QUEST SOFTWARE®

JProbe® 8.1

Demos and Tutorials

If you have any questions regarding your potential use of this material, contact:
Quest Software World Headquarters
LEGAL Dept
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
email: legal@quest.com
Refer to our Web site for regional and international office information.

Trademarks

Disclaimer

The information in this document is provided in connection with Quest products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Quest products. EXCEPT AS SET FORTH IN QUEST'S TERMS AND CONDITIONS AS SPECIFIED IN THE LICENSE AGREEMENT FOR THIS PRODUCT, QUEST ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL QUEST BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF QUEST HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Quest makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Quest does not make any commitment to update the information contained in this document.

Third Party Information

See *Third_Party_Contributions.htm* in your JProbe *\doc* installation directory.

Demos and Tutorials
February 2009
Version 8.1

Table of Contents

Introduction to This Guide

This chapter provides information about what is contained in the Demos and Tutorials. It also provides information about the JProbe documentation suite and Quest Software.

This chapter contains the following sections:

About JProbe

JProbe is an enterprise-class Java profiler that provides intelligent diagnostics on memory usage, performance, and test coverage. It allows developers to quickly pinpoint and repair the root cause of application code performance and stability problems that obstruct component and integration integrity.

JProbe provides three types of analysis:

- Memory analysis allows a developer to identify and resolve Java memory leaks and object cycling to ensure optimal program efficiency and stability.
- Performance analysis allows a developer to identify and resolve Java bottlenecks and deadlocks to ensure optimal program performance and scalability.
- Coverage analysis allows a developer to identify unexecuted lines of code during unit testing to ensure test coverage and program correctness.

JProbe also offers an Eclipse plug-in that provides intelligent code performance analysis and problem resolution from within the Eclipse Java IDE.

About This Guide

This guide contains a summary of all the demo applications that ship with JProbe. It also contains tutorials for some of the applications.

This guide is intended for Java developers who want to learn how to configure JProbe to work with their application and run a JProbe analysis.

How to Use This Guide

A good place to start is with a demo designed for your Java platform, that is, Java SE or Java EE. Next decide which JProbe analysis tool you are most interested in and work through one of the tutorials for that tool.

By the end of your first tutorial, you should know the basic steps involved in running a JProbe analysis. In particular, you will have learned how to complete the following tasks:

- integrate JProbe with a Java SE or Java EE application
- start and run a JProbe analysis session

- identify a problem with the application using the data that JProbe collected
- investigate the problem
- rerun the session with improved code

Later you may choose to review some of the other demos and tutorials to gain a broader understanding of the types of problems you can identify with JProbe.

Where to Find Information Not in This Guide

The following table shows where you can find other types of information:

Information about:	**Refer to:**
Configuring JProbe to run your application or application server	• *JProbe User Guide* (PDF/online help) • *JProbe Plugins for Eclipse Guide*
Running sessions from the JProbe Console	*JProbe User Guide* (PDF/online help)
Using JProbe Plugins for Eclipse	• *JProbe Plugins for Eclipse Guide* • *JProbe User Guide* (online help)
Automating JProbe analysis sessions using command line utilities	*JProbe Reference Guide* (PDF)
Adding JProbe to an Ant system	*JProbe Ant Task User Manual* (HTML)
System requirements, licensing, and installation notes	*JProbe Installation Guide* (PDF)
Known and resolved issues	*JProbe Release Notes* (HTML)

JProbe Documentation Suite

The JProbe documentation suite is provided in a combination of online help, PDF, and HTML.

- **Online Help:** You can open the online help by clicking the **Help** icon on the JProbe toolbar.
- **PDF:** The complete JProbe documentation set is available in PDF format on SupportLink. The PDF documentation can also be found in the Documentation folder on the JProbe DVD. The default location of the documentation after an installation is *<jprobe_home>/docs*. Adobe® Reader® is required.
- **HTML:** Release Notes are provided in HTML and text format. The default location of this document after an installation is *<jprobe_home>/docs*.

  The *Ant Tasks User Manual* is also provided in HTML format. The default location of this document after an installation is *<jprobe_home>/automation/doc*. To open it, navigate to *index.html*.

Core Documentation Set

The core documentation set consists of the following files:

- *Installation Guide* (PDF)
- *User Guide* (PDF and online help)
- *Reference Guide* (PDF)
- *Plugins for Eclipse Guide* (PDF)
- *Demos and Tutorials* (PDF)
- *Release Notes* (HTML)
- *Ant Tasks User Manual* (HTML)

Feedback on the Documentation

We are interested in receiving feedback from you about our documentation. For example, did you notice any errors in the documentation? Were any features undocumented? Do you have any suggestions on how we can improve the documentation? All comments are welcome. Please submit your feedback to the following email address:

am.docfeedback@quest.com

Please do not submit Technical Support related issues to this email address.

Text Conventions

The following table summarizes how text styles are used in this guide:

Convention	Description
`Code`	Monospace text represents code, code objects, and command-line input. This includes: • Java language source code and examples of file contents • Classes, objects, methods, properties, constants, and events • HTML documents, tags, and attributes
`Variables`	Monospace-plus-italic text represents variable code or command-line objects that are replaced by an actual value or parameter.
Interface	Bold text is used for interface options that you select (such as menu items) as well as keyboard commands.
Files, components, and documents	Italic text is used to highlight the following items: • Pathnames, file names, and programs • The names of other documents referenced in this guide

About Quest Software, Inc.

Quest Software, Inc., a leading enterprise systems management vendor, delivers innovative products that help organizations get more performance and productivity from their applications, databases, Windows infrastructure and virtual environments. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 90,000 customers worldwide meet higher expectations for enterprise IT. Quest provides customers with client management as well as server and desktop virtualization solutions through its subsidiaries, ScriptLogic and Vizioncore. Quest Software can be found in offices around the globe and at *www.quest.com*.

Contacting Quest Software

Email	*info@quest.com*
Mail	Quest Software, Inc. World Headquarters 5 Polaris Way Aliso Viejo, CA 92656 USA
Web site	*www.quest.com*

Refer to our web site for regional and international office information.

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract. Quest Support provides around the clock coverage with SupportLink, our web self-service. Visit SupportLink at: *http://support.quest.com*.

From SupportLink, you can do the following:

- Quickly find thousands of solutions (Knowledgebase articles/documents).
- Download patches and upgrades.
- Seek help from a Support engineer.
- Log and update your case, and check its status.

View the *Global Support Guide* for a detailed explanation of support programs, online services, contact information, and policy and procedures. The guide is available at: *http://support.quest.com/pdfs/Global Support Guide.pdf*.

Quest Communities

Get the latest product information, find helpful resources, and join a discussion with the JProbe Quest team and other community members. Join the JProbe community at: *http://jprobe.inside.quest.com/*.

1

Memory Analysis Demos

This chapter provides a summary of the Memory demo applications that ship with JProbe and contains tutorials for some of these applications.

The source code and compiled classes for the Memory demos are located in the *<jprobe_home>/demos/memory* directory.

This chapter contains the following sections:

Summary of Demos for Memory

The following table describes the purpose of the example applications.

Java SE Application	Purpose	More Information
Account.class	This application creates an account object and it is used by the *AccountInfo.class* application.	See the notes in *Account.java*.
AccountInfo.class	Displays three sets of account information to illustrate the impact of indirect object instantiation.	See the notes in *AccountInfo.java*.
LeakExample.class	This example illustrates how an obsolete collection reference may cause loitering objects. When the buttons are removed from a panel, the `JButton` objects are not removed from the Java heap.	See the notes in *LeakExample.java*. **Tutorial:** "LeakExample Tutorial" on page 16
LeakExample2.class	Similar to *LeakExample.class*, this application registers the buttons as listeners. The program demonstrates loitering objects caused by an obsolete listener.	See the notes in *LeakExample2.java.*
Network.class	This example simulates clients (threads) connecting to a server and querying a database. Temporary objects are created for the login data and for the connection.	See the notes in *Network.java*. **Tutorial:** "Network Tutorial" on page 26

Java SE Application	Purpose	More Information
Sim.class	This example simulates a network model in which a connection is made to verify the identity of the user. If the identity of the user is validated, the application extracts the desired data from the database and saves it in the result set.	See the notes in *Sim.java*.
StalledStack.class	This example shows how a stalled stack reference can hold objects in memory longer than necessary. You could be using the memory consumed by these objects for other tasks. Uses heap triggers.	See the notes in *StalledStack.java*.
Strings.class	Compares two algorithms: one that creates large allocations of string objects and one that does not.	See the notes in *Strings.java*.

JProbe also ships with a Java EE demo application called JProbe Game Pack. For more information, see “JProbe Game Pack for JavaEE” on page 73.

LeakExample Tutorial

The *LeakExample* program illustrates how an obsolete collection reference can anchor entire trees of loitering objects in the Java heap.

This tutorial demonstrates how to use JProbe to identify loitering objects in your code and how to reclaim memory by removing loitering `JButton` objects. The improved code shows a 85.60% improvement in how much memory is used by the instances allocated by *LeakExample* methods, and a 0.71% improvement in the overall memory used for the entire program.

Note **The values cited in this tutorial reflect the *LeakExample* running on Windows XP with Sun JDK 1.6.0_10.** You may see different values on your system, but the improvement in memory use should still be evident.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	*LeakExample.class*
Use Case:	Add buttons to a panel. Remove buttons from a panel.
Architecture:	Uses the class `JButton` to create buttons.
Hypothesis:	`JButton` objects are removed from the heap when the buttons are removed from the panel.

This tutorial assumes that you are running JProbe on your local machine. You can find more information about loitering objects in the *JProbe User Guide*.

This tutorial walks you through the following steps:

- Step 1: Setting Up the Memory Leak Session
- Step 2: Running the Memory Leak Session
- Step 3: Identifying Loitering Objects
- Step 4: Investigating Loitering Objects
- Step 5: Running the Memory Leak Session with Improved Code

Step 1: Setting Up the Memory Leak Session

In this step, you use the JProbe Configuration tool to create the session settings for this tutorial. The following procedures mention only the settings that you need to change or verify. If a setting is not mentioned, leave it blank or in its default state. The procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1 Click **Tools > Create/Edit Settings**.

The Create/Edit Settings dialog box appears.

2 In the Manage Configurations pane, click **Java Application**.

The JProbe Configuration Wizard appears.

3 Click **Add**.

4 In the Configuration Name text box, type *LeakExample,* then click **Next**.

5 In the Main Class field, select the **Execute a class** check box.

6 Click the browse button in the Main Class field and navigate to the *LeakExample.class* file in the *<jprobe_home>/demos/memory/leakexample* directory.

7 Click **OK**.

The following information is displayed:

- Main Class: `demos.memory.leakexample.LeakExample`
- Working Directory: <*jprobe_home*>

8 Click the browse button beside the Classpath field.

9 In the Classpath dialog box, click **Add Working Directory**, then click **OK**.

The working directory appears in the Classpath field.

10 Click **Next**.

The Select a Java Virtual Machine page appears.

11 If you want to change the default JVM, click the browse button beside the Java Executable field and select another JVM in the Java Virtual Machines dialog box.

Alternatively, you can use the JVM that is installed with JProbe to run with the tutorial. The java executable is *<jprobe_home>/bin/jre/bin/java.exe*.

Note Ideally, the JVM you select should be the version that was used to compile your program.

12 Click **OK**, then click **Next**.

The Specify Your Code page appears.

13 In the Category/Program Name text box, type *DemoCategory* (which specifies the name of the category in which you want to include your code), then click **Next**.

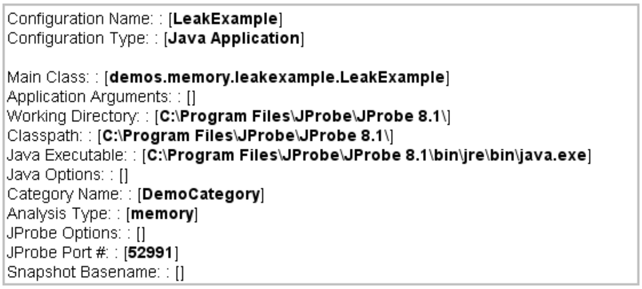
14 In the Select a JProbe Analysis page, ensure that the **Memory** option is selected.

15 On the **Initial Recording** tab, select the **Data Recording Off** check box, then click **Next**.

This disables the data recording at initial JVM start.

16 In the Specify the JProbe Options page, click **Next**.

The Save the Configuration page appears, presenting a summary of the settings defined for your configuration.



```
Configuration Name: : [LeakExample]
Configuration Type: : [Java Application]

Main Class: : [demos.memory.leakexample.LeakExample]
Application Arguments: : []
Working Directory: : [C:\Program Files\JProbe\JProbe 8.1\]
Classpath: : [C:\Program Files\JProbe\JProbe 8.1\]
Java Executable: : [C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe]
Java Options: : []
Category Name: : [DemoCategory]
Analysis Type: : [memory]
JProbe Options: : []
JProbe Port #: : [52991]
Snapshot Basename: : []
```

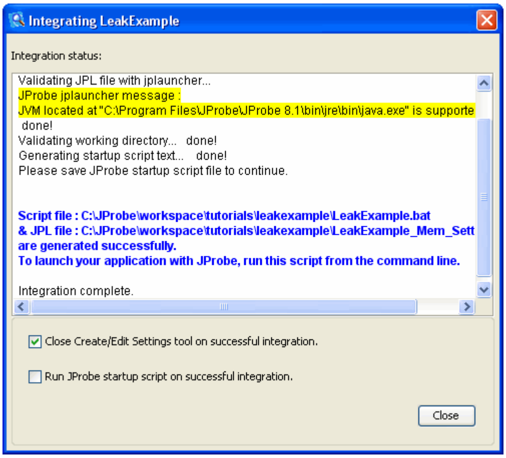
17 Click **Save** and save the configuration file (*LeakExample_Mem_Settings.jpl*) into your working directory.

18 In the Configuration Complete page, select the **Integrate** check box and click **Finish**.

JProbe validates the configuration file and creates a startup script file (for example, in Windows: *LeakExample.bat*, and in Unix/Linux: *LeakExample.sh*).

19 In the Integrating LeakExample dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

The Integrating LeakExample dialog box presents the status of the operation.



Integration status:

```
Validating JPL file with jplauncher...
JProbe jplauncher message :
JVM located at "C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe" is supporte
 done!
Validating working directory...   done!
Generating startup script text...    done!
Please save JProbe startup script file to continue.


Script file : C:\JProbe\workspace\tutorials\leakexample\LeakExample.bat
& JPL file : C:\JProbe\workspace\tutorials\leakexample\LeakExample_Mem_Sett
are generated successfully.
To launch your application with JProbe, run this script from the command line.

Integration complete.
```

☑ Close Create/Edit Settings tool on successful integration.

□ Run JProbe startup script on successful integration.

20 Select both check boxes (**Close Create/Edit Settings tool on successful integration** and **Run JProbe startup script on successful integration**), and click **Close**.

The JProbe Execution Console opens, then the *LeakExample* program starts, displaying a window with **Add** and **Remove** buttons. You are now ready to run a Memory analysis session.

Step 2: Running the Memory Leak Session

In this step, you exercise a use case on *LeakExample* that requires you to add buttons to a panel and then remove them. As the use case runs, you can see how many `JButton` objects have been added since the exercise started and how many remain in the heap when it ends.

Note This procedure assumes that the *LeakExample* program is already running (for instructions on how to execute the startup script, see step 20 in section “Step 1: Setting Up the Memory Leak Session” on page 17.

Alternatively, you can run the startup script from the command line:

In a Windows command window: `>LeakExample.bat`

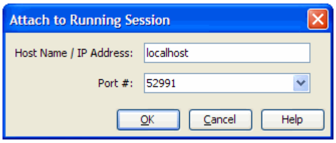
In a Unix or Linux sh shell: `>LeakExample.sh`

In a Unix or Linux csh or ksh shell: `>./LeakExample.sh`

To run the session:

1 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to Running Session dialog box appears, displaying the correct host and port number.



Attach to Running Session

Host Name / IP Address: localhost

Port #: 52991

2 Click **OK**.

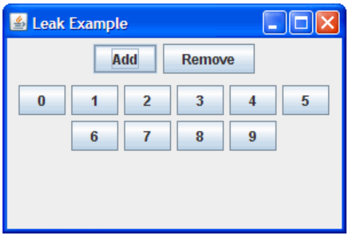
After a few seconds the Runtime Summary view appears, with the Memory Pools tab on the foreground.

3 Click **Set Recording Level** on the toolbar.

4 In the Set Recording Level dialog box, select **Record Allocations and Stack Traces For All Instances**, then click **OK**.

5 In the Leak Example program window, click the **Add** button ten times.

Ten buttons (numbered 0 to 9) are added on the program window.



6 Click the **Remove** button ten times.

The ten buttons are deleted from the program window.

7 Click **Set Recording Level** on the toolbar.

8 In the Set Recording Level dialog box, select **Data Recording Off**, then click **OK**.

JProbe takes a snapshot and displays it in the Snapshot Navigator panel.

9 Close the Leak Example program window.

JProbe disconnects from the running session. After a few seconds, the Instances view appears, displaying instances that were created during the use case.

Step 3: Identifying Loitering Objects

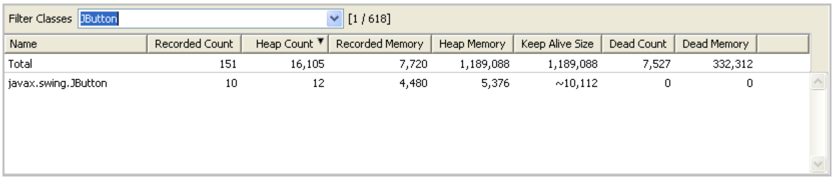
In this step, you look for loitering objects in the heap. Based on the hypothesis, you should expect the count change for the `JButton` class to be zero, because you removed all the buttons you added. In fact, the buttons are not removed and continue to loiter in the heap.

To identify loitering objects:

1. In the Instances view, select **Heap Count** from the Investigate by list.
2. Type `JButton` in the Filter Classes field and press **Enter** to locate the `JButton` class.

   Note This field is case-sensitive.

   The Instances list now displays only the `JButton` class.



Filter Classes JButton [1 / 618]

Name	Recorded Count	Heap Count	Recorded Memory	Heap Memory	Keep Alive Size	Dead Count	Dead Memory
Total	151	16,105	7,720	1,189,088	1,189,088	7,527	332,312
javax.swing.JButton	10	12	4,480	5,376	~10,112	0	0

   The Recorded Count for `JButton` is 10, and the Dead Count is 0. This means that the buttons were removed from the Leak Example program window, but not from the Java heap.

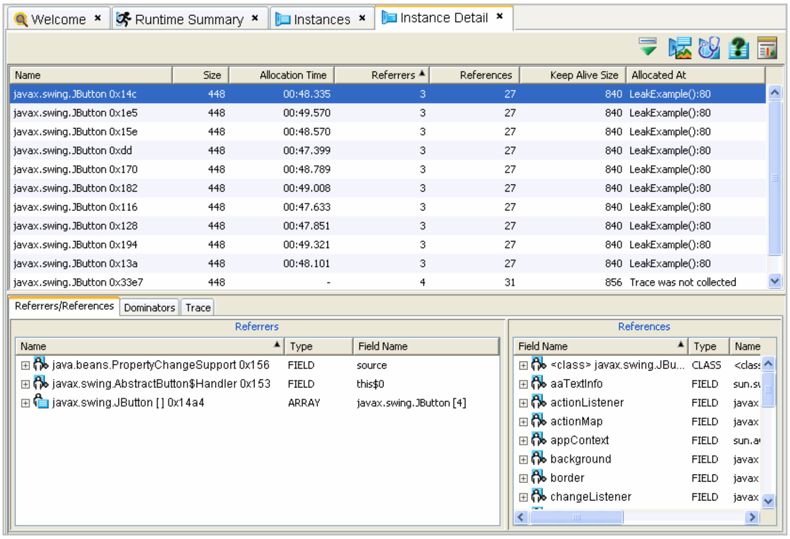
Step 4: Investigating Loitering Objects

In this step, you find the live object that continues to hold a reference to the loitering instances of `JButton`. From the Instances view, you can drill down to more detail on `JButton` in the Instance Detail view. Then you can open the Source view to see the code for the loiterer. You will find that the loitering `JButton` objects are being held by obsolete collection references from the array `JButton[]`.

