CHAPTER 6

Debugging the Code

Debugging is anticipated with distaste,
performed with reluctance,
and bragged about forever.

—Anonymous

he concept of software testing is broader than just debugging your programs. You need to think critically about testing long before you get to the point of developing and implementing your applications. This type of awareness and planning will go a long way toward ensuring that your applications can be brought quickly to production quality.

Finding design and coding defects early in the development lifecycle can result in huge benefits. Perhaps you will only need to spend hours correcting code errors that are discovered early in the process, rather than the days to weeks otherwise required to retrofit code errors discovered after a product is deployed or shipped to a customer.

In other words, testing, quality assurance, and other configuration-management methods should pervade your system development process from start to finish. These measures will help you to ensure that the highest quality software product is delivered using Oracle9i JDeveloper.

It is not the intent of this chapter to discuss the topic of software quality in general. Rather, this chapter discusses some of the techniques that can be easily used within JDeveloper to help you unit-test Java code as it is being written. The chapter explains how the major features of JDeveloper's native debugger handle the debugging process and the essential debugging activities—program control and value checking. It also provides an overview of the Profiler that you can use to understand low-level details about the code as it is executing. The chapter concludes with a hands-on practice that applies these concepts to correcting a simple logic error.

Most of the techniques in this chapter apply equally to debugging a Java application, applet, EJB, servlet, or JSP application that is running in the client machine's Java Virtual Machine (JVM). When you deploy and run JSP, servlet, or EJB code, you may need to perform additional debugging on the server. Although this chapter does not explain remote debugging in detail, the section called "Remote Debugging" explains where to find more information on the subject of debugging code on the server.

Overview

Debugging is the process of locating and fixing errors in software programs. Most developers would readily agree that debugging source code, regardless of the language, is tedious and time-consuming. The objective of debugging is to find and fix the problem as quickly and easily as possible. A number of ways exist to debug a program, many of which do not involve a dedicated debugging tool. However, Java includes a full-featured debugger that helps you to find problems in the code. JDeveloper provides a graphical interface for this debugger and adds many features to assist in the debugging process.

Before explaining how debugging works in a JDeveloper environment, it is useful to review the types of program errors and the activities required in the debugging process to understand where the errors occur. This discussion will provide a context for the description of debugging in JDeveloper.

Types of Program Errors

There are four general types of program errors that you will run into:

- **Syntax errors** These are the result of incorrectly formed statements. They are easily detected by the JDeveloper compiler, which displays errors in the Log window. You can just double click an error in this window, and the Code Editor will highlight the problem line of code.
- **Data condition errors** These are a result of inputting or passing a value to the program that causes the program to abort, such as a divide-by-zero error. The error is reported by the Java runtime system (JVM). Data condition errors are more difficult and challenging than syntax errors because testing may miss the data value that causes the error. Developer experience and better program design can eliminate these errors. The watch feature of the debugger assists in fixing this type of bug.
- **Logic errors** This type of error also shows up only at runtime. A logic error is a result of poor design or of coding mistakes and often manifests itself if the program does something that it is not intended to do. This type of error is the most difficult to catch because it may not occur consistently or early in the development process. In fact, these errors can occur for the first time long after a program is released for production use. The best guard against logic errors is a thorough test plan and testing cycle. Once you experience this type of error, the debugger will help you locate the problem by allowing you to step through the code and to set conditional breakpoints.
- **Resource errors** This type of error occurs due to lack of a required resource such as network availability, server availability, memory, disk space, or other "physical" (nonprogrammatic) resource. In Oracle9i JDeveloper, you can use the Profiler (described later) to assist in solving resource error problems.

Correcting Syntax Errors

Of course, little advice for fixing a syntax error is necessary. Developers normally read the error text carefully and closely examine the code based on that text.

If an error indicates that a method that you call does not exist, check the spelling and use of upper- and lowercase because the Java language is case sensitive. If the spelling is correct, you should check to see that you are passing the correct argument types. The Javadoc for a method will indicate which arguments are expected and allowed. If the spelling and arguments are correct, you may be missing an import statement. You also need to be sure that the appropriate package is accessible by including it as a library in the Project Settings dialog (**Project | Project Settings** or select Project Settings from the right-click menu on a project node).

Other common errors that are easy to check for include the following:

- Mismatched curly brackets or parentheses
- Misuse of upper- and lowercase
- Argument datatype problems

- Unmatched or incorrect single or double quotation marks
- Missing semicolons at the end of an executable line

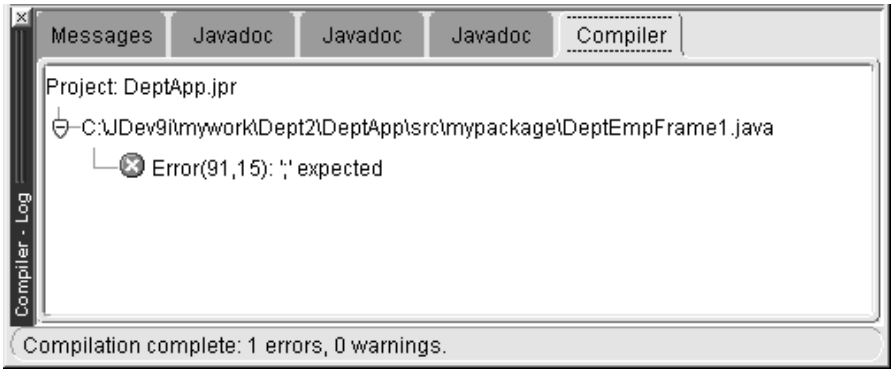
Some of these errors are visible in the Code Editor. For example, when you place the cursor next to a paired symbol such as a curly bracket or parenthesis, the editor highlights (in a color such as light blue) the nearest match as follows:

```
if (storeIndex != i) {
  numList[storeIndex] = numList[i];
  numList[i] = storeMax;
}
```

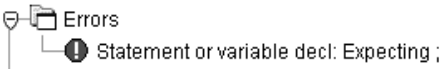
If you do not see the match that you were expecting, you can fix the mistake. The Code Editor also indicates a missing paired symbol using a red highlight in the same way as shown here for a missing close parenthesis:

```
catch(Exception ex)
{
  ex.printStackTrace(;
}
```

The Log window compiler messages will be of some help in identifying these errors as shown here for a missing semicolon:



Double clicking the error in the Log window highlights the line of code in the Code Editor. In addition, as you are typing in the Code Editor, you will see syntax errors such as missing semicolons in the Structure window as follows:



Double clicking the message in the Structure window will move the cursor to the problem line of code. If you have many missing semicolons or other punctuation, you may have to look above the problem line to see if the problem occurs earlier in the code. All of these Code Editor features allow you to check the syntax before incurring the overhead of compilation.

Other than mistakes due to a misunderstanding or misuse of the Java language, the errors mentioned will probably comprise 80 percent of the syntax errors that you experience. Since syntax errors must be corrected before you run the program, and since the debugger only works at runtime, this chapter will not discuss them further.

Correcting Data Condition and Logic Errors

The JDeveloper debugger assists in resolving runtime data condition and logic errors. This chapter focuses mainly on the debugging process that you can apply to these types of errors. One of the most pernicious errors for novice Java developers to watch out for results from the automatic casting of numeric datatypes. Be especially careful of strange rounding errors involving integer datatypes.

Another logic error is casting objects incorrectly. If an object is typed at a high level in the class hierarchy (for example, as an Object), it can be recast as anything under that level. Errors can occur when that object is cast to a type that does not match its use (for example, as a return value for a method). Using *typesafe methods*, which are checked for type upon compilation and avoid runtime type mismatch errors, and interfaces greatly reduces type mismatching problems. (Casting and interfaces are discussed further in Chapter 4.)

The best defense against logic errors is well-constructed, object-oriented code.

CAUTION

The compiler will not catch some logic errors that are a misuse of operators. For example, the expression "`if (isValid = false)`" will assign the value "false" to the boolean variable `isValid` instead of testing the value of `isValid` (as with "`if (isValid == false)`").

Correcting Resource Errors

Resource errors may be intermittent, and these are difficult problems to fix. If you can consistently reproduce the problem, the JDeveloper Profiler will help you to determine the point at which a program is failing. If you determine that there are no logic or data condition errors, you can then turn to external resources. Follow the process of elimination by substituting hardware or network resources and testing the problem. This should lead you to a solution. Sometimes such errors result from a recursive routine that is not terminating.

NOTE
The JDeveloper CodeCoach feature, described in Chapter 2, does not specifically address identifying and fixing bugs. However, CodeCoach does assist in writing more efficient code, which potentially uses fewer resources. Use of this tool could prevent resource errors from occurring.

Debugging Activities

The debugging process normally combines two interrelated activities: running and stepping through the code, and examining the values of variables, parameters, data, and array items. The process follows the steps and activities diagrammed in Figure 6-1. You normally test an application first in normal runtime mode to see if there are errors. If you find errors, you run the debugger and set up breakpoints to stop execution of the code. At those points you can stop the program execution and examine data values. When you are finished with those tasks, you exit the debugger.

Help with the Debugger

The JDeveloper help system topics that apply to debugging fall under the "Testing and Optimizing Application Code\Debugging in JDeveloper" node in the Contents page. Drill down through this node in the Contents page for applicable topics.

Do You Really Have to Run the Debugger?

The JDeveloper debugger is a powerful tool for finding and repairing defects in code that you create. As with many powerful features, there is a bit of complexity. You may not need to use the debugger for a problem that seems to be relatively simple. Instead, simple troubleshooting

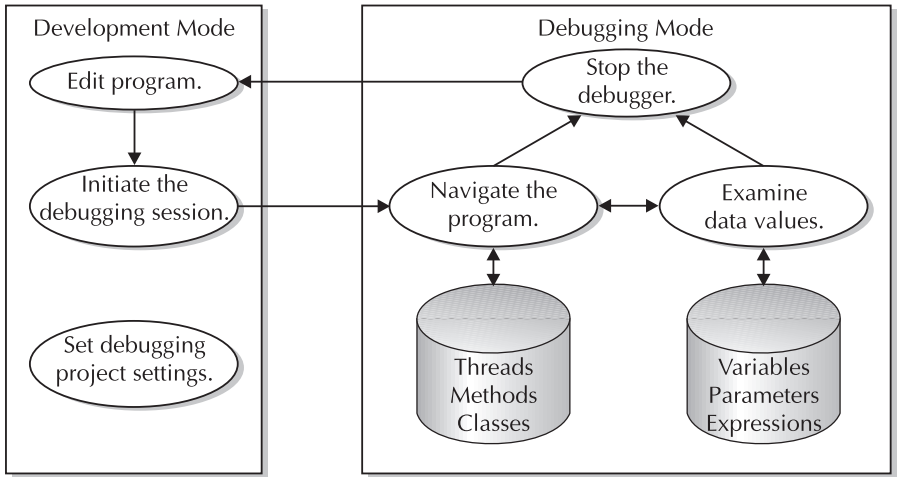


FIGURE 6-1. *A typical debugging session*

techniques may lead you to a solution without having to run the debugger. This section describes some of those techniques. They are your first line of defense in the battle against bugs. Several of the techniques address database connectivity because it is outside of normal Java work. Once you are certain that the database connection and BC4J interactions work, you can concentrate on debugging the application code.

