

Debugging

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

—Brian W. Kernighan

Debugging in the Java Development Tooling Environment

No matter how hard you try to make your code bullet-proof, at some point you will find yourself confronted with a bug that needs to be examined under the microscope. The Eclipse debug environment gives you the tools you need to examine and exterminate bugs as they surface.

Of course, out of the box, Eclipse only gives you enough debugging capabilities as plain-old Java objects can handle. If you want to debug servlets, you can always use a remote agent (which will be discussed at this end of this chapter), but you are much better off getting a plug-in such as Sysdeo or MyEclipse that gives you native app server support. Also, when you look at the remote agent, it will be in the context of a standalone program, not in terms of connecting to an app server.

On the subject of debugging, much can be said, but the less the better. The methodology of Test-Driven Development is mentioned in various places in this book without going into it in any real depth (you can always read Chapter 6, “High-grade Testing Using JUnit”), but take the opening quote to heart: If you find yourself spending a great deal of time in front of your debugger, perhaps you need to write more, or better, tests. Of course, when you reach the point where all else fails, there is always the all-purpose `println()`.



3

IN THIS CHAPTER

- ▶ Debugging in the Java Development Tooling Environment 57
- ▶ The Debug Perspective 58
- ▶ Debugging Standalone Java Code 61
- ▶ Remote Debugging 66

Debugging in Eclipse happens within the Debug perspective. You can open the Debug perspective in all the usual ways:

- From the main menu, select Window, Open Perspective, Debug.
- From the main menu, select Window, Open Perspective, Other and then select Debug from the Select Perspective dialog.
- From the main menu, select Run, Debug As, and then select the program type to run after opening the Debug perspective.
- From the main menu, select Run, Debug to open the Launcher dialog. Once you have created or modified a run configuration, click Debug to run your program after opening the Debug perspective.
- From the toolbar, click the bug icon and Eclipse will start the last program you were debugging after opening the Debug perspective.
- From the toolbar, click the arrow to the right of the bug icon and select the program to be run after opening the Debug perspective.

As usual, the most important thing to recognize is which of the many ways available to accomplish a task is the most comfortable for you.

Let's look at the Debug perspective and then look at debugging standalone Java code, plugins, and server-side Java code as well as remote debugging.

The Debug Perspective

The default Debug perspective is made up of the following views:

- **Debug**—This view is where you track and control the execution of your code (see Figure 3.1). You can have more than one program running at a time, and you can open or close the list of associated threads by double-clicking the selected program. Standard debugging functionality is also available through the buttons on the Debug view toolbar—you can pause, resume, kill, and disconnect processes as well as clear the view of any terminated processes. On any given line of code, you can also go into any given method or constructor (Step Into), execute the current line (Step Over), or leave the current execution point and return to the caller (Step Return). In addition, the Step With Filters/Step Debug feature allows you to step into a method, but skip any code that meets the filter criteria (for example, anything defined within a particular package could be skipped).

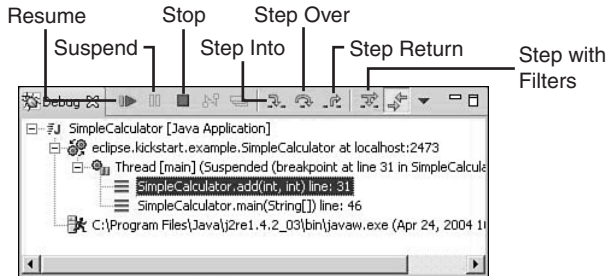


FIGURE 3.1 The Debug view displaying the running threads, the stack leading to the program's current location, and the current suspended thread.

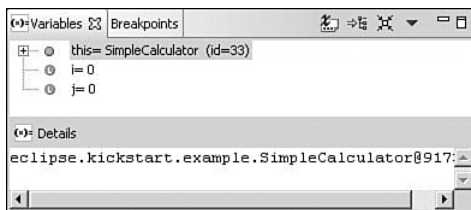


FIGURE 3.2 The Variables view displaying the running threads, the stack leading to the program's current location, and the current suspended thread.

