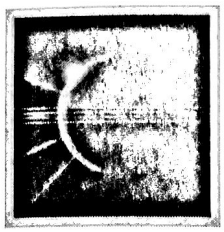

CHAPTER 4

Debugging Java

We've discussed writing and running Java code. Perhaps you've tried this with our exercises or some of your own code. If you're like us, *occasionally* you may find you've coded incorrectly and an error shows up at runtime ☺.

The Eclipse Java Development Tools (JDT) includes state-of-the-art debugging capabilities. The Java Debugger allows you to detect and correct errors in your code as your code executes. It can debug your code locally or remotely. You can control the execution of your code with varying degrees of refinement, set simple and conditional breakpoints, and examine and change variable values.

JDT supports **hot code replace**, a feature new to JRE 1.4 that allows you to replace class definitions once execution has started. This is valuable because it means you can change your code while it is executing to fix errors without having to restart and attempt to re-create the same conditions.

In this chapter we'll start with the basics like setting breakpoints and controlling program execution, and then go into more detail on debugging Java with JDT.

Overview

JDT debugging capability comprises the Debug perspective, several views, and enhancements to the Java editor to allow you to find and fix run-time errors in your programs. You control the execution of your Java program by setting breakpoints and watchpoints, examining and changing the contents of fields and variables, stepping through the execution of your programs, and suspending and resuming threads. Figure 4.1 shows the default configuration of the Debug perspective as it appears when you first start a debugging session.

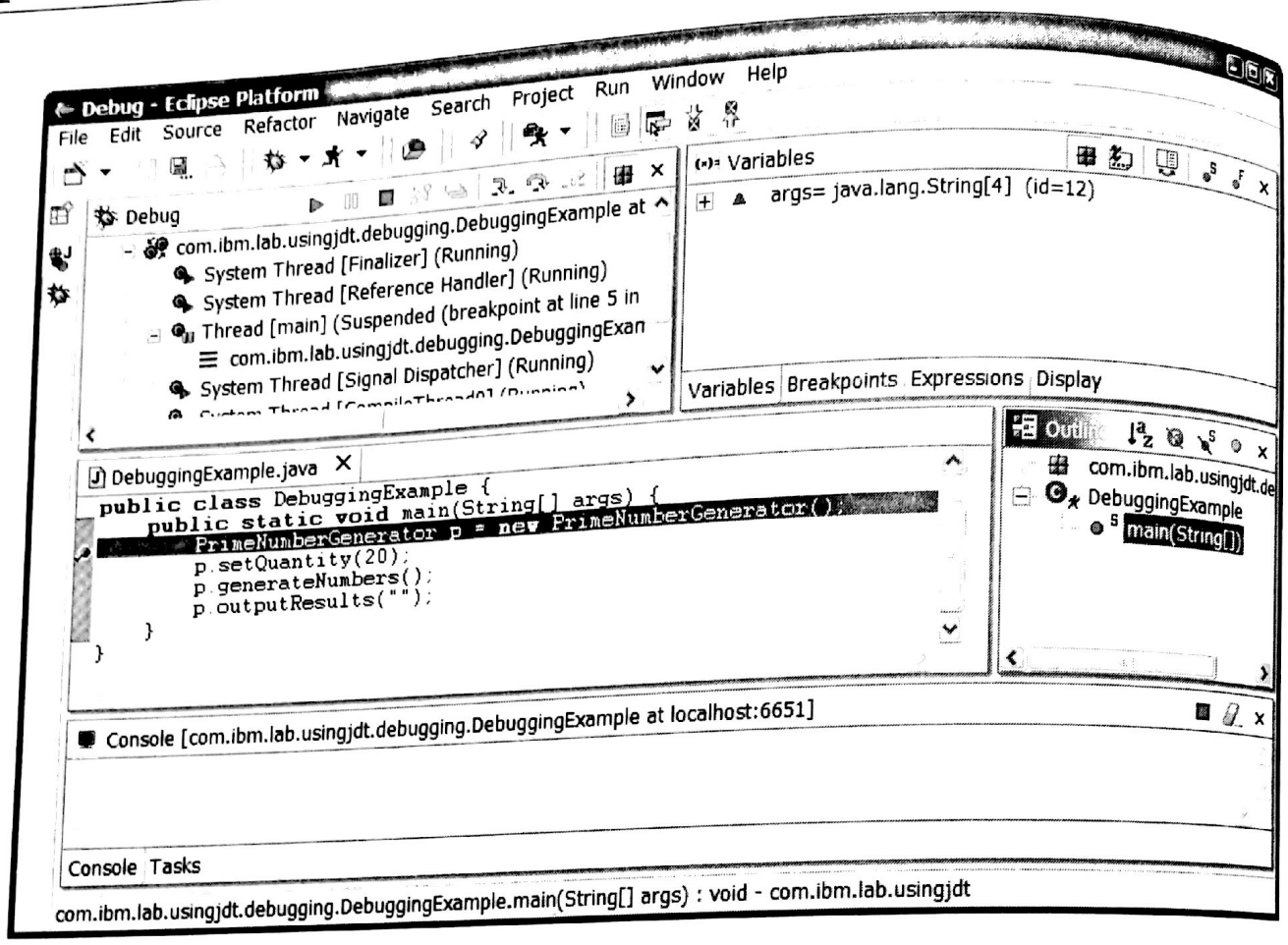

Figure 4.1 Debug Perspective

In the Debug view in the upper left are entries for the programs you're debugging or ones that you were debugging. In this case, we have launched an instance of `DebuggingExample`. The current stack frame (where execution is suspended) is selected. The Variables view, in the upper right, shows the values of local variables. Stacked behind this are the Breakpoints, Expressions, and Display views for examining the status of the executing program. The code for the current stack frame is in the editor in the middle on the left. The line with the breakpoint that caused execution to suspend is selected. The contents in the Outline view (in the middle on the right) correspond to the source in the editor. On the bottom, the Console view shows the output from the program.