Apply General Troubleshooting Techniques

The earlier section "Types of Program Errors" discusses common errors you would look for first. After looking for those errors, you can use the following techniques to troubleshoot the application:

- The first thing to do is to ensure that all projects have a current build. Click on each project node one at a time and click the Rebuild icon. Alternatively, you can select Rebuild Workspace from the right-click menu on a workspace node to build all projects in that workspace.
- Examine the Log window for compilation errors. The message text should indicate the line in which the error occurred. Double click the error message to navigate to the problem code. This should help you identify and fix code syntax problems.
- The curly bracket and parenthesis sets are highlighted, as mentioned in Chapter 2. If you place the cursor by one of a pair of symbols (such as the curly bracket), that symbol and its match will be highlighted. This allows you to check for properly formed code blocks and argument lists.
- For Java client programs (Java applications and applets), you can also double check that all properties and objects have been correctly defined by comparing the code with the properties and objects.
- For Java client applications, check that you have properly defined the hierarchy of objects by examining the Structure window nodes when the UI Editor is open. If something is out of place, you can select Cut from the right-click menu on that object. Then select the node that the object should appear under and select Paste from the right-click menu to position the object under that node.

Test and Edit the Connection

There are some tests that become part of the normal development process. For example, when you define a connection, you can test it to ensure that it accesses the database correctly. If you need to test it after the initial definition, use the following steps:

1. Select Edit from the right-click menu on a specific connections node (such as HR) in the Navigator window Connections node.
2. Select the Test tab and click the Test Connection button. A message will appear in the *Status* field to indicate the results of the test.

If there are errors, check and adjust the settings on the other pages of this dialog and retest the connection using the Test button. You may need to consult a database administrator to verify the proper host, port, and database (SID) name.

Test and Edit the BC4J View Object

Once you have defined view objects (as described in Part II of this book), you can test the query that each one represents using the following steps:

1. On the view object node under the package node in the BC4J project, select Edit <view object> from the right-click menu. The View Object dialog will open.
2. Click the Query tab and click the Test button. A message will appear at the top of the page indicating the state of the query.

If there are errors, modify the properties on the other pages so that the query is valid. You can click the Expert Node button to edit the query text if you need to test variations on the automatically generated query text. You need to repeat this test for each view object that does not work.

Test the BC4J Application Module

After defining a BC4J project, you can test its code using the Oracle Business Component Browser. Use the following steps:

1. Open the BC4J project node in the Navigation window and find the node that represents the application module. The application module is usually named with a suffix of "Module." For example, "DeptEmpModule."
2. On the application module node, select Test from the right-click menu. The Connect dialog will open. Verify the settings and click Connect. If another login dialog appears, enter the password and click OK. (If there is only one application module in the package, you can click the Run icon in the IDE toolbar to test the application module. This method bypasses the Connect dialog.)
3. The Oracle Business Component Browser window will appear. This window contains a navigator of view objects and a viewer. For testing purposes, you just need to double click a node for one of the view objects or select Show from the right-click menu on that node.

The view area will display an application that you can use to test the view object. The application contains a navigation bar that you can use to test the data-browsing features. If this application works, you can be certain that your business components are working. There are other features of this application that you can learn about by reading the help topic that appears when you select **Help | Contents**. Practices in Chapter 3 and Part II allow you to try out this browser.

Displaying Messages

A simple but effective technique that works in some situations is to temporarily add code to display a message while the program is running. The message can contain information about the method that is being run, and it can include variable values. This is usually useful only if you have a clear picture of the potential problem area and just need to quickly verify a value or ensure that the execution path reached a certain line of code. For other debugging situations, the debugger might be easier and more informative. The drawback with using messages is that you have to modify the code and strip out the messages when you are finished debugging.

There are two methods you can use when you want to display a message. Both can accomplish the main debugging objectives of program control and value checking.

NOTE
Some developers define a Boolean variable in the code and set it to "true" to turn on the messages. Each section of code that has a message tests the variable to see if the message should be displayed (for example, "`if (debug)System.out.println("message");`"). When the developer is finished with the messages, she or he sets the Boolean variable to "false" so that messages will be suppressed. If the variable is defined as private, final, and static (for example, "`public final static boolean debug = true`"), the compiler will strip out the code that is not executed, which makes the compiled class file smaller.

Console Window

You can embed a call to this method in key positions in your code. For example, if you wanted to determine the value of an `int` variable called `totalSalary`, you would add the following line in a Java application or applet:

```
System.out.println("total salary = " + totalSalary);
```

This would display the text "total salary = 999" (if the variable value were 999) in the Log window. In a JSP file, you would use the following line:

```
<% out.println("This is a debugging message"); %>
```

This will print the statement in the HTML page that the JSP page generates. `System.out.println` in a JSP scriptlet tag (`<% %>`) will print to the Java console on the server.

TIP
Left justify temporary print statements in your code so that they will be easier to find when you want to remove them.

NOTE
*The Project Settings dialog (**Project | Project Settings**) contains the nodes Configurations\Development\Compiler and Configurations\Development\Runner. These pages allow you to specify which messages appear in the Log window for files in that project during compile time and runtime, respectively.*

Message Dialog

For Java applications and applets, you can display a modal message dialog (message box) that contains text optionally concatenated with variable values. This allows you to stop and look at the dialog before the program continues. A message box is useful if you need to check the output

on the screen in the middle of an operation. The console window technique described earlier does not provide this capability.

The code that you write calls a static (shared) method in JOptionPane, a class from the Swing library. JOptionPane offers a number of features, including specifying more than one button and providing a text input area that the user can fill in. It is an easy way to show a dialog box. For debugging message purposes, you only need its simplest format, shown in this example:

```
JOptionPane.showMessageDialog(this,
  "The debugging message. maxNum = " + maxNum);
```

This code displays the JOptionPane dialog and shows the value of a variable set before the dialog is called. The following shows the output of this code. This technique results in showing the same kind of message you would show using the console window technique, but this technique stops the processing and waits for the user to click OK. Since instantiating the JOptionPane class using the new keyword uses a bit of memory each time you display the dialog, it is better to use the syntax shown and directly call the static method from JOptionPane.

This technique is not applicable to JSP applications because JSP pages run on the server and do not use Swing components such as JOptionPane. For extra functionality, use the Dialog class as described in the sidebar "Using the Dialog Class."

Using the Dialog Class

Although JOptionPane will suffice for most debugging message requirements, you may need more functionality and additional components. The Dialog class demonstrated in the following steps displays a window with a customized dialog:

1. On any workspace node, select New from the right-click menu.
2. Double click "Project Containing a New Application" in the General\Projects category, and name the project and its directory "TestDialogJA."
3. Specify the package name as "testdialog," and click the Finish button on the Paths page. The New Application dialog will open.
4. Leave the defaults and click OK to create the application file and display the New Frame dialog.
5. Click OK to create the frame file.
6. If the frame file is not already open in the UI Editor, on the Frame1.java node, select UI Editor from the right-click menu. (The Document Bar may already contain a UI Editor session for Frame1.java.)

7. Add a JButton component to the frame in the UI Editor from the Swing page of the Component Palette. Select the button and change the *text* property to "Test Dialog" using the Property Inspector. Resize the button so that the text is completely visible.
8. Click Rebuild and Save All. Run the Application1.java file to check the design. (Specify the run target as Application1 if a dialog appears while running the file.) Close the runtime window.

   While the practices in this book normally include clicking Rebuild to build the project, the Runtime node of the Project Settings is set by default to compile before running.
9. You will now create a dialog file. On the TestDialogJA project file node, select New from the right-click menu.
10. Double click Dialog in the Client Tier\Swing/AWTcategory to display the New Dialog dialog.
11. Name the dialog "TestDialog," and click OK to create the file.
12. Open the UI Editor for TestDialog.java if it is not already open. Change the *layout* property of "this" to "null" by clicking "this" in the Structure window and setting the *layout* property. Although you can place any component in this dialog panel, for this demonstration, just add a JTextField component from the Swing page of the Component Palette.
13. Switch to the frame file UI Editor. Click the Test Dialog button for the frame file, and click the Events tab in the Property Inspector.
14. Enter "TestDialogButton_clicked" in the *actionPerformed* event and press ENTER.
15. Control will switch to a method stub in the Code Editor. In a blank line between curly brackets, insert the following code:

```
TestDialog newDialog = new TestDialog();
newDialog.show();
```

16. Click Save All. Run the Application1.java file.
17. Click the Test Dialog button, and the dialog with the text field will display. Use the window closing buttons to close both windows.

This demonstrates how you can display a dialog window from an event such as a button press. Since the dialog can contain any component, you can build an input or selection area that will modify how the application works.

A variation on this technique is moving the instantiation (new) line to the class level (under `public class Frame1` in this example). This allows the dialog to be called from any method in that class. You can also instantiate the dialog with arguments (parent window, dialog title, and modal flag) as follows:

```
TestDialog newDialog = new TestDialog(this, "A test dialog", true);
```

If you set the *modal* property of the dialog's "this" window to "true," the window will be modal. (That is, it must be closed before focus can shift to the other window.)

The help system topic "Creating a Dialog Box" (called "dialog boxes creating" in the Index tab) contains more information about the Dialog class.

HTML Messages

For JSP applications and applet HTML startup files, you can embed simple text in the body section to show the HTML section that is being displayed. Simple message text embedded in the HTML body section can show that a particular tag was reached. You can also use hidden field values to show the parameter values that were passed to the page. (Consult your favorite HTML reference for details about hidden fields.) As a simple example, you might have a page containing the following HTML:

```
<HTML>
<BODY>
   <TABLE>

  . . .
   </TABLE>
after the table
</BODY>

</HTML>
```

The string "after the table" will be displayed if the HTML code successfully handles the HTML tags above it. You may not need to debug the HTML code if your HTML editor handles syntax errors automatically.