- **Variables**—The Variables view displays the object and local variables available at the current position in the execution stack (see Figure 3.2). The top half of the view lists the variables, whereas the lower half displays the value of the selected variable. The menu bar for this view has five selections:
 - Display the type names next to the variable names
 - Display the logical structure
 - Select from one of the logical structures
 - Collapse an expanded node
 - A menu that allows you to change the orientation of the variable list to the Details window, turn Word Wrap on and off, display or hide constants, static variables, full package names, null array entries, and select how to display primitive data values (hex, ASCII, or unsigned)
- **Breakpoints**—The Breakpoints view allows you to enable, disable, or remove listed breakpoints (one at a time or all at once). If you double-click a breakpoint, the file in which it is located will open to the location of the breakpoint. You can also set values such as the breakpoint hit count and display the breakpoint's properties by right-clicking the selected breakpoint.
- **Expressions**—The Expressions view is similar to the Variables view except that you can arbitrarily add a variable or an expression for the debugger to display. The menu items available are the same as for the Variables view. This view is most valuable in displaying a variable that is changed by different levels of code.

- **Display**—The Display view can be thought of as an extension of the Scrapbook page, only the results are displayed in the same view and are resolved within the context of the current stack frame. You can either cut and paste code from the current editor and execute, display, or inspect it (where inspecting the code opens the Expressions view) or you can type in code and then select and execute it.

The Variables, Breakpoints, and Expressions views are stacked in the top-right corner of the Debug perspective by default, but the Expressions view does not appear unless you select Window, Show View from the main menu. The Display view also does not appear unless you select it but rather opens in the lower part of the perspective. You are free to move or resize these views in any way you like. Three of the remaining views you have already seen and used (Outline, Console, and Tasks), leaving only the Progress view:

- **Progress**—This view displays JVM activity while the process is running. Every line of code causes some activity in Eclipse as well as within itself.

The Debug perspective also contains an area reserved for editors. When you select an entry in the stack frame, the editor will jump to the location specified at the breakpoint in the correct level of the stack, if the code is available.

Using Debug Views Outside of the Debug Perspective

Eclipse makes no distinction between a view and the perspective in which it is displayed. Views are views: You can open other views in the Debug perspective (for example, the Package Explorer view), and you can open debug views (for example, the Variables view) in other perspectives. However, where other views could be helpful to have in Debug, the reverse is generally not true: Debug views are not of much value outside of the Debug perspective. Displaying the Debug view's current stack trace in the Java perspective or Registers in the Resource view does not help you to accomplish the task of debugging in an easier way.

SHOP TALK

Debugging as a Complement to Testing

To debug, or not to debug; that is the question. Bug-hunting can be a very satisfying part of the development process because it allows you to focus on a very specific problem-solving exercise, but it can also sometimes cause you to fail to see the forest for the trees. At this stage in the evolution of various development processes, it is easy to forget that without a good test suite in place, you will spend more and more of your time tracking down bugs than writing code. Try to think of debugging as a development tool to keep your tests working and not as a tool to keep your code working. Once you bypass the testing step, no amount of debugging is going to save you. Always have a test written to prove that a feature works as advertised, and use the debugger to help you figure out why a straightforward-looking piece of code is failing.

Without doing anything fancy, let's walk through debugging `SimpleCalculator` using as many of the debugger features as possible. Using the debugger in most other situations is similar, with the exception of necessary setup. For example, debugging server-side code assumes the server can either run within Eclipse or is available for a remote connection.

Debugging Standalone Java Code

Start Eclipse if it is not already running. Create a new project, a Java class, and a JUnit test to go with it. If you created the Calculator project from Chapter 2, "Writing and Running a Java Application," then simply refer to that project for the next few pages. If you have not created the Calculator project, perform the following steps:

1. Press `Ctrl+N` and from the New dialog select Java Project and click Next. In the Project Name field enter "Calculator".
2. Leave the Location section set to its default workspace location.
3. In the Project Layout section, select Create Separate Source and Output Folders and click Configure Defaults. When the Preferences dialog opens, click the Folders button. Leave the Source Folder Name set to `src`, but change the Output Folder Name to `classes`. Click OK.
4. Click Finish.