The Fundamentals

When you debug Java, your basic activities are

- Stepping through the execution of your program with actions in the Debug view

- Following the source in the editor as it executes
- Managing your breakpoints from the editor and the Breakpoints view
- Examining variable values in the Variables view
- Evaluating expressions and viewing the results in the Expressions and Display views
- Following the output of your program in the Console view

Getting started with debugging is simple. You set a breakpoint in your code, start a debugging session, control execution of your code, and examine the state of your program as it runs. The following sections describe these steps.

Setting a Breakpoint

The simplest way to set a breakpoint is to double-click in the editor's marker bar on the line you want the breakpoint defined. You can also position the insertion cursor in a line and then press **Ctrl+Shift+B**.

Starting a Debugging Session

To start a debugging session, select a Java program or a Java element containing the `main` method you want to debug, and select **Debug > Debug As > Java Application** from the menu or the toolbar. The behavior here is the same as the respective **Run >** actions to run a Java program. JDT executes the Java program, suspends execution prior to the line with a breakpoint you defined, and opens the Debug perspective. If you do not start execution with one of the **Debug** actions, execution will not suspend on breakpoints you have defined.

Controlling Program Execution

You control program execution from the Debug view with the following actions.

- **Step Over** or **F6** executes a statement and suspends execution on the statement after that.

- **Step Into** or **F5**, for method invocations, creates a new stack frame, invokes the method in the statement, and suspends execution on the first statement in the method. For other statements, like assignments and conditions, the effect is the same as **Step Over**.

- **Step Return** or **F7** resumes execution to the end of the current method and suspends execution on the statement after the method invocation, or until another breakpoint is encountered.

- **Resume** or **F8** causes execution to continue until the program ends or another breakpoint is encountered.
- **Terminate** stops the current execution without executing any more statements.

Examining an Executing Program

When your program's execution is suspended in the Debugger, you can examine its execution state in the following ways.

- **Manipulating variable values**—examine variable values in the Variables view, and change a value by double-clicking it.
- **Viewing field values**—see the value of a field by hovering over it in the editor.
- **Evaluating expressions**—select an expression in the editor or the Display view, and then select **Display** or **Inspect**. The results are shown in the Display view and Expressions view, respectively.
- **Viewing method invocations**—see the series of nested method invocations that led to the current stack frame by selecting previous stack frames in the Debug view.
- **Viewing program output**—see program output in the Console view as you step through your program's execution.

Additional Debugging Capabilities

Many of the views allow you to show and hide static and final fields and qualified names. This is useful for reducing content in a view if it becomes cluttered. Values presented in many of the views are results returned by `toString`. Override this method in your classes to provide more information during debugging.

Evaluating Expressions

The Expressions and Variables views have Detail panes that appear in the bottom half of the view (see Figure 4.2). Open these with **Show Detail Pane** from the view title bar. The Detail panes are useful for displaying values as you scroll through entries in the top half of the pane. You can also edit expressions in the Detail pane, including using content assist, and evaluate them by selecting **Inspect** or **Display** from the context menu or **File > Inspect** or **File > Display**. The keyboard shortcuts are **Ctrl+Q** and **Ctrl+D**, respectively.

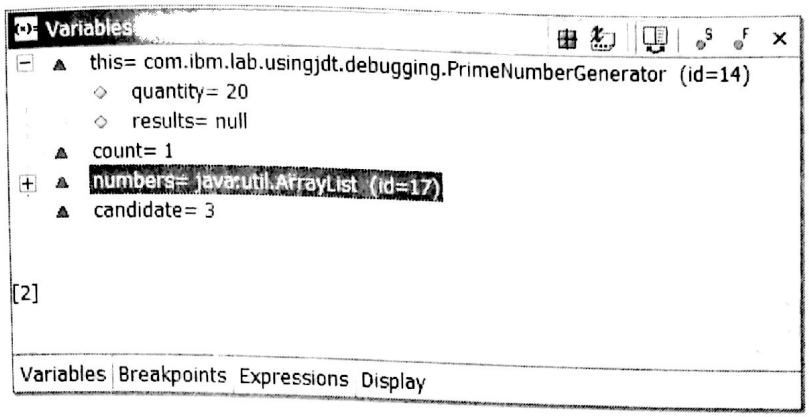

Figure 4.2 Detail Pane

You must have a stack frame selected to evaluate an expression. When you evaluate an expression, it is done in the context of the current (selected) stack frame. You cannot evaluate expressions if you have manually suspended execution. The expression must be a complete, valid one. Recall the keyboard shortcuts for selecting expressions: **Alt+Shift+Up**, **Alt+Shift+Down**, **Alt+Shift+Left**, and **Alt+Shift+Right**. Finally, some expressions, for example, variable declarations, cannot be evaluated.

Changing Variable and Field Values

To change values in the Variables and Expressions views, double-click on a value and enter a new one. Changing the text in the Detail pane of the Expressions and Variables views does not change the value. As you step through program execution and values change, the Variables view notes this by changing the color of its entries. For example, in Figure 4.3, when `count` is incremented, its entry in the view will show in red.