TIP

To help identify problems in running applets, browsers usually offer a way to display messages from the client JVM. In Netscape, you can select ***Communicator*** *|* ***Tools*** *|* ***Java Console*** *to show a window that will display messages about the session. In Internet Explorer, select* ***Tools*** *|* ***Internet Options*** *and click the "Java console enabled" checkbox in the Microsoft VM section of the Advanced tab. You can activate a similar window for the Java plug-in after you have started it in the browser by selecting Show Console from the right-click menu on the Java plug-in icon in the Windows task bar. Messages that you write to the console using System.out.println (for example, in a JSP application) will appear in this window.*

The JDeveloper Debugger

If you have determined that the alternative debugging techniques such as those just mentioned will not suffice, you will want to run a debugging session using the debugging mode. JDeveloper uses the Java 2 debugging API. It includes the ability to handle many JDK versions (version 1.2 and later), and to debug code on remote machines. Debugging is an essential activity that is worth the initial time investment to learn and set up. The more time you spend up front learning how to use the debugger, the easier it will be to respond to a problem program. The time to learn debugging is not when you encounter a problem in a high-pressure development situation.

Starting a Debugging Session

You start a debugging session from the IDE by clicking the Debug button, pressing SHIFT-F9, or selecting **Debug | Debug "<*project name*>"** from the menu (where "<*project name*>" is the name of the project, such as SortJA.jpr). The Debug "<project name>" button on the toolbar also starts a debugging session. You need to check the *Include Debug Information* checkbox on the Compiler page of the Project Settings dialog (**Project | Project Settings**). You also need to set the state of the "Set Start Debug Option" button in the debugging toolbar. The arrow beside this button displays a pulldown radio group that you can set to either Step Into (to stop at the first line of executable code), Step Over (to run through the code until reaching a method where tracing is enabled), or Run Until a Breakpoint Occurs (to run through code until the first breakpoint). The option setting is new as of release 9.0.3. The file compiles with special debugging information and runs in a modified JDeveloper window such as that shown in Figure 6-2.

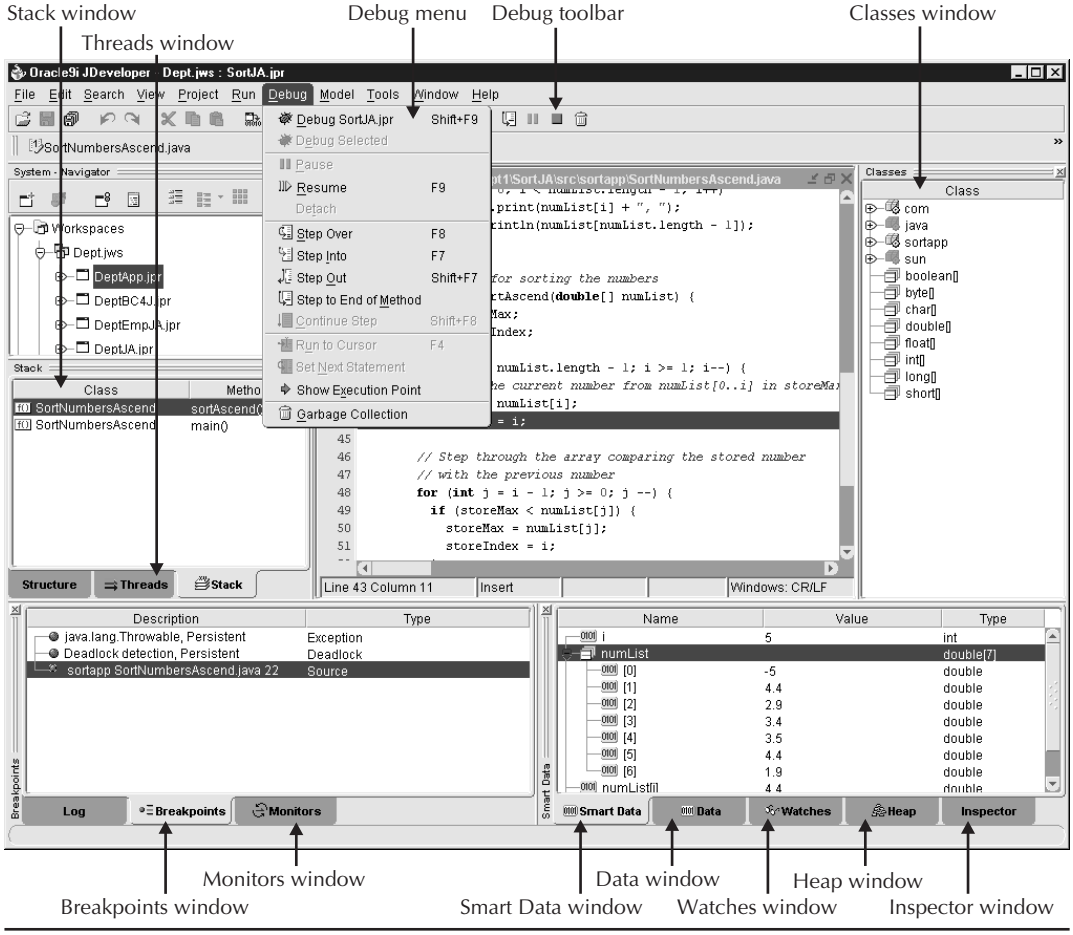


FIGURE 6-2. *JDeveloper running in debug mode*

NOTE
Another way to stop execution in the code is to start the debugger and click Pause to stop. The trick is to stop in the right place, but if the code execution is paused or stopped (such as when the program awaits user input), this is a reasonable technique to use.

A number of special debugging features appear when you run a file in debugging mode. Debugging menu options are enabled in the Debug menu.

Debug Windows

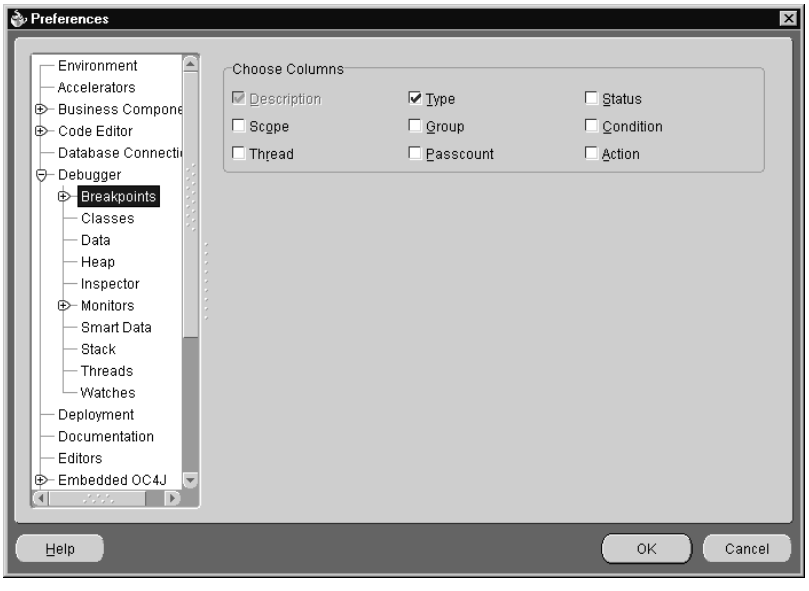
While in debugging mode, you can modify the default debugging session display using the **View | Debug Windows** submenu to display or hide the following windows:

- **Breakpoints** This window displays all breakpoints (program execution stopping points) that you have set or that are set by JDeveloper (such as exceptions and deadlocks). This window is viewable outside of debugging mode.
- **Threads** The Threads window shows all program execution lines. Since you can write multi-threaded programs in Java, you may need to examine the state of the current simultaneously executing threads.
- **Stack** This window shows the sequence of method calls that preceded the execution point (the *stack*).
- **Smart Data** This window shows variables, constants, and arguments that are used close to the execution point (the line of code that is being traced).
- **Data** The Data window displays values of all variables, constants, and arguments that are in scope for the current method.
- **Watches** You use this window to display the current values of the expressions you are watching.
- **Classes** This window displays the packages and classes that will be traced in the debug session. You can include or exclude tracing of specific packages using the right-click menu.
- **Heap** Use this window to examine the use of memory by objects and arrays in the program. This allows you to verify that an object is still in memory and to verify that garbage collection is occurring. The Memory Profiler described later contains more details about runtime memory.
- **Monitors** The Monitors window tracks the synchronization of data and activities between threads of execution. For example, you can use this window to determine which thread is waiting for another thread to complete. This kind of check is helpful in detecting deadlocks.
- **Inspector** This window is not available in the View menu, but is available as the Inspect option from the right-click menu after selecting a variable or expression (such as `storeMax * 100`) in the Code Editor, Watches window, Data window, or Smart Data window. You can open many Inspector windows and use each one to track a single variable or expression.

Some of these windows appear automatically in debug mode. As with other windows in the JDeveloper IDE, all can be anchored into the IDE frame as Figure 6-2 shows. You can place any window in any frame area in the IDE by dragging it over the frame using the draglines in the window title. If there is more than one window in a frame area, the windows will be selectable with tabs. You can drag each tab out of the frame to another frame or to a floating window.

Each window has a right-click menu with appropriate functions such as the following menu from the Breakpoints window:

The right-click menus of most debug windows include a Settings item. This item displays the Preferences dialog page that manages the options for that window, such as the following dialog for the Breakpoints window:



The Debug Toolbar Buttons

The buttons in the Debug tab toolbar shown next offer the main functions that you need when running a debug session. Many of these buttons are also in a toolbar that appears in the Log window when you run a program in debug mode.

The functions and the keyboard shortcuts (for the JDeveloper default key mapping) that you can alternatively use to activate them follow:

- **Debug <project>** Execute the program from the start to the first breakpoint or to the end of the program if no breakpoints are encountered (SHIFT-F9).
- **Set Start Debugging Option** Select an option for what happens when you start debugging (Step Into, Step Over, or Run Until a Breakpoint Occurs). This icon will change according to the option you select.
- **Resume** Continue a paused program from the current line of execution to the next breakpoint or to the end of the program if no subsequent breakpoints are defined (F9).
- **Step Over** Execute the next method call or instruction without tracing, and stop at the next executable line (F8).
- **Step Into** Execute the program and trace the code line by line (F7).
- **Step Out** Step out of the current method, and return to the next instruction of the calling method (SHIFT-F7).
- **Step to End of Method** Execute the program to the end of the current method or to the next breakpoint in the current method.
- **Pause** Temporarily stop the program as it executes.
- **Terminate** Stop the program that is in debugging mode. The debug settings, such as breakpoints and watches that you set during the debugging session, will be saved (CTRL-F2).
- **Garbage Collection** If the virtual machine supports manual triggering of garbage collection, force garbage collection and show the results in the Classes window.