Now you need to create two classes: the class to be run and the class that tests it. In order to create them, perform the following steps:

1. Press `Ctrl+N` to open the New dialog. If only two selections appear, put a check next to Show All Wizards toward the lower-left corner of the dialog. When the remaining resource wizards appear, select Java, Class and click Next.
2. On the Java Class page, enter a Package of `eclipse.kickstart.example` and Class Name of `SimpleCalculator`. Click Finish.
3. Select `SimpleCalculator` in the Package Explorer view.
4. Press `Ctrl+N` again to open the New dialog. Select Java, JUnit, JUnit Test Case and click Next.
5. On the JUnit Test Case page, all the test case information should be filled in. If it is not, click Cancel, select `SimpleCalculator`, and return to step 4. If the JUnit Test Case page is filled in, check `setUp()` and `tearDown()` and click Finish.

If you did not implement the `SimpleCalculator` class from Chapter 2, open the `SimpleCalculator` class and add the following code:

```
package eclipse.kickstart.example;

import java.text.DecimalFormat;
```

```
public class SimpleCalculator implements Calculator {

    private static final String USAGE =
    ↪ "java eclipse.kickstart.example.SimpleCalculator val1 val2";

    public int add(int i, int j) {
        return I + j;
    }

}
```

If you did not implement `SimpleCalculatorTest` in Chapter 2, open the `SimpleCalculatorTest` class and add the following methods (if you already implemented this class then just add the code in bold):

```
package eclipse.kickstart.example;

import junit.framework.TestCase;

public class SimpleCalculatorTest extends TestCase {

    private Calculator _calculator;

    public void testAdd() {
        int expected = 0;
        int actual = _calculator.add(0, 0);
        assertEquals(expected, actual);

        expected = 1;
        actual = _calculator.add(0, 1);
        assertEquals(expected, actual);

        try {
            actual = _calculator.add(Integer.MAX_VALUE, 1);
            fail("add() passed on an overflow value.");
        } catch (RuntimeException e) {
            // If we come here then an exception was
            // thrown and the test passed.
        }
    }

    protected void setUp() throws Exception {
        _calculator = new SimpleCalculator();
    }
}
```

```

        protected void tearDown() throws Exception {
            _calculator = null;
        }
    }
}

```

The code in the try/catch block is checking if `SimpleCalculator`'s `add()` method can handle an integer value overflow. The expected behavior is for the method to throw an exception if the value is too large to contain in an `int`, and the caller then needs to do something about it. If the test passes, you should get a green bar in the JUnit GUI. (It will fail, of course. Make the code changes, select `SimpleCalculatorTest`, and then select Run, Run As, JUnit Test. Red bar!)

Let's set a simple breakpoint in `SimpleCalculator`'s `add()` method at the line that reads `return i + j`. Double-click in the editor margin to the far left, just past the line numbers. A blue dot appears. Press F11 or select from the main menu Run, Debug As, JUnit Test. The debugger stops the code in `add()` and displays the various stack frames in the Debug view. Select the stack frame for `SimpleCalculator.testAdd()` (see Figure 3.3). The selected code is for the first of the three tests run in `testAdd()`. Because you want the debugger to stop on the third test, you have two choices on how to proceed: You can either set the breakpoint in the `testAdd()` method of `SimpleCalculatorTest` or you can modify the breakpoint to become activated based on a condition. For the purposes of this example, you are going to do the latter.

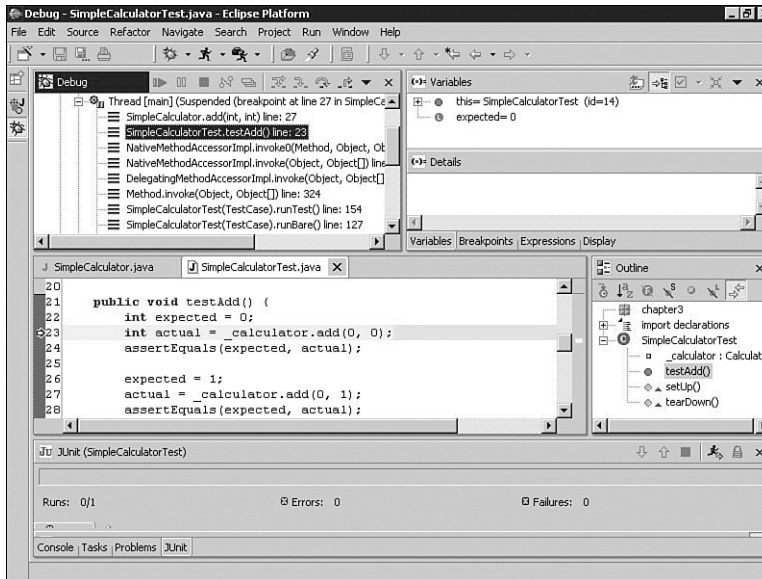


FIGURE 3.3 The Debug perspective with the stack frame for `SimpleCalculatorTest` selected.

