displays the error message. Position the text cursor in red underlined text and press **F2** to see the error message. Click on an error indicator in the overview ruler to cause the code in error to be selected. Clicking on the error indicator again scrolls the editor to the error and selects the text. You can also select **Go to Next Problem (Ctrl+.)** or **Go to Previous Problem (Ctrl+,)** to scroll through and select compiler errors. When you do so, the error message is displayed in the message area.

In the Tasks view, double-click on an error to go to that line in the file with the error. If the file is not open, it is opened in an editor. If the error message is truncated in the Tasks view because of the width of the column, hover over it to display the full message or select the task to see the full text in the message area.

Fixing Java Errors with Quick Fix

- For errors having 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. Click on the icon to see a list of possible solutions (see Figure 3.14). You can also position the cursor within the red underlined text and then press **Ctrl+I** to display the suggestions. Select an item in the pop-up to see the proposed fix or a description. Press **Enter** make the correction.

If the file you are editing is marked as read-only, quick fix will not work correctly. You'll see the quick fix icon on the marker bar, but clicking on it will not result in any suggested fixes.

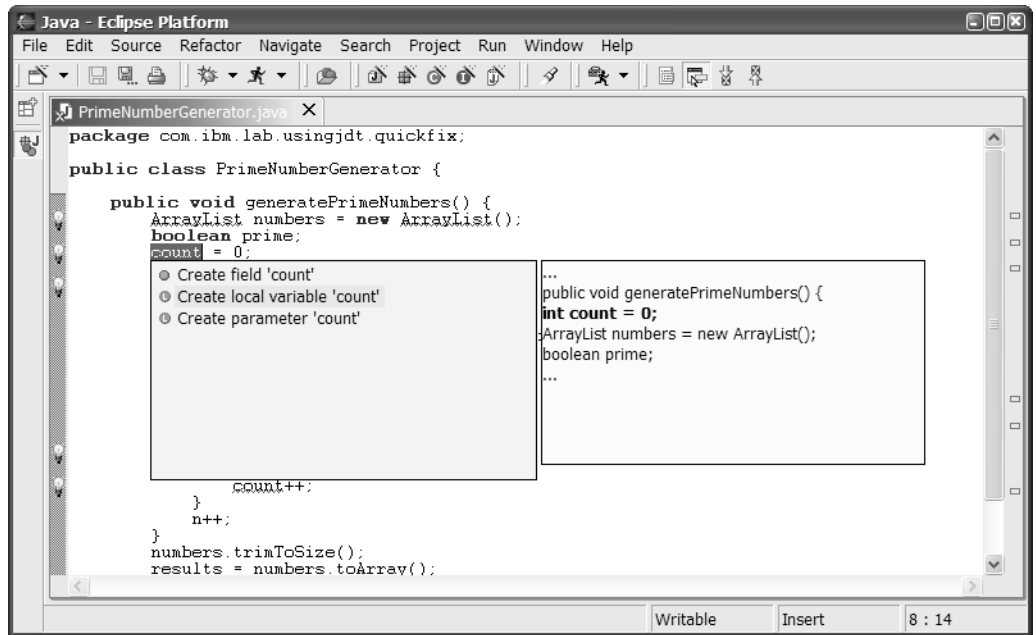


Figure 3.14 Quick Fix Suggestions

Refactoring

Refactoring refers to a class of operations you use to reorganize and make other global changes to your code. You may need to do this in response to API changes, to improve maintainability, or to make naming more consistent. JDT provides state-of-the-art Java refactoring capability that allows you to make changes and update references, including string literals and Javadoc comments. This includes the ability to do 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 to a superclass.
- **Extract Method**
This creates a new method from the selected expressions.
- **Modify Parameters**
This lets you change parameter names and order and update all references to the method being modified.
- **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.
- **Inline Local Variable**
This does the opposite of **Extract Local Variable**. You can use a variable's initialization expression to replace references to it.
- **Self Encapsulate**
You can replace references to a field with calls to its getter and setter methods. This refactoring operation assumes the getter and setter methods do not yet exist.

To refactor your code, select a Java element in one of the views (or the name of an element in the editor) or an expression in the editor, then select **Refactor >** from the context menu or the main menu. You can also 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. 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 at the beginning of the “Writing Java Code” section make this easy.

When you request a refactoring operation, the Refactoring wizard 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 **Finish** to perform the refactoring, or select **Next >** for more options.

Selecting **Next** displays a list of warnings and potential errors that will occur if you apply the proposed changes to your code. If you select **Finish**, this list of warnings and potential errors will only be displayed if your code contains an error or warning of greater severity than you've specified in your **Java Refactoring** preferences. The page after that, shown in Figure 3.16,

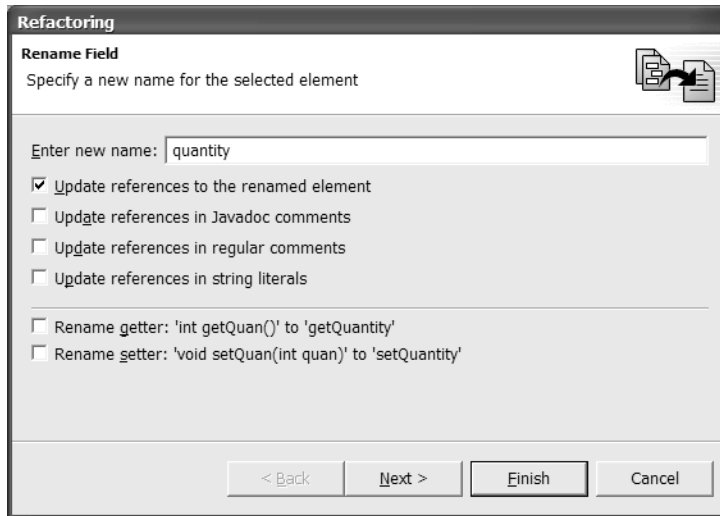


Figure 3.15 Refactoring Wizard

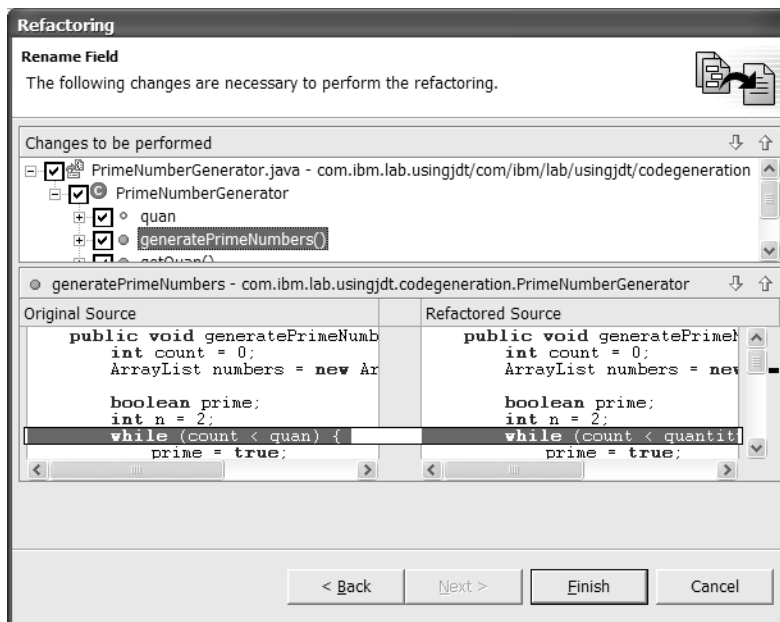


Figure 3.16 Previewing Refactoring Changes

shows each of the changes and a before-and-after view of the code. You can select to apply none, some, or all of the changes. Expand the elements in the top pane and select an element or subelement to see only the source code for that element.

To undo or redo a refactoring operation, use **Refactor > Undo** and **Refactor > Redo**. 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 only of changes made as a result of the refactoring operation in the active editor.

NOTE When you drag and drop elements in the views to affect refactoring, the **Refactor > Undo** and **Redo** actions are not available.

Browsing Javadoc

There are several ways to browse Javadoc. In the editor, position the cursor in a Java element and then select **Edit > Show Tooltip Description** or press **F2**. A small pop-up displays the Javadoc for the selected element, as shown in Figure 3.17. With **Show Text Hover** selected (the default), move your mouse over a Java element to see its Javadoc.

You can display the Javadoc for a Java element in a browser by selecting a Java element in the editor and then selecting **Navigate > Open External Javadoc** or by pressing **Shift+F2**.