NOTE

*The keyboard shortcuts (or "accelerators") listed in this section are those assigned by the default key map. You can select a different key map using the Accelerators page of the Preferences dialog (**Tools | Preferences**).*

The Debug Menu Items

Most Debug menu items are enabled only when you are in debug mode. The Debug menu options repeat the toolbar options and add the following functions:

- **Debug <selected file>** Start the debug session for the file selected in the Navigator window. If the selected file is the *default run target* (file that runs when the project is run), this option is the same as the Debug <project> option.
- **Detach** Stop the debugging session, but do not stop the running program. This is useful for remote debugging on an application server.
- **Continue Step** Resume tracing (into a step) after the debugger stops at an exception breakpoint (SHIFT-F8).
- **Run to Cursor** Run the program from its current execution point to the line of program code containing the cursor (F4).
- **Set Next Statement** Skip from the stopping point to the code at the location of the cursor, even if the cursor is before the stopping point. Code lines between the stopping point to the cursor location is not executed. This option is useful for backtracking to previously executed lines of code after you have changed data values.
- **Show Execution Point** Display the section of code that is currently running.

Some of the options in the toolbar and menu are also available in the right-click menu in the Code Editor.

Controlling Program Execution

One main debugging objective is to track and control the *execution point*—the section of code that is about to be executed. In a typical session, the debugger stops the program execution, highlights the execution point, and waits for you to resume the program. You use one of the debug actions (button, keyboard shortcut, or menu selection) to resume the program execution.

Debugging Actions

While the program is stopped, you can evaluate the stack and the order in which the program is executed to determine whether the program execution path is the problem. The Threads window and Stack window display this information. In the pause, you can also evaluate data values (as mentioned in the later section "Examining Program Data Values").

As you step through the code in this way, you are *tracing* its path. You can choose to skip tracing each line of code in a particular method that is called by stepping over the call. The method will still execute, but you will not stop in the method code. This is useful if you are certain that a method works and you do not want to take the time to look at it in detail. If you want to examine the method that is called next, you *step into* the method. If you start tracing in a method and determine that the rest of the code works, you can choose to *step out* of that method.

These are the main actions in a debugging session. There are other actions you can take in the debugger that are available from the right-click menus in the various debugger windows.

Other techniques such as stepping back, disabling tracing for classes that have not been loaded, and tracing into a class that has no source code file, are explained further in the help system node "Testing and Optimizing Application Code\Debugging in JDeveloper."

Breakpoints

A *breakpoint* is a defined pause point for the debugging session. It is the primary method that you use to control the program execution. When the debugger reaches a breakpoint, it stops and waits for an action.

JDeveloper supports the following breakpoint types:

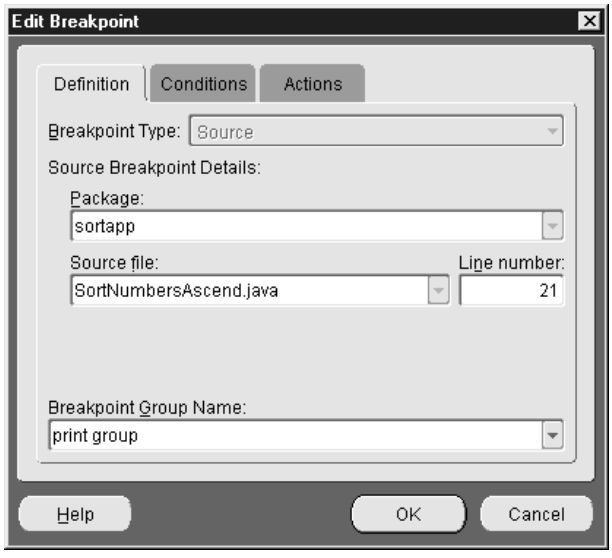
- **Source breakpoint** This is set to pause at a particular line of code specified by the developer.
- **Exception breakpoint** This is set to pause when an exception for a particular exception class is thrown. An exception breakpoint is automatically set for `java.lang.Throwable`, but you may create exception breakpoints for other exception classes.
- **Deadlock breakpoint** This is automatically defined by JDeveloper to detect situations where one thread is waiting for another thread that is, in turn, waiting for it (a deadlock). This feature is only available for some JVMs. You can disable this deadlock using the Breakpoints window right-click menu, but you cannot define a deadlock breakpoint.
- **Method breakpoint** This occurs when a particular method is called. The method is defined in the Edit Breakpoint or Add Breakpoint dialog.
- **Class breakpoint** This occurs when any method in a particular class is called. The class name is entered in the Edit Breakpoint or Add Breakpoint dialog.

All breakpoint types except for the deadlock breakpoint may be created using the New Breakpoint dialog available from the right-click menu in the Breakpoints window. All breakpoint types may be edited, disabled, and enabled in the same menu.

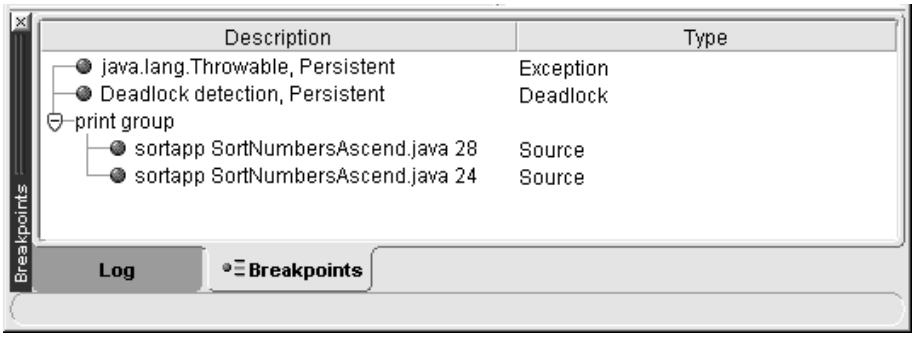
Defining Breakpoints Source breakpoints are marked with a red circle icon in the Code Editor's left margin so that you can easily identify the lines of code at which the execution will stop. The list of breakpoints appears in the Breakpoints window.

You can enable and disable or delete breakpoints using options in the right-click menu in this window. You can also delete or set a breakpoint in the Code Editor by clicking in the line and pressing F5 or by clicking the left margin on that line.

Breakpoint Groups You can modify breakpoints as a group. A *breakpoint group* is a set of breakpoints that share the same name. You can enter or select the group name in the Definition tab of the Edit Breakpoint dialog (accessible from the right-click menu on the breakpoint in the Breakpoints window) as shown here:



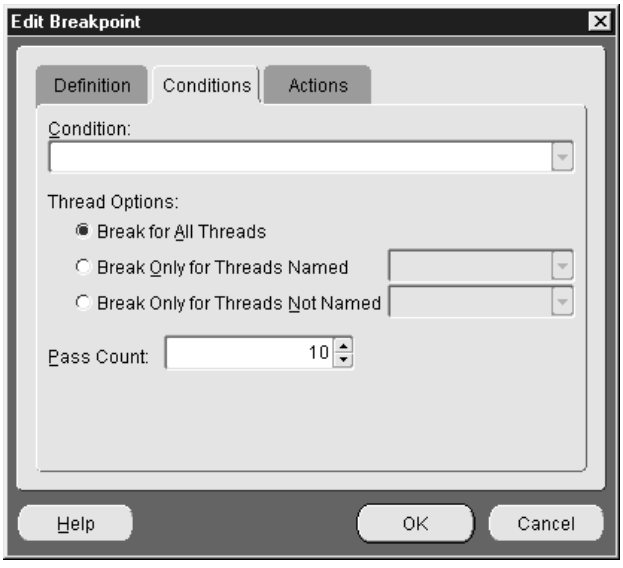
The group name will then appear in the Breakpoints window as follows:



You can select Edit Group from the right-click menu on the group name node in the Breakpoints window to set conditions and actions for the entire set of breakpoints. You can also enable or disable the group using the Enable Group or Disable Group right-click menu options, respectively, in that window.

Conditions You can define a breakpoint condition so that the program will stop at the breakpoint only if a variable value is within a certain range or a flag variable is set. You can also set a *pass count* that specifies that the execution will stop only after breakpoint is reached a certain number of times. For example, if the pass count is set to "3," the breakpoint will only stop execution the third time it is reached. This is an additional condition that is placed on the breakpoint that is added (using AND logic) to the other conditions that are defined.

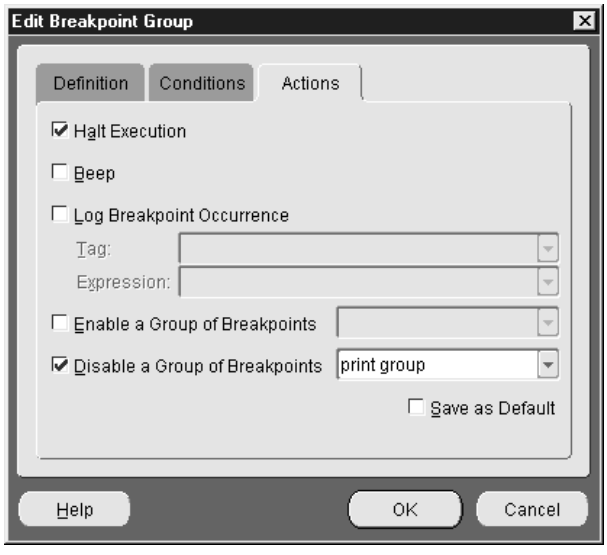
You can set and define conditions for breakpoints on the Conditions tab of the Edit Breakpoint dialog shown next that appears when you select Edit Breakpoint from the right-click menu on the breakpoint listed in the Breakpoints window.



NOTE
The Edit Breakpoint dialog for an existing breakpoint is the same as the New Breakpoint dialog for a new breakpoint. Both are available from the right-click menu in the Breakpoints window.

Breakpoint Actions You can define a number of actions to take place when a breakpoint occurs. The default action that this chapter assumes is "Halt Execution," which stops execution at the breakpoint line. You can also log the breakpoint occurrence, enable or disable a group of breakpoints, or issue a beep from the system's speaker.

All actions are set on the Actions tab of the Edit Breakpoints dialog described earlier. The following shows this dialog and how you would disable all the breakpoints in a group that you named "print group":



TIP

*The Breakpoints page of the Preferences dialog (**Tools | Preferences**) allows you to set the columns that are displayed in the Breakpoints window. A child page, Default Actions, lets you set the selections that automatically appear in the Edit Breakpoint dialog. Setting defaults in this way is useful if you want to have an action (such as a system beep) occur for each breakpoint.*

Examining Program Data Values

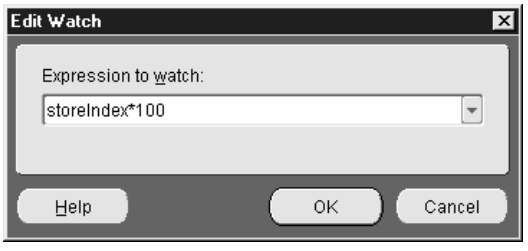
The other main debugging objective is to examine the values of *expressions* that represent variables, constants, and data structure values and the operators that act upon them. You may not use method calls in a debugging expression. You also may not use local variables or static variables that are unavailable (because of scope) to the code line you are executing. The following windows described earlier allow you to examine expressions:

Data Window As mentioned, this tracks variables, constants, and arguments that are close to the execution point.