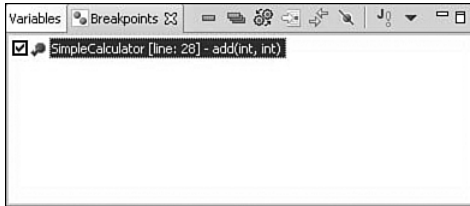


FIGURE 3.4 The Breakpoints view displaying the single available breakpoint.

Go to the Breakpoints view in the upper-right corner (see Figure 3.4). If it is not visible, click the Breakpoints tab. Double-click the breakpoint in the Breakpoints view to bring the editor forward with the breakpoint line selected. Because you are debugging the code and you are at the particular line, you will also see an arrow pointing toward the line in question. Right-click where the blue dot is to open a pop-up menu and then select the Breakpoint Properties menu item. This menu item

will open a Properties dialog (see Figure 3.5) for this breakpoint, allowing you to do the following:

- Disable the breakpoint.
- Set the breakpoint to be called only after it has been accessed a certain number of times.
- Set the breakpoint to halt execution when a condition has been met.

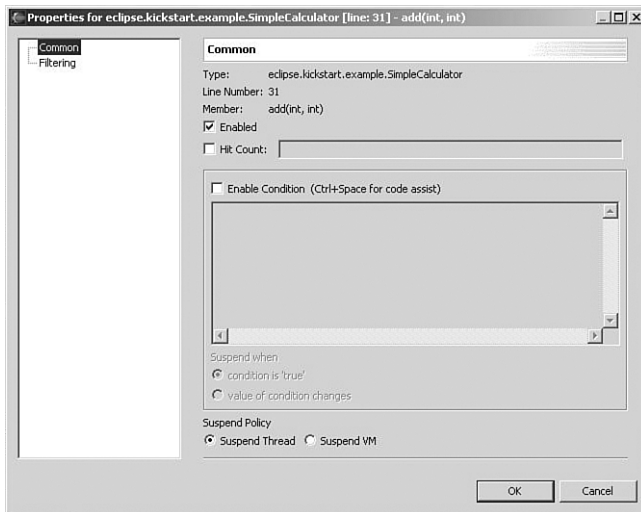


FIGURE 3.5 The Breakpoint Properties dialog.

Let's try all three. Uncheck Enabled and click OK. To resume execution, go to the Debug view and click the right-pointing triangle or just press F8. The breakpoint, now disabled, was ignored, and the test completed with the red bar.

Reenable the breakpoint by right-clicking the breakpoint and, when the Properties dialog opens, checking Enabled. Now check Hit Count and enter 3 into its input field. Click OK. Press F11 to restart the debug session.

Now, when the breakpoint is hit in the `add()` method, it will only stop the third time the breakpoint is accessed. To prove it, select the stack frame just below the breakpoint frame (the one for `SimpleCalculatorTest.testAdd()`). Instead of stopping at the first call to `add()` in `testAdd()`, it has stopped at the third call. This is useful when you have a bug that only surfaces when a piece of code executes a certain number of times. Press F8 to resume and complete the test run (still a red bar!).

What about those situations where you need to stop only after a particular condition is met? For example, when the value of a variable enters a certain range, you would like the breakpoint to activate. Once again, right-click the breakpoint and open the Breakpoint Properties dialog. Uncheck Hit Count and check Enable Condition. In the text area, enter `i == Integer.MAX_VALUE`. Code Assist is active in this text area, so feel free to use it in completing `Integer` and `MAX_VALUE`. Click OK and press F11 to restart the debugging session. Click the Variables tab to bring the Variables view forward. The variable `i` is set to 2147483647, which, not coincidentally, equals `Integer.MAX_VALUE`. Just to double-check, click the stack frame for `SimpleCalculatorTest.testAdd()` again. The breakpoint stopped at the third test once again. Press F8 to complete the test run.