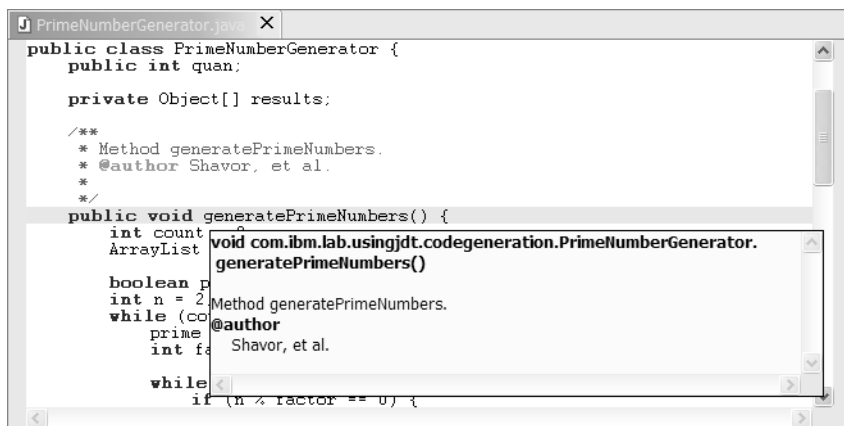


Figure 3.17 Displaying Javadoc

To browse Javadoc from the editor, you need to provide JDT with a starting point. To point to Javadoc for a JRE, use your **Java Installed JREs** preferences as shown in Figure 3.18. Edit an **Installed JRE** entry and specify the **Javadoc URL**. You can use a `file:` URL to point to Javadoc on your file system or an `http:` URL to point to Javadoc on another machine. This should point to the root folder containing the Javadoc; that is, the folder containing the `java`, `javax`, and `org` folders.

To specify the Javadoc location for code in a Java project, from the Navigator context menu, open the Properties dialog on the project. Set the **Javadoc location** property for the project. If you are referencing declarations in Java code in a JAR file, set the Javadoc location in the properties on the JAR file. This approach also works for the JRE `rt.jar` file.

Using Code Templates

Templates are code outlines 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

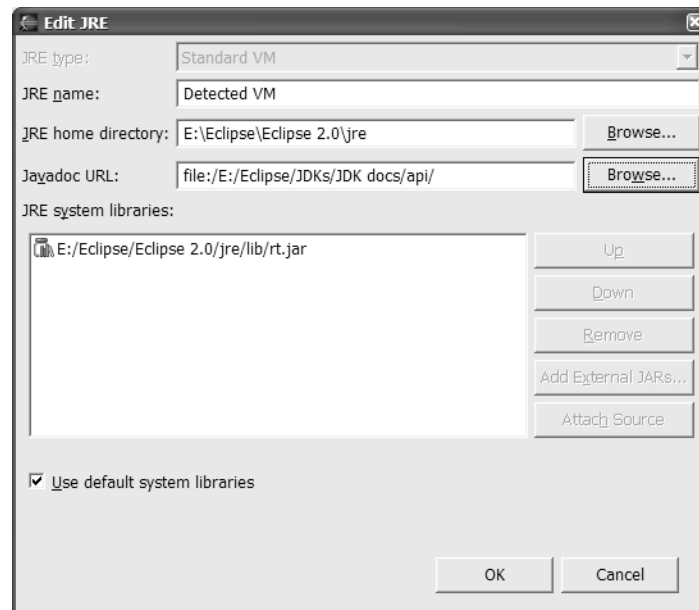


Figure 3.18 Specifying Javadoc for a JRE

contain Java or Javadoc source, variables that are substituted when the template is inserted in the Java code, and a specification of where the cursor should be positioned when editing of the template is complete. JDT suggests variable substitutions based on the template, your Java source, and where in your Java source you are inserting the template. For example, if you select to insert the template to use a `for` statement to iterate over an array, and your method has an array as a local variable or parameter, JDT will assume this local variable or parameter is the array to use and substitute its name for the template variable.

You insert a template in your Java source through content assist, that is, by pressing **Ctrl+Space**. Based on what you had just typed and your context (whether you're in a Java code or Javadoc section of the file), you will get a list of possibilities for completing what you had been typing. If what you had just typed matches the name of a template, the template appears as a choice. The **Code Assist** page of the **Java Editor** preferences lists several options for code generation such as **Insert single proposals automatically** and **Enable auto activation** (see Figure 3.19).

NOTE Code Assist is the term used in Eclipse 2.0. The Eclipse team plans to change this to **Content Assist** in 2.1, as this is the more commonly used term.

After you insert a template, if there were variables in the template, the first one is selected to allow you to modify the variable. Use the **Tab** key to navigate to the next variable in the template. After you navigate through the variables, another **Tab** will place the cursor in what the template designer believed to be the next logical place where you'd want the cursor in order to continue coding. For example, in the `for` template, the cursor will be placed in the body of the `for` statement.

It's easy to create your own templates or to modify those provided. For instance, you might want to change the **filecomment** or **typacomment** template to add a copyright or legal notice or add your own template for a `try/catch` block to invoke a special error handling routine. You do this in your **Java** preferences under **Templates**.

In Figure 3.20 you can see a list of templates that come with Eclipse. Select a template to preview it in the bottom pane. Templates can be enabled and disabled. Disabling a template will cause it not to appear as a suggestion in the content assist prompt. To share templates with your team, use the

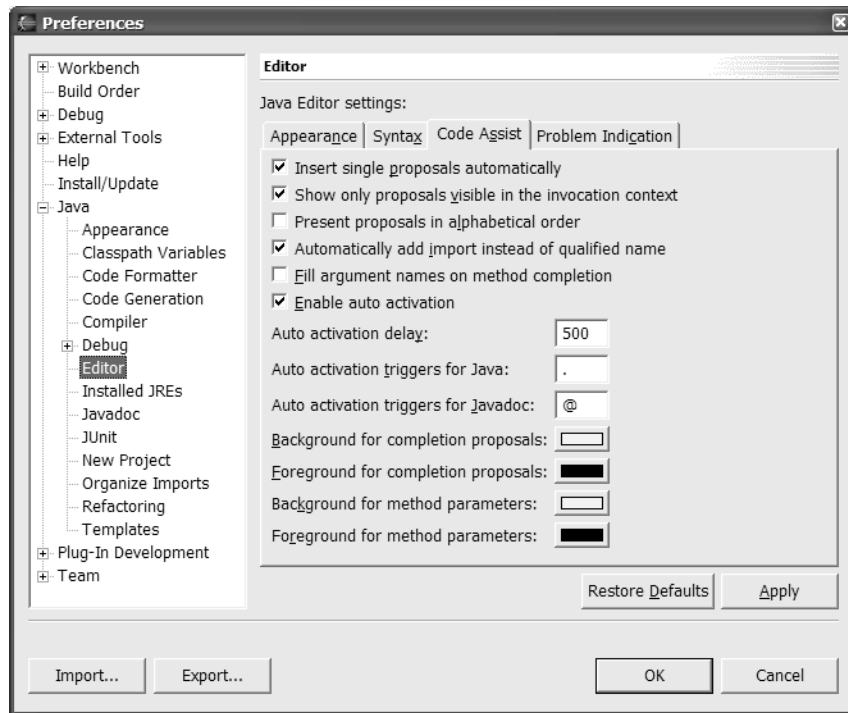


Figure 3.19 Java Editor Code Assist Preferences

Import... and **Export...** buttons on the **Java Templates** preferences page to read and write templates as *.xml files. The **Use Code Formatter** option indicates you wish to format your code after the template has been inserted according to your **Code Formatter** preferences.

You can create and edit templates in the Edit Template dialog, shown in Figure 3.21, by selecting **New...** or **Edit...** **Name** is mandatory, and **Description** is optional, though useful. Both appear in the content assist prompt. **Context** indicates whether the template is for Java or Javadoc generation and affects when a template appears in the content assist prompt. To insert a variable, use the **Insert Variable...** button on the dialog, or press **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.