To investigate loitering objects:

1. In the Instances view, select the `JButton` row and click **Instance Detail** on the toolbar.

The Instance Detail view appears, listing all the instances of `JButton` in the heap.



Name	Size	Allocation Time	Referrers	References	Keep Alive Size	Allocated At
javax.swing.JButton 0x14c	448	00:48.335	3	27	840	LeakExample():80
javax.swing.JButton 0x1e5	448	00:49.570	3	27	840	LeakExample():80
javax.swing.JButton 0x15e	448	00:48.570	3	27	840	LeakExample():80
javax.swing.JButton 0xdd	448	00:47.399	3	27	840	LeakExample():80
javax.swing.JButton 0x170	448	00:48.789	3	27	840	LeakExample():80
javax.swing.JButton 0x182	448	00:49.008	3	27	840	LeakExample():80
javax.swing.JButton 0x116	448	00:47.633	3	27	840	LeakExample():80
javax.swing.JButton 0x128	448	00:47.851	3	27	840	LeakExample():80
javax.swing.JButton 0x194	448	00:49.321	3	27	840	LeakExample():80
javax.swing.JButton 0x13a	448	00:48.101	3	27	840	LeakExample():80
javax.swing.JButton 0x33e7	448	-	4	31	856	Trace was not collected

Referrers

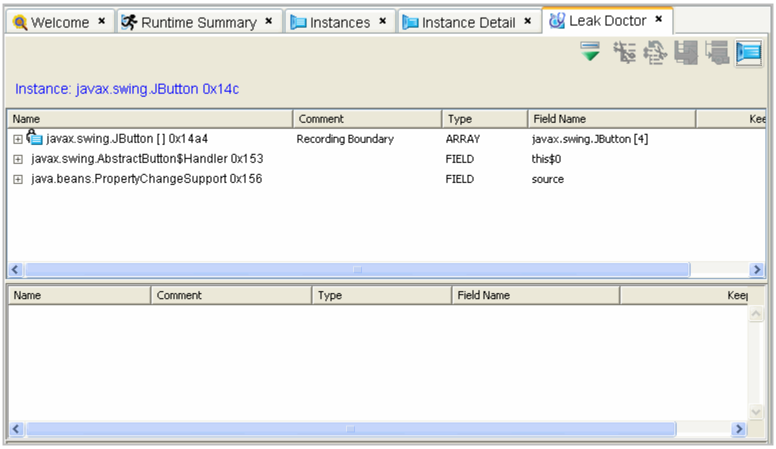
Name	Type	Field Name
java.beans.PropertyChangeSupport 0x156	FIELD	source
javax.swing.AbstractButton$Handler 0x153	FIELD	this$0
javax.swing.JButton [] 0x14a4	ARRAY	javax.swing.JButton [4]

References

Field Name	Type	Name
<class> javax.swing.JBu...	CLASS	<clas
aaTextInfo	FIELD	sun.s
actionListener	FIELD	javax
actionMap	FIELD	javax
appContext	FIELD	sun.a
background	FIELD	javax
border	FIELD	javax
changeListener	FIELD	javax

2 In the upper table, click the first instance with an allocation time displayed, then click **Leak Doctor** on the toolbar.

The Leak Doctor view appears.



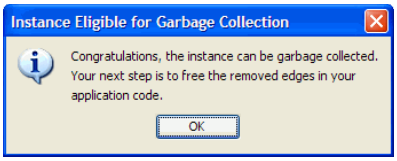
Instance: javax.swing.JButton 0x14c

Name	Comment	Type	Field Name	Kee
javax.swing.JButton [] 0x14a4	Recording Boundary	ARRAY	javax.swing.JButton [4]	
javax.swing.AbstractButton$Handler 0x153		FIELD	this$0	
java.beans.PropertyChangeSupport 0x156		FIELD	source	

Name	Comment	Type	Field Name	Keep

3 Select the first instance and click **Remove The Edge** on the toolbar.

A dialog box confirms that this instance is eligible to be garbage collected if you free the removed edge from your application code.



Congratulations, the instance can be garbage collected. Your next step is to free the removed edges in your application code.

4 Click **OK**.

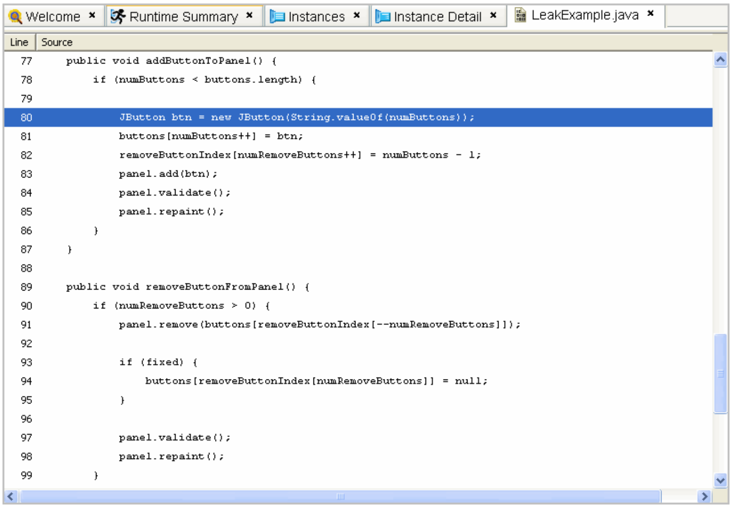
The selected instance moves from the upper table to the lower one. Your next step is to find the method that allocated the instance in order to understand why it is not being removed.

5 Close the Leak Doctor view by clicking the **x** on the tab.

6 In the Instance Detail view, click the **Trace** tab to see the stack trace.

7 Right-click the `LeakExample.addButtonToPanel()` row and select **Show Allocated At Source** to examine the source code for this allocating method.

The Source view displays the *LeakExample.java* source code. The line that allocates the `JButton` is selected. It is located within the `addButtonToPanel()` method. Below this method is the method that is supposed to remove the buttons: `removeButtonFromPanel()`.



```
public void addButtonToPanel() {
    if (numButtons < buttons.length) {

        JButton btn = new JButton(String.valueOf(numButtons));
        buttons[numButtons++] = btn;
        removeButtonIndex[numRemoveButtons++] = numButtons - 1;
        panel.add(btn);
        panel.validate();
        panel.repaint();
    }
}

public void removeButtonFromPanel() {
    if (numRemoveButtons > 0) {
        panel.remove(buttons[removeButtonIndex[--numRemoveButtons]]);

        if (fixed) {
            buttons[removeButtonIndex[numRemoveButtons]] = null;
        }

        panel.validate();
        panel.repaint();
    }
```

Notice that the line of code that removes buttons from the `buttons[]` array is encased in an `if` statement; in essence, it is missing. You have confirmed that the `buttons[]` array is the live object that is holding the loiterers in memory.

Note In Step 5: Running the Memory Leak Session with Improved Code you learn how to remedy this problem, by running the code with the `fix` program argument, thus proving that the original hypothesis is correct.

8 Close the Source view.

9 **Optional**: To find out how much memory is consumed by the loiterer, review the Heap Memory column in the Instances view and the Keep Alive Size column in the Instance Detail view.

Note The **~** sign in the Instances view's Keep Alive Size column indicates an estimated value for this metric. To calculate the actual size, right-click the `JButton` instance and select **Calculate Actual Keep Alive Size** from the list.

You may be surprised at how many instances are held in memory by a single `JButton` instance. The memory consumed by the loiterer and all its anchored instances is about 10,240 bytes; the Keep Alive Size for a `JButton` instance is 856.

Note These numbers may vary depending on which JVM you used to run the Leak Example.

10 Close the Instances and Instance Detail views.

Step 5: Running the Memory Leak Session with Improved Code

In this step, you need to add to the code a line that removes the buttons from the `buttons[]` array. The *LeakExample* demo contains the fixed line of code; you just need to add a program argument to your session settings to activate it. Note that in this example, removing the loiterer does not free all the memory, because the memory calculation includes some recursive references and because other objects in the program continue to need some of the objects referenced by the loiterer.

After you rerun the session with the fixed code, you can clearly see in the Instances view that both the Recorded Count and Dead Count for `JButton` are now 0, as predicted by the hypothesis. However, in a real-life scenario where the effect of changes is more widespread, you may need to compare the snapshots to see all the differences. This tutorial walks you through how to do that comparison.

To verify the fixed code:

1 Click **Tools** > **Create/Edit Settings**.

The Create/Edit Setting dialog box appears.

2 In the Manage Configurations pane, select **LeakExample**.

The configuration settings for the LeakExample appear in the right panel.

3 Click the **Java Application** tab and then click **Edit**.

4 Click the browse button beside the Application Arguments field.

5 In the Application Arguments dialog box, type `fix` in the upper field and click **OK**.

The argument appears in the Application Arguments field.

6 Click **Save** and then **Close**.

7 Run the LeakExample startup script and then follow the instructions in Step 2: Running the Memory Leak Session to exercise the same use case.

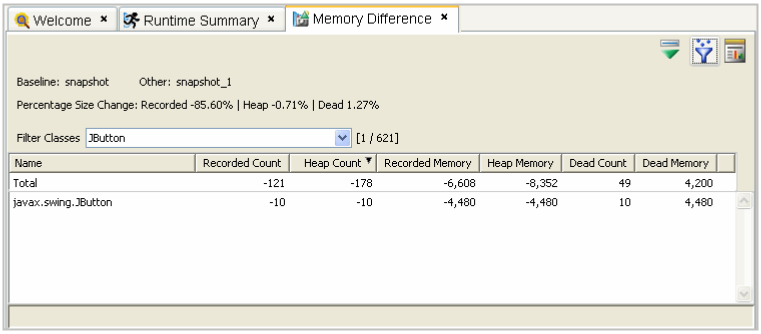
8 In the navigator, right-click the new snapshot with the improved code and select **Snapshot Differencing**.

The Memory Difference dialog box opens. The new snapshot is displayed in the Snapshot to Compare list.

9 Select the original snapshot with the loitering objects from the **Baseline Snapshot** list.

10 Click **OK**.

The Memory Difference view appears, displaying the differences in data between the two snapshots. Use the filters to display only the `JButton` objects.



Baseline: snapshot   Other: snapshot_1

Percentage Size Change: Recorded -85.60% | Heap -0.71% | Dead 1.27%

Filter Classes JButton [1 / 621]

Name	Recorded Count	Heap Count ▼	Recorded Memory	Heap Memory	Dead Count	Dead Memory
Total	-121	-178	-6,608	-8,352	49	4,200
javax.swing.JButton	-10	-10	-4,480	-4,480	10	4,480

You can see that in the new snapshot (obtained by using the improved code) there are ten fewer `JButton` objects than in the original snapshot. The Recorded Count and Heap Count are -10, and the Heap Memory is reduced by approximately 4,480 bytes. Therefore, the code modification fixed the problem.

In the upper side of the view, you can also see that the *LeakExample* code now uses memory more efficiently. The improved code reduced the Recorded Memory use by 85.60%.

Network Tutorial

The *Network* program illustrates how over-allocating short-lived objects can cause the garbage collector to run longer than necessary. Garbage collection takes time and resources.

This tutorial demonstrates how to use JProbe to identify excessive garbage collections in your code. The fixed code shows that it is often more efficient to reuse objects or cache data.

Note **The values cited in this tutorial reflect the *Network* example running on Windows XP with Sun JDK 1.6.0_10.** You may see different values on your system, but the improvement in garbage collection overhead should still be evident.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	*Network.class*
Use Case:	Connect from a client to a server, query a database, and return a result to the client.
Architecture:	See the comments in the *Network.java* source file.
Hypothesis:	The program does not create unnecessary temporary objects.

The tutorial walks you through the following steps:

- Step 1: Setting Up the Network Session
- Step 2: Running the Network Session
- Step 3: Identifying Large Allocations of Short-Lived Objects
- Step 4: Investigating Large Allocations of Short-Lived Objects
- Step 5: Running the Network Session with Improved Code

Step 1: Setting Up the Network Session

To run a garbage collection analysis, you need to set up the session in the JProbe Create/Edit Settings tool. The following procedures mention only the settings that you need to change or verify. If a setting is not mentioned, leave it blank or in its default state. The procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1. Click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.

2. In the Manage Configurations pane, click **Java Application**.

   The JProbe Configuration Wizard appears.

3. Click **Add**.
4. In the Configuration Name text box, type *Network*, then click **Next**.
5. Under Main Class, click **Execute a class**.
6. Click the browse button beside the Main Class field and navigate to the *Network.class* file in the *<jprobe_home>/demos/memory/network* directory.
7. Click **OK**.

   The following information is displayed:

   - Main Class: `demos.memory.network.Network`
   - Working Directory: <*`jprobe_home`*>

8. Click the browse button beside the Classpath field.
9. In the Classpath dialog box, click **Add Working Directory**, then click **OK**.

   The working directory appears in the Classpath field.

10. Click **Next**.

    The Select a Java Virtual Machine page appears.

11. If you want to change the default JVM, click the browse button beside the Java Executable field, select another JVM in the Java Virtual Machines dialog box, then click **OK**.

    Alternatively, you can use the JVM that is installed with JProbe to run with the tutorial. The java executable is *<jprobe_home>/bin/jre/bin/java.exe*.

    Note Ideally, the JVM you select should be the version that was used to compile your program.

12 To ensure accurate allocation methods, disable the just-in-time compiler.

Note This step applies only to Sun or IBM JVMs. It does not apply to JRockit JVM.

a Click the browse button beside the Java Options field.

b In the Java Options dialog box, in the upper field, type `-Xint`.

c Click **Parse Arguments**.

The argument appears in the first line of the lower field.

d Click **OK**.

The Java Options field displays the program argument.

13 Click **Next**.

The Specify Your Code page appears.

14 In the Category/Program Name text box type *DemoCategory* (which specifies the name of the category in which you want to include your code), then click **Next**.

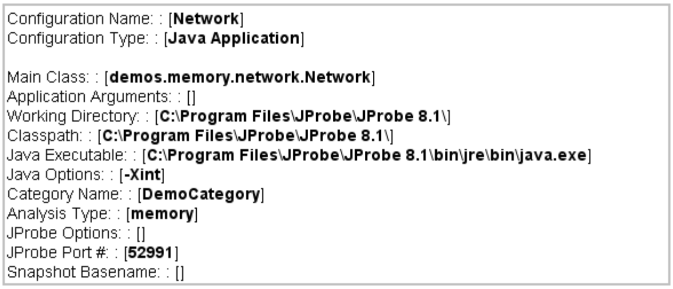
15 In the Select a JProbe Analysis page, ensure that the **Memory** option is selected.

16 On the **Initial Recording** tab, select the **Data Recording Off** check box, then click **Next**.

This disables the data recording at initial JVM start.

17 In the Specify the JProbe Options page, click **Next**.

The Save the Configuration page appears, presenting a summary of the settings defined for your configuration.



```
Configuration Name: : [Network]
Configuration Type: : [Java Application]

Main Class: : [demos.memory.network.Network]
Application Arguments: : []
Working Directory: : [C:\Program Files\JProbe\JProbe 8.1\]
Classpath: : [C:\Program Files\JProbe\JProbe 8.1\]
Java Executable: : [C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe]
Java Options: : [-Xint]
Category Name: : [DemoCategory]
Analysis Type: : [memory]
JProbe Options: : []
JProbe Port #: : [52991]
Snapshot Basename: : []
```

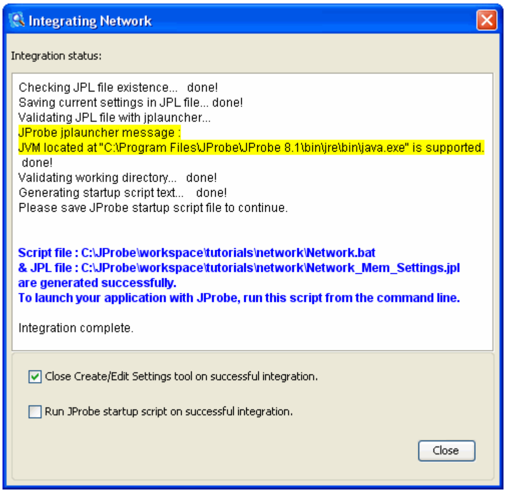
18 Click **Save** and save the configuration file (*Network_Mem_Settings.jpl*) into your working directory.

19 In the Configuration Complete page, select the **Integrate** check box and click **Finish**.

JProbe validates the configuration file and creates a startup script file (for example, in Windows: *Network.bat*, and in Unix/Linux: *Network.sh*).

20 In the Integrating Network dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

The Integrating Network dialog box presents the status of the operation.



Integration status:

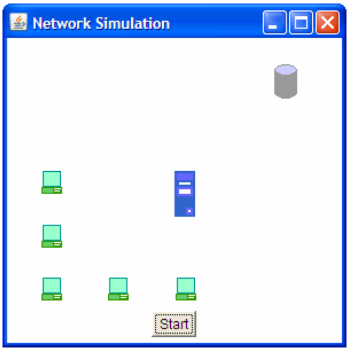
```
Checking JPL file existence...   done!
Saving current settings in JPL file... done!
Validating JPL file with jplauncher...
JProbe jplauncher message :
JVM located at "C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe" is supported.
 done!
Validating working directory...   done!
Generating startup script text...    done!
Please save JProbe startup script file to continue.


Script file : C:\JProbe\workspace\tutorials\network\Network.bat
& JPL file : C:\JProbe\workspace\tutorials\network\Network_Mem_Settings.jpl
are generated successfully.
To launch your application with JProbe, run this script from the command line.

Integration complete.
```

☑ Close Create/Edit Settings tool on successful integration.

☐ Run JProbe startup script on successful integration.

21 Select both check boxes (**Close Create/Edit Settings tool on successful integration** and **Run JProbe startup script on successful integration**), and click **Close**.

The JProbe Execution Console opens, then the *Network Simulation* program starts, displaying a window with icons representing elements of a network. You are now ready to run a Memory analysis session.



Step 2: Running the Network Session

In this step, you exercise the use case on the *Network Simulation* program. All you need to do is to click the **Start** button in the *Network Simulation* program; the program simulates clients (threads) connecting to a server and querying a database. It runs to completion in about one minute (depending on your system setup), generating the data that you need to assess the performance of the garbage collector.

Note This procedure assumes that the *Network Simulation* program is already running (for instructions on how to execute the startup script, see step 21 in section "Step 1: Setting Up the Network Session" on page 27.

Alternatively, you can run the startup script from the command line:

In a Windows command window: `>Network.bat`

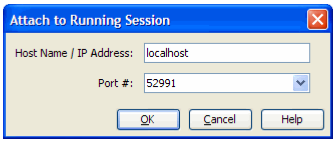
In a Unix or Linux sh shell: `>Network.sh`

In a Unix or Linux csh or ksh shell: `>./Network.sh`

To run the session:

1 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to Running Session dialog box appears, displaying the correct host and port number.



Attach to Running Session

Host Name / IP Address: localhost

Port #: 52991

2 Click **OK**.

After a few seconds the Runtime Summary view appears, with the Memory Pools tab on the foreground.

3 From the Pools and GC Interval list, select **Five Minutes**.

4 Click **Set Recording Level** on the toolbar.

5 In the Set Recording Level dialog box, select **Record Allocations, Stack Traces, and Garbage Data For All Instances**, then click **OK**.

This enables the recording of allocations, traces and garbage collection on all instances.

6 In the Network Simulation program window, click **Start**.

When the red lines disappear in the Network Simulation window, the program has finished.

7 Click **Set Recording Level** on the toolbar.

8 In the Set Recording Level dialog box, select **Data Recording Off**, then click **OK**.

JProbe takes a snapshot and displays it in the Snapshot Navigator panel.

9 In the Network Simulation window, click **Stop**.

10 Close the Network Simulation window.

JProbe disconnects from the running session. After a few seconds, the Instances view appears, displaying instances that were created during the use case.