In case you are tired of seeing the red bar, add the following code to `SimpleCalculator`'s `add()` method:

```
public int add(int i, int j) {
    int result = 0;

    long op1 = i;
    long op2 = j;
    long longResult = op1 + op2;
    if (longResult > Integer.MAX_VALUE) {
        throw new RuntimeException(
            "The addition of the values would result in an overflow.");
    } else {
        result = (int) longResult;
    }
    return result;
}
```

Let's look at the Expressions view before we move away from the `SimpleCalculator` example. Set a breakpoint at the first line of the `add()` code and another at the `if` statement. Press F11 to restart the debug session. When the breakpoint stops at the top of `add()`, select the variable `longResult`, right-click to open the pop-up menu, and select Watch. Because `longResult` is a local variable, the debugger complains that it cannot evaluate the expression. Press F8 to resume execution to the next breakpoint. Now that `longResult` is declared, the Expressions view shows the variable set to 0 (which is correct because the first JUnit test checked that `0 + 0` equals 0). Press F8 again, and the Expressions view once again complains that `longResult` has encountered evaluation errors. Press F8 again, and now `longResult` equals 1.

Continue pressing F8 until the program completes. If you want to clear the Debug view of all the terminated processes you created, click the fifth icon from the left in the menu bar of the Debug view (the one that looks like a stack of rectangles). To remove the breakpoints, click the Breakpoints tab, right-click in the Breakpoints view, and select Remove All from the pop-up menu.

Close all the editors.

Debugging Server-Side Java

Server-side debugging of Java code is not supported in Eclipse, but it can be added through additional plug-ins. The section titled “Debugging Servlets, JSPs, and EJBs” of Chapter 9, “J2EE and the MyEclipse Plug-In,” covers J2EE development using the MyEclipse plug-in, including the installation of the plug-in, how to set up the configuration of a server, implementing a servlet and JSP, and debugging server-side components within Eclipse.

Once the plug-in J2EE support is installed and the server configuration is entered, debugging proceeds as before. Set breakpoints, examine variables, edit files, and observe the various views in the same way you would for a standalone.

Remote Debugging

The debugging of an external process from within Eclipse is supported through remote debugging. The process can be running on the same machine as the debugger or a separate machine on the network. Unfortunately, the VMs of various vendors do not all turn on their debug-listening capabilities in the same way. The following example uses the standard Java VM and its documented command-line options.

The first step is to ensure that the compiler has debugging turned on. Open the Eclipse Preferences dialog and go to the Java compiler page (Window, Preferences, Java, Compiler). Click the Compliance and Classfiles tab and look at the settings for Classfile Generation. For this example, make sure all the boxes are checked, although in general, Preserve unused local variables can be unchecked (see Figure 3.6). Clicking OK will force a rebuild of the current projects.

In the Calculator project, create a new class named `CalculatorInput`. This class will read a string of digits from the command line, add them up, and print the result of the addition. All the generated comments have been removed:

```
public class CalculatorInput {  
  
    private BufferedReader _in;  
    private SimpleCalculator _calculator;  
    public CalculatorInput() {
```

```
InputStreamReader reader = new InputStreamReader(System.in);
_in = new BufferedReader(reader);

_calculator = new SimpleCalculator();
}
public static void main(String[] args) {
    CalculatorInput app = new CalculatorInput();
    try {
        app.start();
    } catch (IOException e) {
        System.out.println("Exception found: " + e.getMessage());
    }
}

public void start() throws IOException {
    String calcStr = null;

    System.out.println(
        "Enter a space-separated list of numbers to add (q to exit):");
    calcStr = _in.readLine();
    int [] value = null;
    int result;
    while(calcStr.equals("q") == false) {
        result = 0;
        value = parseValues(calcStr);
        for (int i = 0; i < value.length; i++) {
            result = _calculator.add(value[i], result);
        }

        System.out.println("Result: " + result);
        calcStr = _in.readLine();
    }
}

private int[] parseValues(String calcStr) {
    int [] result = null;

    Vector v = new Vector();
    StringTokenizer tokenizer = new StringTokenizer(calcStr);
    while (tokenizer.hasMoreTokens()) {
        String strVal = (String) tokenizer.nextToken();
        v.add(strVal);
    }

    result = new int[v.size()];
}
```

```

    for (int i = 0; i < result.length; i++) {
        result[i] = Integer.parseInt((String) v.get(i));
    }
    return result;
}
}
}