Smart Data Window As mentioned, this tracks variables, constants, and arguments that are close to the execution point.

Watches Window This window shows the data values that are part of the expressions being tracked. A *watch* is an expression that contains program variables or other data elements and their operators. You set up a watch and monitor its value as the program executes. The watch displays the value for the context (current) execution.

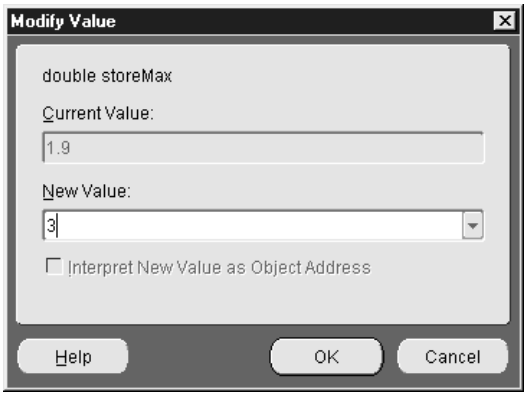
You can add a watch as the program runs by selecting the variable in the Code Editor and selecting Watch (CTRL-F5) from the right-click menu. You can modify the watch expression as shown next by selecting Edit Watch (or Edit Expression) from the right-click menu in the Watches Window.



NOTE
You can delete a watch expression by highlighting it in the Watches window and selecting Remove Watch from the right-click menu.

Inspector Window An inspector also allows you to examine data values just as with a watch. Display the Inspector window by selecting a variable in the Code Editor, Data window, Smart Data window, or Watches window and then selecting Inspect from the right-click menu. The expression is always evaluated within the current scope of execution. You can open more than one inspector at a time to track different expressions.

Modify Value Dialog This dialog (shown next) displays the data state of a variable, constant, or argument within the context of the current program execution point.



It is available by selecting the Modify Value option from the right-click menu on a variable in the Data window, Smart Data window, or Watches window. You can enter a new value in the

New Value field and click OK to set the value in the debugging session. This allows you to test a bug fix that would modify a value. The value type must match the type of the variable to which it is assigned. The change you make does not change your code. You can only alter primitive variables, strings, and reference pointers.

View Whole Value If the value you are watching is too large for the cell in one of the data-related debug windows (such as the Data window or Smart Data window), you can show a multi-line window that contains the entire value by selecting View Whole Value from the right-click menu.

> **NOTE**
> *You can modify the fields that appear when you expand a node in the data-related debug windows by selecting Edit Filters from the right-click menu. The Edit Filters dialog allows you to hide fields (such as value, offset, count, and hash) for selected classes. This can simplify the display so that you can concentrate on a particular set of fields.*

Remote Debugging

Debugging an application that is running on a remote server (such as a JSP application or servlet) is a challenge because error and execution messages must be generated by the server instead of by the IDE. Although the basic approaches to debugging code are possible, such as adding calls to `System.out.println()` and logging this output to a file, these debugging methods do not always suffice, and you need to run the debugger on the server. When you run the debugger in the JDeveloper IDE, the program you are running (called a *debuggee process*) is attached to the debugger automatically. When you are running remotely on an application server, you start the program and then attach it to the debugger.

JDeveloper's support for remote debugging of server-side Java includes the features built into the local IDE debugger, such as controlled execution, breakpoints, watch expressions, and examination of variable values and properties during execution. Therefore, with JDeveloper, the same interface is used for both local and remote debugging.

More Information

Although the topic of remote debugging is beyond the scope of this book, most techniques mentioned in this chapter apply. The additional configuration steps and specifics on attaching and detaching remote processes are well documented in the JDeveloper help system. Start in the Contents tab node "Testing and Optimizing Application Code\Debugging in JDeveloper\ Debugging Remote Java Programs." In addition, Oracle's developer website, otn.oracle.com, contains white papers on remote debugging.

The Profiler

The Profiler is new to Oracle9*i* JDeveloper. This tool assists in debugging resource issues and in helping you make informed optimization decisions for your code. It collects and displays information that supplements the information that you can gather in the debugger. The Profiler tracks three main types of information: events, execution, and memory. Each profile type (or mode) has its own window and configuration settings. Although the Profiler is a single tool, the JDeveloper

help system documentation uses the terms "Memory Profiler," "Events Profiler," and "Execution Profiler" to represent the three modes in which you can run the Profiler.

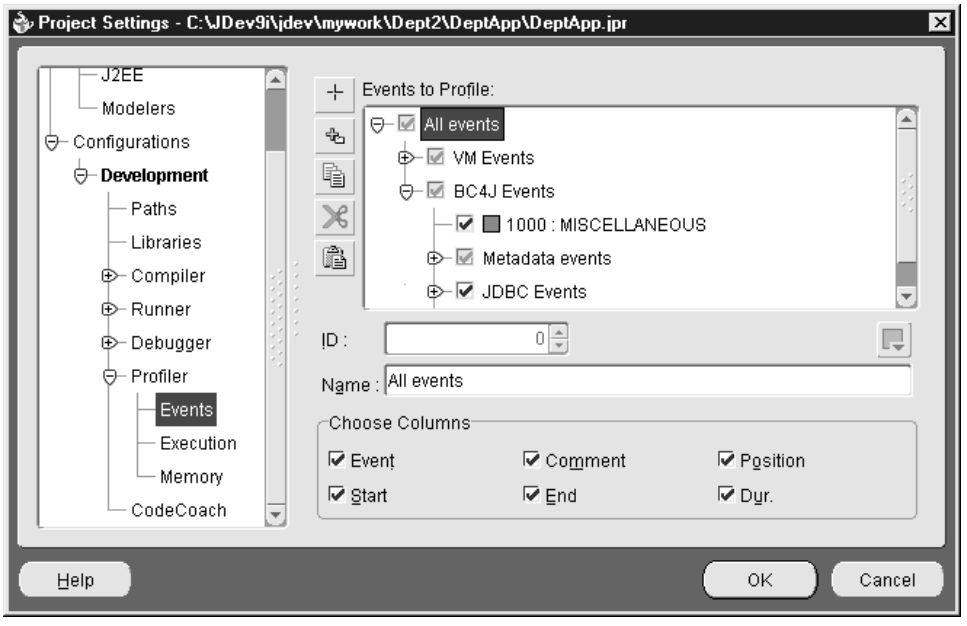
The following provides an overview of the general steps that you take to run this tool and provides a view of the different profiles. This will serve as an introduction to the technical details explained in the JDeveloper help system node "Testing and Optimizing Application Code\Profiling a Project" to which you can refer for further information.

NOTE
You can also use the Profiler to collect information about a remotely running program. In this environment, the local development machine is running the Profiler but not the program being profiled. Since the program being profiled and the Profiler are running on different machines, the Profiler and program do not compete for resources as they do when they are running on the same machine.

Running the Profiler

The steps for running the Profiler are similar for all three types of profiles and consist of the following activities:

- **Setting Profiler options** You need to configure the Profiler using the Project Settings dialog available by selecting Project Settings from the right-click menu on the project node in the Navigator window. Navigate to the Configurations\Development\Profiler node, and click the profile type that you want to set (Events, Execution, or Memory). Each has a separate page in the Project Settings dialog. For example, the following shows the settings for the Event Profiler:



- **Defining the Profiler class set** You have to declare which classes the Profiler will track (the *Profiler class set*). Navigate to the Profiler page of Project Settings as mentioned before, and list the packages and classes to include and exclude. This page also allows you to declare if you will be profiling remotely.
- **Starting profiling** You start the Profiler using an option in the Run menu. For example, to run the Event Profiler, select the program in the Navigator window, and select **Run | Event Profile <project>**. This starts the program and displays the Profiler window.
- **Examining results** When you reach a point in the program that you would like to examine, you click the Pause (yellow bars) button in the Profiler window. This refreshes the display in the Profiler to reflect the current state of the program. You can sort the table displays by clicking the column heading buttons. For the event and execution modes, you click the Clear button (eraser) to reset the display. Then click the Resume (green arrow) button to restart the program and collect more profile data. Click Pause again when you reach the end of the section of code that you are profiling. You can save the information collected into an HTML file by selecting Save to HTML from the right-click menu.
- **Terminating the Profiler** When you are finished with the cycle of pausing, examining results, and resuming, you can close the Profiler window or select **Run | Terminate | <project>**. Closing the Profiler will also stop the program execution, but stopping the program will not close the Profiler.

Event Profiler

This profile allows you to view the timings of individual actions in your program and compare them with timings in the rest of the program. It helps you identify the exact area that is causing a slowdown in processing. As Figure 6-3 shows, each event includes timing information and a description that helps you identify the event.

You can set up your own events using the *Profiler API,* a package included with JDeveloper. Use this if there is an event that you want to profile but that is not included in the classes that your program uses. The API class is documented in the help node "Testing and Optimizing Application Code\Profiling a Project\Reference: Class ProfilerAPI."

Execution Profiler

This type of profile accumulates timing information about the current thread at regular intervals (defined in the Project Settings dialog). It allows you to identify the methods and threads that take the most time and therefore provide opportunities for tuning. Figure 6-4 shows the Execution Profiler window. Clicking a method line in the table on the left displays information about the method that calls it and about the methods that it calls.

Memory Profiler

The Memory Profiler helps you identify the cause of memory leaks. It also samples at an interval defined in the Project Settings dialog. A feature of this profile is that you can review a number of samples so that you can compare how memory was used at different times in the program execution. The Memory Profiler window is shown in Figure 6-5. The slider at the top of the window allows you to go back to a previous sampling and to compare how memory was used at different times.