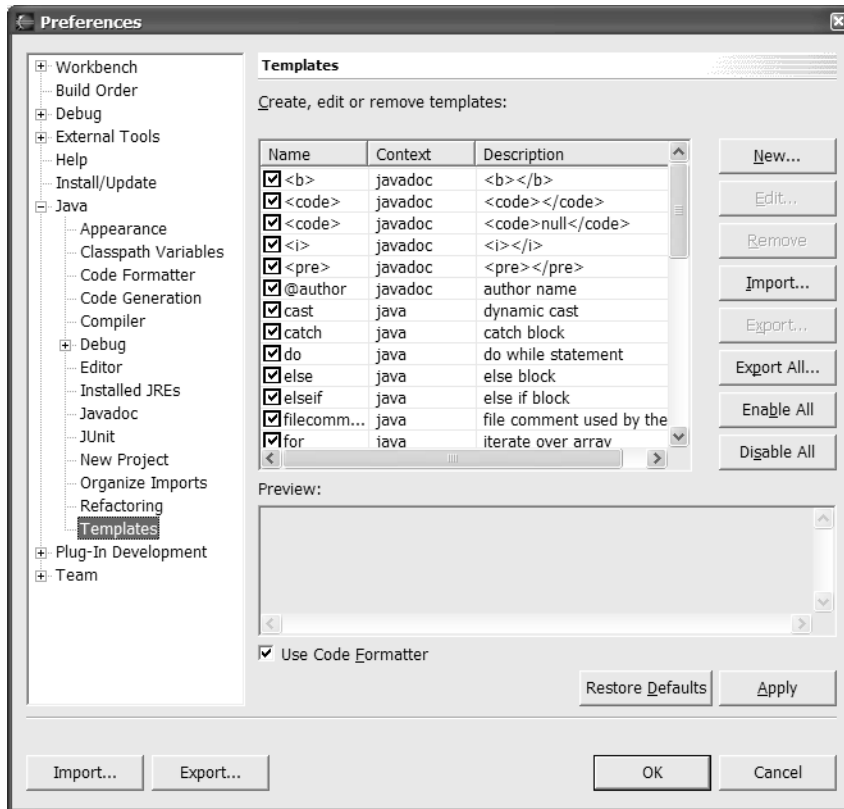


Figure 3.20 Java Templates Preferences

Externalizing Strings

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. 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 **Errors and Warnings** page of the **Java Compiler** preferences.

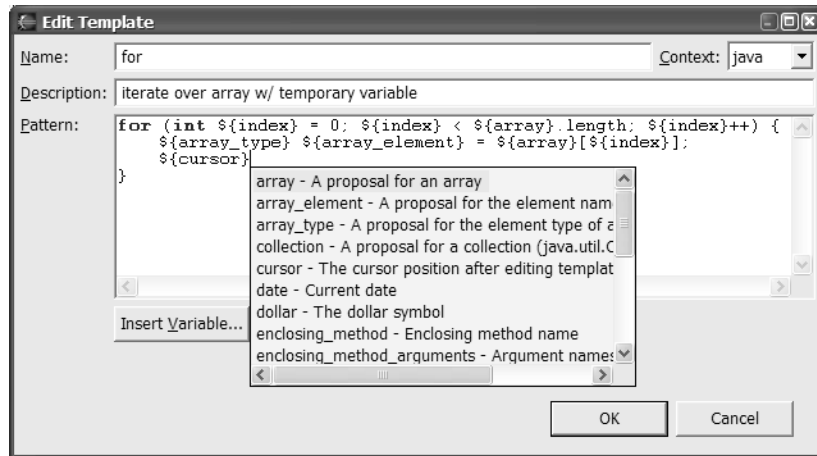


Figure 3.21 Edit Template Dialog

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 (see Figure 3.22).

Select a file and then **Externalize...** to go to the Externalize Strings wizard. Alternatively, select a *.java file and then select **Source > Externalize Strings...** from the menu to display the wizard.

On the first page of the Externalize Strings wizard, shown in Figure 3.23, you specify which strings are to be externalized (for translation) and keys for accessing the strings. Select an entry and then **Translate** to externalize the string. Select **Never Translate** if you do not want the string externalized. In

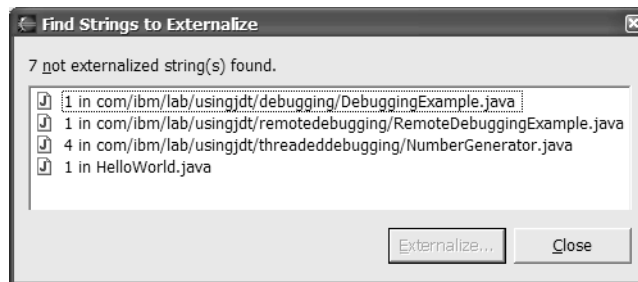


Figure 3.22 Finding Strings to Externalize

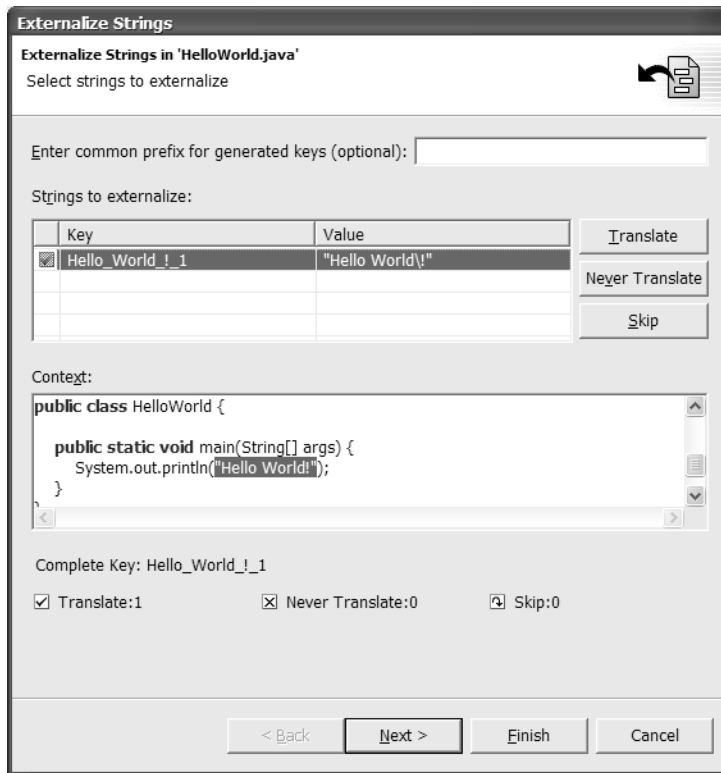


Figure 3.23 Specifying Strings to Externalize

each of these cases, JDT annotates your source code with comments indicating your choice. Selecting **Skip** goes to the next string without taking any action. If you have your preferences set to flag nonexternalized strings as errors or warnings, and you select **Translate** or **Never Translate** for the string, it will no longer be flagged as an error or warning. If you select **Skip**, 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. Even if you do not intend to translate a string, it still may be useful to externalize it if you reference the same string value in several places in your code, for example, strings that show up in your user interface. This way, you can be certain they remain the same.

On the second page of the wizard, shown in Figure 3.24, you specify the name and location of the file that contains the externalized strings and a class

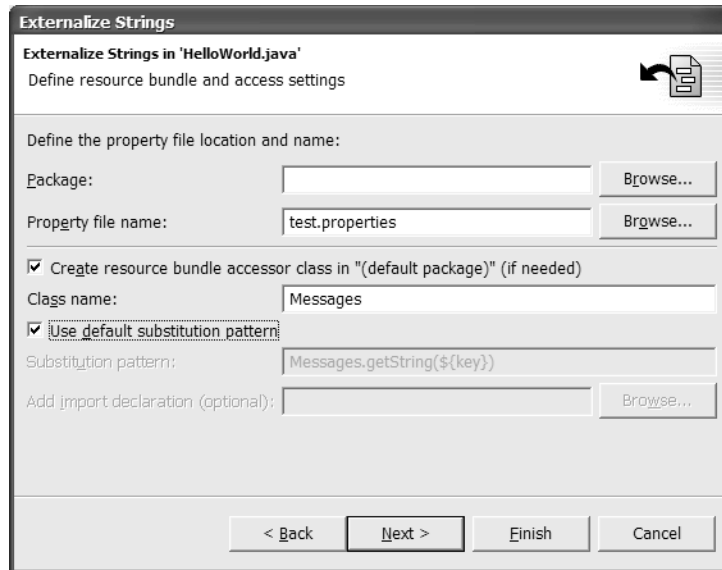


Figure 3.24 Specifying Location of Externalized Strings

to access the strings. JDT generates this class and the specified method and adds it to your project.

The final page of the wizard is a before-and-after view of your code (see Figure 3.25). In the top pane, you select to apply or not apply changes for each of the proposed modifications. Select **Finish** 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 will also remove the generated accessor class.

Generating Javadoc

You generate Javadoc by exporting it. To do so, you first need to set your **Java Javadoc** preferences to point to `javadoc.exe`. This program performs the Javadoc generation. The `javadoc.exe` program is not shipped as part of Eclipse; it comes with a JDK distribution.