```

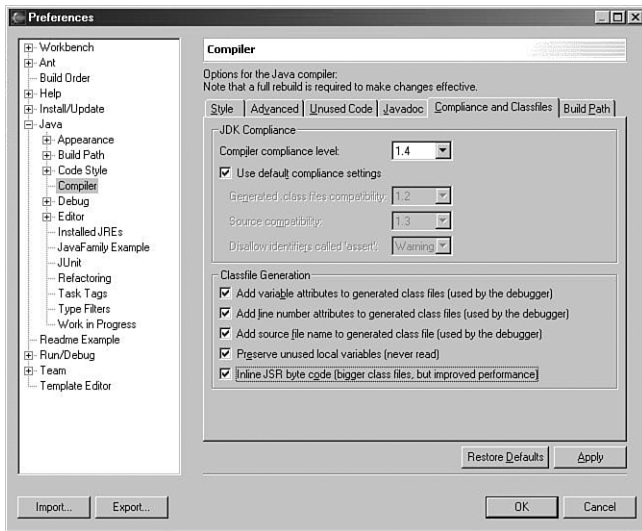


FIGURE 3.6 The Preferences dialog with all classfile generation options checked.

Do not worry about the missing package imports. When you have typed the preceding code into Eclipse, you can press `Ctrl+Shift+O` and all the missing imports will be added to the file. Save `CalculatorInput`.

Export the `Calculator` project by right-clicking the project name and selecting `Export` from the pop-up menu. When the `Export` dialog opens, select `JAR file` as the export destination. Click `Next`. In the list to the right of the project, uncheck `.classpath`, `.project`, and `calculator.jpge`. Exporting them will not cause any damage, but they are not needed. In the `Select Export Destination` field, enter `c:\calc.jar` or any safe location where you can find the file later, but still name the `JAR` file `calc.jar`. Click `Next`. The `JAR Packaging Options` page can be left alone, so click `Next`. In the `JAR Manifest Specification` page, click the `Browse` button, located toward the bottom of the page, and select `CalculatorInput` from the `Select Main Class` dialog. Click `OK`. The new class, `eclipse.kickstart.example.CalculatorInput`, is now assumed to be the class to be run when you run the `JAR` file from the command line. Click `Finish` to complete the export process.

Open a command-line window and change directory to wherever it is you saved `calc.jar`. If you are running on Windows, the easiest location to run this from is the `C` drive, which is

why you entered `c:\calc.jar` as the target export directory. Make sure you can run Java from the command line. If not, update your path variable to include Java's bin directory so that you can run `CalculatorInput` from the command line.

Enter the following on the command line:

```
C:\> java -Xdebug -Xrunjwp:transport=dt_socket,address=8000,suspend=n,server=y
➤ -jar calc.jar
```

All this must be on one line. Press Enter, and the program prompts you to enter a string of space-separated numbers, or you can enter the letter *q* to exit. If the program does not prompt you, check that you have entered everything as listed here.

The JVM -X Options

If you are using the standard Javasoft-supplied JVM, here is a caveat on the preceding command-line listing: The `-X` command-line option for the JVM defines options that may go away someday. The two options have been around for a few years, but there are no guarantees they will stay that way. Always refer to your vendor documentation to discover what options are available to turn on remote debugging.

The command-line options are as follows:

- `-Xdebug`—Notifies the VM that an external process may connect to it.
- `-Xrunjwp:`—Contains the list of comma-separated configurations needed by the VM to allow an external process to connect to it:
 - `transport=dt_socket`—The VM can expect the external process to connect via a socket.
 - `address=8000`—The external process will connect using this port.
 - `suspend=n`—The VM should not suspend the program while it waits for the external program to attach.
 - `server=y`—The VM should behave like a server.

Enter the string `0 1` and press Enter. The program will display `Result: 1`. Let's attach the Eclipse debugger to this process.