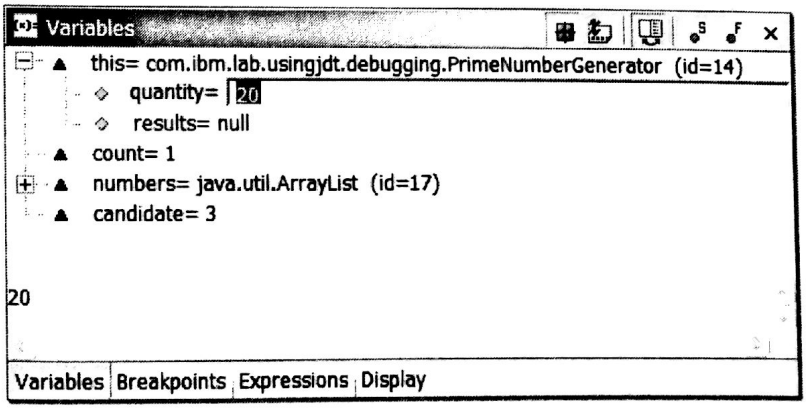

Figure 4.3 Changing a Field Value

Changing values in the Variables and Expressions views changes them in the context of the current stack frame. If you use **Inspect** to evaluate an expression and that expression returns object references, those references point to objects in the current stack frame. At this point, changing a value of an object shown by reference in the Expressions view changes the value in the current stack frame. This impacts the execution of your program.

Debugging with the Java Editor

The Java editor shows the code for the stack frame selected in the Debug view. This is the same Java editor that appears in the Java perspectives. Content assist, code generation, and refactoring work the same as in the Java perspectives. If you have not configured hot code replace and you make changes to a class that has been loaded, the changes will not take effect until you restart the program. A class that has been loaded has its breakpoint icons decorated with a checkmark.

If you attempt to **Step Into** a method for which the Debugger does not have a source, you'll see a notification in the editor (see Figure 4.4). At this point, if you have the source, you can attach it. (We'll get to this in the section "Associating a Source with Your Programs" later in this chapter.) If not, select the previous stack frame (the one that invoked the method having no source) and then select **Step Over**.

From the context menu, select a line and then select **Run to Line** or **Ctrl+R** to have execution resume and then suspend on the selected line or at the next breakpoint encountered. This is neat because you do not have to first set a breakpoint.

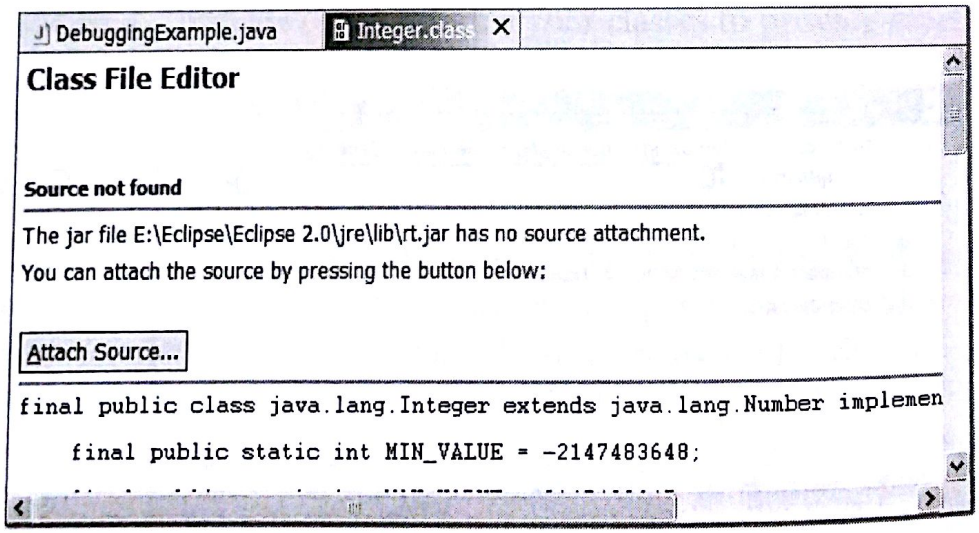

Figure 4.4 Source Not Found

If you're in the Debug perspective and you switch to the Java perspective to edit another file, when you switch back to the Debug perspective, the editor will still show the last file you were editing, not the one you were debugging. At this point, select the file you were debugging (if you recall) or select a stack frame entry in the Debug view to get back to where you were in your debugging session.

Manipulating the Programs You're Debugging

The Debug view shows the programs you're currently debugging (and others you were debugging that have terminated) and their associated processes and threads (see Figure 4.5).

You will see a Java program entry for each program you debugged or are debugging. Each of these has a Debug Target. A **Debug Target** is a running instance of a JVM executing code. Debug Targets have properties that are useful for understanding the context in which the code is executing, such as the `main` method being executed, Java command line and VM parameters, and classpath information. To see this information, select a Debug Target in the Debug view and then select **Properties** from the context menu (see Figure 4.6).

Your initial focus will usually be on the main thread and the stack frame(s) under it. This is the thread of execution you're debugging and seeing in the editor. Stack frames appear with the most recent method invocation at the top of the list. Select the stack frames in this view to see the method invocations that led to the current state of execution.