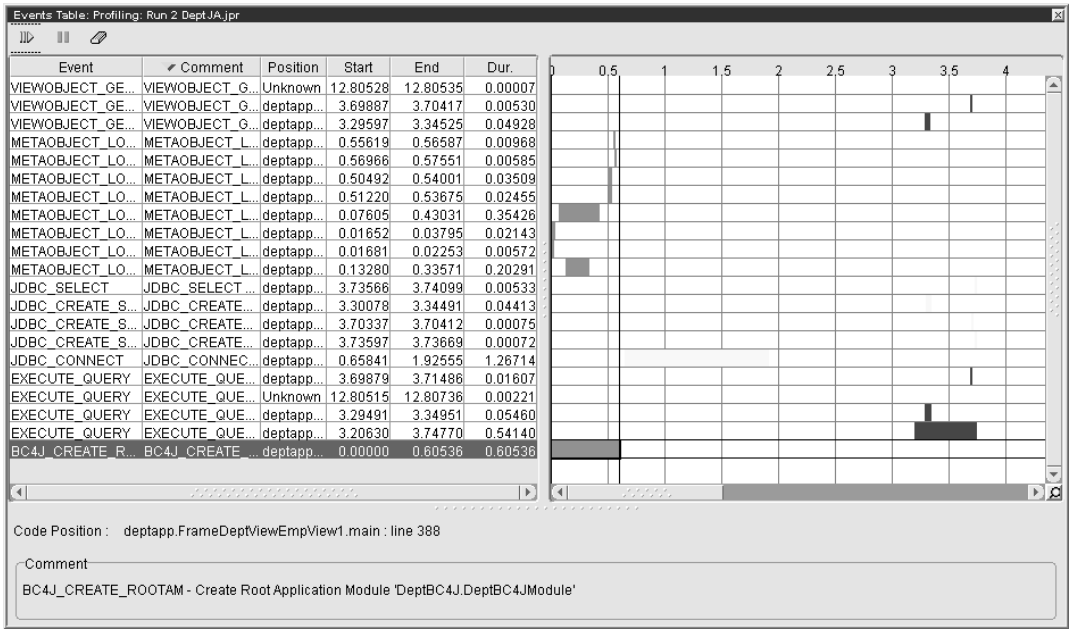


FIGURE 6-3. *Event Profiler window*

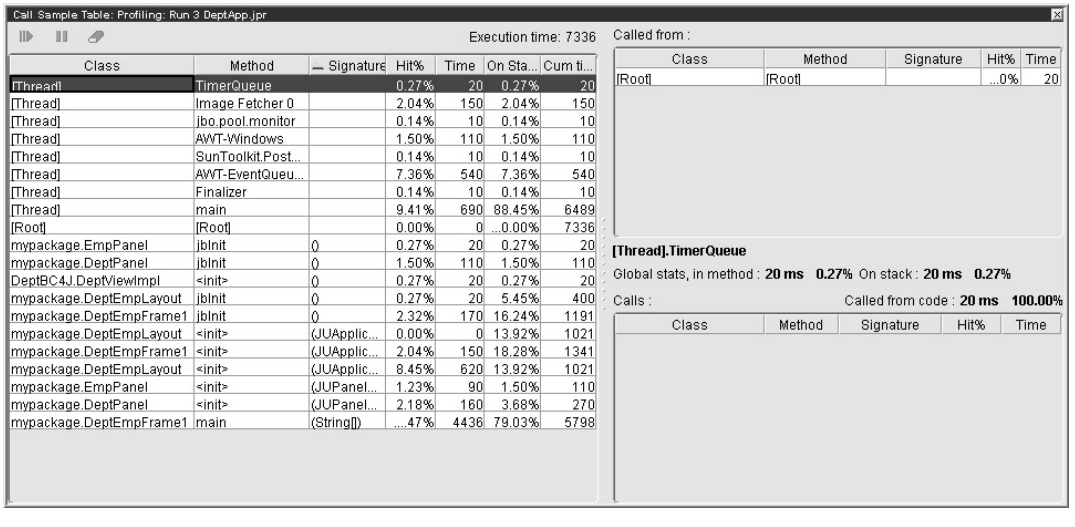
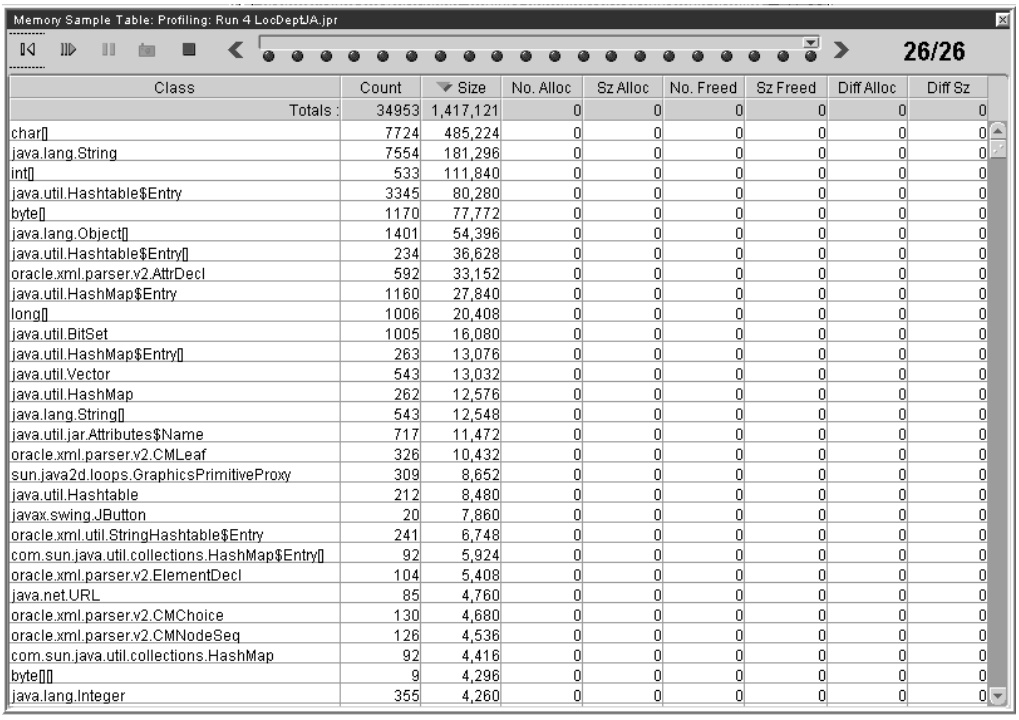


FIGURE 6-4. *Execution Profiler window*


Memory Sample Table: Profiling: Run 4 LocDeptJA.jpr

26/26

Class	Count	Size	No. Alloc	Sz Alloc	No. Freed	Sz Freed	Diff Alloc	Diff Sz
Totals :	34953	1,417,121	0	0	0	0	0	0
char[]	7724	485,224	0	0	0	0	0	0
java.lang.String	7554	181,296	0	0	0	0	0	0
int[]	533	111,840	0	0	0	0	0	0
java.util.Hashtable$Entry	3345	80,280	0	0	0	0	0	0
byte[]	1170	77,772	0	0	0	0	0	0
java.lang.Object[]	1401	54,396	0	0	0	0	0	0
java.util.Hashtable$Entry[]	234	36,628	0	0	0	0	0	0
oracle.xml.parser.v2.AttrDecl	592	33,152	0	0	0	0	0	0
java.util.HashMap$Entry	1160	27,840	0	0	0	0	0	0
long[]	1006	20,408	0	0	0	0	0	0
java.util.BitSet	1005	16,080	0	0	0	0	0	0
java.util.HashMap$Entry[]	263	13,076	0	0	0	0	0	0
java.util.Vector	543	13,032	0	0	0	0	0	0
java.util.HashMap	262	12,576	0	0	0	0	0	0
java.lang.String[]	543	12,548	0	0	0	0	0	0
java.util.jar.Attributes$Name	717	11,472	0	0	0	0	0	0
oracle.xml.parser.v2.CMLeaf	326	10,432	0	0	0	0	0	0
sun.java2d.loops.GraphicsPrimitiveProxy	309	8,652	0	0	0	0	0	0
java.util.Hashtable	212	8,480	0	0	0	0	0	0
javax.swing.JButton	20	7,860	0	0	0	0	0	0
oracle.xml.util.StringHashtable$Entry	241	6,748	0	0	0	0	0	0
com.sun.java.util.collections.HashMap$Entry[]	92	5,924	0	0	0	0	0	0
oracle.xml.parser.v2.ElementDecl	104	5,408	0	0	0	0	0	0
java.net.URL	85	4,760	0	0	0	0	0	0
oracle.xml.parser.v2.CMChoice	130	4,680	0	0	0	0	0	0
oracle.xml.parser.v2.CMNodeSeq	126	4,536	0	0	0	0	0	0
com.sun.java.util.collections.HashMap	92	4,416	0	0	0	0	0	0
byte[][]	9	4,296	0	0	0	0	0	0
java.lang.Integer	355	4,260	0	0	0	0	0	0

FIGURE 6-5. *Memory Profiler window*

NOTE

The Profiler button on the toolbar changes to reflect the last profile type that was run. For example, if you last ran the Execution Profiler, the button will run that profile for the selected project.

Hands-on Practice: Debug a Java Application

The best way to understand how the debugger works is to apply it to a task and to examine the activities required to find a problem. In this practice, you will walk through all of the major steps for carrying out the two essential debugging activities of controlling program execution and examining program data values. The practice follows these phases:

I. **Create a buggy application**

II. **Prepare for the debugging session**

III. **Control program execution**

- Run the debugger and step through the code

- Disable and enable tracing of classes

IV. Examine Data Values

NOTE
Most debug functions may be called using the menu, right-click menu, toolbar, or keyboard shortcut, and this hands-on practice demonstrates a variety of methods. You can explore the other ways to call these functions (as mentioned in the sections before "The Debug Toolbar Buttons" and "The Debug Menu Items") to see which way is easiest for you.

I. Create a Buggy Application

Normally, you do not intentionally create buggy code. The sample application for this practice requires code with bugs to illustrate the debugging process and techniques. The application used in this practice processes a list of numbers entered as command-line parameters and sorts the numbers in ascending order. The results print in the Log window. A logic error in the program results in an incorrect sort, and the aim of the practice is to discover this error. Once the bug is found, a fix can be applied.

1. Create a project in an existing workspace by selecting the workspace node and selecting New Empty Project from the right-click menu.
2. Use "SortJA" for the directory and file name.
3. On the project node, select New Class from the right-click menu.