To attach the Eclipse debugger to an external Java program, it has to be told which machine the external program is running on and what port it can use to communicate with it. This information is entered through the Launcher dialog. Select from the main menu `Run, Debug` to open the Launcher dialog specific to debugging. Select `Remote Java Application` as the

configuration type and click New. This configuration information will be used by the debugger to connect to the external program. Enter the following configuration information:

- Name: Remote Calculator
- Project: Calculator
- Connection Type: Standard (Socket Attach)
- Connection Properties: Host: localhost
- Connection Properties: Port: 8000

Click Debug to start up the debugger and have it attach to `CalculatorInput`. The Debug perspective will open and the Debug view will display the Calculator as running in a Java HotSpot Client VM. Right-click the third line of the stack frame (`Thread [main] (Running)`) and select Suspend from the pop-up menu. The external VM will suspend the named thread “main” and the debugger displays what the current stack looks like (see Figure 3.7). The second-to-last line of the stack is the call from `CalculatorInput.start()`. Select this line and the `CalculatorInput` file will open in the editor at the proper location.

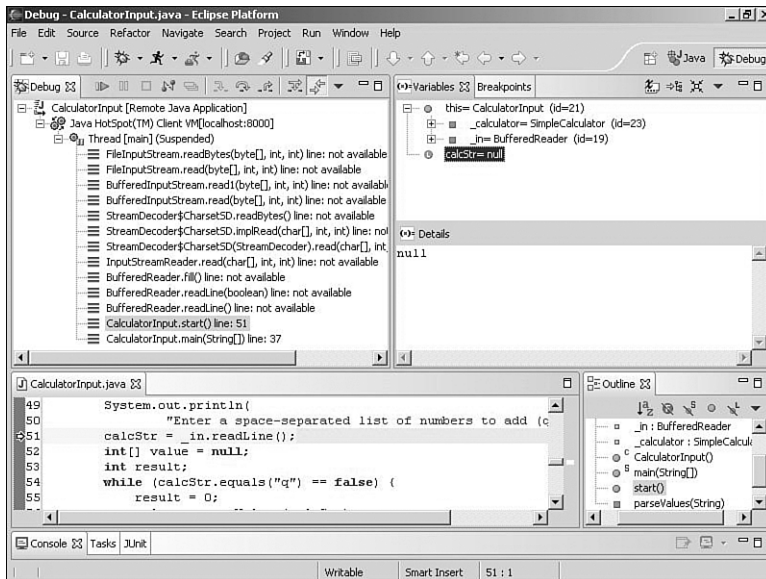


FIGURE 3.7 The debugger with the Debug view showing the stack trace for the remote `CalculatorInput` program. The editor is open to the line at which the process is suspended.

The Variables view displays the `this` and `calcStr` references as set in the external process. Set a breakpoint at the first line inside the `while` loop and press F8 to resume the thread. Bring the command-line window forward, enter the string `"2 2"` and press Enter. The debugger will suspend execution of `CalculatorInput` and display the line within the code where the breakpoint was hit. The Variables view displays three variables: `this`, `calcStr`, and `value`, which was created after the call to `in.readLine()`.

From within the Variables view, double-click `calcStr` and change its value from `"2 2"` to `"5 5"`. When the Set Variable Value dialog opens, do not enter quotes around the values. Click OK and then press F8 to resume execution. The result of the addition of `"2 2"` is now 10. From the command line window, type `q` to end the `CalculatorInput` session.

Once the debugger has connected to the external process, you can use many of the standard features of the debugger, but remember that you cannot do things like hot-swap code. If you change the code in the debugger while you are debugging, the debugger will flag the code as out of sync with the running code and will not allow you to use the file for the debugging session.

In Brief

The Eclipse debugger can examine Java code locally or remotely. In Chapter 13, "The Eclipse Plug-in Architecture," you will find information about debugging plug-ins. Here are the salient points about out-of-the-box Eclipse debugging:

- The Debug perspective presents the basic grouping of views to help you debug your code. You can use a number of additional views by selecting Window, Show View, Other and looking in the Debug category.
- The debugging of standalone Java code is quite straightforward and complements JUnit testing. Standard features such as Step Into, Step Over, and Resume are all supported. Use Step Filter tells the debugger what code it can safely skip when it is stepping into code during a debug session.
- Remote debugging is activated by running the process to be debugged using command-line options that make the JVM listen on a socket for external commands. Once the Eclipse debugger is connected to the external process, it becomes just another debug target with most of the available features of the debugger.