There are a number of useful options in the Debug view available from the toolbar and/or context menu. While a program is executing, you can

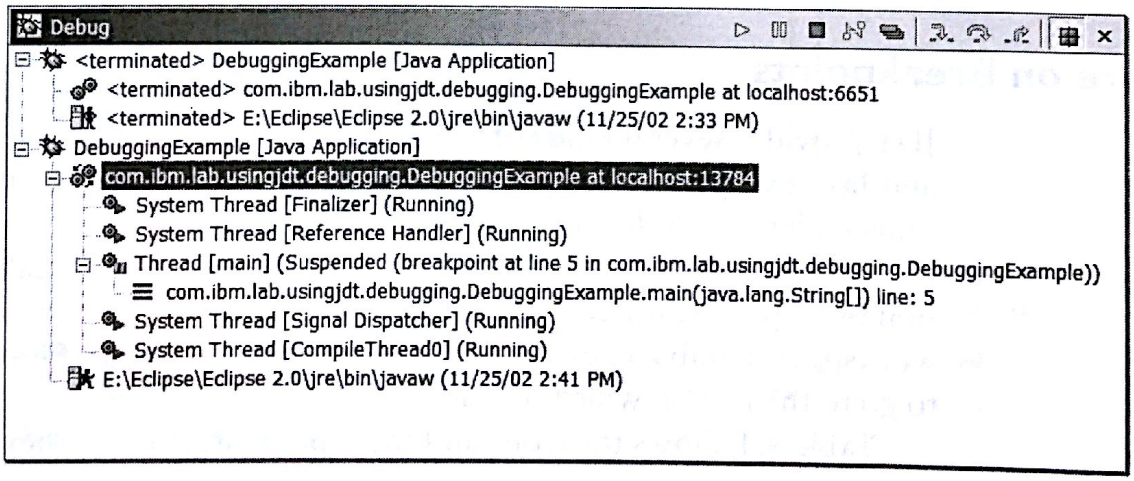

Figure 4.5 Debug View

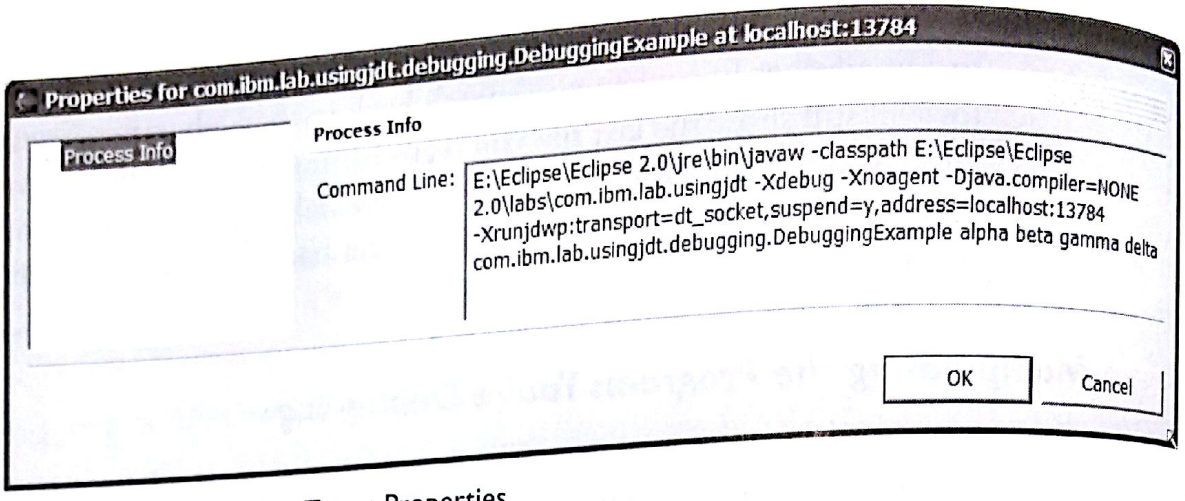

Figure 4.6 Debug Target Properties

manually suspend it with **Suspend**. The current stack frame is displayed and the line that was executing is selected. This is useful if, for example, your program gets into an infinite loop. **Relaunch** leaves the current process as it is and starts another execution of the same program. When you terminate a program or it finishes execution, it remains in the Debug view. This allows you to view the output of terminated programs in the Console view. If the list becomes cluttered, clean it up with **Remove All Terminated Launches**. Finally, you should clean up debugging sessions you're through with, as the running JVMs associated with the debugging sessions can degrade performance. Use **Terminate and Remove** from the context menu to do this.

If you don't want to bother with the "Source not found" dialog when you use **Step Into**, use step filters from the Debug view context menu. This causes **Step Into** to instead **Step Over** for selected packages and classes. These filters are defined in your **Java Debug Step** preferences under **Filtering** (see Figure 4.7).

More on Breakpoints

JDT provides several kinds of breakpoints: line, method, field (watchpoints), and Java exception breakpoints. In addition, some of these can have conditions under which they become enabled, or active.

The Breakpoints view (see Figure 4.8) shows all line breakpoints, exception breakpoints, watchpoints, and method breakpoints defined in your workspace. Double-click on a breakpoint or select **Go to File for Breakpoint** to go to the line on which it is defined.

Table 4.1 shows the icons and the type of breakpoint they indicate.