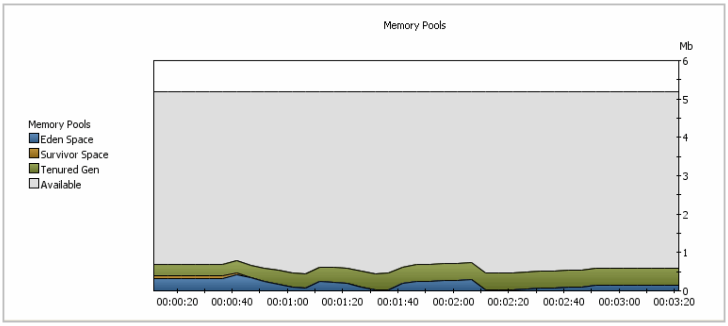
Step 3: Identifying Large Allocations of Short-Lived Objects

The Heap Usage Chart indicates an excessive number of garbage collections.

To identify the short-lived objects:

1 In the Runtime Summary view, click the Memory Pools tab.

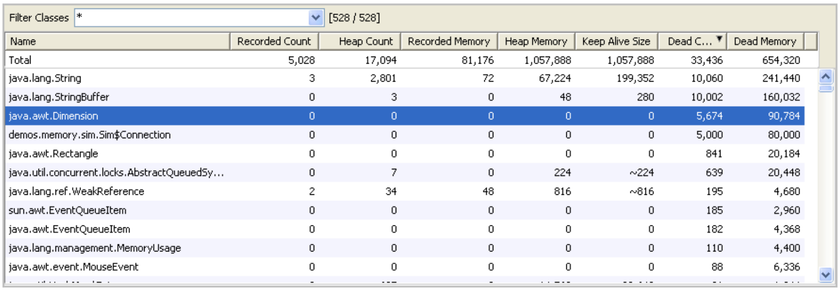
The peaks and valleys in the Memory Pools graph show that the objects being collected are not alive very long before they are garbage collected.



2 In the Instances view, select **Dead Count** from the Investigate by list.

3 Click the **Dead Count** column header twice to sort the table by the classes with the most garbage collected instances.

The top classes by Dead Count instances are `String`, `StringBuffer`, and `Sim$Connection`. None of these instances remain alive at the end of the session.



Filter Classes * [528 / 528]

Name	Recorded Count	Heap Count	Recorded Memory	Heap Memory	Keep Alive Size	Dead C...	Dead Memory
Total	5,028	17,094	81,176	1,057,888	1,057,888	33,436	654,320
java.lang.String	3	2,801	72	67,224	199,352	10,060	241,440
java.lang.StringBuffer	0	3	0	48	280	10,002	160,032
java.awt.Dimension	0	0	0	0	0	5,674	90,784
demos.memory.sim.Sim$Connection	0	0	0	0	0	5,000	80,000
java.awt.Rectangle	0	0	0	0	0	841	20,184
java.util.concurrent.locks.AbstractQueuedSy...	0	7	0	224	~224	639	20,448
java.lang.ref.WeakReference	2	34	48	816	~816	195	4,680
sun.awt.EventQueueItem	0	0	0	0	0	185	2,960
java.awt.EventQueueItem	0	0	0	0	0	182	4,368
java.lang.management.MemoryUsage	0	0	0	0	0	110	4,400
java.awt.event.MouseEvent	0	0	0	0	0	88	6,336

We know that `StringBuffer` is created by `String` objects, so `String` and `Sim$Connection` are good candidates for further investigation.

Step 4: Investigating Large Allocations of Short-Lived Objects

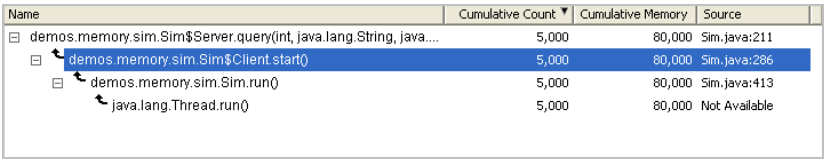
In this step you investigate the short-lived objects identified in Step 3: Identifying Large Allocations of Short-Lived Objects by looking at the source code. Remember that in this example the code contains the fixed code as well. The problem areas are identified in the code comments.

Note To discover where instances are allocated, drill down on an allocation hotspot to display its stack trace in the Merged Allocation Points view. You can then look for your allocating method in the stack trace and drill down on it to see the source code.

To investigate the garbage collected objects:

1 Right-click `Sim$Connection` and select **Open Merged Allocation Points View**.

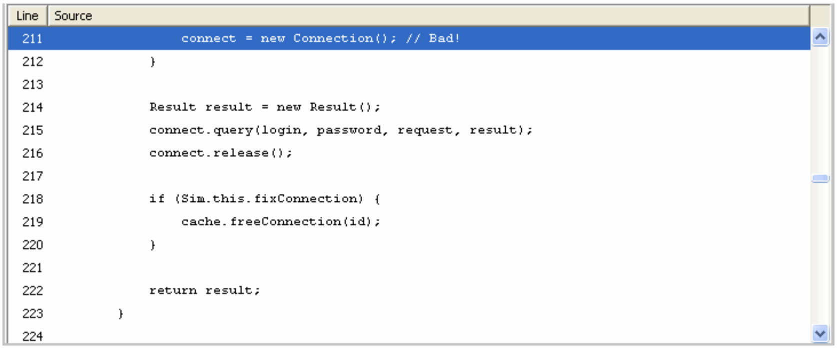
The upper pane of the Merged Allocation Points view displays the `Sim$Server.query` method, which allocates instances of `Sim$Connection`. The Source column indicates the line of code where this method occurs.



Name	Cumulative Count	Cumulative Memory	Source
⊟ demos.memory.sim.Sim$Server.query(int, java.lang.String, java....	5,000	80,000	Sim.java:211
⊟ demos.memory.sim.Sim$Client.start()	5,000	80,000	Sim.java:286
⊟ demos.memory.sim.Sim.run()	5,000	80,000	Sim.java:413
java.lang.Thread.run()	5,000	80,000	Not Available

2 Right-click `Sim$Server.query` and select **Show Allocated At Source**.

The Source view opens, displaying the *Sim.java* source code at the line indicated in the Source column (211).



```
Line  Source
211                 connect = new Connection(); // Bad!
212             }
213
214             Result result = new Result();
215             connect.query(login, password, request, result);
216             connect.release();
217
218             if (Sim.this.fixConnection) {
219                 cache.freeConnection(id);
220             }
221
222             return result;
223         }
224
```

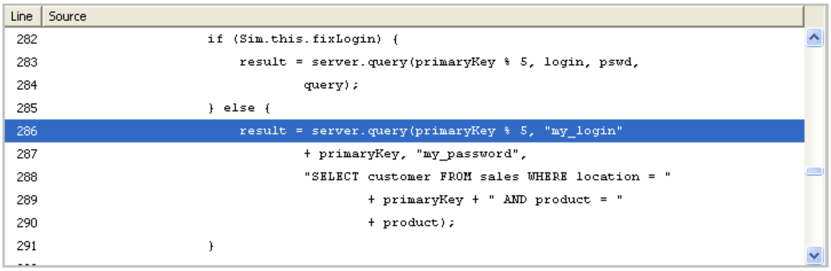
Notice that each time a client sends a query to the server, the server creates a new connection to the database that lasts until the connection is terminated. Also notice that `Connection` is an inner class of `Sim`.

3 In the Instances view, right-click `String` and select **Open Merged Allocation Points View**.

4 In the Merged Allocation Points view, right-click `System Code` and select **Expand To Next Branch Point**.

5 In the lower panel of this view, right-click `Sim$Client.start` and select **Show Allocated At Source**.

The Source view opens, displaying the *Sim.java* source code at the line indicated in the Source column (286).



```
Line  Source
282                 if (Sim.this.fixLogin) {
283                     result = server.query(primaryKey % 5, login, pswd,
284                             query);
285                 } else {
286                     result = server.query(primaryKey % 5, "my_login"
287                             + primaryKey, "my_password",
288                             "SELECT customer FROM sales WHERE location = "
289                                     + primaryKey + " AND product = "
290                                     + product);
291                 }
```

Notice that each time a client sends a query to the server, it creates temporary strings to pass the login, password, and query. This is the source of most of the `String` and `StringBuffer` instances that we saw in the Instances view. You can now proceed with fixing the code.

Step 5: Running the Network Session with Improved Code

Two key problems were discovered in the code: temporary connection objects are created for each connection to the database, and temporary strings are created to pass login, password, and query information. The program contains fixes to reduce the number of temporary objects. You need to add program arguments to use the fixed code. You can run the code with one or both of the fixes.

For the connection issue, use the `-fc` application argument. This problem is solved by implementing a cache. If you review the cache fix in *Sim.java*, you will notice that there are actually three caching options documented: `SimpleCache`, `PoolCache`, and `LocalCache`. By default, the `LocalCache` fix is used. If you change the cache, you need to recompile the program.

For the login issue, use the `-fl` application argument. The problem is solved by introducing static `String` and `StringBuffer` classes that can be reused.

To verify that the fixed code improves memory use:

1. In the JProbe Console, select the snapshot and click **Tools > Create/Edit Settings**.

   The Create/Edit Settings dialog box opens, displaying the settings for the *Network* program. To run the fixed program, you need to add program arguments.

2. Click the **Java Application** tab and click **Edit**.
3. Click the browse button beside the Application Arguments field.
4. In the upper text box in the Application Arguments dialog box, type:

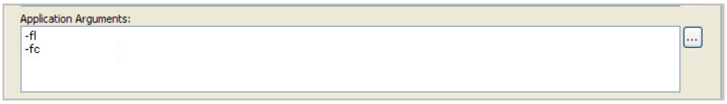
   ```
   -fl -fc
   ```

5. Click **Parse Arguments**.

   The arguments appear in separate lines in the lower list.

6. Click **OK**.

   The Application Arguments field in the Java Application tab displays the new arguments.

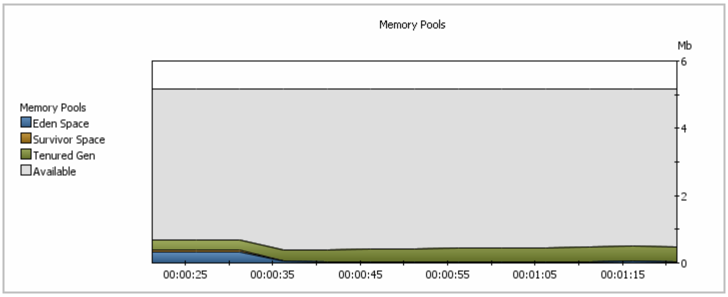
   Application Arguments:

   ```
   -fl
   -fc
   ```

7. Click **Save** and then **Close**.

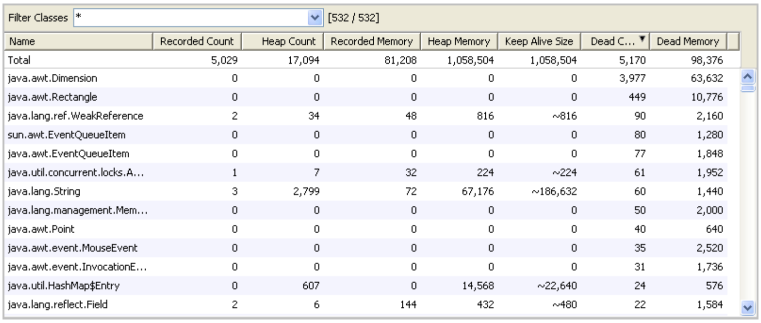
8 Follow the instructions in Step 2: Running the Network Session to exercise the same use case.

In the Memory Pools graph, the peaks and valleys on the graph are less pronounced, which suggests that fewer objects are collected each time the garbage collector runs.



9 In the Instances view, if necessary, select Dead Count from the **Investigate by** list.

10 In the lower pane, click the **Dead Count** column header twice to sort the table by the classes with the most garbage collected instances.



Filter Classes * [532 / 532]

Name	Recorded Count	Heap Count	Recorded Memory	Heap Memory	Keep Alive Size	Dead C...	Dead Memory
Total	5,029	17,094	81,208	1,058,504	1,058,504	5,170	98,376
java.awt.Dimension	0	0	0	0	0	3,977	63,632
java.awt.Rectangle	0	0	0	0	0	449	10,776
java.lang.ref.WeakReference	2	34	48	816	~816	90	2,160
sun.awt.EventQueueItem	0	0	0	0	0	80	1,280
java.awt.EventQueueItem	0	0	0	0	0	77	1,848
java.util.concurrent.locks.A...	1	7	32	224	~224	61	1,952
java.lang.String	3	2,799	72	67,176	~186,632	60	1,440
java.lang.management.Mem...	0	0	0	0	0	50	2,000
java.awt.Point	0	0	0	0	0	40	640
java.awt.event.MouseEvent	0	0	0	0	0	35	2,520
java.awt.event.InvocationE...	0	0	0	0	0	31	1,736
java.util.HashMap$Entry	0	607	0	14,568	~22,640	24	576
java.lang.reflect.Field	2	6	144	432	~480	22	1,584

The Dead Count of `String` now reports 60 instances, down from more than 10,000 instances in the original example. The total number of Dead instances has also decreased, from 33,000 in original example to 5,100 in the fixed code.

You will notice the following:

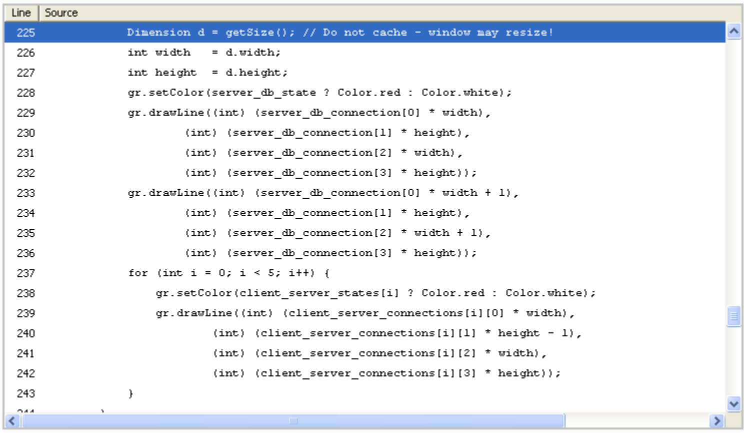
- The `String` count is much lower (60).
- The `Sim$Connection` count is even lower (5).
- `java.awt.Dimension` now has the highest dead count.

11 To investigate `java.awt.Dimension`, right-click it and select **Open Merged Allocation Points View**.

12 In the upper pane of the Merged Allocation Points view, right-click **Sytem Code** end select **Expand To Next Branch Point**.

13 Right-click **My Code** and select **Replace Category With More Detail**.

14 Right-click the class with the large number of instances (3,977) and select **Show Allocated At Source**.



```
        Dimension d = getSize(); // Do not cache - window may resize!
        int width   = d.width;
        int height  = d.height;
        gr.setColor(server_db_state ? Color.red : Color.white);
        gr.drawLine((int) (server_db_connection[0] * width),
                (int) (server_db_connection[1] * height),
                (int) (server_db_connection[2] * width),
                (int) (server_db_connection[3] * height));
        gr.drawLine((int) (server_db_connection[0] * width + 1),
                (int) (server_db_connection[1] * height),
                (int) (server_db_connection[2] * width + 1),
                (int) (server_db_connection[3] * height));
        for (int i = 0; i < 5; i++) {
            gr.setColor(client_server_states[i] ? Color.red : Color.white);
            gr.drawLine((int) (client_server_connections[i][0] * width),
                    (int) (client_server_connections[i][1] * height - 1),
                    (int) (client_server_connections[i][2] * width),
                    (int) (client_server_connections[i][3] * height));
        }
    }
```

You can see that the instances are the result of a `getSize` call on the main *Network Simulation* window. Because it can be resized, the dimensions should not be cached.

2

Performance Analysis Demos

This chapter provides a summary of the Performance demo applications that ship with JProbe and contains tutorials for some of these applications.

The source code and compiled classes for the Performance demos are located in the *<jprobe_home>/demos/performance* directory.

This chapter contains the following sections:

Summary of Demos for Performance

The following table describes the purpose of the example applications.

Java SE Application	Purpose	More Information
Diner.class	The application hangs and does not terminate. This example shows how you can use JProbe to identify the threads involved in a deadlock.	See the notes in *Diner.java*. **Tutorial:** "Philosopher's Diner Tutorial" on page 40.
Files.class	Compares the performance of two algorithms: a buffered Reader/Writer versus an unbuffered DataInputStream/ Data OutputStream.	See the notes in *Files.java*.
MethodCalls.class	This application demonstrates the JProbe's ability to track method calls by allowing the user to control the number of calls to specific methods. Each button has a corresponding `ActionListener`. When a button is pressed, the `actionPerformed` method displays a message in the text area. The user can see how Performance tracks method calls by comparing Performance's reported number of calls to each method and the data displayed in the text area.	See the notes in *MethodCalls.java*.
Objects.class	Compares the performance of using objects versus primitives.	See the notes in *Objects.java*.

Java SE Application	Purpose	More Information
Polynomial.class	The application calculates a polynomial expression using one of two algorithms. This example shows how you can use JProbe to compare performance and identify the more efficient algorithm.	See the notes in *Polynomial.java*. **Tutorial:** "Polynomial Tutorial" on page 50
Strings.class	This application uses two alternative approaches to strip embedded tabs from strings. You can track the two methods and compare their performance.	See the notes in *Strings.java*.

JProbe also ships with a Java EE demo application called JProbe Game Pack Demo for Java EE. For more information, see "JProbe Game Pack for JavaEE" on page 73.

Philosopher’s Diner Tutorial

In this tutorial, you investigate a deadlock situation. The symptom of the problem is that the program hangs and does not terminate.

Based on the class Dining Philosophers deadlock demonstration, the tutorial program simulates five Philosophers seated around a table, each with a bowl of rice in front of him or her. To eat their rice, there are only five chopsticks available to share among the Philosophers. To eat the rice, a Philosopher must have two chopsticks. Once a Philosopher is finished using a chopstick, that chopstick is available to any other Philosopher seated at the table. There is no prescribed sharing pattern among the Philosophers; sharing is random. The eventual result of this random sharing is a deadlock, when each Philosopher waits indefinitely for another chopstick to become available.

This tutorial illustrates how you can detect where a thread causing a deadlock is created within your own code.

Causing Threads

A causing thread is one that is directly responsible for the deadlock. For example, a situation might be that Philosopher 4 and Philosopher 2 are causing the deadlock because Philosopher 4 is waiting for Philosopher 2 and vice versa.

Affected Threads

An affected thread is one that cannot make progress and is not part of the deadlock cycle. Typically it is waiting for either a thread that is part of the cycle or another affected thread. For example, a situation might be that Philosopher 0 is affected because he/she is waiting for the chopstick held by Philosopher 2. Due to the fact that Philosopher 2 and Philosopher 4 are deadlocked, the chopstick never becomes available and Philosopher 0 waits indefinitely. Similarly, Philosopher 1 and Philosopher 3 are waiting for Philosopher 0, but because Philosopher 0 cannot acquire a chopstick, neither they would acquire any chopsticks.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	*Diner.class*
Use Case:	Detect threads involved in the deadlock and identify the location for code modifications.

Architecture: The threads are located in the `demos.performance.diners.philosopher.run()` method.

1 Run the tutorial first using the following filter setting:

```
Action = Method Level
```

2 Stop the executing *Diner* demo and add a filter for `demos.performance.diners.philosopher.run()` with the following detail:

```
Action = Line Level
```

Hypothesis: Using filters can help you identify where a deadlock occurs in the code.

The tutorial walks you through the following steps:

- Step 1: Setting Up the Diner Session
- Step 2: Running the Diner Session
- Step 3: Investigating the Deadlock
- Step 4: Running the Diner Session with Improved Filters
- Step 5: Finding the Cause of the Deadlock in the Source Code

Step 1: Setting Up the Diner Session

In this step, you set up a session to detect deadlock situations. You use the JProbe Create/Edit Settings tool to create the session settings. The following procedures mention only the settings that you need to change or verify. If a setting is not mentioned, leave it blank or in its default state. The procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1 Click **Tools** > **Create/Edit Settings**.

The Create/Edit Setting dialog box appears.

2 In the Manage Configurations pane, click **Java Application**.

The JProbe Configuration Wizard appears.