Once you have set the location of `javadoc.exe`, you can export Javadoc by selecting **Export...** from the Package Explorer context menu and then

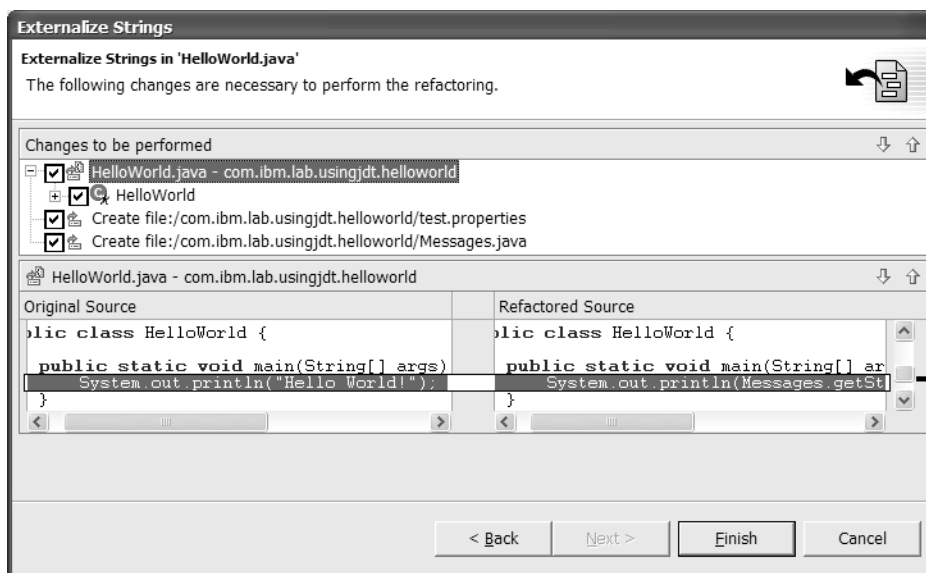


Figure 3.25 Externalize Strings Code Changes

selecting **Javadoc** as the export destination. You have a number of options for generating Javadoc for one or more of your Java projects. Figure 3.26 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 “Java Preferences” information in the “Reference” section of the *Java Development User Guide*.

When you generate the Javadoc, you will see a prompt asking if you want to update the **Javadoc Location** properties of the projects you are generating Javadoc for. You should select to do so. This will enable you to browse the generated Javadoc in the Java editor. Output from the generation shows up in the Console view. You should review this to ensure there were no errors.

Writing Java for Nondefault JREs

Eclipse uses a JRE for two purposes: to run Eclipse itself and to run your Java code. This can be the same JRE or different JREs. The specifications are

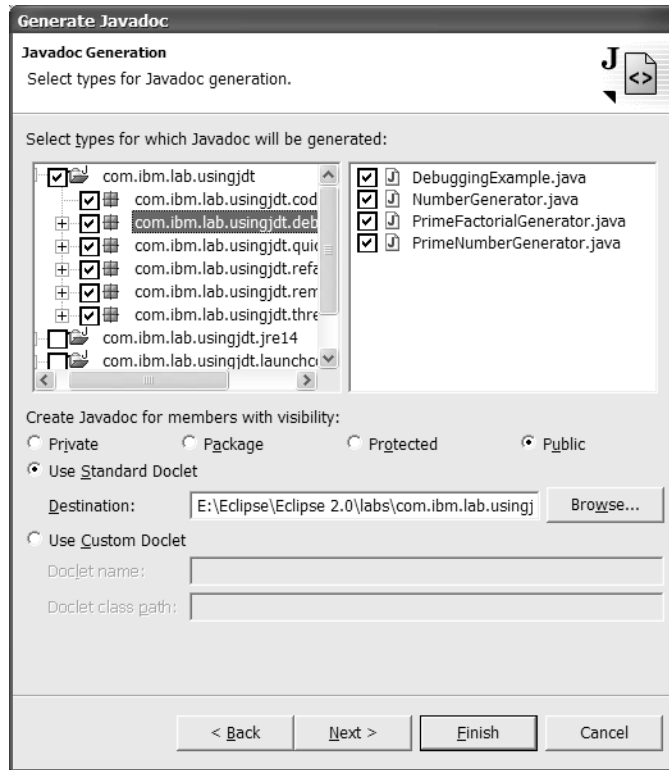


Figure 3.26 Generating Javadoc

independent; that is, you can write JRE 1.4 dependent code in an Eclipse running on a 1.3 JRE and vice versa. However, by default the JRE you use to run Eclipse is the same one the JDT uses to run your Java code. If you want to develop Java code that uses a different JRE from the one that runs Eclipse, you need to reconfigure Eclipse to define the JRE you are going to use and to set your JDK compliance preferences.

You specify the JRE your code needs in the **Installed JREs** preferences under **Java** (see Figure 3.27). Set the default JRE (with the associated check box) to be the one you just installed. This specifies the JRE used when you develop Java code. It does not affect the JRE used to run Eclipse.

Set your **Compiler compliance level** setting, on the **JDK Compliance** page of the **Java Compiler** preferences, to either **1.3** or **1.4**, depending on what your code requires (see Figure 3.28). This is a global preference that applies to all projects. We do not recommend attempting to manage JRE 1.3

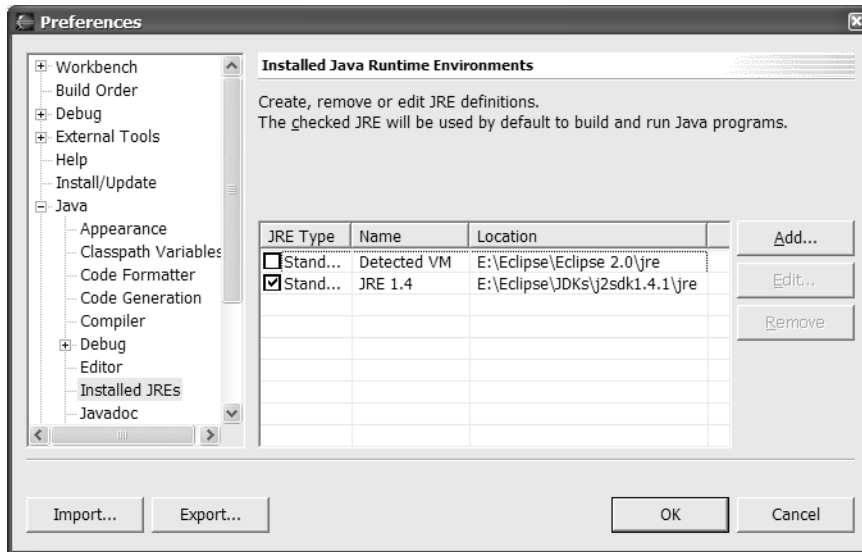


Figure 3.27 Setting Installed JREs Preferences

dependent code and JRE 1.4 dependent code in the same workspace. See the section “Multiple JREs, Differing JDK Level” later in this chapter for more information.

Each Java project maintains its own classpath. JDT sets a Java project’s classpath based on the project’s **Java Build Path** properties, which includes a JRE. If you have multiple projects that depend on different JREs, you have several configuration options. Your options depend primarily on the JDK level of the JREs your code requires.

Multiple JREs, Same JDK Level

You can have multiple Java projects that require different JREs, but the JREs are the same JDK compliance level, either 1.3 or 1.4. You have two options: You can configure to run multiple workspaces, one for each different JRE your code requires, or you can configure to run one workspace containing code requiring different JREs. We recommend the multiple workspace approach because it is simpler and less error prone.

To configure to run with multiple workspaces, use the following procedure.

1. Use the `-data` command line parameter to specify a different workspace for each different JRE. Create scripts and/or shortcuts to invoke Eclipse with your different workspaces.

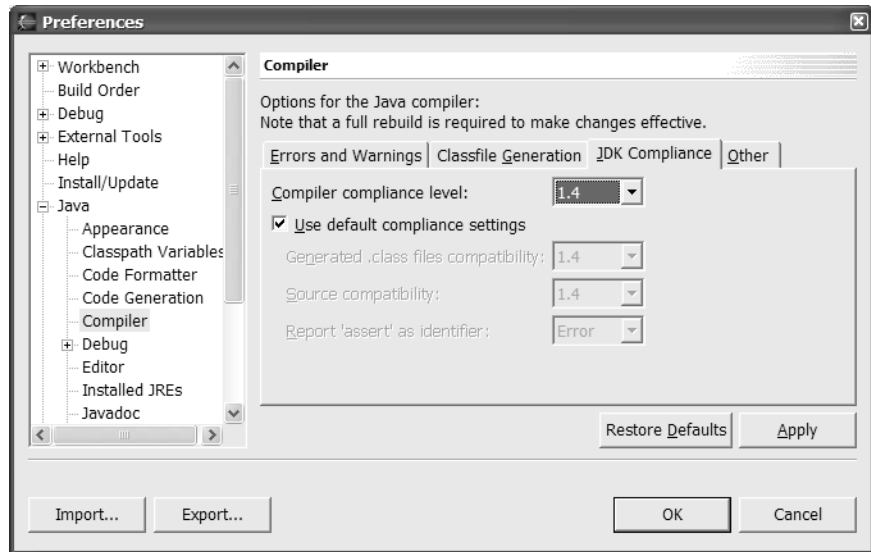


Figure 3.28 Java Compiler JDK Compliance Preferences

2. In each of your workspaces, if the JRE your code needs is not the one Eclipse is running on, specify the required JRE in your **Installed JREs** preferences. Set the default JRE (with the associated check box) to be the one you installed.
3. In each of your workspaces, set the **Compiler compliance level** setting in your **Java Compiler** preferences.