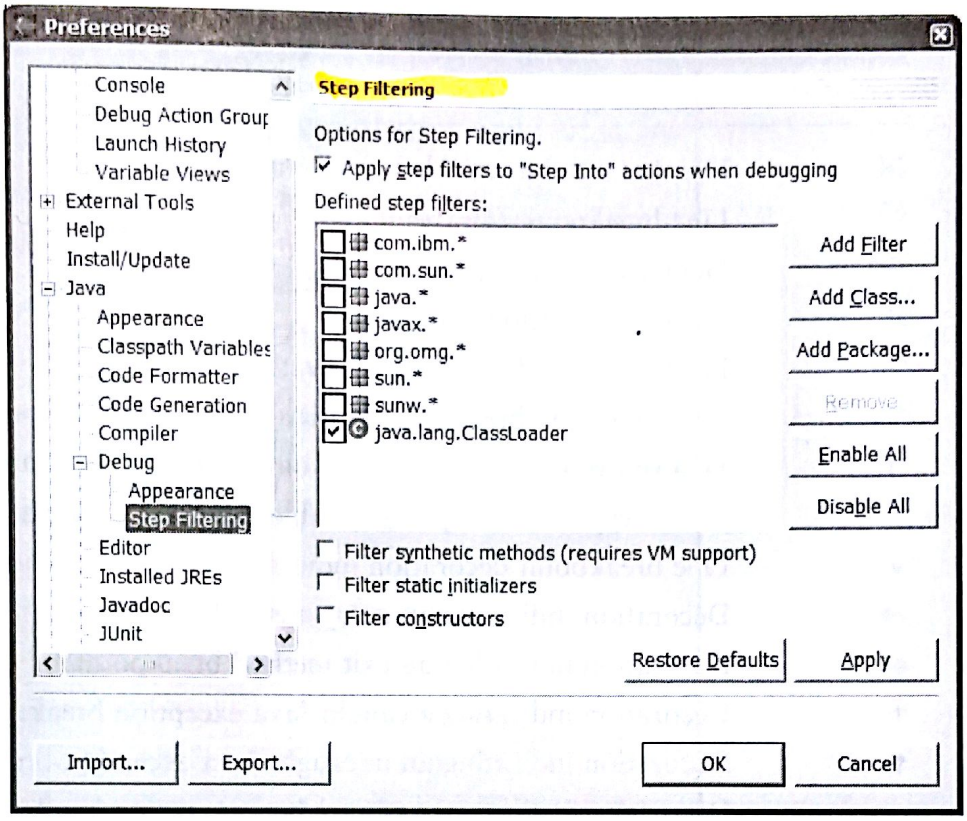

Figure 4.7 Java Debug Step Filtering Preferences

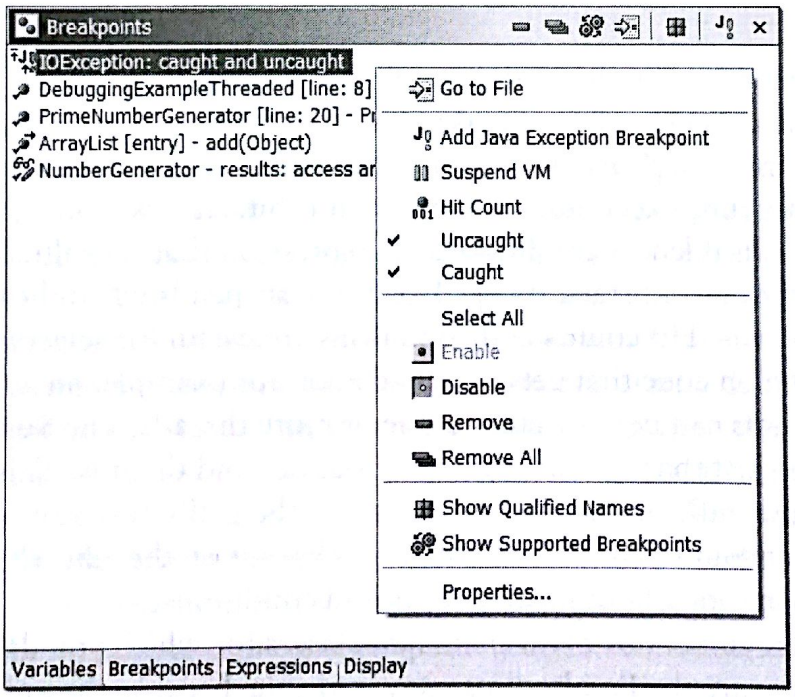

Figure 4.8 Breakpoints View

Table 4.1 Breakpoint Icons

Icon	Description
	Line breakpoint (enabled)
	Line breakpoint (disabled)
	Field access watchpoint
	Field modification watchpoint
	Field access and modification watchpoint
	Java exception breakpoint (with a yellow exclamation mark)
	Java run-time exception breakpoint (with a red exclamation mark)
	Java exception breakpoint (disabled) (with a gray exclamation mark)
	Line breakpoint decoration indicating the class is loaded
	Decoration indicating an entry method breakpoint
	Decoration indicating an exit method breakpoint
	Decoration indicating a caught Java exception breakpoint
	Decoration indicating an uncaught Java exception breakpoint
	Decoration indicating a scoped Java exception breakpoint

Select a breakpoint and then select **Properties...** from the context menu to set the properties for a breakpoint, including hit counts and conditions (see Figure 4.9). When a hit count of *n* is defined on a breakpoint, the breakpoint suspends execution the *n*th time it is hit. At this point, the breakpoint becomes disabled. A **condition** is an expression that is evaluated in the context of the current stack frame. Execution suspends when the condition evaluates to `true`. Hit counts and conditions are useful for selectively suspending execution on code that gets executed a lot, for example, an `add` method. Breakpoints can be restricted to one or more threads. The **Suspend VM** option does just that—it suspends all processes and threads. This is useful for getting a handle on the execution state of the entire program when the breakpoint is encountered. Hover over a breakpoint on the editor's marker bar to see information about its hit counts and conditions.

When you define breakpoints on classes for which you do not have a source, ensure the Breakpoints view is visible, because the only indication the breakpoint was defined is by a new entry in the Breakpoints view.