   Additional Information: This opens the New Class dialog.
4. Enter "SortNumbersAscend" for the *Name* field and "sortapp" for the *Package* field. Leave the other defaults as shown in Figure 6-6, and click OK to create the file.
5. Click Save All.
6. Open the Code Editor for this class file, and edit it to match the following code. (This code is available on the authors' websites.)

```
package sortapp;

public class SortNumbersAscend {

  /**
   * main
   * @param args
   */
  public static void main(String[] args) {
    // Fill a list with the command line parameters
    int n = args.length;
    double[] numList = new double[n];
    for (int i = 0; i < n; i++) {
```

```
      numList[i] = Double.parseDouble(args[i]);
    }

    // display the numbers in the order they were entered
    System.out.println("The given set of numbers is: ");
    printNumList(numList);

    // sort the numbers in ascending order by calling a method
    sortAscend(numList);

    // Print the sorted list
    System.out.println("The numbers arranged in ascending order: ");
    printNumList(numList);
    }

    // The method for displaying the numbers in the message window
    static void printNumList(double[] numList) {
      for (int i = 0; i < numList.length - 1; i++)
        System.out.print(numList[i] + ", ");
      System.out.println(numList[numList.length - 1]);
    }

    // The method for sorting the numbers
    static void sortAscend(double[] numList) {
      double storeMax;
      int    storeIndex;

      for (int i = numList.length - 1; i >= 1; i--) {
        // store the current number from numList[0..i] in storeMax
        storeMax = numList[i];
        storeIndex = i;

        // Step through the array comparing the stored number
        // with the previous number
        for (int j = i - 1; j >= 0; j--) {
          if (storeMax < numList[j]) {
            storeMax = numList[j];
            storeIndex = i;
          }
        }
        // Swap the numbers if the current number is greater
        if (storeIndex != i) {
          numList[storeIndex] = numList[i];
          numList[i] = storeMax;
        }
      }
    }
}
```

7. Click Save All.

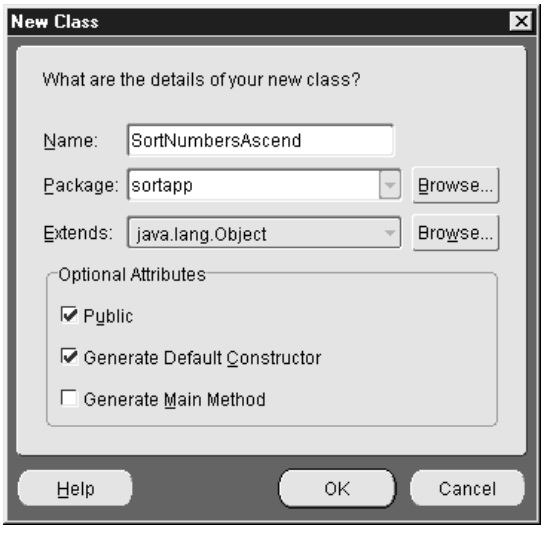


FIGURE 6-6. *New Class dialog*

What Just Happened? You created a new workspace and project. You then added a Java class file with some buggy code that you can use for testing the debugger. The intended logic for this program follows:

1. Loop through the command-line parameter list of numbers, and build an array variable with one number in each array element.
2. Display the unsorted list in the Log window.
3. Sort the numbers using the `sortAscend()` method.
4. Display the sorted list.

The logic for the `sortAscend()` method follows:

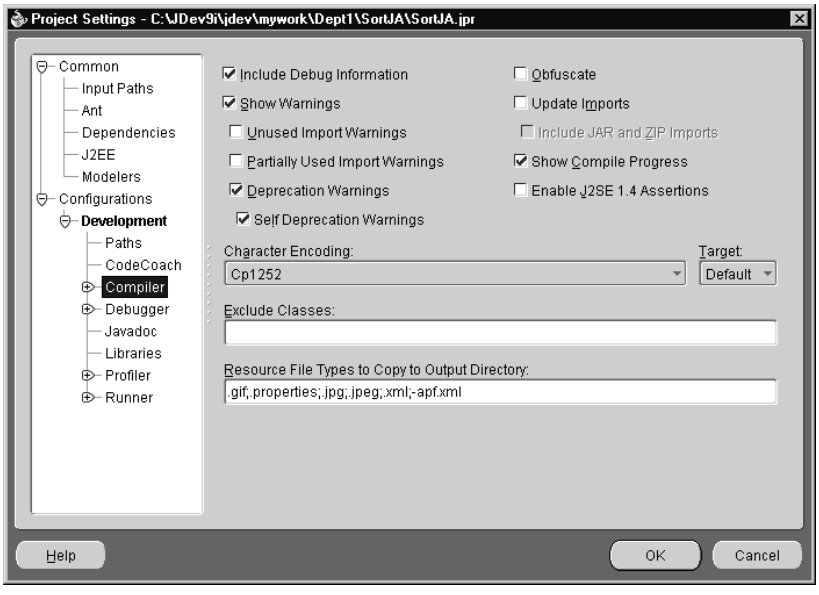
1. Loop through the array that was passed from the `main()` method.
2. Store the value of the current element in `storeMax` and current index in `storeIndex`.
3. Loop through all numbers in the list before the current element. If `storeMax` is less than the number, swap the current element number with the number that is less than `storeMax`.

II. Prepare for the Debugging Session

The first stage in running the debugger is to prepare the project settings. You then compile your program to generate the symbolic debugging information required by the debugger.

1. On the SortJA project node, select Project Settings from the right-click menu.
2. Navigate to the Configurations\Development\Compiler page (as shown next), and ensure that there are check marks in *Include Debug Information* and *Show Warnings*.

   Additional Information: The *Include Debug Information* field specifies whether special debugging information will be compiled into the .class file. This information is required for debugging. The *Show Warnings* field indicates whether compilation error messages will appear in the Log window.



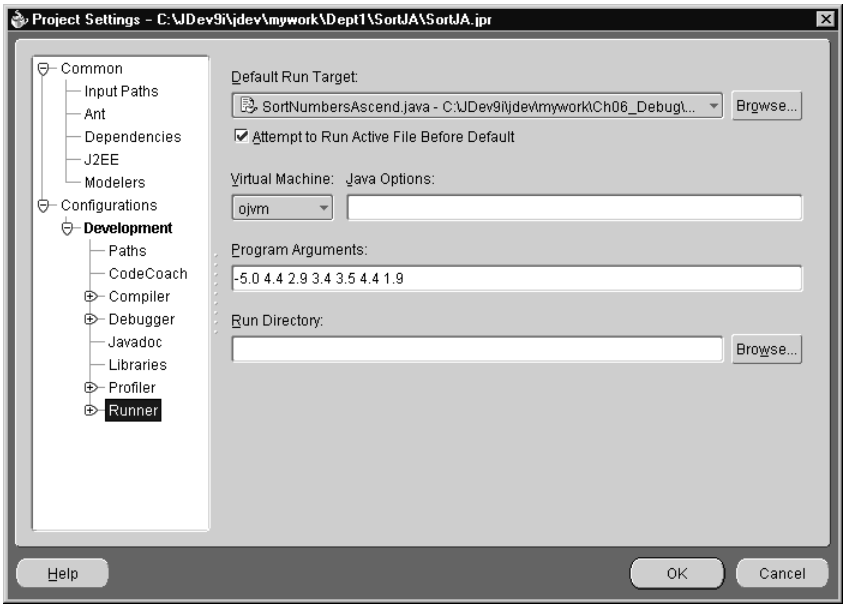
3. Navigate to the Configurations\Development\Runner\Options page, and be sure the *Make Project* checkbox is checked.

   Additional Information: This causes JDeveloper to automatically compile the code when you click Run or Debug. The alternative is to explicitly click Make before clicking Run or Debug.

4. Click the Runner node in the Project Settings dialog, and fill in the *Program Arguments* field with the following string of numbers, each separated by a blank space:

```
-5.0 4.4 2.9 3.4 3.5 4.4 1.9
```

The next illustration shows the Project Settings dialog with this string entered in the *Program Arguments* field.



Additional Information: This string of numbers (the numbers to be sorted) will be passed to the application at runtime.

5. Click OK to close the Project Settings dialog. Click Save All.

What Just Happened? You set the project settings and command-line parameters to prepare for the debugging session. The debugger requires special instructions in the class file that are inserted when you set the project settings.

CAUTION

Remove the check on the "Include Debug Information" checkbox before generating the final production code so that the final code does not incur the slight overhead of the size of the extra debugging information. This checkbox is the same as using the "-g" switch when running the javac compiler from the command line.

III. Control Program Execution

Before starting the debugger, be certain that you have properly set the project settings as in the last phase. This phase will start the debugging session and illustrate how you track the program execution.

Run the Debugger and Step Through the Code

You can run the debugger by selecting Debug (using a menu item, button, or keyboard shortcut). The code will compile and run in debug mode using whichever Set Start Debugging Option you

set (Step Into, Step Over, Run Until a Breakpoint Occurs). Step Into will stop at the first executable statement whether or not breakpoints were defined. Step Over will stop at the first executable method that is set to trace. Run Until Breakpoint Occurs will stop at the first breakpoint. Since this phase of the practice uses no breakpoints, the Step Into technique is used.

1. Select SortNumbersAscend.java in the Navigator, and select Set Start Debugging Option to Step Into on the option button in the toolbar. Click the Debug <project> button.

   Additional Information: If the code compiles successfully, the debugger will start and pause at the first executable line of code. The Smart Data, Data, and Watches tabs will be added in the Log window area. The current execution line will be indicated with an arrow in the left margin of the Code Editor.

   The debugger will add a Stack tab to the Structure window to display the method execution thread. It will also add a Breakpoints tab to the Log window to display the stopping points that you have set.

CAUTION
As mentioned, data will be shown in the Data tab only if the project property ""Include Debug Information" checkbox is checked in the Project Settings dialog.

2. Exit debug mode by clicking the Terminate button. This aborts the program without executing the rest of the code.
3. Click Debug <project> to restart the debugger. The red arrow indicating the execution point should point to the following line, which is the first executable line of code in the `main()` method:

   ```
   int n = args.length;
   ```

4. Click the cursor on the line of source code, making the first call to the method `printNumList(numlist)`.
5. Select **Debug | Run to Cursor**.

   Additional Information: This moves the execution to the location of the cursor. The Log window shows the output from the execution of the line of code prior to the call to the `printNumList()` method. This line causes the string "The given set of numbers is:" to be printed in the Log window.

6. Click the Step Over button in the toolbar.

   Additional Information: The Log window shows the numbers that were input as parameters as the output from the `printNumList()` method. This is because the Step Over command allows all of the code invoked by the call to execute to completion before pausing again.

7. Place the cursor on the line of code that makes the second call to the `printNumlist(numList)` method.

8. Press F4 (the same function as selecting **Debug | Run to Cursor**).

   Additional Information: You will see the output of the text string argument to the `println()` method—the text string "The numbers arranged in ascending order:".

9. Select **Debug | Step Into**.

   Additional Information: The execution point jumps to the first line of the `printNumList()` method.

10. Watch the Log window as you click the Step Into button on the debugging toolbar twice. Continue clicking the button four more times to watch the loop iterate.

11. Select **Debug | Step to End of Method**.

    Additional Information: The program executes the remaining iterations of the loop, and the execution point moves to the closing curly bracket of the `printNumList()` method.

12. Click the Terminate button to end the debugging session. If you look in the Log window, you will see that the output order of the numbers is the same as the input order. The sort function is not working, and this is the bug you need to fix.