To configure to run with multiple JREs in one workspace, do the following.

1. Install all JREs required by the code you plan to develop in your workspace in your **Installed JREs** preferences.
2. Set the **Compiler compliance level** setting in your **Java Compiler** preferences.
3. For each project, set the project's build path to use the correct JRE through the Java Project Properties dialog by selecting **Properties** from the Package Explorer view context menu. Select **Java Build Path** and then the **Libraries** page. Remove the existing JRE entry (rt.jar file). Select **Add External Jars...** or **Add Variable...** to add the rt.jar of the JRE you want to use.

Multiple JREs, Differing JDK Levels

If you need to work in one workspace on code that requires different JREs at different JDK levels, practically speaking, you only have one choice. You need to separate the code into different workspaces, according to the steps outlined above. Given that the **Compiler compliance level** preference is global, and that each project maintains its own build path and classpath information, changing from 1.3 JDK compliance to 1.4 or vice versa will cause all open Java projects to be recompiled. If the compliance is set to JDK 1.4 level and your 1.3 dependent projects are rebuilt, your files may compile, but the generated `.class` files will not run with a 1.3 JRE (more precisely, be understood by a 1.3 JVM). If the compliance is set to JDK 1.3 level and your 1.4 JRE dependent projects are rebuilt, the compiler will not understand 1.4-specific syntax and you will not be able to produce 1.4 format `.class` files.

Running Java Code

To run Java code, you must be in one of the Java perspectives. There are three basic ways to run code in your Java projects: You can run (launch) a Java program with the **Run** action, you can run a Java program with the **Debug** action, and you can evaluate (execute) an Java expression in a scrapbook page. With the **Run** action, your program executes and you do not have an opportunity to suspend its execution or examine variable or field values. In debug mode, you suspend, resume, and examine a program's execution. We'll look more at debugging in Chapter 4.

You can run code even if it still has compiler errors. If the scope of an error is limited to a method, you can run code in the class, except for the method with the error. If the scope of an error is at the class level, for example, a problem with a static declaration, you can run code in other classes but not the one with the error.

Using the Run Action

- ✦ Java programs, that is, classes with `main` methods, are identified with the **Run** label decoration. To run a Java program, select a class or a Java element containing a class, and select **Run** from the menu or the **Run** pull-down menu, and then select **Run As > Java Application**. JDT executes the `main` method and sends output to the Console view. If you have previously run Java programs, you will have entries under **Run > Run History** and **Run** from the toolbar.



Select from these to re-run programs. You can also press **Ctrl+F11** or simply select **Run** from the toolbar to re-run the last program you ran.

When you run a Java program, you run it as a Java application, JUnit test, or run-time workbench. We're going to focus on Java applications here. Running as a run-time workbench is for testing extensions to Eclipse. We'll get to that in Chapter 8. Running as a JUnit test is beyond the scope of this book. For more information on this, refer to "Using JUnit" in the "Tasks" section of the *Java Development User Guide*.

To run a Java program, JDT needs two things: a main method and a launch configuration. Depending on what view was active and what you had selected when you requested to run a program, if a unique class with a main method can be determined, that main method will be run. For example, if the Java editor is active on an element with a main method, that main method will be run. If you have a project selected in the Package Explorer view and there are multiple classes in that package with main methods, you will be prompted to select one.

If the program encounters a run-time error, the exception information goes to the Console view and is displayed in error text color to distinguish it from other output types. Color-coding for output text in the Console view is defined in your **Console** preferences under **Debug**.

Managing Launch Configurations

Launch configurations define information for running Java programs. With launch configurations you can specify input parameters, JVM arguments, and source for your code, and set the run-time classpath and JRE. If you do not specify a launch configuration when you run a Java program with the **Run** action, a default is created for you. To define a launch configuration, select   the **Run** pull-down menu and then **Run...** In the Launch Configurations dialog (see Figure 3.29), select **Java Application** for **Launch Configurations** and then select **New**. If you had previously run Java programs with the **Run** action, you will see the default launch configurations that were created for you (see Figure 3.29).

On the **Main** page, **Project** is optional. If you specify it, its build path is used to set the classpath, source lookup, and JRE. Use **Search...** to search a project's build path for Java programs. If a valid project is specified, the build path of that project is searched for classes with main methods matching the search pattern (you can use wildcards) in **Main class**. If a project is not specified, the build paths of all the projects in your workspace are searched for classes with main methods matching the search pattern.

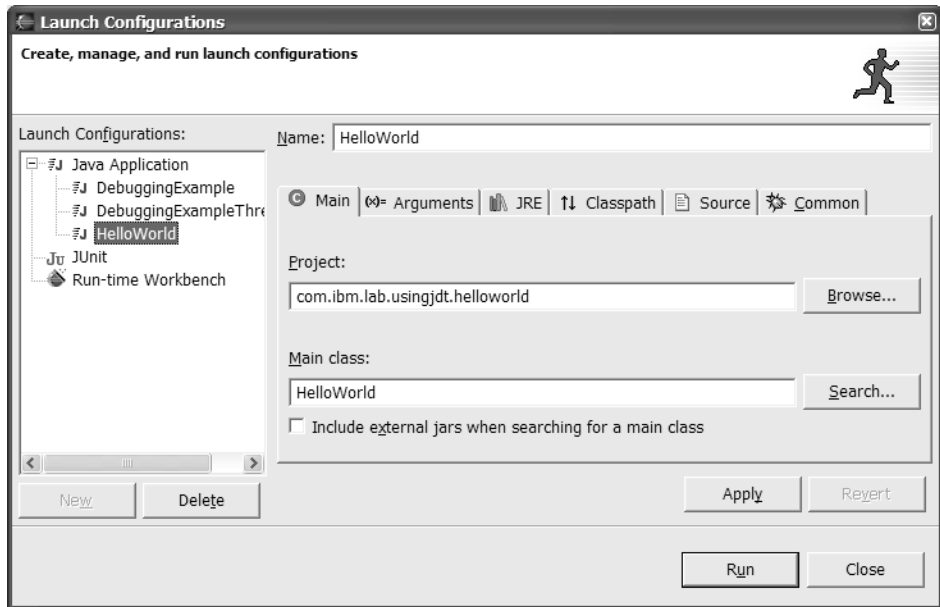


Figure 3.29 Creating a Launch Configuration

On the **Arguments** page, you can set the input parameters for the Java program and any JVM parameters. The Java program parameters are strings separated by spaces. For more information on JVM arguments, see “Running Eclipse” in the “Tasks” section of the *Workbench User Guide*.

On the **JRE** page, you set the JRE used to execute the Java program. Use this to override the JRE defined in the build path properties of the project containing the program. Choose from the list of choices, which is populated with the JREs you have defined in your **Installed JREs** preferences, or add a new JRE. This is useful, for example, to specify a 1.4 JRE in order to do hot code replace when you debug your code.

On the **Classpath** page, the classpath is set based on the build path information from the project specified on the **Main** page. To override or add to this, deselect **Use default classpath**. If you do not have a project specified, you need to do this and specify classpath information. If you choose not to use the default classpath, add references in the classpath to those projects and JAR files containing declarations the Java program references, including the project or JAR file containing the main method being executed.

On the **Common** page you specify information about how to save the launch configuration. By default, launch configurations are saved as metadata

in your workspace. If you select **Shared**, you specify a Java project into which to save the launch configuration as a `.launch` file. In this way, it's easy to keep and manage launch configurations with the code they run and to share them with others accessing the project; this is especially important for teams. The **Run mode** and **Debug mode** selections allow you to specify the perspective to display when the launch configuration is used to run or debug, respectively, the Java program. You can also indicate which favorites list displays the launch configuration, the **Run** or the **Debug** toolbar pull-down menus.

Evaluating Expressions in Scrapbook Pages

Java scrapbook pages (*.jpage files) allow you to edit and evaluate Java code expressions and display or inspect the results. They are a quick and easy way to test code and experiment with Java code expressions. You can evaluate an expression that's a partial statement, a full statement, or a series of statements. To create a scrapbook page, select **Create a Scrapbook Page** or use the **New** wizard. Scrapbook pages can be located in Java projects, folders, source folders, and packages.