3 Click **Add**.

4 In the Configuration Name text box, type *Diner*, then click **Next**.

5 In the Main Class field, select the **Execute a class** check box.

6 Click the browse button beside the Main Class field and navigate to the *Diner.class* file in the *<jprobe_home>/demos/performance/diners* directory.

7 Click **OK**.

The following information is displayed:

- Main Class: `demos.performance.diners.Diner`
- Working Directory: <*`jprobe_home`*>

8 Click the browse button beside the Classpath field.

9 In the Classpath dialog box, click **Add Working Directory**, then click **OK**.

The working directory appears in the Classpath field.

10 Click **Next**.

The Select a Java Virtual Machine page appears.

11 If you want to change the default JVM, click the browse button beside the Java Executable field, select another JVM in the Java Virtual Machines dialog box, and click **OK**.

Note Ideally, the JVM you select should be the version that was used to compile your program.

12 Click **Next**.

The Specify Your Code page appears.

13 In the Category/Program Name text box type *DemoCategory* (which specifies the name of the category in which you want to include your code), then click **Next**.

14 In the Select a JProbe Analysis page, ensure that the **Performance** option is selected.

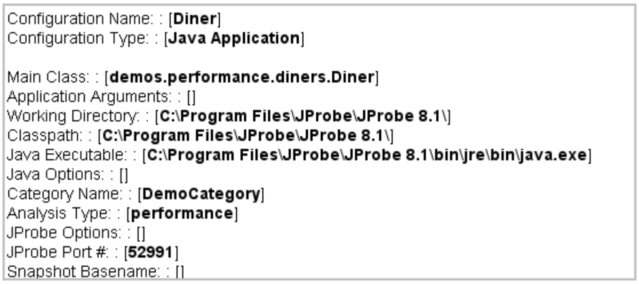
15 On the **General** tab, select the **Detect Deadlocks** check box.

16 On the **Automation** tab, select the **Data Recording Off** check box, then click **Next**.

This disables the data recording at initial JVM start.

17 In the Specify the JProbe Options page, click **Next**.

The Save the Configuration page appears, presenting a summary of the settings defined for your configuration.



```
Configuration Name: : [Diner]
Configuration Type: : [Java Application]

Main Class: : [demos.performance.diners.Diner]
Application Arguments: : []
Working Directory: : [C:\Program Files\JProbe\JProbe 8.1\]
Classpath: : [C:\Program Files\JProbe\JProbe 8.1\]
Java Executable: : [C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe]
Java Options: : []
Category Name: : [DemoCategory]
Analysis Type: : [performance]
JProbe Options: : []
JProbe Port #: : [52991]
Snapshot Basename: : []
```

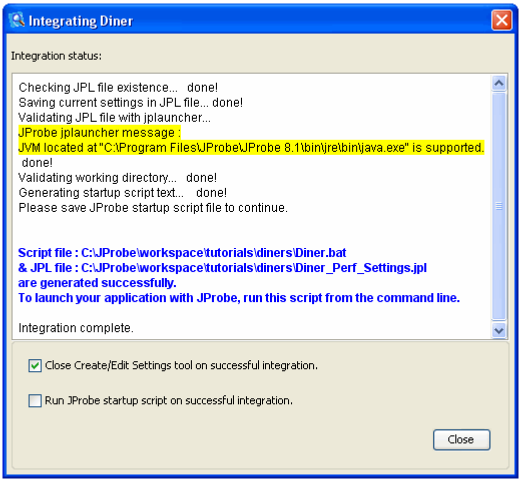
18 Click **Save** and save the configuration file (*Diner_Perf_Settings.jpl*) into your working directory.

19 In the Configuration Complete page, select the **Integrate** check box and click **Finish**.

JProbe validates the configuration file and creates a startup script file (for example, in Windows: *Diner.bat*, and in Unix/Linux: *Diner.sh*).

20 In the Integrating Diner dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

The Integrating Diner dialog box presents the status of the operation.



Integration status:

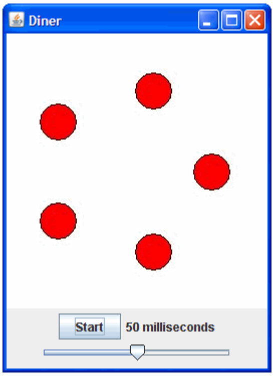
```
Checking JPL file existence...   done!
Saving current settings in JPL file... done!
Validating JPL file with jplauncher...
JProbe jplauncher message :
JVM located at "C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe" is supported.
 done!
Validating working directory...   done!
Generating startup script text...    done!
Please save JProbe startup script file to continue.


Script file : C:\JProbe\workspace\tutorials\diners\Diner.bat
& JPL file : C:\JProbe\workspace\tutorials\diners\Diner_Perf_Settings.jpl
are generated successfully.
To launch your application with JProbe, run this script from the command line.

Integration complete.
```

☑ Close Create/Edit Settings tool on successful integration.

☐ Run JProbe startup script on successful integration.

21 Select both check boxes (**Close Create/Edit Settings tool on successful integration** and **Run JProbe startup script on successful integration**), and click **Close**.

The JProbe Execution Console opens, then the *Diner* program starts, displaying a window with icons representing the philosophers. The slider controls the number of milliseconds that a Philosopher waits (“sleeps”) between chopstick attempts (“eating”). You are now ready to run a Performance analysis session.



Step 2: Running the Diner Session

In this step, you start the session from the command line. Let the Diner program run until a deadlock occurs. That said, in some cases a deadlock may not occur because the programmed behavior of the philosophers is random. The program is sensitive to timing on the computer and in the JVM.

Note This procedure assumes that the *Diner* program is already running (for instructions on how to execute the startup script, see step 21 in section “Step 1: Setting Up the Diner Session” on page 41.

Alternatively, you can run the startup script from the command line:

In a Windows command window: `>Diner.bat`

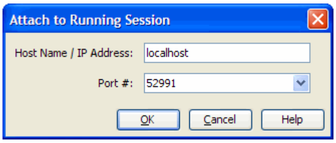
In a Unix or Linux sh shell: `>Diner.sh`

In a Unix or Linux csh or ksh shell: `>./Diner.sh`

To run the session:

1 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to Running Session dialog box appears, displaying the correct host and port number.



Host Name / IP Address: localhost

Port #: 52991

2 Click **OK**.

After a few seconds the Runtime Summary view appears, with the Memory Pools tab on the foreground.

3 Click **Set Recording Level** on the toolbar.

4 In the Set Recording Level dialog box, select **Full Encapsulation**, then click **OK**.

This enables JProbe to collect data for all methods and the methods they call.

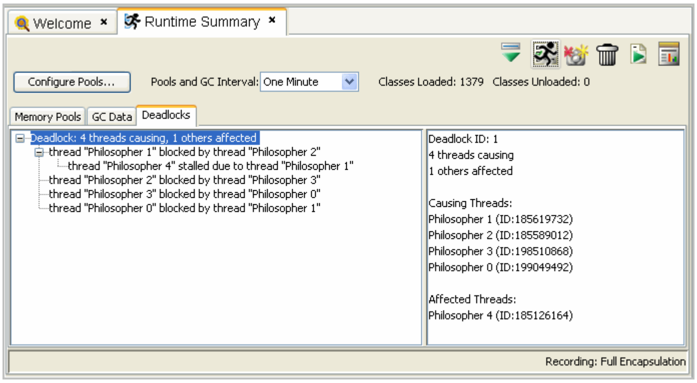
5 In the *Diner* program window, click **Start**.

The program stops when a deadlock occurs because each philosopher has only one chopstick.

Tip If a deadlock occurs immediately, stop the program (click **Stop**) and start it again (click **Start**).

Tip If a deadlock does not occur at all, click **Stop**, adjust the slider to a lower value, then click **Start** to rerun the program. Repeat until a deadlock occurs.

When the deadlock occurs, the Runtime Summary view appears with the Deadlocks tab on the foreground.



Pools and GC Interval: One Minute

Classes Loaded: 1379 Classes Unloaded: 0

```
Deadlock: 4 threads causing, 1 others affected
  thread "Philosopher 1" blocked by thread "Philosopher 2"
    thread "Philosopher 4" stalled due to thread "Philosopher 1"
  thread "Philosopher 2" blocked by thread "Philosopher 3"
  thread "Philosopher 3" blocked by thread "Philosopher 0"
  thread "Philosopher 0" blocked by thread "Philosopher 1"
```

```
Deadlock ID: 1
4 threads causing
1 others affected

Causing Threads:
Philosopher 1 (ID:185619732)
Philosopher 2 (ID:185589012)
Philosopher 3 (ID:198510868)
Philosopher 0 (ID:199049492)

Affected Threads:
Philosopher 4 (ID:185126164)
```

Recording: Full Encapsulation

6 Click **Set Recording Level** on the toolbar.

7 In the Set Recording Level dialog box, select **Data Recording Off**, then click **OK**.

JProbe takes a snapshot and displays it in the Snapshot Navigator panel.

8. Close the *Diner* program window.

   JProbe disconnects from the running session. After a few seconds, the Call Graph view appears.

9. Close the Call Graph view by clicking the “x” on the tab; you do not need this view to investigate deadlocks.

Step 3: Investigating the Deadlock

As presented in Step 2: Running the Diner Session, the JProbe Execution Console reported a deadlock among four threads. Information about the deadlocks is contained in the Deadlocks tab of the Runtime Summary view.

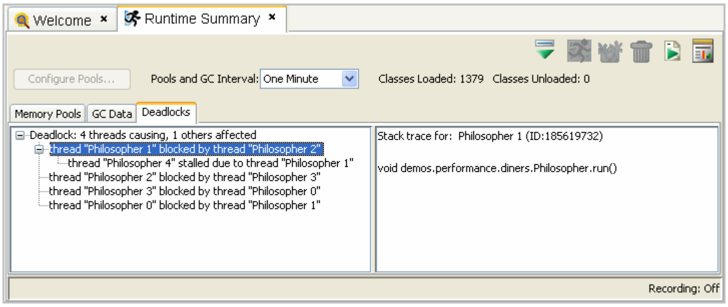
To see the deadlock information:

1. In the Runtime Summary view, click the **Deadlocks** tab (if not selected).

   You can see that four threads are causing the deadlock, while one thread is affected by the deadlock.

   Note The thread IDs change each time you run this program.

2. Select a thread in the left panel.



Pools and GC Interval: One Minute

Classes Loaded: 1379 Classes Unloaded: 0

```
⊟ Deadlock: 4 threads causing, 1 others affected
   ⊟ thread "Philosopher 1" blocked by thread "Philosopher 2"
        thread "Philosopher 4" stalled due to thread "Philosopher 1"
     thread "Philosopher 2" blocked by thread "Philosopher 3"
     thread "Philosopher 3" blocked by thread "Philosopher 0"
     thread "Philosopher 0" blocked by thread "Philosopher 1"
```

```
Stack trace for:  Philosopher 1 (ID:185619732)

void demos.performance.diners.Philosopher.run()
```

Recording: Off

   The stack trace for that thread is displayed. The thread is created by the `Philosopher.run()` method. If you select any of the other threads, you find that they are also created by the same method.

Step 4: Running the Diner Session with Improved Filters

You have identified that the threads are created by the `run()` method in the *Philosopher* class. In this step, you first edit the session settings to focus on the `run()` method, then run the *Diner* program again. You will now be able to identify in the source code the line number where the thread is created.

To edit the configuration settings and re-run the session:

1. Click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.

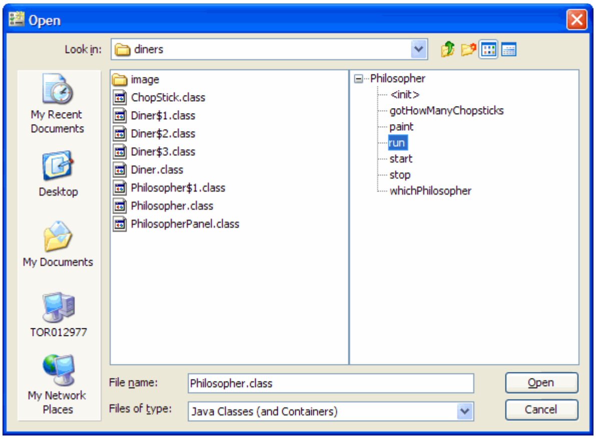
2. In the Manage Configurations pane, select **Diner** and then click **Edit**.

3. Click the **Analysis Type** tab, then the **Filters** tab.

   You are now going to select a filter type that allows you to identify the causing thread.

4. Click in the row below the existing filter and click the browse button.

5. Navigate to *<jprobe_home>/demos/performance/diners/Philosopher.class*, expand its method list, and select **run**.



Look in: diners

image
ChopStick.class
Diner$1.class
Diner$2.class
Diner$3.class
Diner.class
Philosopher$1.class
Philosopher.class
PhilosopherPanel.class

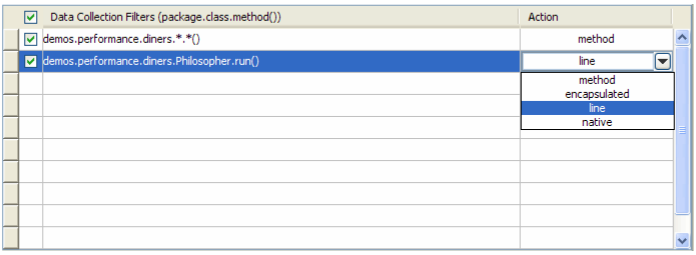
Philosopher
- <init>
- gotHowManyChopsticks
- paint
- run
- start
- stop
- whichPhilosopher

File name: Philosopher.class
Files of type: Java Classes (and Containers)

6. Click **Open**.

   The method is displayed in the Data Collection Filter list.

7. Click on the Action cell for this row and select **line**.



☑ Data Collection Filters (package.class.method())	Action
☑ demos.performance.diners.*.*()	method
☑ demos.performance.diners.Philosopher.run()	line

8 Click **Save**.

You can now rerun the session to find out the line number in the source code where the thread is created.

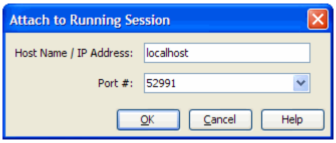
9 Click **Run**, then click **OK** in the Run JProbe Configuration dialog box.

The JProbe Execution Console opens, then the *Diner* program starts, displaying a window with icons representing the philosophers. The slider controls the number of milliseconds that a Philosopher waits (“sleeps”) between chopstick attempts (“eating”). You are now ready to run a Performance analysis session.

10 Close the Create/Edit Settings dialog box.

11 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to a Running Session dialog box appears, displaying the correct host and port number.



Attach to Running Session

Host Name / IP Address: localhost

Port #: 52991

12 Click **OK**.

After a few seconds the Runtime Summary view appears, with the Memory Pools tab on the foreground.

13 Click **Set Recording Level** on the toolbar.

14 In the Set Recording Level dialog box, select **Full Encapsulation**, then click **OK**.

This enables JProbe to collect data for all methods and the methods they call.

15 In the *Diner* program window, click **Start**.

When the deadlock occurs, the Runtime Summary view appears with the Deadlocks tab on the foreground.

16 Close the *Diner* program window.

JProbe disconnects from the running session. After a few seconds, the Call Graph view appears.

Step 5: Finding the Cause of the Deadlock in the Source Code

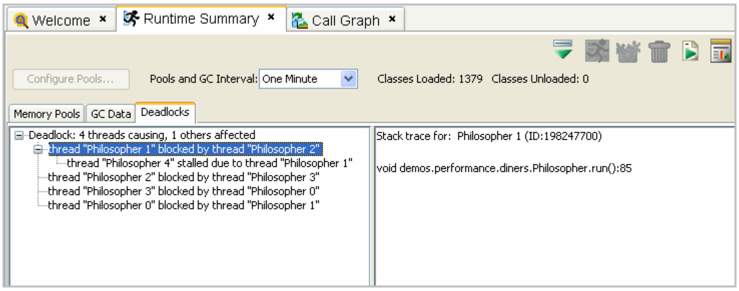
The filter causes line numbers to be appended to the methods in the stack trace. This makes it easy to locate the problem area in the code.

To identify the affected threads in stack traces:

1 In the Runtime Summary view, click the **Deadlocks** tab.

2 Select a thread in the left panel.

This time the method (shown in the right panel) has a number appended to it (`85`). The number represents a line number in the source code.



Pools and GC Interval: One Minute

Classes Loaded: 1379 Classes Unloaded: 0

```
Deadlock: 4 threads causing, 1 others affected
    thread "Philosopher 1" blocked by thread "Philosopher 2"
        thread "Philosopher 4" stalled due to thread "Philosopher 1"
    thread "Philosopher 2" blocked by thread "Philosopher 3"
    thread "Philosopher 3" blocked by thread "Philosopher 0"
    thread "Philosopher 0" blocked by thread "Philosopher 1"
```

```
Stack trace for:  Philosopher 1 (ID:198247700)

void demos.performance.diners.Philosopher.run():85
```

3 In the Call Graph view, right-click the method `demos.performance.diners.Philosopher.run()` and select **Show Source**.

The Source view appears, with line number `85` in red. You can see that the `synchronized (ch1)` block is causing the deadlock.

Polynomial Tutorial

The *Polynomial* tutorial illustrates how an inefficient algorithm can significantly impact the performance of your code, and how you can use JProbe to compare performance and identify the more efficient algorithm.

Tip You only want to optimize algorithms in the critical path of your program; there is no point tuning algorithms that are called very rarely. You also need to evaluate the overall impact on the runtime of the program. If an inefficient algorithm takes a total of a few seconds to execute, you may make it run faster, but the impact on the overall runtime of the program will be negligible.

The program runs two algorithms for computing the following polynomial expression:

$a_n * x^n + ... + a_2 * x^2 + a_1 * x + a_0$, where n=750

The original algorithm uses nested loops to calculate x^n. The improved algorithm implements Horner’s Rule for evaluating polynomial expressions, that is, it factors out powers of x in the form:

$(((...(a_n * x + a_{n-1})...) x + a_2) x + a_1) + a_0$

Note **The values cited in this tutorial reflect the *Polynomial* running on Windows XP with Sun JDK 1.6.0_10**. You may see different values on your system, but the performance improvement between the algorithms should still be evident.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	*Polynomial.class*
Use Case:	Calculate a polynomial expression.
Architecture:	Both polynomial calculations are in the `evaluate()` method. To run the program, you need to set a program argument: • `N` = Use nested loops (original algorithm) • `H` = Use Horner’s Rule (alternate algorithm)
Hypothesis:	Horner’s Rule is faster.

The tutorial walks you through the following steps:

- Step 1: Setting Up the Polynomial Session

- Step 2: Running the Polynomial Session
- Step 3: Identifying and Investigating the Performance Bottleneck
- Step 4: Running the Polynomial with Improved Code
- Step 5: Measuring the Performance Improvement

Step 1: Setting Up the Polynomial Session

To collect timing data on the original algorithm, you need to set up the session in the JProbe Create/Edit Settings tool. The following procedures mention only the settings that you need to change or verify. If a setting is not mentioned, leave it blank or in its default state. The procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1. Click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.

2. In the Manage Configurations pane, click **Java Applications**.

   The JProbe Configuration Wizard appears.

3. Click **Add**.
4. In the Configuration Name box, type *Polynomial*, then click **Next**.
5. Under Main Class, click **Execute a Class**.
6. Click the browse button beside the Main Class field and navigate to the *Polynomial.class* file in the *<jprobe_home>/demos/performance/polynomial* directory.
7. Click **OK**.

   The following information is displayed:

   - Main Class: `demos.performance.polynomial.Polynomial`
   - Working Directory: `<`*`jprobe_home`*`>`

8. To use the nested loop algorithm, you need to enter a program argument.

   a. Click the browse button beside the Application Arguments field.

   b. In the upper field of the Application Arguments dialog, type: `N`

   c. Click **Parse Arguments**.

The argument appears in the first line of the lower field.

d Click **OK**.

The Application Arguments field displays the program argument.

9 Click the browse button beside the Classpath field.

10 In the Classpath dialog box, click **Add Working Directory**, then click **OK**.

The working directory appears in the Classpath field.

11 Click **Next**.

The Select a Java Virtual Machine page appears.

12 If you want to change the default JVM, click the browse button beside the Java Executable field, select another JVM in the Java Virtual Machines dialog box, and click **OK**.