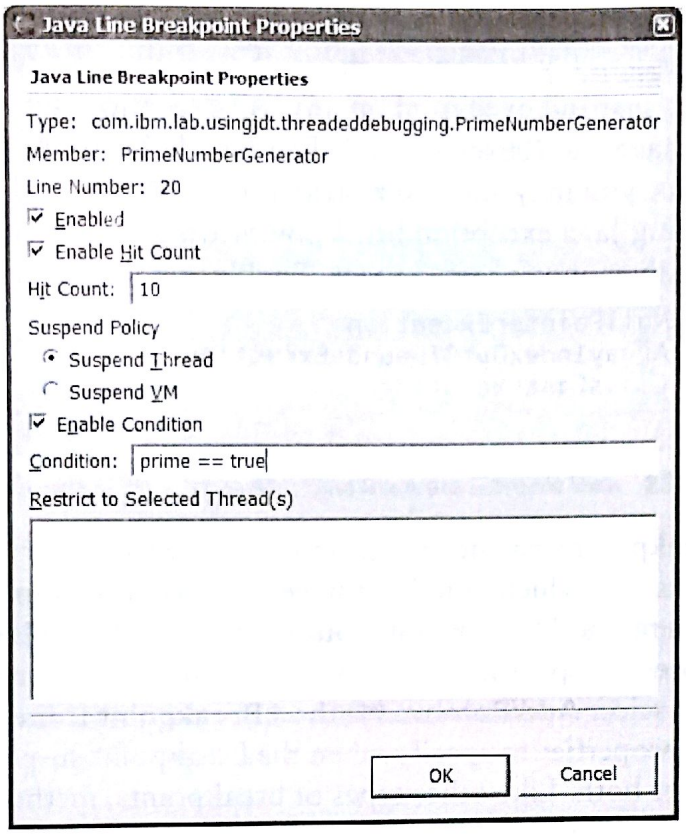

Figure 4.9 Breakpoint Properties

Java Exception Breakpoints

Java exception breakpoints allow you to suspend execution when the program you are debugging throws an exception. Java exception breakpoints suspend execution when any code throws the exception defined in the breakpoint, including classes in run-time libraries (for example, the JRE). If your program is throwing an exception and you're not sure where it's originating, define a breakpoint on the exception and start a debugging session on the program. The Debug perspective will open when the exception is thrown, allowing you to examine the execution state at that point.

To specify a Java exception breakpoint, select **Add Java Exception Breakpoint** in the Breakpoints view. Edit the breakpoint's properties to specify whether execution suspends when the exception is caught, not caught, or both. Java exception breakpoints can have hit counts, but not conditions. In addition, you can restrict a Java exception breakpoint to a set of classes and/or packages.

Use the class structure of exceptions to specify individual breakpoints that suspend execution when any of their subclass exceptions are thrown. For example, to suspend execution on any I/O exception, set an exception breakpoint on `java.io.IOException`. When you define more general exception breakpoints, you may want to restrict them only to uncaught exceptions.

The following Java exception breakpoints are generally useful. Define them and leave them in your workspace. Enable and disable them as required.

```
java.lang.NullPointerException
java.lang.ArrayIndexOutOfBoundsException
java.lang.ClassCastException
```

Method Breakpoints

A **method breakpoint** suspends execution on entry to and/or exit from a method in a class for which you don't have a source, for example, in a JAR file. To define a method breakpoint, you first need to "open" it in the editor to have its contents appear in the views. Select a method in one of the Java views and then select **Add/Remove Method Breakpoint** from the context menu. Edit its properties to specify when the breakpoint suspends execution: on entry, exit, or both. Like other types of breakpoints, method breakpoints can have hit counts and conditions.

Watchpoints

A **watchpoint** (or field breakpoint) suspends execution on the line of code that is about to access the field on which the watchpoint is defined. When you want to observe how a field is accessed, it's often easier to use a watchpoint on the field than it is to try to set breakpoints on all the lines that might access the field. You can define watchpoints on fields in run-time libraries or JAR files; that is, fields in classes for which you do not have a source. To define a watchpoint, select a field in one of the Java views and then select **Add/Remove Watchpoint** from the context menu. Edit the breakpoint's properties to specify when the watchpoint should suspend execution: on access, modification, or both. Watchpoints can also have hit counts and conditions.

Program Output in the Console View

In a debugging session, the Console view provides the ability to open a Java type from the output text (see Figure 4.10). Select text in the Console view or position the cursor on a line and then select **Open on Type** from the context

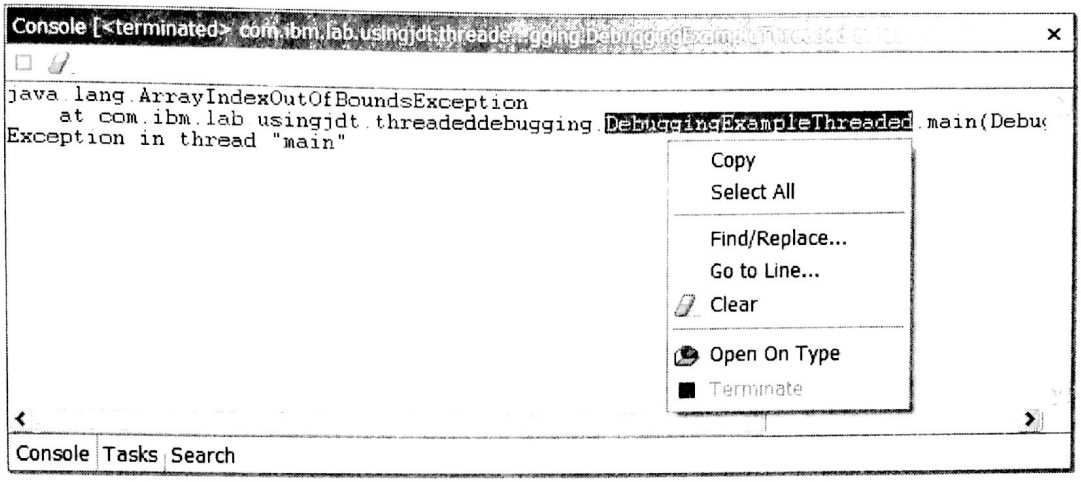

Figure 4.10 Opening a Type from the Console View

menu to have the selected expression or line scanned for a type name. If a valid name is found, the type is opened in the editor. Double-clicking on text will also open a type with a valid name in the editor. This is a convenient way to quickly go to Java code references in diagnostic or error output.