Enter an expression in the Scrapbook page. This could be something simple like `System.out.println("Hello World")`, or an invocation of your Java code, for example, its `main` method. Content assist (**Ctrl+Space**) is available in scrapbook pages. Select the expression and then select **Display** from the toolbar or the context menu. JDT evaluates the expression, sends output to the Console view, and displays the `toString` value returned by the expression you selected in the scrapbook page. Select **Inspect** to display the results in the Expressions view. (We'll discuss the Expressions view in more detail in Chapter 4.) **Run Snippet** simply runs the code and sends the output to the Console view.

There are a couple of scrapbook page properties worth noting. To view or change these, select a scrapbook page and then **Properties** from the context menu. **Working directory** by default is the Java project in your workspace containing the scrapbook page. This is for relative file references. You can also change the JRE used for the scrapbook page. The default JRE is the one specified in your **Java** preferences under **Installed JREs**.

Scrapbook pages get their classpath from the containing project's build path. If in a scrapbook page you want to reference a Java element that is not on the build path of the containing Java project, you need to add to the Java project's build path. Scrapbook pages also allow you to specify `import` statements. You do this by selecting **Set Imports** from the context menu of a scrapbook page or **Set Import Declarations for Running Code** from the

toolbar. You need to set `import` statements for references to Java declarations in your projects. This is a common oversight. If the type or package you are attempting to import is not listed in the Add dialog, it means you need to add it to the build path of the project containing the scrapbook page. If you are referencing an element that has multiple declarations, you will need to add an `import` statement to uniquely identify the element.

- Each scrapbook page has its own associated JVM. The JVM is started the first time you evaluate an expression in a scrapbook page after it is created or opened. The JVM for a scrapbook page runs until the page is closed
- or explicitly stopped with **Stop Evaluation** from the context menu or the toolbar. When a scrapbook page is in the process of evaluating, the icon
 - 📄 changes to a red *J* on a gray background. If the expression you're evaluating results in a loop or a hang, select **Stop Evaluation**. In some cases this may not stop execution. If it doesn't stop, simply close the scrapbook page and then re-open it.

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, creating and importing Java elements, and details on local history for Java elements.

More on Java Projects

Java projects add a number of properties to projects, including external tool builders, the Java build path, and the Javadoc location. We discussed Javadoc location earlier in the “Generating Javadoc” section in this chapter, and external tools builders in the section “Customizing Eclipse” in Chapter 2. A Java project's build path property specifies how JDT organizes Java code, output files, and other resources in your project.

Java Source Code Organization

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 having 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 output folder. 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 output folder. For example, when you save a .java file and the file and project are built, the resulting .class file or files are put in the output folder. Other files in the source folders are copied to the output folder, unless you specify otherwise in your **Java Compiler** preferences. This approach allows you to organize and subdivide your Java packages, while the output folder contains all run-time resources.

There are two preference settings that allow you to customize this source code organization and build processing. **Java New Project** preferences allow you to specify the default organization for when you create new Java projects. You can later override this organization on the New Project wizard's **Source** page. 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 **Other** page of **Java Compiler** preferences.

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 run-time classpath for the code. When you first create a project, its build path is set to 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 that was generated from its build path information by defining a launch configuration.

Classpath errors show up with other errors in the Tasks view. If you have classpath errors and have trouble diagnosing them, open the properties on the project and check the **Java Build Path** pages. The icons for the entries indicate if they cannot be found.

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 build path properties for A. You do *not* affect this if you simply set project A's **Project References** properties to refer to 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 JDT. You can examine them, but making changes will have no impact.

Creating Java Projects

Eclipse maintains information for Java projects, including build path information, the Javadoc location, and external tools builder information. You have the opportunity to set this information when you create a new project on the New Java Project wizard (see Figure 3.30). To set the information, select **Next >** on the first page of the New Java Project wizard rather than **Finish**. You can also go back later after the project is created to modify this information by editing the project's properties.

On the **Source** page, you can specify the organization of your source code and output folders. **Use the project as source folder** is the organization described above in which output (*.class) files are put with source (*.java) files. **Use source folders contained in the project** allows you to separate source from output files. If you select this, you need to specify which folders contain Java source and/or resources; otherwise, the files will not be built. Use **Create New Folder...** to add source folders. Use **Add Existing Folders...** after the project has been created to add an existing folder to the project as a source folder. The build (compile) process takes inputs from source folders you add or designate and from folders they contain. In this way, as your project grows, you can subdivide your source to make it more manageable.

The **Projects** page allows you to add other Java projects to the build path of the new project (see Figure 3.31). You need to do this if you are going to make references from the project you are creating to Java declarations in other projects.

The **Libraries** page allows you to add other Java resources to your project's build path (see Figure 3.32). JDT automatically adds the default JRE. Select **Add Jars...** to add JAR or *.zip files from other projects, as opposed to adding references to resources in projects not contained in JARs or *.zip files, as you do on the **Projects** page. Generally, if you have the resources in a

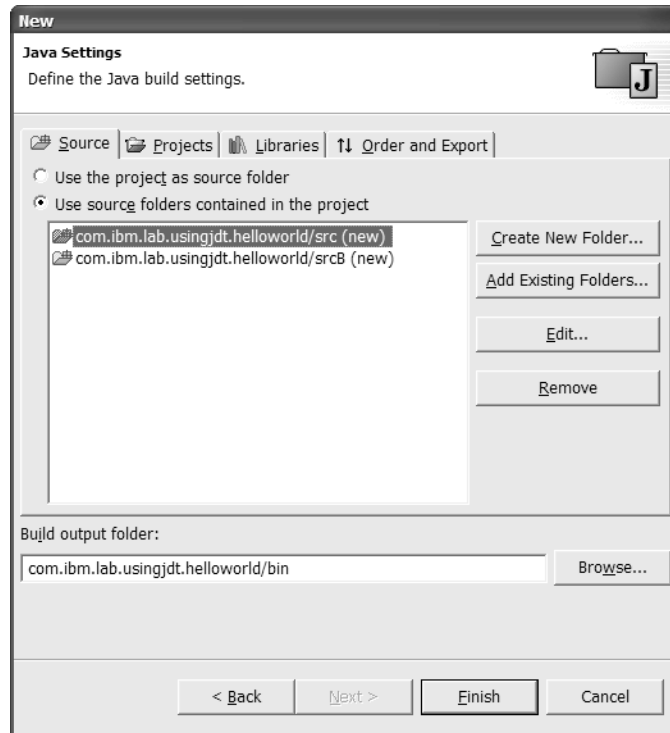


Figure 3.30 Java Build Path Properties Source Page

project, you should reference them that way, not within the JAR. Select **Add External JARs...** to add JAR or *.zip files from your file system. For example, if you wanted to develop a Java servlet, you would need to add `servlet.jar` to the project's build path. Use **Add Variable...** to add a classpath variable to your project's build path. The options under **Advanced...** allow you to add other source folders to the build path. **Attach Source...** allows you to add source to the selected build path entry. Do this to enable source-level debugging when the build path entry does not include source.

When you want to reference JAR files on your file system, especially if you do so in multiple projects, you should consider defining a classpath variable to refer to the JAR file and then including the variable in your projects' build paths. In this way, if the location of the JAR file changes, you only need to update the classpath variable and not the build paths of each of the projects. To define a classpath variable, use your **Java** preferences and select **Classpath Variables**.

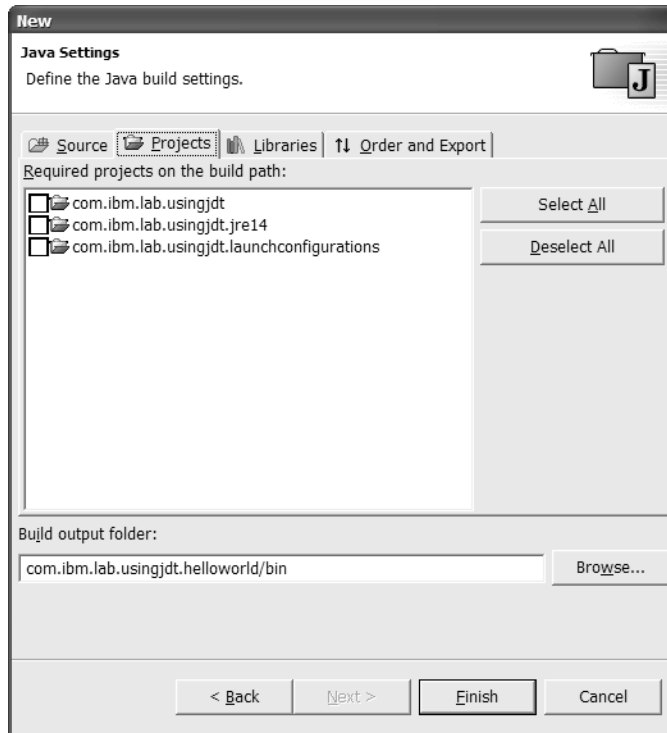


Figure 3.31 Java Build Path Properties Projects Page

If the build path icon is a hollow circle, it means that the library or file could not be found. Look for this when you have classpath errors.