Note Ideally, the JVM you choose should be the version that was used to compile your program.

13 Click **Next**.

The Specify Your Code page appears.

14 In the Category/Program Name text box type *DemoCategory* (which specifies the name of the category in which you want to include your code), then click **Next**.

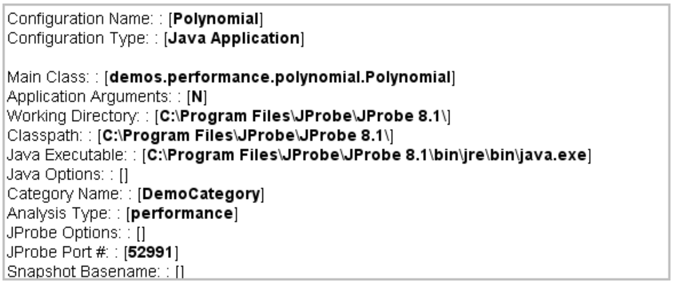
15 In the Select a JProbe Analysis page, ensure that the **Performance** option is selected.

16 On the **Automation** tab, select the **Full Encapsulation** check box, then click **Next**.

This enables JProbe to collect data for all methods and the methods they call, from the initial JVM start.

17 In the Specify the JProbe Options page, click **Next**.

The Save the Configuration page appears, presenting a summary of the settings defined for your configuration.



```
Configuration Name: : [Polynomial]
Configuration Type: : [Java Application]

Main Class: : [demos.performance.polynomial.Polynomial]
Application Arguments: : [N]
Working Directory: : [C:\Program Files\JProbe\JProbe 8.1\]
Classpath: : [C:\Program Files\JProbe\JProbe 8.1\]
Java Executable: : [C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe]
Java Options: : []
Category Name: : [DemoCategory]
Analysis Type: : [performance]
JProbe Options: : []
JProbe Port #: : [52991]
Snapshot Basename: : []
```

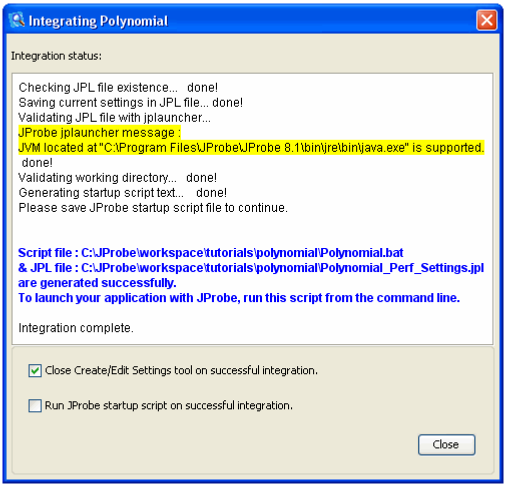
18 Click **Save** and save the configuration file (*Polynomial_Perf_Settings.jpl*) into your working directory.

19 In the Configuration Complete page, select the **Integrate** check box and click **Finish**.

JProbe validates the configuration file and creates a startup script file (for example, in Windows: *Polynomial.bat*, and in Unix/Linux: *Polynomial.sh*).

20 In the Integrating Polynomial dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

The Integrating Polynomial dialog box presents the status of the operation.



```
Checking JPL file existence...   done!
Saving current settings in JPL file... done!
Validating JPL file with jplauncher...
JProbe jplauncher message :
JVM located at "C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe" is supported.
 done!
Validating working directory...   done!
Generating startup script text...   done!
Please save JProbe startup script file to continue.


Script file : C:\JProbe\workspace\tutorials\polynomial\Polynomial.bat
& JPL file : C:\JProbe\workspace\tutorials\polynomial\Polynomial_Perf_Settings.jpl
are generated successfully.
To launch your application with JProbe, run this script from the command line.

Integration complete.
```

☑ Close Create/Edit Settings tool on successful integration.
□ Run JProbe startup script on successful integration.
Close

21 Select the **Close Create/Edit Settings tool on successful integration** check box, and click **Close**.

You are now ready to run a Performance analysis session.

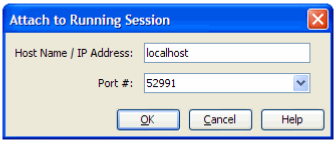
Step 2: Running the Polynomial Session

In this step, you exercise the use case on the *Polynomial* program. All you need to do is to start the *Polynomial* program. It runs to completion in about one minute (depending on your system setup), generating the data that you need to assess the performance of the “nested loops” algorithm.

To run the session:

1 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to a Running Session dialog box appears, displaying the correct host and port number.



Attach to Running Session

Host Name / IP Address: localhost

Port #: 52991

2 Click **OK**.

The Connection Indicator dialog box indicates that JProbe is looking for your session.

3 Start the *Polynomial* program from the command line:

- In a Windows command window: `>Polynomial.bat`
- In a Unix or Linux sh shell: `>Polynomial.sh`
- In a Unix or Linux csh or ksh shell: `>./Polynomial.sh`

The *Polynomial* program starts and runs in a command window.

When the program is finished, the command window closes. JProbe takes a performance snapshot and displays it in the Snapshot Navigator panel and in the Call Graph view.

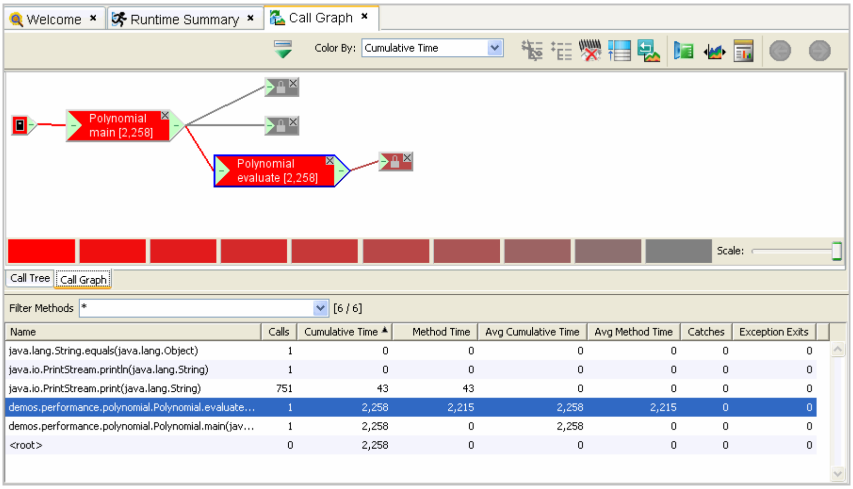
Step 3: Identifying and Investigating the Performance Bottleneck

By default, the snapshot taken at the end of the session is selected and the Call Graph window opens automatically.

Note This simple program contains only the problem algorithm. In a real world program, you would have to locate the target algorithm in the graph or method list. The Filter Methods field can help you narrow down the number of methods displayed. Include the full *package.class*.

1 Select the `Polynomial.evaluate()` method in the graph or the list.

The method is highlighted in both the graph and the list.



Filter Methods * [6 / 6]

Name	Calls	Cumulative Time	Method Time	Avg Cumulative Time	Avg Method Time	Catches	Exception Exits
java.lang.String.equals(java.lang.Object)	1	0	0	0	0	0	0
java.io.PrintStream.println(java.lang.String)	1	0	0	0	0	0	0
java.io.PrintStream.print(java.lang.String)	751	43	43	0	0	0	0
demos.performance.polynomial.Polynomial.evaluate...	1	2,258	2,215	2,258	2,215	0	0
demos.performance.polynomial.Polynomial.main(jav...	1	2,258	0	2,258	0	0	0
<root>	0	2,258	0	0	0	0	0

The method was called only once and took 2,258 milliseconds to execute.

Note This value may vary when running this tutorial on a different platform. The actual time is less important than the comparative difference between the two algorithms.

2 In the Snapshot Navigator, right-click the snapshot, select **Save Snapshot As**, name the snapshot *Polynomial*, and click **Save**.

3 Close the Call Graph view.

Step 4: Running the Polynomial with Improved Code

The original algorithm is slower than expected. Based on the hypothesis, the algorithm that implements Horner’s Rule should run faster.

To use the Horner’s Rule algorithm, you need a new program argument. You can either edit the existing configuration or create a separate configuration to make it easy to switch between tests. In this tutorial, you create a copy of the existing configuration and edit it.

To run and assess the Horner's Rule algorithm:

1. Click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.

2. In the Manage Configurations pane, under Java Application, click **Polynomial** if it is not already selected.

3. Click **Copy**.

   The settings are copied from the original configuration.

4. In the Configuration Name text box, type *PolynomialFixed* as the name for the new configuration.

5. Click the **Java Application** tab to change the program argument.

   a. Click the browse button beside the Application Arguments field.

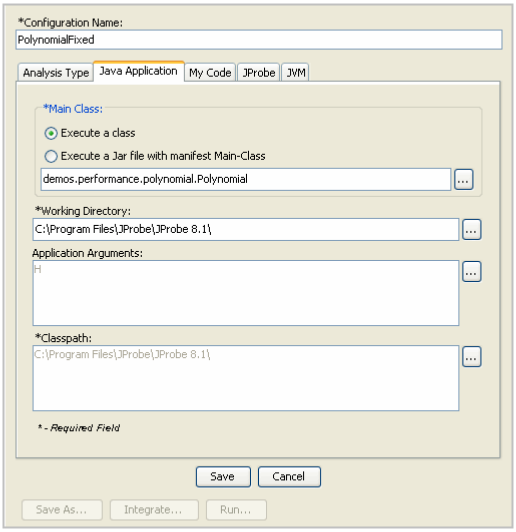
   b. In the upper field of the Application Arguments dialog box, delete `N` and type: `H`

   c. Click **Parse Arguments**.

      The argument appears in the first line of the lower field.

   d. Click **OK**.

   The Application Arguments field displays the program argument.



*Configuration Name: PolynomialFixed

*Main Class:
● Execute a class
○ Execute a Jar file with manifest Main-Class
demos.performance.polynomial.Polynomial

*Working Directory: C:\Program Files\JProbe\JProbe 8.1\

Application Arguments: H

*Classpath: C:\Program Files\JProbe\JProbe 8.1\

* - Required Field

6. Click **Save** then **Save As**, to save the configuration file (*PolynomialFixed_Perf_Settings.jpl*) into your working directory.

7. Click **Integrate**.

   JProbe validates the configuration file and creates a startup script file (for example, in Windows: *PolynomialFixed.bat*, and in Unix/Linux: *PolynomialFixed.sh*).

8. In the Integrating PolynomialFixed dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

   The Integrating Polynomial dialog box presents the status of the operation.

9. Select the **Close Create/Edit Settings tool on successful integration** check box, and click **Close**.

   You are now ready to run a Performance analysis session using the fixed code.

10. On the JProbe toolbar, click **Attach to a Running Session** .

    The Attach to a Running Session dialog box appears, displaying the correct host and port number.

11. Click **OK**.

    The Connection Indicator dialog box indicates that JProbe is looking for your session.

12. Start the fixed *Polynomial* program from the command line:

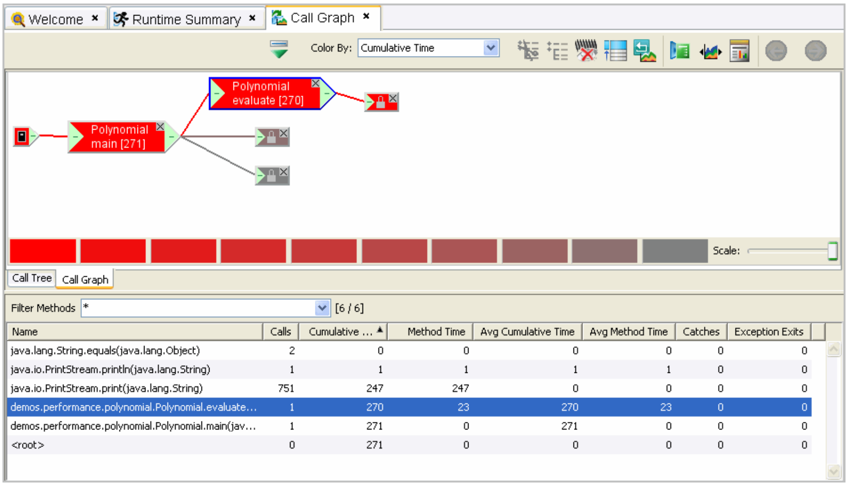
    - In a Windows command window: `>PolynomialFixed.bat`
    - In a Unix or Linux sh shell: `>PolynomialFixed.sh`
    - In a Unix or Linux csh or ksh shell: `>./PolynomialFixed.sh`

    The *PolynomialFixed* program starts and runs in a command window.

    When the program is finished, the command window closes. JProbe takes a performance snapshot and displays it in the Snapshot Navigator panel and in the Call Graph view.

13. In the Call Graph view, click the Call Graph tab, and select the `Polynomial.evaluate()` method.

    The method is highlighted in the Call Graph and the list.



Filter Methods * [6 / 6]

Name	Calls	Cumulative ...	Method Time	Avg Cumulative Time	Avg Method Time	Catches	Exception Exits
java.lang.String.equals(java.lang.Object)	2	0	0	0	0	0	0
java.io.PrintStream.println(java.lang.String)	1	1	1	1	1	0	0
java.io.PrintStream.print(java.lang.String)	751	247	247	0	0	0	0
demos.performance.polynomial.Polynomial.evaluate...	1	270	23	270	23	0	0
demos.performance.polynomial.Polynomial.main(jav...	1	271	0	271	0	0	0
<root>	0	271	0	0	0	0	0

The method was called only once and took 270 milliseconds to execute.

14 In the Snapshot Navigator, right-click the snapshot, select **Save Snapshot As**, name the snapshot *PolynomialFixed*, and click **Save**.

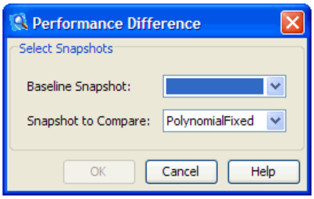
Step 5: Measuring the Performance Improvement

You know that the second algorithm runs much faster than the first one. In this step, you will quantify the performance improvement using the Snapshot Difference window to compare snapshots.

To measure the performance improvement:

1 In the Snapshot Navigator, right-click the **PolynomialFixed** snapshot and select **Snapshot Differencing**.

The Performance Difference dialog box appears, with PolynomialFixed displayed in the Snapshot to Compare list.

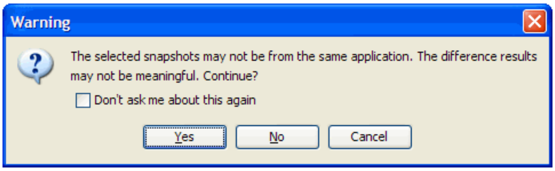


Baseline Snapshot:

Snapshot to Compare: PolynomialFixed

2 Select **Polynomial** from the Baseline Snapshot list and click **OK**.

3 If the following warning dialog appears, click **Yes**.

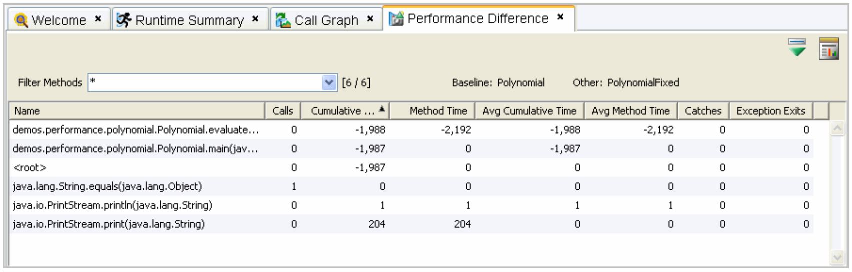


Warning

The selected snapshots may not be from the same application. The difference results may not be meaningful. Continue?

□ Don't ask me about this again

The Snapshot Difference view appears, displaying the differences between the two methods.



Filter Methods * [6 / 6] Baseline: Polynomial Other: PolynomialFixed

Name	Calls	Cumulative ...	Method Time	Avg Cumulative Time	Avg Method Time	Catches	Exception Exits
demos.performance.polynomial.Polynomial.evaluate...	0	-1,988	-2,192	-1,988	-2,192	0	0
demos.performance.polynomial.Polynomial.main(jav...	0	-1,987	0	-1,987	0	0	0
<root>	0	-1,987	0	0	0	0	0
java.lang.String.equals(java.lang.Object)	1	0	0	0	0	0	0
java.io.PrintStream.println(java.lang.String)	0	1	1	1	1	0	0
java.io.PrintStream.print(java.lang.String)	0	204	204	0	0	0	0

The number of calls to `Polynomial.evaluate()` did not change, but the Cumulative Time has decreased by 1,988.

Note Negative values represent a performance improvement. Therefore, the Horner's Rule algorithm runs considerably faster than the nested loop algorithm.

3

Coverage Analysis Demos

This chapter provides a summary of the Coverage demo applications that ship with JProbe and a tutorial for one of these applications.

The source code and compiled classes for the Coverage demos are located in the *<jprobe_home>/demos/coverage* directory.

This chapter contains the following sections:

Summary of Demos for Coverage

The following table describes the purpose of the example applications.

Java SE Application	Purpose	More Information
Adventure.class	A text-based adventure game in which you navigate through a house. The application ships with two test case input files. The test cases do not provide 100% coverage of the program code.	See the notes in *Adventure.java*. **Tutorial:** “Adventure Tutorial” on page 63
SwitchCaseTest.class	This example demonstrates conditional coverage using a simple switch/case statement.	See the notes in *SwitchCaseTest.java*.
TryCatchFinallyTest.class	This example demonstrates how JProbe tracks try-catch-finally blocks.	See the notes in *TryCatchFinallyTest.java*.

JProbe also ships with a Java EE demo application called JProbe Game Pack Demo for Java EE. For more information, see “JProbe Game Pack for JavaEE” on page 73.

Adventure Tutorial

This basic tutorial shows you how to evaluate the effectiveness of two test cases for a text-based adventure game. The test cases are supplied as text files, which are specified in program arguments. The files for the tutorial are available in the *<jprobe_home>/ demos/coverage/adventure* directory.

This tutorial does not create a baseline snapshot of the *Adventure* program because the test cases hit methods in all classes. For more information about the baseline coverage snapshot, see the *JProbe User Guide*.

Note **The values cited in this tutorial reflect the *Adventure* program running on Windows XP with Sun JDK 1.6.0_10.** You may see different values on your system.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	*Adventure.class*
Test Case:	*AdvTest1.txt, AdvTest2.txt*

The tutorial leads you through the following steps:

- Step 1: Setting Your Global Options
- Step 2: Setting Up the Session for the First Test Case
- Step 3: Running the First Test Case
- Step 4: Setting Up and Running the Second Test Case
- Step 5: Merging the Test Case Results
- Step 6: Assessing Your Test Case Coverage

Step 1: Setting Your Global Options

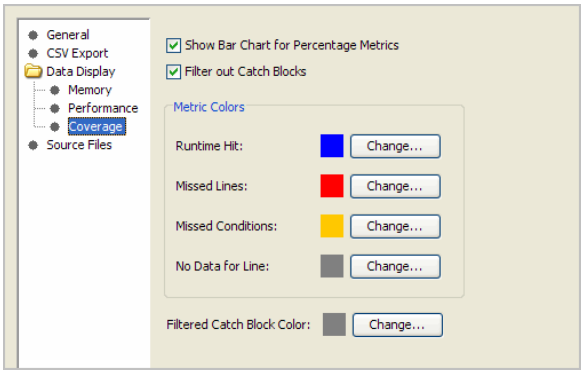
Catch blocks are often hard to test. For this tutorial, we are going to remove the results for catch blocks by setting a global option.

To set global options for the Coverage analysis tool:

1 In the JProbe Console, click **Tools** > **Options** on the toolbar.

2 Click **Data Display** > **Coverage**.

3 Select the **Filter out Catch Blocks** check box.



☑ Show Bar Chart for Percentage Metrics

☑ Filter out Catch Blocks

Metric Colors

Runtime Hit: 🟦

Missed Lines: 🟥

Missed Conditions: 🟨

No Data for Line: ⬛

Filtered Catch Block Color: ⬛

4 Click **OK**.

Step 2: Setting Up the Session for the First Test Case

In this step, you create a configuration for the Adventure program using the JProbe Create/Edit Settings dialog box. The configuration includes the path to a text file that contains the first test case.