Debug Launch Configurations

We introduced launch configurations in the section "Running Java Code" in Chapter 3. They are a way to define a set of properties for running Java programs. You can also use launch configurations to define the same properties to launch your program in a debugging session. To do this, select the **Debug** pull-down menu and then **Debug...**.

On the launch configuration **Source** page, you specify the location of the Java source being debugged, so that it can be presented in the Debug perspective. Uncheck **Use default source lookup path** and add projects and/or JARs containing the source. You need to do this if you do not have a project specified on the **Main** page. You may also want to do this for libraries you reference.

Associating a Source with Your Programs

As you are stepping through code in a debugging session, you may see a dialog in the editor indicating that it cannot find a source for an associated method. If the missing source is for the JRE, use your **Installed JREs** preferences to associate the source. Select the JRE and then **Edit...** to display the Edit JRE dialog (see Figure 4.11). Deselect **Use default system libraries**, select

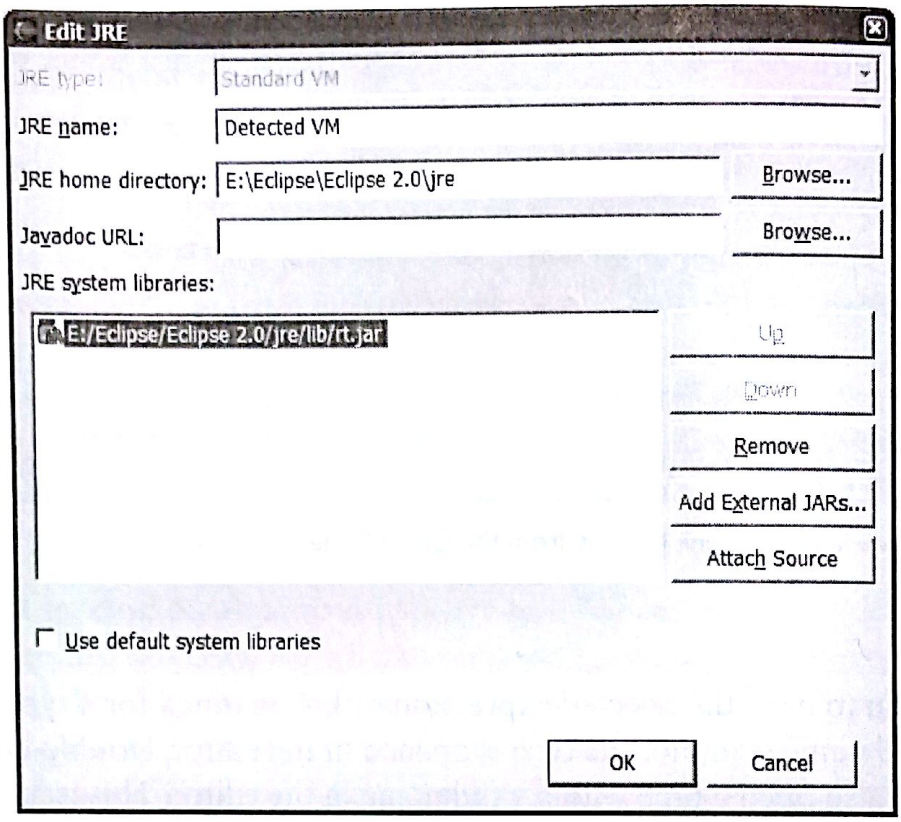

Figure 4.11 Associating a Source for a JRE

the JRE system library, select **Attach Source**, and then browse the file system for the `.zip` or `.jar` file with the associated source.

If the missing source is not for the JRE, select **Attach Source...** in the Source not found dialog and then use one of the options to browse the file system for the associated source. To associate a source with a JAR file you have added to a project's build path, edit the project's **Java Build Path** properties. Select the **Libraries** page, select a library (JAR file), and then select **Attach Source...**.

Hot Code Replace

Hot code replace is a powerful tool that enables you to quickly try fixes and/or alternative code while in a debugging session, without having to restart your program and reproduce the state you were debugging. Even if you intend to deploy on a JRE that does not support hot code replacement, developing on one prior to final testing can be more productive.

To use hot code replace, you need to debug your program with a JVM that supports it. To install a JRE (and associated JVM), do so with your **Java Installed JREs** preferences. Set this to be the default JRE. Or, create a launch

configuration for your program, and on the **JRE** page specify an installed JRE that supports hot code replace. Even though you will be running your program on a 1.4 JRE with its associated JVM, you do not have to set your **Java Compiler** preferences to a JDK compliance level of 1.4. There are two use cases here. First, your program depends on JRE 1.4 APIs. In this case, you should define the 1.4 JRE as your default JRE in your **Java Installed JREs** preferences. The second case is that you are using a 1.4 JRE only to debug your program to enable hot code replacement but you intend to deploy the program to run with a 1.3 JRE. In this case, your default JRE should be a 1.3 JRE and you should leave your JDK compliance level as 1.3. You should use a launch configuration to specify a 1.4 JRE to use while debugging to enable hot code replace. Doing this ensures that `.class` files are created in the right format and that you don't inadvertently access 1.4 specific declarations.