The **Order and Export** page (see Figure 3.33) 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, it allows you to select an entry (that is, add a check mark) to cause its declarations to be visible outside the project.

Creating Folders

- 📁 There are two types of folders in Java projects: Source Folders and Nonsource (“simple”) Folders. This is an important distinction because it affects how builds are performed. When you create a source folder, JDT adds the source

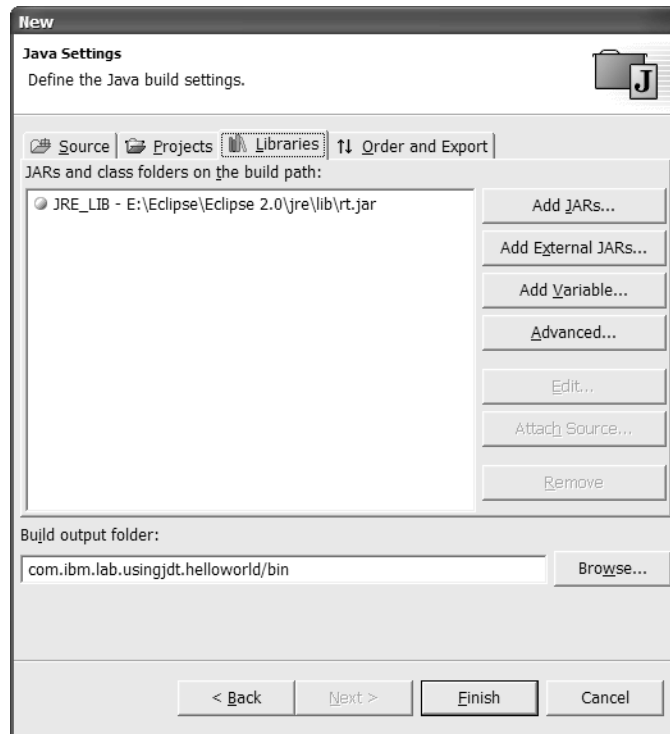


Figure 3.32 Java Build Path Properties Libraries Page

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. You can put nonsource folders in Java projects and folders; you can put source folders 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 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.

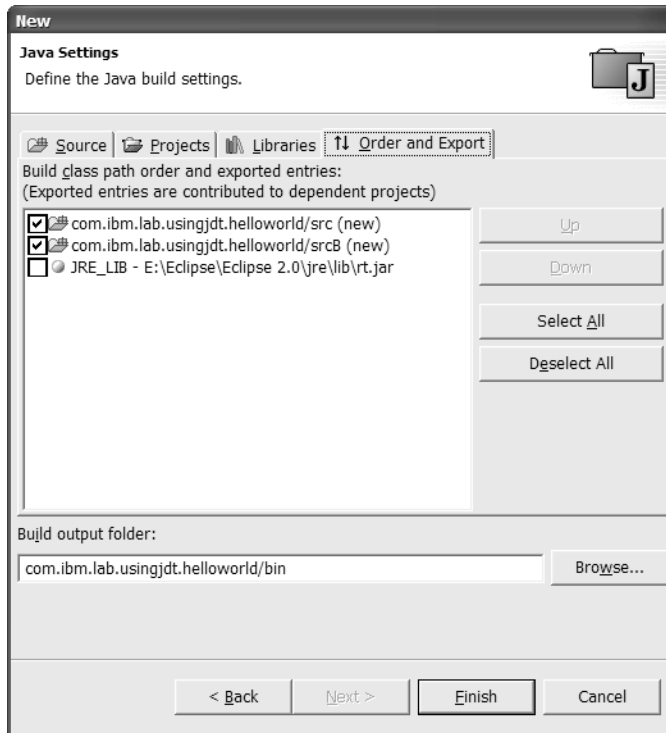


Figure 3.33 Java Build Path Properties Order and Export Page

Creating Classes and Interfaces

The options available when creating a new class or interface are straightforward. Several are worth noting. **Enclosing Type** is used for nested classes and 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. An easy way to create a subclass is to select a class in one of the Java views, and then select to create a new class. The **Superclass** of the new class is the one you had selected. 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 content assist and a code template. If you do not select to have **Inherited abstract methods** generated, you will see errors in the Tasks 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 **Override Methods...** from the context menu.

Importing Java Elements

In the section “Resource Management” in Chapter 2, we discussed importing resources with the Import wizard. Recall that you see 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, there are additional considerations. Generally, it is easier to deal with projects than individual Java elements because Java projects contain build path and tool builder information in the project’s `.classpath` and `.project` files. When you select to import **Existing Project into Workspace** in the Import wizard, a folder structure is a valid Eclipse project if it includes a `.project` file. Recall that importing an existing project into your workspace does not cause the contents of the project to be moved into your workspace folder. Rather, a definition for the project is added with a reference to its existing location. So, if you import an existing project from one workspace to another and you don’t intend to work on it further in the original workspace, be sure to delete the project definition there, *but not the contents*. This would delete the project’s contents for both workspaces.

The easiest way to import and export code and in general share it is to do so with an SCM repository and by means of a Team Project Set. We’ll see this in Chapter 5.

Local History for Java Elements

The section “Resource Management” in Chapter 2 described how you could compare and replace 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 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 3.34 shows comparing a single method, `generatePrimeNumbers()`, with a previous edition.

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 the Outline view and then selecting **Restore from Local History** from the context menu.

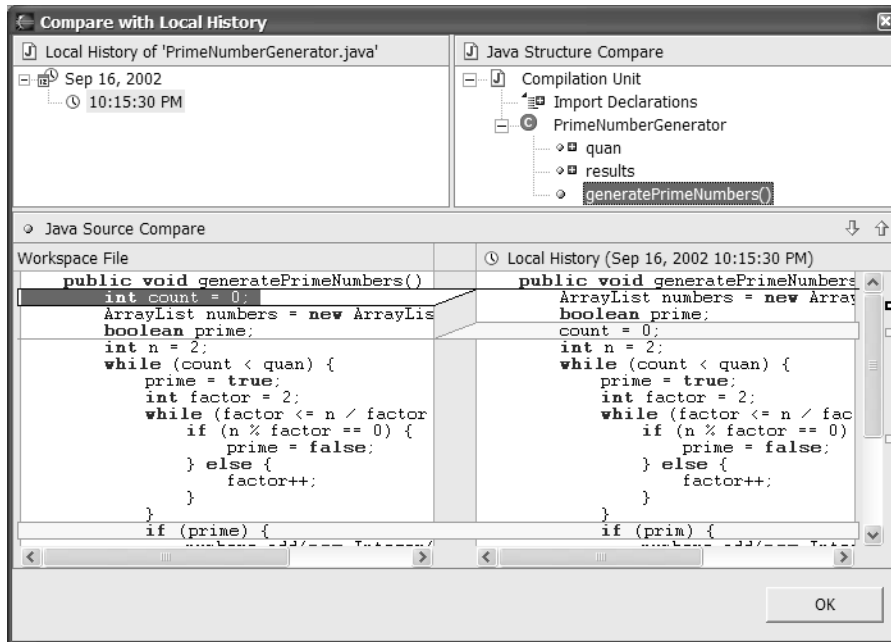


Figure 3.34 Comparing a Java Element from Local History

Tuning the Performance of JDT

In the section “Customizing Eclipse” in Chapter 2, we discussed a number of factors that affect performance of Eclipse, including startup time and memory usage. These included the number of installed tools; how many views, editors, and projects are open; and the number of resources they contained. There are several other factors that can also 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 your performance degrades, there are a number of things you can do to improve performance. Some of these things apply across all projects, while others are specific to the JDT. Obviously, each also involves tradeoffs of performance for capability.


- 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.
- Under **Workbench** preferences, deselect **Perform build automatically on resource modification**. If you do this, you will need to perform builds manually by selecting **Project > Rebuild Project** from the menu.

- Under **Java** preferences, for **Update Java views**, select **On save only**.
- Under **Java > Editor** preferences, on the **Code Assist** page, deselect **Enable auto activation**. On the **Problem Identification** page, deselect any of the three options.

More on JDT Views and Perspectives

Let's end this chapter with a little more detail on some of the views and perspectives that comprise 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 have static members, fields, and nonpublic members hidden. 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 of the Package Explorer view, you can filter the contents by working set or by a number of Java-specific criteria (see Figure 3.35). This is useful, for example, if you want to suppress the display of a project's referenced libraries.