To set up the first test case:

1 Click **Tools** > **Create/Edit Settings**.

The Create/Edit Setting dialog box appears.

2 In the Manage Configurations pane, click **Java Application**.

The JProbe Configuration Wizard appears.

3 Click **Add**.

4 In the Configuration Name text box, type *Adventure_TestCase1*, then click **Next**.

5 In the Main Class field, select the **Execute a class** check box.

6 Click the browse button beside the Main Class field and navigate to the *Adventure.class* file in the *<jprobe_home>/demos/coverage/Adventure* directory.

7 Click **OK**.

The following information is displayed:

- Main Class: `demos.coverage.adventure.Adventure`

- Working Directory: <*jprobe_home*>

8 Add the fully qualified path to the text file containing the first test case.

a Click the browse button beside the Application Arguments field.

b In the Application Arguments dialog, type the following in the upper field: `<`*`jprobe_home`*`>/demos/coverage/adventure/AdvTest1.txt`

Note If there is a space in your JProbe home directory path, enclose the argument in quotes.

c Click **Parse Argument**.

The argument appears in the first line of the lower field.

d Click **OK**.

The Arguments field displays the program argument.

9 Click the browse button beside the Classpath field.

10 In the Classpath dialog box, click **Add Working Directory**, then click **OK**.

The working directory appears in the Classpath field.

11 Click **Next**.

The Select a Java Virtual Machine page appears.

12 If you want to change the default JVM, click the browse button beside the Java Executable field, select another JVM in the Java Virtual Machines dialog box, and click **OK**.

Note Ideally, the JVM you select should be the version that was used to compile your program.

13 Click **Next**.

The Specify Your Code page appears.

14 In the Category/Program Name text box type *DemoCategory* (which specifies the name of the category in which you want to include your code), then click **Next**.

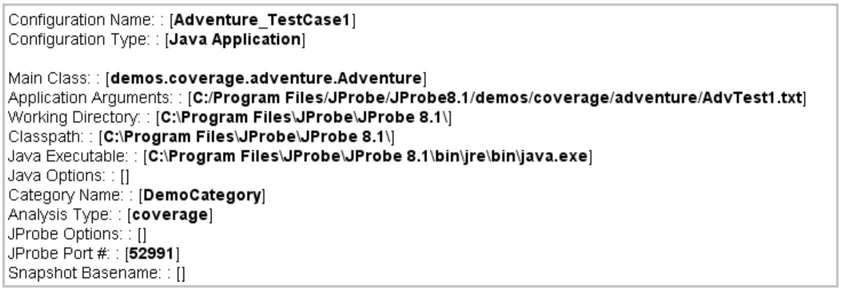
15 In the Select a JProbe Analysis page, ensure that the **Coverage** option is selected.

16 Click the **Filters** tab.

You should see the following default Include filter in the table: `demos.coverage.adventure.*.*().`

17 Click **Next** and **Next** again to pass the Specify the JProbe Options page.

The Save the Configuration page appears, presenting a summary of the settings defined for your configuration.



```
Configuration Name: : [Adventure_TestCase1]
Configuration Type: : [Java Application]

Main Class: : [demos.coverage.adventure.Adventure]
Application Arguments: : [C:/Program Files/JProbe/JProbe8.1/demos/coverage/adventure/AdvTest1.txt]
Working Directory: : [C:\Program Files\JProbe\JProbe 8.1\]
Classpath: : [C:\Program Files\JProbe\JProbe 8.1\]
Java Executable: : [C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe]
Java Options: : []
Category Name: : [DemoCategory]
Analysis Type: : [coverage]
JProbe Options: : []
JProbe Port #: : [52991]
Snapshot Basename: : []
```

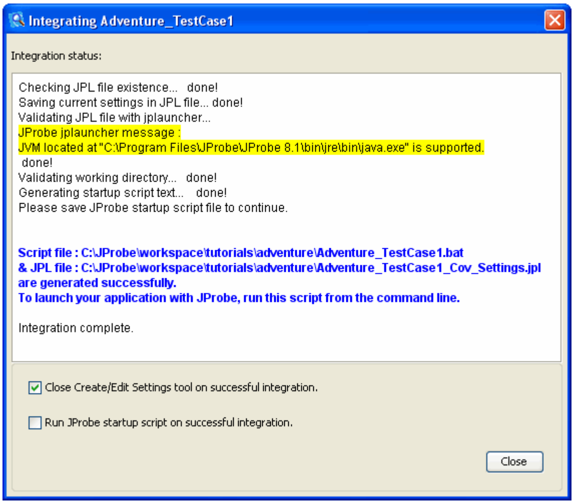
18 Click **Save** and save the configuration file (*Adventure_TestCase1_Cov_Settings.jpl*) into your working directory.

19 In the Configuration Complete page, select the **Integrate** check box and click **Finish**.

JProbe validates the configuration file and creates a startup script file (for example, in Windows: *Adventure_TestCase1.bat*, and in Unix/Linux: *Adventure_TestCase1.sh*).

20 In the Integrating Adventure_TestCase1 dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

The Integrating Adventure_TestCase1 dialog box presents the status of the operation.



```
Checking JPL file existence...   done!
Saving current settings in JPL file... done!
Validating JPL file with jplauncher...
JProbe jplauncher message :
JVM located at "C:\Program Files\JProbe\JProbe 8.1\bin\jre\bin\java.exe" is supported.
 done!
Validating working directory...   done!
Generating startup script text...    done!
Please save JProbe startup script file to continue.


Script file : C:\JProbe\workspace\tutorials\adventure\Adventure_TestCase1.bat
& JPL file : C:\JProbe\workspace\tutorials\adventure\Adventure_TestCase1_Cov_Settings.jpl
are generated successfully.
To launch your application with JProbe, run this script from the command line.

Integration complete.
```

☑ Close Create/Edit Settings tool on successful integration.

□ Run JProbe startup script on successful integration.

21 Select the **Close Create/Edit Settings tool on successful integration** check box, and click **Close**.

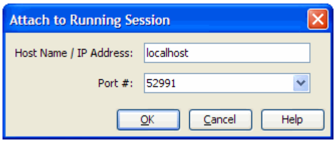
You are now ready to run a Coverage analysis session (test case #1).

Step 3: Running the First Test Case

To run the first test case:

1 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to Running Session dialog box appears, displaying the correct host and port number.


Attach to Running Session

Host Name / IP Address: localhost

Port #: 52991

2 Click **OK**.

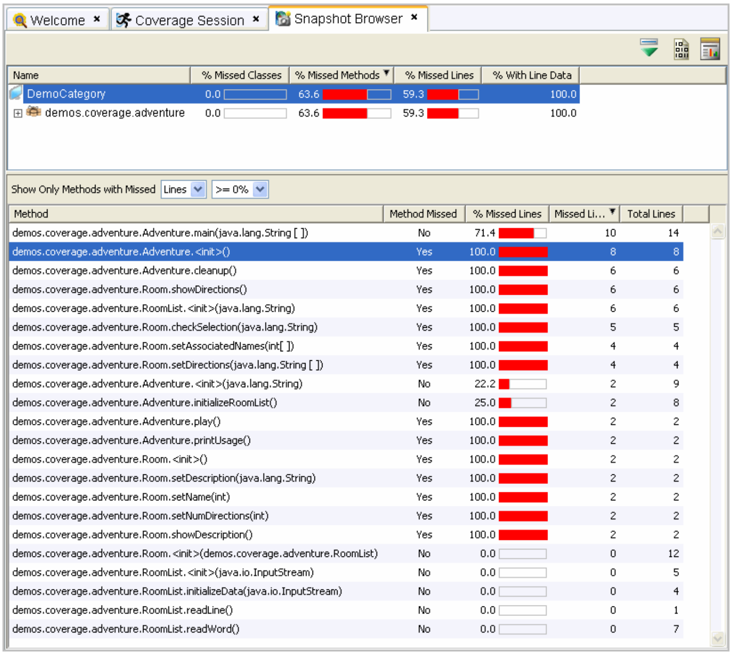
The Connection Indicator dialog box indicates that JProbe is looking for your session.

3 Start the *Adventure* program (test case #1) from the command line:

- In a Windows command window: `>Adventure_TestCase1.bat`
- In a Unix or Linux sh shell: `>Adventure_TestCase1.sh`
- In a Unix or Linux csh or ksh shell: `>./Adventure_TestCase1.sh`

The JProbe Execution Console opens and the program runs using the text file as test case input for choices.

When the program is finished, the command window closes. JProbe takes a coverage snapshot and displays it in the Snapshot Navigator panel and in the Snapshot Browser view.



Name	% Missed Classes	% Missed Methods	% Missed Lines	% With Line Data
DemoCategory	0.0	63.6	59.3	100.0
demos.coverage.adventure	0.0	63.6	59.3	100.0

Show Only Methods with Missed Lines >= 0%

Method	Method Missed	% Missed Lines	Missed Li...	Total Lines
demos.coverage.adventure.Adventure.main(java.lang.String [ ])	No	71.4	10	14
demos.coverage.adventure.Adventure.<init>()	Yes	100.0	8	8
demos.coverage.adventure.Adventure.cleanup()	Yes	100.0	6	6
demos.coverage.adventure.Room.showDirections()	Yes	100.0	6	6
demos.coverage.adventure.RoomList.<init>(java.lang.String)	Yes	100.0	6	6
demos.coverage.adventure.Room.checkSelection(java.lang.String)	Yes	100.0	5	5
demos.coverage.adventure.Room.setAssociatedNames(int[ ])	Yes	100.0	4	4
demos.coverage.adventure.Room.setDirections(java.lang.String [ ])	Yes	100.0	4	4
demos.coverage.adventure.Adventure.<init>(java.lang.String)	No	22.2	2	9
demos.coverage.adventure.Adventure.initializeRoomList()	No	25.0	2	8
demos.coverage.adventure.Adventure.play()	Yes	100.0	2	2
demos.coverage.adventure.Adventure.printUsage()	Yes	100.0	2	2
demos.coverage.adventure.Room.<init>()	Yes	100.0	2	2
demos.coverage.adventure.Room.setDescription(java.lang.String)	Yes	100.0	2	2
demos.coverage.adventure.Room.setName(int)	Yes	100.0	2	2
demos.coverage.adventure.Room.setNumDirections(int)	Yes	100.0	2	2
demos.coverage.adventure.Room.showDescription()	Yes	100.0	2	2
demos.coverage.adventure.Room.<init>(demos.coverage.adventure.RoomList)	No	0.0	0	12
demos.coverage.adventure.RoomList.<init>(java.io.InputStream)	No	0.0	0	5
demos.coverage.adventure.RoomList.initializeData(java.io.InputStream)	No	0.0	0	4
demos.coverage.adventure.RoomList.readLine()	No	0.0	0	1
demos.coverage.adventure.RoomList.readWord()	No	0.0	0	7

Step 4: Setting Up and Running the Second Test Case

To run JProbe with a different test case, you need to change the program arguments for your configuration. Because these are tests that may be modified and re-run, do this by creating a second configuration based on the original, by copying and editing it.

To set up and run the second test case:

1. Click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.

2. In the Manage Configurations pane, under Java Application, click **Adventure_TestCase1** if it is not already selected.

3. Click **Copy**.

   The settings are copied from the original configuration.

4. In the Configuration Name box, type *Adventure_TestCase2* as the name for the new configuration.

5. Click the **Java Application** tab to change the program argument.

  a Click the browse button beside the Application Arguments field.

  b In the Application Arguments dialog box, select the existing argument in the lower box and click **Delete**.

  c Edit the argument in the upper box as follows:

  <jprobe_home>`/demos/coverage/adventure/AdvTest2.txt`

  Note If there is a space in your JProbe home directory path, enclose the argument in quotes.

  d Click **Parse Argument**.

  The argument appears in the first line of the lower field.

  e Click **OK**.

The Application Arguments field displays the program argument.

6 Click **Save** then **Save As**, to save the configuration file (*Adventure_TestCase2_Cov_Settings.jpl*) into your working directory.

7 Click **Integrate**.

JProbe validates the configuration file and creates a startup script file (for example, in Windows: *Adventure_TestCase2.bat*, and in Unix/Linux: *Adventure_TestCase2.sh*).

8 In the Integrating Adventure_TestCase2 dialog box, use the browse button to navigate to your working directory, and click **Save** to save the startup file.

The Integrating Adventure_TestCase2 dialog box presents the status of the operation.

9 Select the Close Create/Edit Settings tool on successful integration check box, and click **Close**.

You are now ready to run a Coverage analysis session (test case #2).

10 On the JProbe toolbar, click **Attach to a Running Session** .

The Attach to a Running Session dialog box appears, displaying the correct host and port number.

11 Click **OK**.

The Connection Indicator dialog box indicates that JProbe is looking for your session.

12 Start the *Adventure* program (test case #2) from the command line:

- In a Windows command window: `>Adventure_TestCase2.bat`

- In a Unix or Linux sh shell: `>Adventure_TestCase2.sh`
- In a Unix or Linux csh or ksh shell: `>./Adventure_TestCase2.sh`

The JProbe Execution Console opens and the program runs using the text file as test case input for choices.

When the program is finished, the command window closes. JProbe takes a coverage snapshot and displays it in the Snapshot Navigator panel.

13 Close the Snapshot Browser view.

Step 5: Merging the Test Case Results

In this step, you merge the snapshots taken during Step 3: Running the First Test Case and Step 4: Setting Up and Running the Second Test Case, to get a complete picture of the coverage provided by these two test cases.

You merge the snapshots using the `jpcovmerge` command line tool. Before you can use this tool, you must save to the disk the two snapshots to be compared.

To merge the snapshots:

1. Save the snapshot taken during Step 3: Running the First Test Case to the disk.
   a. Right-click the snapshot and select **Save Snapshot As**.
   b. In the Save As dialog box, save the snapshot as *<working_directory>\first.jpc*.
2. Repeat step a to step b to save the snapshot taken during Step 4: Setting Up and Running the Second Test Case as *<working_directory>\second.jpc*.
3. Click **Start > Run** and type the following, then click **OK**:

```
<jprobe_home>\bin\jpcovmerge <working_directory>\first.jpc
<working_directory>\second.jpc
<working_directory>\merged.jpc
```

4. On the JProbe Console, click **File > Load Snapshot**.
5. In the Open dialog box, select *<working_directory>\merged.jpc* and click **Open**.

   The merged snapshot is selected and its content is displayed in the Snapshot Browser view.

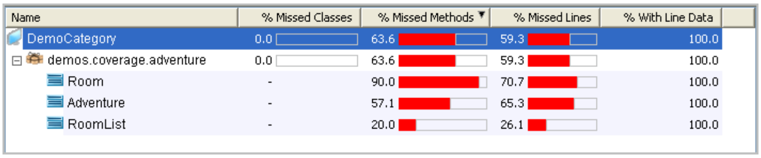
Step 6: Assessing Your Test Case Coverage

After merging your test cases, you are now ready to investigate your results. By default, JProbe displays results in terms of misses. A miss means that the code was not called during your test cases.

To investigate your results:

1. In the Snapshot Browser view, expand the class tree in the upper pane to show all of the classes in `demos.coverage.adventure`.
2. Click the **% Missed Methods** column heading in the upper pane to sort the table from highest-to-lowest percentage of missed methods.

   Methods are missed in three classes: `Adventure`, `Logic`, and `RoomList`. We will investigate the first two classes.

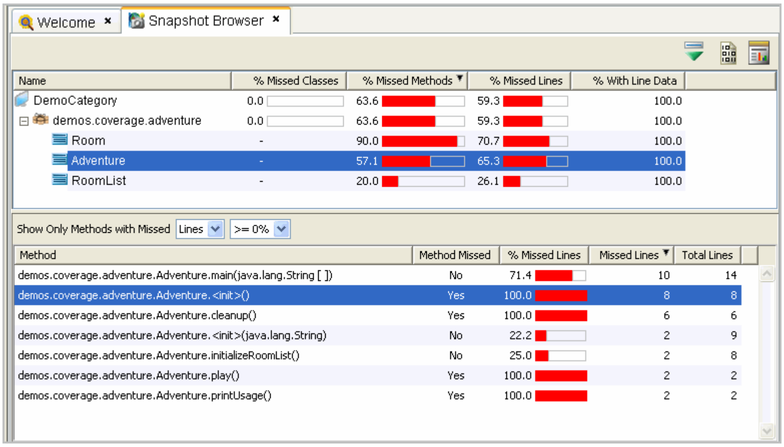


Name	% Missed Classes	% Missed Methods	% Missed Lines	% With Line Data
DemoCategory	0.0	63.6	59.3	100.0
demos.coverage.adventure	0.0	63.6	59.3	100.0
Room	-	90.0	70.7	100.0
Adventure	-	57.1	65.3	100.0
RoomList	-	20.0	26.1	100.0

   By default, the Adventure program is selected in the top pane, which means that the lower pane contains all the methods in the program.

3. In the upper pane, select the **Adventure** class.

   The methods for the Adventure class are displayed in the lower pane.



Name	% Missed Classes	% Missed Methods	% Missed Lines	% With Line Data
DemoCategory	0.0	63.6	59.3	100.0
demos.coverage.adventure	0.0	63.6	59.3	100.0
Room	-	90.0	70.7	100.0
Adventure	-	57.1	65.3	100.0
RoomList	-	20.0	26.1	100.0

Show Only Methods with Missed Lines >= 0%

Method	Method Missed	% Missed Lines	Missed Lines	Total Lines
demos.coverage.adventure.Adventure.main(java.lang.String [ ])	No	71.4	10	14
demos.coverage.adventure.Adventure.<init>()	Yes	100.0	8	8
demos.coverage.adventure.Adventure.cleanup()	Yes	100.0	6	6
demos.coverage.adventure.Adventure.<init>(java.lang.String)	No	22.2	2	9
demos.coverage.adventure.Adventure.initializeRoomList()	No	25.0	2	8
demos.coverage.adventure.Adventure.play()	Yes	100.0	2	2
demos.coverage.adventure.Adventure.printUsage()	Yes	100.0	2	2

4. Right-click the `main()` method and click **View Source** to open the *<jprobe_home>\demos\coverage\adventure\Adventure.java* source code.

   The source code opens at a block of missed lines that are in an `if` statement, so you know that the test case missed a condition. You have identified an opportunity to expand the test suite with a new test case. In this case, you would pass two input files as program arguments.

5. Click the ▾ button at the top of the coverage bar (to the right of the scroll bar) to move to the next set of missed lines.

   We can see that code is missed because input is not coming from the console. You can ignore these misses.

6. Continue in this way until you have examined all the missed blocks of code, then move on to the next missed class.

   You have identified an area for improvement. If you like, you can modify the test suite and redo the tutorial to see the improvement in overall coverage.

4

JProbe Game Pack for JavaEE

This section describes how to deploy the JProbe Game Pack demo software and presents how to use JProbe with JavaEE applications that require a third-party application server. The tutorials describe how to find two different kinds of memory leaks with the Memory analysis tool and how to investigate a performance bottleneck with the Performance analysis tool.

Note The Game Pack tutorials were created using JProbe 8.1, BEA WebLogic 10.0 application server, and JDK 1.6.0_10 on Windows XP. You can use any of the supported application servers listed in “System Requirements” on page 74, but your results may be different from those seen in the tutorials.

This chapter contains the following sections:

Deploying the JProbe Game Pack Demo

This section describes how to deploy the JProbe Game Pack demo on JBoss-Tomcat and WebLogic application servers. For details, see:

- System Requirements
- Setting Environment Variables
- Installing the Game Pack Demo
- Deploying the Game Pack Demo on Your Application Server
- Creating a User Account for Game Pack
- Running Game Pack with the JProbe Application
- Game Pack Known Issues

System Requirements

The Game Pack demo requires the following environment:

- JProbe 8.1
- One of the following operating systems:
  - Microsoft® Windows® 2003 or XP PRO SP2 or later
  - Red Hat® AS 4.0 or 5.x
  - Sun Solaris® SPARC 8.0, 9.0, or 10.0
  - IBM AIX® 5L 5.3 or 6.1
- One of the following application servers:
  - JBoss™ 3.2.1 with Apache Tomcat 5.0.24
  - JBoss™ 4.0.1 SP1 with Apache Tomcat 5.0.30
  - JBoss™ 4.2.2.GA
  - BEA® WebLogic® Server 8.1 SP2
  - BEA® WebLogic® Server 9.1 or 9.2
  - BEA® WebLogic® Server 10.0
- Java EE SDK 1.6.0 or later
- One of the following browsers:
  - FireFox

    - Microsoft IE
    - Apple Safari
- Jakarta Ant 1.6.3 or later

For a list of JProbe supported platforms and installation instructions, see the *JProbe Installation Guide*. The guide is available as a PDF file on the CD and in the JProbe installation directory *<jprobe_install>\doc*.