Disable and Enable Tracing of Classes

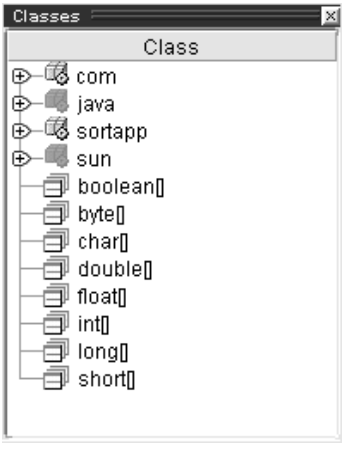
In this application you have only one file, which you created and need to debug. However, when you have many source code files in your project and want to debug only one of these files, you will need to change the list of classes that the debugger will step into.

You can change the list of classes that will be traced at runtime or at design time. The following steps demonstrate both techniques:

1. Start the debugging session again by clicking the Debug button. Since the start option is set to Step Into, the debugger will stop at the first line of executable code.

2. If the Classes window is not displayed, select **View | Debug Windows | Classes**.

   Additional Information: The Classes window will appear as shown here with all classes loaded in this session.



A gray icon for packages and classes indicates that they will be excluded from tracing. A colored package icon indicates a package that has classes that will be included in the tracing. For example, in the illustration before, the com and sortapp packages are included, and the java and sun classes are excluded.

3. On a node in the Classes window, select Tracing from the right-click menu.

   Additional Information: This displays the Tracing dialog, where you can enter class packages and classes to include or to exclude.

4. To test this feature, add ";sortapp.SortNumbersAscend" (including the leading semicolon) to the end of the *Tracing Classes and Packages to Exclude* field. Click OK.

TIP
You can use the Edit buttons in this dialog to construct the list of packages and classes. Using the dialogs from the Edit button will eliminate the problem of mistyping a name or punctuation, but may be a bit slower.

   Additional Information: The icon for SortNumbersAscend under the sortapp package node in the Classes window will turn gray. This action turns off tracing of the main application class file for demonstration purposes. If your project contained more than one file, you would use this technique to skip tracing in class files that were bug free. You need to restart the debugging session for the change in traced classes to take effect.

5. Click the Terminate button to end the debugging session.

6. Restart the debugging session by clicking Debug.

   Additional Information: The debugger will start the application, but will execute the application without stopping at any line of code.

NOTE
If you set breakpoints in an excluded class file, the debugger will still stop at those breakpoints.

7. On the SortJA project node in the Navigator, select Project Settings from the right-click menu. The Project Settings dialog will appear.

   Additional Information: Since the Classes window is only available while debugging, you cannot reset the exclusions list using the same method that you used to set it. The Project Settings dialog contains the same lists, and this dialog is available whether you are in design mode or debug mode.

8. Navigate to the Configurations\Development\Debugger node, and remove the ";sortapp.SortNumbersAscend" string from the *Tracing Classes and Packages to Exclude* field (including the leading semicolon). Click OK to dismiss the dialog.

9. Run the application again using the Debug button. The debugger will again stop at the first line of executable code. Click Terminate to stop the debugging session.

What Just Happened? You started the debugging session and practiced stepping into and stepping over code. You also viewed the messages that you can see in the Log window and tried two methods for disabling and enabling the tracing of specific classes and their child classes. Disabling tracing on a particular class or tree of classes will save your having to step out of or step over a set of code that you know to be problem free.

IV. Examine Data Values

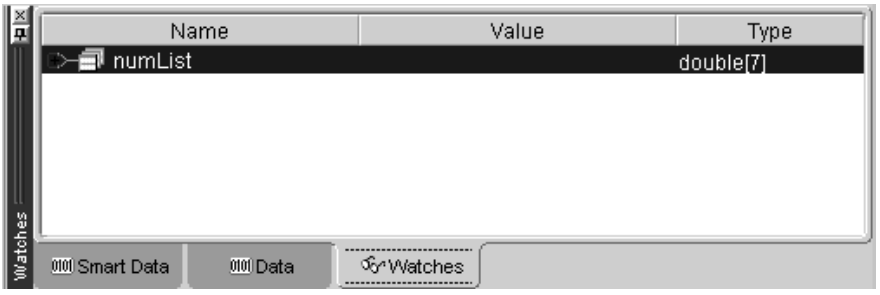
Now that you know how to control the program execution, you can practice defining watches, setting breakpoints, and examining data element values. This will help you locate the problem area in the code.

1. Click Debug to start debugging. The execution point is in the first executable line of the `main()` method (`int n = args.length;`).

NOTE
If any of the steps in this section do not work, terminate the debugging session and start the debugging session again.

2. Click the Watches window (or the Watches tab if this window is merged with other windows). Since you have not defined any watches, the window will be blank.
3. Click Step Into twice to move the execution line to the `for` statement.
4. In the Code Editor, double click to select the variable word `numList` that appears in the line before the `for` statement:
   `double[] numList = new double[n];`
5. On the selected variable `numList`, select Watch from the right-click menu. This will display the Add Watch dialog, as shown here:

6. Click OK, and you will see the `numList` variable added to the Watches window as shown next.



7. Click Step Into twice. Expand the numList node in the Watches window (using the + to the left of the word) to see that the array has been declared and the values of all elements in the array are set to "0" except for the first element.

8. Click Step Into twice more, and you will see that the value of the second element of the array has been filled in as the loop proceeds through the list of numbers.

   Additional Information: Since there are no errors until the sort method, you can skip all lines of code until the call to the `sortAscend()` method. You could just click the Step Over button until you reached that code, but there is a better way using a breakpoint.

   If you set a breakpoint on the line on which you want to stop, you can easily skip all lines of code before that. There are several ways to set a breakpoint:

   - Click the left border next to the line of code.

     Or:

   - On the line of code, select Toggle Breakpoint from the right-click menu.

     Or:

   - Press F5.

9. Create a breakpoint for the line that calls sortAscend() using one of the methods just listed. (You do not need to be in debug mode to set breakpoints.)

10. Click the Resume button to execute all lines until the breakpoint.

    Additional Information: The Watches window will show that values have been assigned to all elements of the `numList` array.

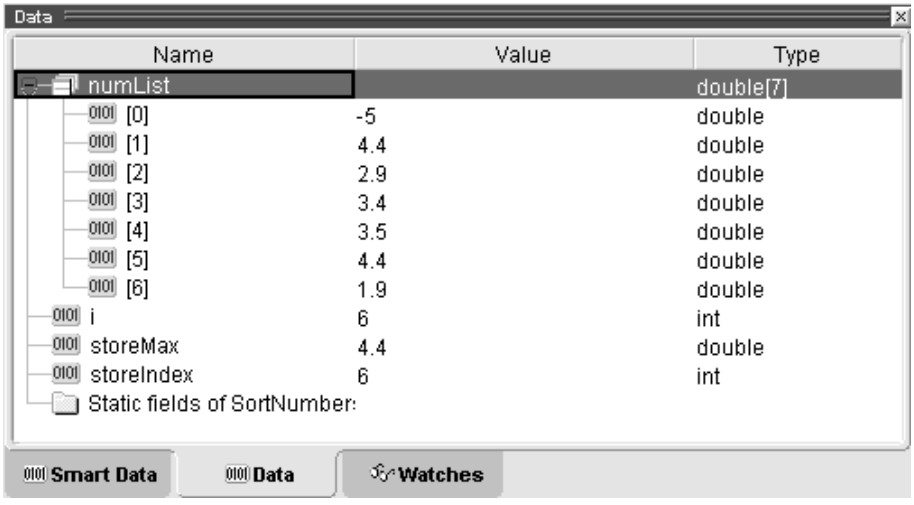
11. Click the Step Into button to trace the code in the `sortAscend()` method.

12. Place the cursor at the `if` statement immediately after the comment "Swap the numbers…". On that line, select Run to Cursor from the right-click menu.

    Additional Information: The execution point will transfer from the breakpoint to the `if` statement.

13. Click the Data window (as shown in the following illustration), and look at the values for the variables `i`, `storeMax`, and `storeIndex`. You can expand the numList node to see the individual elements in the array.

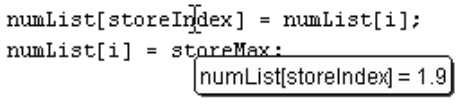
Additional Information: The sort routine correctly picked the largest number as 4.4 for the first iteration of the outer loop, but the array position of this number is not [6] and is the same as the value in `i`. This means that the exchange of values did not occur, as it should for a proper sort.



The watch for `numList` that was set earlier tracked the values of this variable, and these are displayed in both the Watches window and the Data window. All variables that appear in the current line of code are also displayed in the Data window, but since they have no watch defined, they are not shown in the Watches window.

TIP

You can examine the value of a variable by holding the mouse cursor over the variable name in the Code Editor. The value will be displayed in a tooltip next to the name, as shown here:



14. If you examine the code before the execution point, you can determine that the correct value is not being saved in the `storeIndex` variable. In the Code Editor, change the assignment of `storeIndex` in the `j` loop to "`j`" instead of "`i`" as follows:

```
storeIndex = j;
```

The file is still in debug mode while you are editing it.

15. Click Terminate to stop the program.
16. Click Run to run the program. Verify the results in the Log window, and ensure that the sort is correct.

What Just Happened? You successfully debugged a logic error by setting up a watch and a breakpoint and examining data values during the program execution. This simple example showed the basic techniques that you could use to find and fix problems in your code. The sidebar "The Modify Value Dialog" demonstrates how you can change the values of variables during the debugging session. There are some other advanced features, as mentioned earlier, that you will want to explore once you have mastered these basics. The JDeveloper help system is the source for information about these techniques.

The Modify Value Dialog

You can use the Modify Value dialog to view and change values of variables. The following steps demonstrate how to work with this dialog using the example program in this practice:

1. Remove all breakpoints by selecting Remove All from the right-click menu in the Breakpoints window. The Breakpoints window usually appears as a tab in the Log window area. If the Breakpoints window is not displayed, select **View | Debug Windows | Breakpoints**.
2. Start a debugging session using Debug. The start option (on the Set Start Debugging Option button pulldown) should still be set to "Step Into."
3. In the `sortAscend()` method, click in the following line and select Run to Cursor from the right-click menu to run all code before that line:
   ```
   if (storeMax < numList[j])
   ```
4. In the Data window, click the `storeMax` line and select Modify Value from the right-click menu to display the Modify Value dialog.
5. Enter "2" in the *New Value* field and click OK. The value "2" will replace the previous value of 1.9 and will display in the Data window. The program will then use the new number when that variable is next referenced.
6. Click Terminate to end the session.