Package Explorer View

If the package names in the Package Explorer view are long and/or you have the width of the view sized so 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 package name segments (except the last one)** 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 to it be compressed to `o.e.jface`,
- 0 results in `jface`, and
2. results in `or.ec.jface`.



Figure 3.35 Filtering the Package Explorer View

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** or **Forward** to focus the contents on the selected containing 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 default, instead of expanding the element in the view.

Finally, if you work with JAR files in your projects, in your **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 3.36). There are three presentations available from the

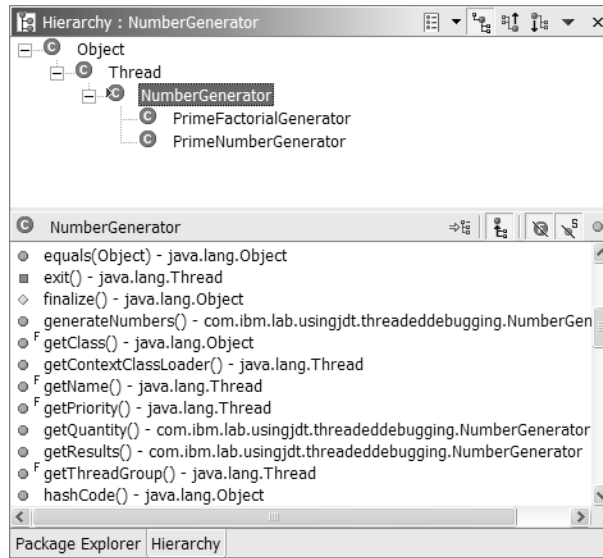



Figure 3.36 HierarchyView

- toolbar: the Type Hierarchy (**Ctrl+I**), the Supertype Hierarchy (**Ctrl+2**), and the Subtype Hierarchy (**Ctrl+3**). These three presentations revolve around a focal point type in the view, indicated by the label decoration. This is the type you opened the view on. Change this with **Focus On...** from the context menu.
- LockView 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 a Java element from one of the Java views and drop it on the Hierarchy view to open it there. The view maintains a list of the elements you have opened here. Rather than opening an element again, you can select **Previous Hierarchy Inputs** to see a list of those available. Change the orientation

- of the view from vertical to horizontal by selecting **HorizontalView Orientation** from the **Menu** pull-down on the Hierarchy view toolbar.

Tasks 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 manage if you filter them. Select  **Filter...** on the title bar and then select **On selected resource only** or **On selected resource and its children**.

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 **Previous Search Results** 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 3.37). When you select **Open Type Hierarchy**, 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. VisualAge for Java users, who like the ability to double-click on a class to open it in a window, may find this useful.

Java Browsing Perspective

The Java Browsing perspective, shown in Figure 3.38, provides a slightly more structured approach to navigating through Java resources than the Java

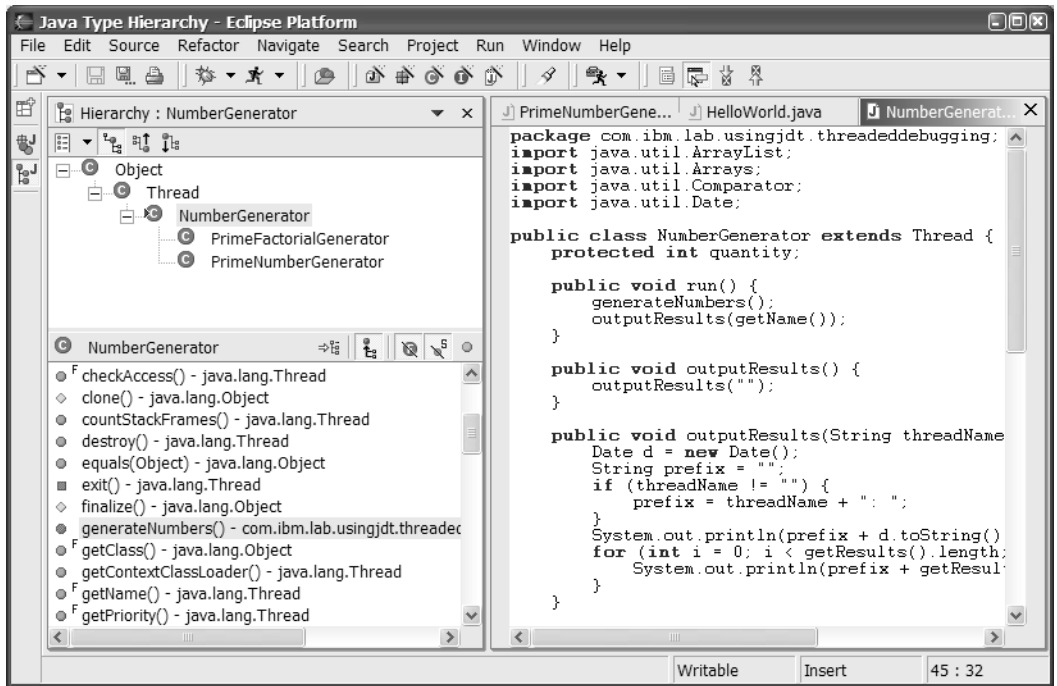


Figure 3.37 Java Type Hierarchy Perspective

perspective. In addition to an editor pane, it contains Projects, Packages, Types, and Members views. The editor pane is set to **Show Source of Selected Element Only**. The organization of this view will be familiar to VisualAge for Java users. 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** 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 3.38. 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**.

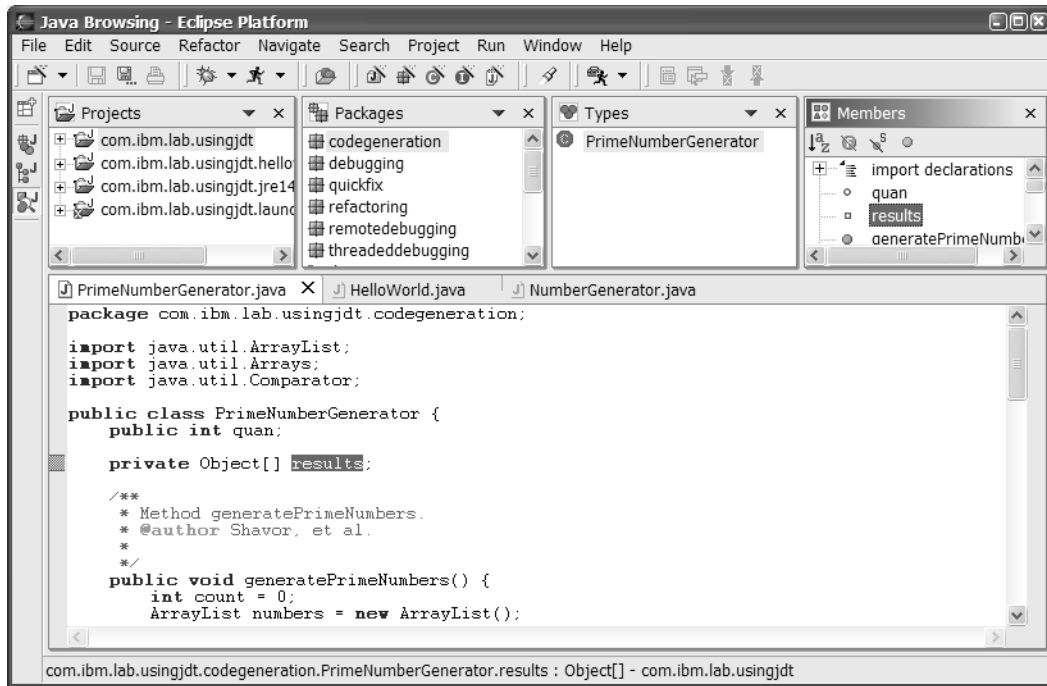


Figure 3.38 Java Browsing Perspective

Exercise Summary

The exercise in Chapter 29 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 using a scrapbook page.

2. Quick Fix

You'll use 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 JDT when using refactoring to clean up, reorganize, and extend a program.

5. Launch Configurations

You'll see how to use launch configurations to run Java programs, and you'll pass parameters to a program, pass JVM parameters, and reference declarations in a JAR file.

6. JRE 1.4 Code

In the final part of the exercise, you'll see how to configure JDT in order to develop code that requires JRE 1.4.

Chapter Summary

This chapter provided a comprehensive overview of using the Java Development Tools (JDT) to explore, write, and run Java. It 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 content assist, code generation, and refactoring. It covered how different kinds of errors are marked and how to fix errors with quick fix. The chapter discussed how to set Javadoc locations and view Javadoc. It 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. It explained how to run Java code and how to evaluate Java expressions in a scrapbook page, run Java programs using the **Run** action, and use launch configurations to run Java programs, including specifying program and JVM parameters.