You can download the supported versions of JBoss with Apache Tomcat from SourceForge.net at: *http://sourceforge.net/project/showfiles.php?group_id=22866*.

You can download Ant 1.6.3 or later from the Apache Jakarta Project at: *http://jakarta.apache.org/ant/*. Extract the files to a directory.

Setting Environment Variables

Before you begin, set up the following environment variables:

- `ANT_HOME=<`*`ant_install_dir`*`>`
- `JPROBE_HOME=<`*`jprobe_install_dir`*`>`
- If you are using JBoss, `JBOSS_HOME=<`*`jboss_install_dir`*`>`
- If you are using WebLogic Server, `WL_HOME=<`*`wlserver##_install_dir`*`>`

Add the following paths to your `PATH` environment variable (use the syntax appropriate for your operating system):

- `%ANT_HOME%\bin`
- `%JPROBE_HOME%`
- If you are using JBoss, `%JBOSS_HOME%`
- If you are using WebLogic Server, `%WL_HOME%`

Installing the Game Pack Demo

For detailed installation instructions, see the *JProbe Installation Guide*.

If you chose to install examples during the JProbe installation, the Game Pack demo files are installed automatically in the following directory structure:

JPROBE_HOME

demos

gamepack

- *build* (build and class files)
- *dist* (deploy files)
- *lib* (JAR files)
- *src* (source code)
- *support* (support files)

Deploying the Game Pack Demo on Your Application Server

This section presents how to deploy the Game Pack demo on the following application servers:

- Deploying the Game Pack Demo on JBoss
- Deploying the Game Pack Demo on BEA WebLogic Server 8.1 SP2
- Deploying the Game Pack on BEA WebLogic Server 9.x
- Deploying the Game Pack on BEA WebLogic Server 10.0

Deploying the Game Pack Demo on JBoss

Use the build properties file and build script provided. The following procedure assumes that you have successfully installed and configured: Ant 1.6.3 or later, a supported version of JBoss with Apache Tomcat, and JProbe 8.1. In addition, you need to have defined an environment variable called `JBOSS_HOME` which points to your JBoss home directory.

To deploy the Game Pack demo on JBoss:

1. In the *<JPROBE_HOME>\demos\gamepack\build* directory, edit the *build.properties* file.
   a. Delete the pound sign (`#`) from the `J2EEServer=JBossWithTomcat` statement.
   b. Add a pound sign (`#`) in front of the `J2EEServer=Weblogic9.1` statement.
   c. Save the file.

   Note A `JBOSS_HOME` environment variable is required for the build to work.
2. At a command prompt, navigate to the *<JPROBE_HOME>\demos\gamepack* directory and enter the following command:

```
ant deploy -f build\build.xml
```

A successful build includes the following types of messages:

```
deploy.tofolder:

[copy] Copying 1 file to C:\Program
Files\ApplicationServers\JBoss\jboss-
4.2.2.GA\server\default\deploy

[echo] INFO: 'C:\Quest_Software\JProbe_8.1\demos\gamepack/
dist/gamepack.ear

' has been deployed to 'C:\Program
Files\ApplicationServers\JBoss\jboss-4.2.2.GA

/server/default/deploy'.

BUILD SUCCESSFUL
```

3 Start JBoss using the *<JBOSS_HOME>\bin\run.bat* file.

4 To view the Game Pack, open a browser and enter:

```
http://localhost:<port>/gamepack/index
```

where <*port*> is Tomcat's http port number. The default value is 8080.

Note The Ant build command given in this procedure packages and deploys the Game Pack files. To rebuild the Game Pack from source and package and deploy it, use: `ant -f build\build.xml`.

Deploying the Game Pack Demo on BEA WebLogic Server 8.1 SP2

To deploy the JProbe Game Pack on WebLogic Server 8.1, you use the WebLogic Configuration Wizard. The following procedure assumes that you have successfully installed and configured BEA WebLogic Server 8.1 SP2 and JProbe 8.1.

To deploy the Game Pack demo on BEA WebLogic Server 8.1 SP2:

1 Start the WebLogic Configuration Wizard.

- On Windows, click **Start > Programs > BEA Weblogic Platform 8.1 > Configuration Wizard**.
- On UNIX, execute `java -jar config.jar` in the *WL_HOME/weblogic81/common/lib* directory.

2 Select the **Create a new WebLogic configuration** check box and click **Next**.

3. In the Locate Additional Templates field, click **Browse** and navigate to *<JPROBE_HOME>/demos/gamepack/support/deployment/Weblogic8.1/templates/domains* directory.

   The WebLogic Configuration Templates tree refreshes to display a Quest template folder.

4. Expand the entire Quest template folder.
5. Select **Gamepack Domain** and click **Next**.
6. For a default configuration, select the **Express** check box, then click **Next**.

   Note Alternatively, for customizing configuration elements, such as server port numbers, select the **Custom** check box. This adds several steps to the configuration process.

7. Make changes to the **User name** and **User password** and click **Next**.
8. Specify the server start mode by selecting the **Development Mode** check box.
9. Select a JDK from the available list (default is recommended), then click **Next**.
10. Specify the directory in which the WebLogic configuration will be created. The default is in the *user_projects* directory of your *WL_HOME*. The default configuration name is *gamepack*.
11. Click **Create** to create the domain in the selected directory.
12. When the configuration is created, click **Done**.
13. Start the Game Pack using your new configuration by executing the `startWebLogic.cmd` command from the *WL_HOME/user_projects/domains/gamepack* directory (or other directory if you made changes in the Configuration Wizard).
14. To view the Game Pack, open a browser and type:

    ```
    http://localhost:7001/gamepack/index
    ```

    You can now stop WebLogic (by using the `stopWebLogic.cmd` command from the *WL_HOME/user_projects/domains/gamepack* directory) and prepare to run the Game Pack in JProbe.

Deploying the Game Pack on BEA WebLogic Server 9.x

To deploy the JProbe Game Pack on BEA WebLogic Server 9.x, you use the WebLogic Configuration Wizard. The following procedure assumes that you have successfully installed and configured BEA WebLogic Server 9.x and JProbe 8.1.

To deploy the Game Pack demo on BEA WebLogic Server 9.x:

1. Start the WebLogic Configuration Wizard.
   - On Windows, click **Start > Programs > BEA Products > Tools > Configuration Wizard**.
   - On UNIX, execute `java -jar config.jar` in the *WL_HOME/weblogic9x/common/lib* directory.
2. Select the **Create a new WebLogic domain** check box and click **Next**.
3. Select the **Base this domain on an existing template** check box.
4. Click **Browse** and navigate to *<JPROBE_HOME>/demos/gamepack/support/deployment/Weblogic9.1/templates/domains* directory, select *gamepack.jar*, and click **OK**.
5. Click **Next**.
6. Configure the **User name** and **User password** and click **Next** (default is recommended).
7. Specify the server start mode by selecting the **Development Mode** check box.
8. Select a JDK from the available list (default is recommended), then click **Next**.
9. To use the default configuration (recommended), select the **No** check box.

   Note Alternatively, for customizing different configuration options, such as listen ports, select the **Yes** check box. This adds several steps to the configuration process.
10. Specify the directory where the WebLogic domain will be created. The default is in the *user-projects/domain* directory of your *WL_HOME*. The default domain name is *gamepack*. You can leave the default directory or change it.
11. Specify the directory where WebLogic Applications will be stored. The default is the *user_projects/applications* directory of your *WL_HOME*. You can leave the default directory or change it.
12. Click **Create** to create the domain in the selected directory.
13. When the configuration is created, click **Done**.
14. Start the Game Pack using your new configuration by executing the `startWebLogic.cmd` command from the *WL_HOME/user_projects/domains/gamepack* directory (or other directory if you made changes in the Configuration Wizard).
15. To view the Game Pack, open a browser and type:

    ```
    http://localhost:7001/gamepack/index
    ```

You can now stop WebLogic, by using the `stopWebLogic.cmd` command from the *WL_HOME/user_projects/domains/gamepack/bin* directory (or other directory if you made changes in the Configuration Wizard), and prepare to run the Game Pack in JProbe.

Deploying the Game Pack on BEA WebLogic Server 10.0

To deploy the JProbe Game Pack on BEA WebLogic Server 10.0, you use the WebLogic Configuration Wizard. The following procedure assumes that you have successfully installed and configured BEA WebLogic Server 10.0 and JProbe 8.1.

To deploy the Game Pack demo on BEA WebLogic Server 10.0:

1. Start the WebLogic Configuration Wizard.
   - On Windows, click **Start > Programs > BEA Products > Tools > Configuration Wizard**.
   - On UNIX, execute `java -jar configwiz.jar` in the *WL_HOME/wlserver_10.0/common/lib* directory.
2. Select the **Create a new WebLogic domain** check box and click **Next**.
3. Select the **Base this domain on an existing template** check box.
4. Click **Browse** and navigate to *<JPROBE_HOME>/demos/gamepack/support/deployment/Weblogic10.3/templates/domains* directory, select *gamepack.jar*, and click **OK**.
5. Click **Next**.
6. Configure the **User name** and **User password** and click **Next** (default is recommended).
7. Specify the server start mode by selecting the **Development Mode** check box.
8. Select a JDK from the available list (default is recommended), then click **Next**.
9. To use the default configuration (recommended), select the **No** check box.

   Note Alternatively, for customizing different configuration options, such as listen ports, select the **Yes** check box. This adds several steps to the configuration process.
10. Specify the directory where the WebLogic domain will be created. The default is in the *user-projects/domain* directory of your *WL_HOME*. The default domain name is *gamepack*. You can leave the default directory or change it.
11. Specify the directory where WebLogic Applications will be stored. The default is the *user_projects/applications* directory of your *WL_HOME*. You can leave the default directory or change it.

12 Click **Create** to create the domain in the selected directory.

13 When the configuration is created, click **Done**.

14 Start the Game Pack using your new configuration by executing the `startWebLogic.cmd` command from the *WL_HOME/user_projects/domains/gamepack* directory (or other directory if you made changes in the Configuration Wizard).

15 To view the Game Pack, open a browser and type:

```
http://localhost:7001/gamepack/index
```

You can now stop WebLogic, by using the `stopWebLogic.cmd` command from the *WL_HOME/user_projects/domains/gamepack/bin* directory (or other directory if you made changes in the Configuration Wizard), and prepare to run the Game Pack in JProbe.

Creating a User Account for Game Pack

The first time that you start the Game Pack, you need to create a user account for yourself.

To create a user account:

1 In the Game Pack demo home page, click **Sign up as new user**.

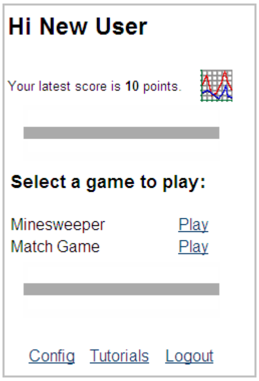
2 Type a user name and password in the **User ID** and **Password** fields.

3 Re-type the password in the **Re-enter Password** field.

4 Type the name you want to use for the games in the **Name** field.

5 Click **Sign-up**.

The Game Pack demo home page re-appears. The name you typed in the Name field appears in the top left corner.



Hi New User

Your latest score is **10** points.

Select a game to play:

Minesweeper Play
Match Game Play

Config Tutorials Logout

You can now proceed to play Minesweeper or Match Game.

Running Game Pack with the JProbe Application

Follow a tutorial that matches the JProbe application you are using. You can open the tutorials from the Game Pack demo by clicking the **Tutorials** link.

To run Game Pack with JProbe, you must complete the following generic tasks:

1. Integrate the JProbe application with your application server.
2. Set up your JProbe session.
3. Run the JProbe session to start the application server.
4. Connect the JProbe Console to the session.
5. When the application server is started, open a browser and go to:

   For JBoss: *http://localhost:8080/gamepack/index*

   For WebLogic Server: *http://localhost:7001/gamepack/index*
6. Exercise the use case.

Game Pack Known Issues

The following known issues have been identified:

- The Game Pack may not work with the evaluation version of WebLogic Server. Use a fully licensed version of WebLogic or use JBoss.

- Minesweeper handles one selection at a time and does not buffer multiple tile clicks.
- Right-clicking Minesweeper and selecting **Open in New Window** may cause unexpected results.
- Opening multiple browser sessions for the same user may cause point calculation issues.

Loitering Objects Tutorial

Minesweeper and the Match Game both offer a Loitering Objects mode. The Loitering Objects mode demonstrates how an obsolete container reference can keep session beans in the Java heap long after their usefulness is gone. In this tutorial, you see how you can reclaim memory by removing loitering `GameHandler` objects.

Before beginning this tutorial, you need to deploy the Game Pack to the application server of your choice. For more information, see "Deploying the JProbe Game Pack Demo" on page 74.

Note The values cited in this tutorial reflect the Game Pack running on JBoss 4.2.2.GA, on Windows XP.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	Game Pack demo: Match Game
Use Case:	Play a few games without quitting.
Architecture:	When the Start button is selected, create a `GameHandler` object to run the game.
Hypothesis:	The `GameHandler` object is removed from the heap when the game ends.

The tutorial walks you through the following steps:

- Step 1: Setting Up the Session
- Step 2: Starting the Session and the Game Pack
- Step 3: Running the Session

- Step 4: Identifying Loitering Objects
- Step 5: Investigating Loitering Objects
- Step 6: Running the Session with Improved Code

Step 1: Setting Up the Session

You use the JProbe configuration tool to create the session settings for this example. In the setup procedure, only the settings that you need to change or verify are mentioned. If a setting is not mentioned, leave it blank or in its default state. The following procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1. On the JProbe Console, click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.
2. In the Manage Configurations pane, select **JBoss** and click **Add**.
3. In the Define a New Configuration screen, type a name in the Configuration Name field, such as LoiteringObjects.
4. Click the appropriate version under **JBoss** and click **Next**.
5. Type the path to your server startup script in the text field or click ... and navigate to it, then click **Next**.
6. In the Specify Your Code screen, in the Elements area, type the path to the deployed *gamepack.ear* file (for example,

   <jprobe_home>\demos\gamepack\dist\gamepack.ear) or click ... and navigate to it.
7. Specify a Category/Program Name, then click **Next**.
8. In the Select a JProbe Analysis screen, under Analysis Type, select **Memory**, then click **Next**.
9. Click **Next** again to pass the Specify the JProbe Options screen.
10. In the Save the Configuration screen, review the settings, then click **Save** and browse to a location to save the settings file that you just created.
11. In the Configuration Complete screen, select **Integrate**, then click **Finish**.

12. In the Integrating dialog box, click ... to navigate to the location where you want to save the startup script (for example, in Windows: *run_WithJProbe.bat*, and in Unix/Linux: *run_WithJProbe.sh*), then click **Save**.
13. When you see the `Integration complete` message, select the **Close Create/ Edit Settings tool on successful integration** check box and click **Close**.

Step 2: Starting the Session and the Game Pack

In this step, you start JBoss using the startup script you created in the Step 1: Setting Up the Session and connect to it from JProbe. Then you open a browser to run the Game Pack demo.

To start the JProbe session and the Game Pack:

1. Start JBoss using the startup script:

   In Windows: `>C:\<jboss_home>\bin\run_WithJProbe.bat`

   In Unix/Linux (sh shell): `>run_WithJProbe.sh`

   In Unix/Linux (csh or ksh shells): `>./run_WithJProbe.sh`

2. In the JProbe Console, click **Attach to Session** .
3. Click **OK** in the Attach to Running Session dialog box.

   The Runtime Summary view opens, with the Memory Pools tab on the foreground.

4. Open a browser and go to *http://localhost:8080/gamepack/index*.

   The Game Pack Demo login page appears.

5. Enter your user ID and password and click **Login**.

   Note The first time you do this, you need to create a user name and password for yourself. For more information, see “Creating a User Account for Game Pack” on page 81.

Step 3: Running the Session

In this step, you work through a use case by playing three consecutive games. It does not matter for the analysis whether you win or lose the games. However, it is important that you start a session before you begin to play, or JProbe will not perform a garbage collection. After you have played three games, you end the session, and JProbe takes a snapshot.

To run a game with the Loitering Objects fault:

1. In the Game Pack demo, click **Play** beside either Minesweeper or Match Game.

   Note Both games have the same loitering object problem.

2. Select the **Loitering Objects** option.

   Note Clicking the link displays the option's definition.

3. In the JProbe Runtime Summary view, click **Set Recording Level** on the toolbar.
4. In the Set Recording Level dialog box, select **Record Allocations and Stack Traces For All Instances**, then click **OK**.
5. In the Game Pack demo, click **Start** and play the game.
6. When the game ends, play the game twice more (without quitting) for a total of three complete games.
7. Click **Quit**.
8. In the Runtime Summary view, click **Set Recording Level** on the toolbar.
9. In the Set Recording Level dialog box, select **Data Recording Off**, then click **OK**.

   JProbe takes a snapshot and displays it in the Snapshot Navigator panel.

10. In the Snapshot Navigator, right-click the snapshot, select **Save Snapshot As**, and navigate to where you want to save the snapshot.
11. Name the snapshot *loitering_objects* and click **Save**.

    The new name is displayed in the Snapshot Navigator.

12. Click **Detach from Running Session** .

    Note You can also close your application server and the Game Pack demo browser.

    The session snapshot appears in the snapshot navigator and (after a few seconds) the Instances view appears.

Step 4: Identifying Loitering Objects

In this step, you look for loitering objects in the heap. The Heap Count column is the first place to look. In general, you should expect objects created during a session to be removed at the end of it. In fact, the `GameHandler` objects are not removed, and three instances of this object continue to loiter in the heap.

To identify loitering objects:

1 If the *loitering_objects* snapshot is not open, right-click it in the snapshot navigator and select **Open Instances View**.

The Instances view appears.

2 In the Filter Classes field, type `*.GameHandler` and press **Enter** to display only the `GameHandler` class.

The Heap Count for GameHandler is 3, not 0 (zero) as hypothesized.

Step 5: Investigating Loitering Objects

In this step, you find the live object that continues to hold a reference to the loitering instances of `GameHandler` in the heap. You start in the Instances view, then drill down to Instance Detail and the Memory Leak views, and discover that the loitering `GameHandler` objects are being held by an obsolete container reference. To look for a solution, you open the Source view and review the code for the allocating method.

To investigate loitering objects:

1 If the snapshot is not open, right-click it in the snapshot navigator and select the **Open Instances View**.

The Instances view appears.

2 Filter the method list by typing `*.GameHandler` in the Filter Classes field and pressing **Enter**.

3 Select the `GameHandler` class and click **Instance Detail**.

The Instance Detail view opens. The three loitering `GameHandler` objects are displayed in the instances list. You can see the stack trace of method calls in the **Allocated At** column.

4 Click the Trace tab and scroll down the method list to the methods belonging to the subpackages of the `demos.gamepack.web.game` package and find a method called `EJBControllerImpl.startGame().`

The `startGame()` method calls the `getGameHandlerRemote()` method, which in turn sets off a series of calls to JBoss methods that eventually causes the bean to be created. Therefore, the `getGameHandlerRemote()` method is the most likely candidate for investigation, because it is the last Game Pack method before the series of application server calls.

5. Right-click the `demos.gamepack.web.game.EJBControllerImpl.getGameHandlerRemote()` method and select **Show Allocated At Source**.

6. If you are prompted for the source code, navigate to *<jprobe_home>/demos/gamepack/src/demos/gamepack/web/game*, select *EJBControllerImpl.java*, and click **Open**.

   The Source view opens, highlighting line 197 of the code. This line represents the method call from the `getGameHandlerRemote()` method to the application server's `create` method, which creates the beans.