Once you're configured for hot code replace, simply set a breakpoint in your code and start a debugging session. When you stop at a breakpoint, change the code in the editor and save it. The changed class is compiled, the stack frame replaced, and debugging resumes. If you do not have auto-build enabled in your preferences, you'll need to do a manual build by selecting **Project > Rebuild Project** or **Rebuild All**.

If you get an error when attempting hot code replace, one or more of your stack frames may be obsolete or invalid. Bypass these stack frames by selecting the most recent (top-most in the Debug view) valid stack frame and stepping over the selected line (method invocation). Otherwise, you will need to terminate your current debugging session and relaunch your program.

There are some limitations with hot code replace worth noting. You can't change the shape of a class, for example, by adding or deleting methods or fields. Also, changing variable references in an inner class can cause the compiler to generate synthetic methods in the outer class, changing its shape. You need to use the `*.class` files that the Java compiler creates; using those generated by `javac` is not supported. Some JVMs can't replace the bottom (`main` method) stack frame if you change it or stack frames above a native method. In this case, you'll need to step into a method first before you can change code and have the JVM replace the stack frame. There are limitations on the kinds of code changes you can make, for example, changes in and around `try/catch` blocks. Some JVMs may require command line parameters; refer to the documentation for the JVM. If you're having difficulty getting hot code replace to work do the following.

- Ensure the JVM you're using supports hot code replace.
- Use the `-showversion` JVM command line parameter to verify the

JVM running in the debugging session is the correct one.
- Try a simple change (not in the `main` method), like changing a string literal or value of an `int`.

Remote Debugging

The Java Debugger allows you to debug programs remotely, including using hot code replace with JVMs that support it. Remote programs include those running on your local machine outside of Eclipse and programs on another machine, either standalone or in a server environment. In order to display a source and enable you to set breakpoints, the Debugger gets the source for the code being debugged from your workspace.

Debugging a remote program is similar to debugging a local one, except that the remote program must be launched first with certain JVM parameters, and you need to specify a Remote Java Application launch configuration. When you launch your remote program with the parameters specified below, it starts and then suspends waiting for a connection from a debugger. The launch configuration contains information for the JDT Debugger to connect to the JVM running the remote program and initiate a debugging session.

To debug a program remotely, you need to start the remote program with the following JVM command line parameters (this should appear on one line).

```
-Xdebug -Xnoagent
        -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000
        -Djava.compiler=NONE
```

The `-Xdebug` parameter enables debugging, `-Xnoagent` disables support for `oldjdb` (Old Java Debugger), `-Xrunjdwp:` specifies Java Debug Wire Protocol (JDWP) options for the connection, and `8000` is the port the Debugger will use to communicate with the JVM. Remember this, as you'll need it to specify it in your launch configuration. The `server=y` parameter tells the remote JVM not to attempt to connect to the Debugger after the VM starts, but rather to wait for an inbound connection. The `suspend=y` parameter causes the JVM to start but suspend execution before the `main` method is loaded, and then wait until a connection is received from the Debugger. The `-Djava.compiler=NONE` parameter disables the Just in Time (JIT) compiler.

To define a Remote Java Application launch configuration, select **Debug > Debug...** to see the Launch Configurations dialog. Select **Remote Java Application** as the type of launch configuration and then select **New**. On the **Connect** page, **Project** specifies the Java project containing the source for the remote program you are debugging. In **Connection Properties**, specify the

same port number that you used in the JDWP options (`address=8000`) to launch the remote Java program. For programs running on the same machine, use `localhost` for **Host**. Specifying **Allow termination of remote VM** allows you to terminate the remote program from the debugging session.

When you have your JVM command line parameters set and your launch configuration defined, first start the remote Java program (with the command line parameters) and then start a debugging session using the launch configuration. A debugging session starts on the remote program. At this point, continue debugging as if you were debugging a local program.

If you are unable to get a remote debugging session started, try the following.

- Run your program without the additional JVM command line parameters to ensure the program will run correctly without having the JVM attempt to configure for remote debugging.
- Check communication between the machines. Ping the remote machine from the local machine using the hostname or IP address you specified in the launch configuration.
- Ensure the port numbers in the launch configuration and in your JVM command line parameters match.
- Ensure that the `dll`s the JVM requires (e.g., `jdwp.dll` and `dt_socket.dll`) on the remote machine are on the search path, for example, in the `PATH` variable. Ensure the `dll`s that the JVM is finding are the correct ones, for example, if there are multiple JREs in the search path of the remote machine.

If all else fails, search the Eclipse newsgroup archives for others who have had difficulty getting remote debugging working and then get on the newsgroup; the community is a helpful one.

Exercise Summary

Chapter 30 has an exercise that reinforces much of the material presented in this chapter. The exercise is broken down into a number of sections, each demonstrating different debugging concepts.

1. Debugging

   You'll start with the basics and then hit all the mainline debugging functions you'll need to debug Java. You'll see different kinds of breakpoints, controlling program execution, examining and changing variable values, and evaluating expressions.

2. Debugging Threads

   In this part, you'll debug a multi-threaded program and see watchpoints and method breakpoints on a class in a JAR file for which you do not have the source.

3. Remote Debugging

   In the final part of the exercise, you'll see how to debug a program running outside of Eclipse.

Chapter Summary

The Java Development Tools (JDT) provide a wide range of Java debugging capability. This chapter described the different kinds of breakpoints and how to define them, enable and disable them, and set their properties, including hit counts and conditions. It discussed how to control program execution through the debugger step commands and how to examine and change variable and field values. It looked at launch configurations again and how to use them to start debugging sessions, including ones on remote programs. It outlined how to debug a program remotely, including the command line parameters required.