   Now that you found the method that creates the loitering `GameHandler` objects, you are close to finding out where the objects should be removed. Scrolling down to line 124, you find that the `resetGameHandlerRemote()` method is the problem method. When the Loitering Objects mode is selected, the `_gameHandlerRemote.remove()` method is not called, so the references to the GameHandler objects are never removed.

Tip It is good programming practice to pair your calls to create and remove objects close together.

Step 6: Running the Session with Improved Code

You can re-run Minesweeper or Match Game in Normal mode with the corrected code. Repeat Step 1: Setting Up the Session through Step 3: Running the Session, selecting **Normal** mode instead of **Loitering Objects** and not re-naming the snapshot. You will see in the Instances view that the Heap Count for GameHandler is now 0, as predicted by the hypothesis.

This example demonstrates how to use JProbe to identify and remove loitering objects from your code.

Object Cycling Tutorial

The Match Game offers an Object Cycling mode. The Object Cycling mode demonstrates how over-allocating short-lived objects can cause the garbage collector to run more frequently than necessary. Garbage collection takes time and resources. The Normal mode shows that it is often more efficient to reuse strings.

Before beginning this tutorial, you need to deploy the Game Pack to the application server of your choice. For more information, see “Deploying the JProbe Game Pack Demo” on page 74.

Note The values cited in this tutorial reflect the Game Pack running on JBoss 4.2.2.GA, on Windows XP.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	Game Pack Demo: Match Game
Use Case:	Play three games without quitting.
Architecture:	**Object Cycling**: Use string concatenation. **Normal**: Append to an existing string buffer object.
Hypothesis:	The program does not create unnecessary temporary string objects.

The tutorial walks you through the following steps:

- Step 1: Setting Up the Session
- Step 2: Starting the Session and the Game Pack
- Step 3: Running the Session
- Step 4: Identifying Object Cycling
- Step 5: Investigating Object Cycling
- Step 6: Running the Session with Improved Code

Step 1: Setting Up the Session

You use the JProbe configuration tool to create the session settings for this example. In the setup procedure, only the settings that you need to change or verify are mentioned. If a setting is not mentioned, leave it blank or in its default state. The following procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1 On the JProbe Console, click **Tools** > **Create/Edit Settings**.

The Create/Edit Setting dialog box appears.

2. In the Manage Configurations pane, select **JBoss** and click **Add**.
3. In the Define a New Configuration screen, type a name in the Configuration Name field, such as *ObjectCycling*.
4. Click the appropriate version under **JBoss** and click **Next**.
5. Type the path to your server startup script in the text field or click … and navigate to it, then click **Next**.
6. In the Specify Your Code screen, in the Elements area, type the path to the deployed *gamepack.ear* file (for example,

   <jprobe_home>\demos\gamepack\dist\gamepack.ear) or click … and navigate to it.
7. Specify a Category/Program Name, then click **Next**.
8. In the Select a JProbe Analysis screen, under Analysis Type, select **Memory**, then click **Next**.
9. Click **Next** again to pass the Specify the JProbe Options screen.
10. In the Save the Configuration screen, review the settings, then click **Save** and browse to a location to save the settings file that you just created.
11. In the Configuration Complete screen, select **Integrate**, then click **Finish**.
12. In the Integrating dialog box, click … to navigate to the location where you want to save the startup script (for example, in Windows: *run_WithJProbe.bat*, and in Unix/Linux: *run_WithJProbe.sh*), then click **Save**.
13. When you see the `Integration complete` message, select the **Close Create/ Edit Settings tool on successful integration** check box and click **Close**.

Step 2: Starting the Session and the Game Pack

In this step, you start JBoss using the startup script you created in Step 1: Setting Up the Session and connect to it from JProbe. Then you open a browser to run the Game Pack demo.

To start the JProbe session and the Game Pack:

1. Start JBoss using the startup script:

   In Windows: `>C:\<jboss_home>\bin\run_WithJProbe.bat`

   In Unix/Linux (sh shell): `>run_WithJProbe.sh`

In Unix/Linux (csh or ksh shells): `>./run_WithJProbe.sh`

2. In the JProbe Console, click **Attach to Session** .
3. Click **OK** in the Attach to Running Session dialog box.

   The Runtime Summary view opens, with the Memory Pools tab on the foreground.
4. Open a browser and go to *http://localhost:8080/gamepack/index*.

   The Game Pack Demo login page appears.
5. Enter your user ID and password and click **Login**.

   Note The first time you do this, you need to create a user name and password for yourself. For more information, see "Creating a User Account for Game Pack" on page 81.

Step 3: Running the Session

In this step, you turn on garbage monitoring so that you can see how many objects are garbage collected during your use case. You work through a use case by playing three consecutive games. It does not matter for the analysis whether you win or lose the games. However, it is important that you start a use case before you begin to play, or JProbe will not perform a garbage collection. After you have played three games, you end the use case, and JProbe takes a snapshot.

To run a game with the Object Cycling fault:

1. In the Game Pack demo, select **Play** beside Match Game.
2. Select the **Object Cycling** option.

   Note Clicking the link displays the option's definition.
3. In the JProbe Runtime Summary view, click **Set Recording Level** on the toolbar.
4. In the Set Recording Level dialog box, select **Record Allocations, Stack Traces, and Garbage Data For All Instances** and click **OK**.
5. In the Game Pack demo, click **Start** and play the game.
6. When the game ends, play the game twice more (without quitting) for a total of three complete games.
7. Click **Quit**.
8. In the Runtime Summary view, click **Set Recording Level** on the toolbar.

9 In the Set Recording Level dialog box, select **Data Recording Off**, then click **OK**.

JProbe takes a snapshot and displays it in the Snapshot Navigator panel.

10 In the Snapshot Navigator, right-click this snapshot, select **Save Snapshot As**, and navigate to where you want to save the snapshot.

11 Name the snapshot *object_cycling* and click **Save**.

The new name is displayed in the Snapshot Navigator.

12 Click **Detach from Running Session** .

Note You can also close your application server and the Game Pack demo browser.

The Instances view appears after a few seconds,.

Step 4: Identifying Object Cycling

In this step, you look for classes and methods that allocate short-lived objects.

The Garbage Collections chart in the GC Data tab displays steep spikes, which means that some set of objects is garbage collected soon after being created. In the Instances view, look for classes with high Dead Count values and no or very few instances still alive. In this example, you can see that many instances of `StringBuffer` were allocated and garbage collected. None of the instances are still alive. When you review the results, you see that most of the `StringBuffer` objects were allocated by `*_StringConcatenation` methods in the `MatchGameRenderer` class.

To identify short-lived objects:

1 If the *object_cycling* snapshot is not open, right-click it in the Snapshot Navigator and select **Open Instances View**.

The Instances view appears.

2 Select **Dead Count** from the *Investigate by* list, and sort the table by **Dead Count**.

You can see that many `String` and `StringBuffer` instances are immediately garbage collected (that is, high **Dead Count** values and low **Recorded Count** values).

3 Drill into the Merged Allocation Points and Call Traces views, by right-clicking the `StringBuffer` instance and selecting the **Open Merged Allocation Points View** and **Open Call Traces View**, respectively.

You can see that most of the StringBuffer objects were allocated by renderGameMap_StringConcatenation, renderGamePlay_StringConcatenation, and renderSnapshot_StringConcatenation methods in the MatchGameRenderer class, demos.gamepack.matchgame package.

Step 5: Investigating Object Cycling

In Java, the JVM converts string concatenations into StringBuffer objects, which means that each concatenation creates a new object with a very short life span. This is a less efficient way to handle strings.

Take a look at the source code to see exactly how the application works. The _StringConcatenation methods are in the MatchGameRenderer.java source code. You can use any editor to examine the allocating methods, but it makes it easier to find the method if your editor has a search feature. Remember that in this case the code contains the fixed code as well.

To investigate the garbage collected objects:

1. Navigate to *<jprobe_home>/demos/gamepack/src/demos/gamepack/matchgame* and open the *MatchGameRenderer.java* file in a source code editor.
2. Search for the top method: renderGamePlay_StringConcatenation.
3. Observe that the method contains many string concatenations. If you scroll down to find renderSnapshot_StringConcatenation and then renderGameMap_StringConcatenation, you can see that these methods also use string concatenation.

Step 6: Running the Session with Improved Code

The file *MatchGameRenderer.java* also contains methods that offer a better way to handle the strings. While you have the source code open, you can scroll to find these improved methods:

- renderGamePlay_StringBufferAppend
- renderSnapshot_StringBufferAppend
- renderGameMap_StringBufferAppend

If you want, you can re-run this tutorial using Normal mode, which uses the improved methods. When you check the Instances view, you should see that the number of short-lived StringBuffer objects is reduced significantly.

This example demonstrates how to use JProbe to identify and remove object cycling problems in your code.

Performance Bottleneck Tutorial

The Minesweeper game can be played in the Method Time mode. This mode demonstrates a performance bottleneck caused by an inappropriate algorithm that is used for rendering the Minesweeper game board as one large image. For comparison, you can play the game in the Normal mode, which creates the game board as a table containing HTML links to images.

Before beginning this tutorial, you need to deploy the Game Pack to the application server of your choice. For more information, see "Deploying the JProbe Game Pack Demo" on page 74.

Note The values cited in this tutorial reflect the Game Pack running on JBoss 4.2.2.GA, on Windows XP.

The following table summarizes the types of information you need to know before starting this tutorial.

Program:	Game Pack Demo, Minesweeper Game Entry point of interest: `GameController.doGet`
Use Case:	Run minesweeper and select one tile.
Architecture:	**Method Time**: The game board is constructed with images on the server-side, encoded as a single image, and sent to the browser. You may notice the image jumps when a tile is selected. **Normal**: The game board is created as a table of links to images.
Hypothesis:	Encoding is slow. Creating the game board with links to images will be faster and smoother, especially if the images are cached by the browser.

The tutorial walks you through the following steps:

- Step 1: Setting Up the Session
- Step 2: Starting the Session and the Game Pack
- Step 3: Running the Session
- Step 4: Identifying the Performance Bottleneck
- Step 5: Running the Session with Improved Code
- Step 6: Measuring the Performance Improvement

Step 1: Setting Up the Session

You use the JProbe configuration wizard to create the session settings for this example. In the setup procedure, only the settings that you need to change or verify are mentioned. If a setting is not mentioned, leave it blank or in its default state. The following procedure assumes that you are running JProbe locally on your computer.

To set up the session:

1. On the JProbe Console, click **Tools > Create/Edit Settings**.

   The Create/Edit Setting dialog box appears.

2. In the Manage Configurations pane, select **JBoss** and click **Add**.
3. In the Define a New Configuration screen, type a name in the Configuration Name field, such as PerformanceBottleneck.
4. Click the appropriate version under **JBoss** and click **Next**.
5. Type the path to your server startup script in the text field or click ... and navigate to it, then click **Next**.
6. In the Specify Your Code screen:

   a. In the Elements area, type the path to the deployed *gamepack.ear* file (for example, *<jprobe_home>\demos\gamepack\dist\gamepack.ear*) or click ... and navigate to it.

   b. Click **Create Filters**.

      The table is populated with the filters available for your application.

   c. Select the My Application Filters check box.

      All application filters are now selected and Action set to **include**. JProbe can use these application filters as default data collection filters when performing a Performance analysis.

d Specify a Category/Program Name, then click **Next**.

7 In the Select a JProbe Analysis screen:

a Under Analysis Type, select **Performance**.

b In the General tab, select the **Detect Deadlocks** check box.

c In the Filters tab, ensure that all collection filters are selected and set Action to *line* for each of them.

d Click **Next**.

8 Click **Next** again to pass the Specify the JProbe Options screen.

9 In the Save the Configuration screen, review the settings, then click **Save** and browse to a location to save the settings file that you just created.

10 In the Configuration Complete screen, select **Integrate**, then click **Finish**.

11 In the Integrating dialog box, click … to navigate to the location where you want to save the startup script (for example, in Windows: *run_WithJProbe.bat*, and in Unix/Linux: *run_WithJProbe.sh*), then click **Save**.

12 When you see the `Integration complete` message, select the **Close Create/ Edit Settings tool on successful integration** check box and click **Close**.

Step 2: Starting the Session and the Game Pack

In this step, you start JBoss using the startup script you created in the Step 1: Setting Up the Session and connect to it from JProbe. Then you open a browser to run the Game Pack demo.

Note The first time you do this, you need to create a user name and password for yourself.

To start the JProbe session and the Game Pack:

1 Start JBoss using the startup script:

In Windows: `>C:\<jboss_home>\bin\run_WithJProbe.bat`

In Unix/Linux (sh shell): `>run_WithJProbe.sh`

In Unix/Linux (csh or ksh shells): `>./run_WithJProbe.sh`

2 In the JProbe Console, click **Attach to Session** .

3 Click **OK** in the Attach to Running Session dialog box.

The Runtime Summary view opens, with the Memory Pools tab on the foreground.

4 Open a browser and go to *http://localhost:8080/gamepack/index*.

The Game Pack Demo login page appears.

5 Enter your user ID and password and click **Login**.

Note The first time you do this, you need to create a user name and password for yourself. For more information, see “Creating a User Account for Game Pack” on page 81.

Step 3: Running the Session

In this step, you exercise the use case by selecting one tile. It does not matter for the analysis whether the tile reveals a number or a mine. In Method Time mode, the entire game board is redrawn on the server-side, encoded, and sent to the browser. You should find that the game responds slowly.

To run a game with a performance bottleneck:

1 In the Game Pack, click **Play** beside Minesweeper.

2 Select the **Method Time** option.

Note Clicking the link displays the option’s definition.

3 In the JProbe Runtime Summary view, click **Set Recording Level** on the toolbar.

4 In the Set Recording Level dialog box, select **Full Encapsulation** and click **OK**.

5 In the Game Pack, click **Start** and click any one tile.

6 Click **Quit**.

7 In the Runtime Summary view, click **Set Recording Level** on the toolbar.

8 In the Set Recording Level dialog box, select **Data Recording Off**, then click **OK**.

JProbe takes a snapshot and displays it in the Snapshot Navigator panel.

9 In the Snapshot Navigator, right-click this snapshot, select **Save Snapshot As**, and navigate to where you want to save the snapshot.

10 Name the snapshot *minesweeper_methodtime* and click **Save**.

The new name is displayed in the Snapshot Navigator.

11. Click **Detach from Running Session** .

    Note You can also close your application server and the Game Pack demo browser.

    The Call Graph view appears after a few seconds.

Step 4: Identifying the Performance Bottleneck

In this tutorial, you use the Call Graph view to identify a hotspot. A hotspot is an expensive method, one that takes more time than necessary to run. The hotspot method may be the performance bottleneck or the method may call another method that causes the slowdown. In this example, you find two expensive third-party methods that are called by one of the Game Pack methods.

To identify the performance bottleneck:

1. Right-click the *minesweeper_methodtime* snapshot and select the **Open Call Graph View**.
2. Select **Cumulative Time** from the *Color By* list.

   Tip You can change the default Red-Gray color scheme. Right-click the color scale (located between the graph and the list) and select a different color scheme.

   Nodes on the left of the graph are bright red, because each node contains the cumulative time of its children and their call trees. As you follow a branch to the right, the red color fades, because the cumulative time of each node includes fewer method call trees.
3. Identify the critical path.

   In this case, we know from our preliminary work that the critical path of the Game Pack demo is started by the call to `GameController.doGet`.

   The parallel branch, started by `GameImageServlet.doGet`, is actually initiated from a method in the main call tree. The `GameImageServlet.doGet` branch does the work of encoding the completed minesweeper game board image for the browser.

   Note In this example, we focus only on the critical path. However, later when you compare these results against the Minesweeper game in Normal mode, you will find that the encoding servlet is no longer required by the improved algorithm, which saves all the time used by this branch.
4. To isolate the critical path, select `GameController.doGet` method and click **Isolate Subtree** on the toolbar.

Percentages are recalculated to include only the call trees for this method. The cumulative time for the `GameController.doGet` method is therefore 100%.

5 Follow the brightest nodes in the branch from left to right until the bright color fades or you reach the end of the branch. In this example, the last nodes are `MediaTracker.waitForID()` and `Toolkit.getDefaultToolkit()`.

Note You may need to expand the branch to see the last nodes. `Toolkit.getDefaultToolkit()` displays in the graph as an encapsulated node. To view this node, you might have to increase the number of nodes that are displayed in the Call Graph.

Tip You may notice a small lock icon on some of the nodes. The lock means that data on the method was encapsulated. By default, details are encapsulated for third party and framework methods. Method level detail is only set for your packages, which focuses your analysis on your own code. For more information, see Encapsulate Filter in the online help.

6 Select `MediaTracker.waitForID()`. In the method list (lower panel), you can see that the method time is 274 ms and it is called 200 times.

7 Select `Toolkit.getDefaultToolkit`. The method time is 1 ms and it is called 200 times.

Both hotspot methods are from the `java.awt` framework methods. You cannot modify the code for these methods; all you can do is change how or if you call them. You need to find which of the Game Pack methods initiates the calls to these methods.

8 Travel the call tree back to `GameImageManager.loadImage` and then to `MineSweeperRenderer.renderMineMapImage_MethodTime`. This is the method that starts the process that calls the `java.awt` methods.

Conclusion: The performance bottleneck is caused by the Game Pack method `renderMineMapImage_MethodTime` calling these expensive, third-party methods for every tile in the game board.

Step 5: Running the Session with Improved Code

After you discover the performance bottleneck, you can choose how best to fix your code. Review the source code. You could attempt to call a less expensive method or you might choose an entirely new algorithm to do the same task. For the Game Pack demo, we decided to use a different way to build the game board, one that does not require that the images be loaded and encoded on the server side. Instead, the game board is simply a table of HTML links. If image caching is enabled in the browser, the images are stored and used locally; otherwise, the links point to images on the Web server.

To run the session with improved code:

1. Restart the session following the instructions in Step 2: Starting the Session and the Game Pack.
2. Repeat Step 3: Running the Session, but select **Normal** mode instead of the **Method Time** option, and save the snapshot as *minesweeper_normal*.
3. Right-click *minesweeper_normal* and select the **Open Call Graph View**.
4. To isolate the critical path, select `GameController.doGet` and click **Isolate Subtree** on the toolbar.

   Percentages are recalculated to include only the call trees for this method. The cumulative time for the `GameController.doGet` method is therefore 100%.
5. Follow the path of red nodes to find the method that contains the new algorithm. In this example, it is called `renderMineMapImage_Normal`.

   The new method `renderMineMapImage_Normal` runs faster than the original `renderMineMapImage_MethodTime` method. Also, the new method does not require the encoding servlet, so overall the program is even faster.
6. Exit the Game Pack Demo and end the session. Close the browser.

Step 6: Measuring the Performance Improvement

You know that the image caching algorithm runs much faster than the original compression algorithm. To quantify the improvement, use the Snapshot Difference tool to compare snapshots.

To measure the performance improvement:

1. In the JProbe Snapshot Navigator, right-click the *minesweeper_normal* snapshot and select **Snapshot Differencing**.

   The Performance Difference dialog box appears. The selected snapshot is displayed in the Snapshot to Compare field.
2. Select *minesweeper_methodtime* from the Baseline Snapshot list, and click **OK**.

   If you isolated on different methods, you see a message informing you that the snapshots have different transformations. The isolate action, among others, is removed automatically to ensure that you are comparing the same data set.

   The Performance Difference view opens. By default, only the classes with differences are displayed.

3 To see the impact that changing the algorithm had on the servlet, isolate the `doGet` method by typing `*.doGet()` in the Filter Methods field.

The `GameController.doGet` and `GameImageServlet.doGet` methods are the only methods displayed. A negative number in the Cumulative Time column means an improvement. The Normal mode offers a 76% improvement over the Method Time mode, which is a significant difference.

Note The actual percentage may be different on your system.

This example demonstrates how, in your own code, an inefficient algorithm can significantly impact performance. Of course, you only want to optimize algorithms in the critical path of your program; there is no point tuning algorithms that are called very rarely. You also need to evaluate the overall impact on the runtime of the program. If an inefficient algorithm takes a total of a few seconds to execute, you may make it run faster, but the impact on the overall runtime of the program would be negligible.

Index