DevPartner Java Edition

Getting Started Guide

Release 4.5

20091013100444

Table of Contents

Preface

This manual describes how to get started using Micro Focus DevPartner Java Edition.

Who Should Read This Manual

This manual is intended for new DevPartner Java Edition users and for users of previous versions who want an overview of new functions and interface changes. It is designed to help you understand how DevPartner Java Edition can help you be a more productive software developer, and to get you started using the software. It is **not** a comprehensive user's guide.

New users should read Chapter 1, "Analyzing Problems in Java Applications" for a survey of DevPartner Java Edition concepts. Subsequent chapters show how to use individual features during a software development cycle.

Users of previous versions of DevPartner Java Edition should read the Release Notes to see how this version differs from previous versions.

This manual assumes that you are familiar with the Windows or UNIX operating environments and with Java software development concepts.

What This Manual Covers

This manual contains the following chapters and appendixes:

- Chapter 1, "Analyzing Problems in Java Applications" describes the problems that DevPartner Java Edition can help Java programmers uncover.
- Chapter 2, "Finding Memory Problems" explains how to use DevPartner Java Edition to analyze how your Java program uses memory.
- Chapter 3, "Ensuring Testing Consistency" describes the concepts that underlie the DevPartner Java Edition software's ability to capture, display, and merge test coverage information.
- Chapter 4, "Finding Performance Problems" explains how to locate performance bottlenecks in your Java applications.
- Chapter 5, "Working with Integrated Development Environments" describes how to use DevPartner Java Edition from within various IDEs.
- Chapter 6, "Working with Application Servers" describes how to profile code running through application servers.

Conventions Used In This Manual

This book uses the following conventions to present information.

- Interactive features of the DevPartner Java Edition user interface appear in **bold typeface**. For example:

  To update the information displaying in the **Application Testing** tab of the Start page, click **Refresh**.

- Computer commands appear in `monospace typeface`. For example:

  Execute the `nmjava` command.

- File names and paths appear in boldfaced monospace typeface. For example:

  The session file is saved in the **`/var/sessionfiles`** folder.

- Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in ***`italic monospace type`***. For example:

  Enter **`http://`*`servername`*`/cgi-win/itemview.dll`**, where ***`servername`*** is the designation of your server.

Getting Help

If ever you have any problems or you would like additional technical information or advice, there are several sources. In some countries, product support from Micro Focus may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described below. If you obtained it from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us as described below.

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Product Support can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name, release (version), and build number of the product.
- Installation information, including installed options, whether the product uses local or network databases, whether it is installed in the default folders, whether it is a standalone or network installation, and whether it is a client or server installation.
- Environment information, such as the operating system and release on which the product is installed, memory, hardware/network specifications, and the names and releases of other applications that were running.
- The location of the problem in the product software, and the actions taken before the problem occurred.
- The exact product error message, if any.
- The exact application, licensing, or operating system error messages, if any.
- Your Micro Focus client, office, or site number, if available.

Contact

Our Web site gives up-to-date details of contact numbers and addresses. The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus Product Support customer, please see your Product Support Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

Chapter 1

Analyzing Problems in Java Applications

This chapter provides an overview of DevPartner Java Edition. It describes the kind of performance problems that might hamper a Java application, and how to use DevPartner Java Edition to diagnose those problems. For a more thorough exploration of Java program performance, see *Java Platform Performance — Strategies and Tactics* by Steve Wilson and Jeff Kesselman, or *Effective Java* by Joshua Bloch.

For information on how to install DevPartner Java Edition and to optimize its overall performance, refer to the *DevPartner Java Edition Installation Guide*. For direct information on how to use the product, refer to the task-oriented online help.

When Good Programs Produce Bad Results

Regardless of experience level, every Java programmer occasionally comes face-to-face with a not-so-obvious application performance problem. The problem might manifest itself as unexpected behavior. For example, a Web application might run smoothly when accessed by 1,800 simultaneous users, but lock up when the number of users exceeds 2,000. An otherwise swiftly running application might take forever to start up.

When an application goes awry, the symptoms it exhibits may have a single cause, or they might result from a combination of causes. For example, an application that runs slowly might be using an inefficient algorithm, or it might have a memory leak, or it may have a combination of a memory leak and an inefficient algorithm.

The kinds of symptoms that good programs exhibit when they go bad fall into several broad categories:

- Scalability problems – Code that works well with one set of users or inputs, but degrades when the set expands.
- Perception of performance – Start-up delays, applications that take too much time drawing a window, cursors that disappear for long periods.
- Excessive CPU use – Sluggish graphics performance, slow turnaround on database transactions.
- Sharing critical resources, such as memory– Conflicts among multiple applications using memory, starving threads in multi-threaded applications, excessive physical and virtual memory use, applications that slow down over time.
- Adequate testing and application stability – Untested program sections, an unstable code base.

The following sections take a closer look at these broad categories. See “Finding Problem Code with DevPartner Java Edition” on page 13 for a strategy for tracking down these kinds of problems.

Scalability

Scalability can be an issue for any Java application, particularly Web-enabled applications. The design of an application should take into account how far its major operations will scale. Designers need to ask questions such as:

- How many simultaneous transactions should my application perform?
- What is the upper limit on the number of users that will not bog down my application?
- How many records per unit of time should my application be able to process?

Many factors can contribute to how scalable an application will be, including the amount of memory it uses, the number of objects it instantiates and discards, the number of explicit notifications it receives from a server process, plus many others. The key task is to understand how much of a given resource each additional user/task consumes.

Perception of Performance

How a program appears to perform will vary from user to user, but almost every user will find certain aspects of performance to be unacceptable, for example:

- The cursor disappears for more than 10 seconds
- Characters typed on the keyboard appear on the screen only after a delay

Other performance perceptions are more subtle: a program that takes a long time to load, or an application where the user does not know what is going on. Application designers should ask questions such as:

- Has the application provided visual clues about what is happening?
- Can program start-up procedures be shortened or deferred?

Using multiple threads is one of many ways to improve the perception of performance.

Intense Computational Operations

Computational problems, such as calculating a Fourier transform, require the construction of simple or complex algorithms. Some algorithms often scale unacceptably as the amount of data to be computed grows. Choosing algorithms is as much an art as a science.

Computational problems might also be caused by the choice of data structures in a program. Data structures are used to store, retrieve, and manipulate data, and it can take time to access this data storage depending on the type of structure.

Before you can experiment with alternate algorithms or data structures, you need to understand what parts of your program might be causing performance bottlenecks.

Memory Behavior

Even though Java was designed to eliminate many of the memory allocation problems that plague other languages, it still has memory issues that can cause poor program performance. In spite of its automatic garbage collectors, Java programs can still create memory leaks, that is, blocks of allocated memory that never get released, either due to programmer inattention or to non-optimal data structure design. In addition, it is important to understand exactly how much total memory an application uses. For example, if an application uses so much memory that it forces the system to swap out to a virtual memory swap file, the application's performance suffers.

A memory leak is the leading suspect when you see a program's use of memory increase over time but never decrease, even after garbage collection. Eventually, such a program will crash.

See "Finding Memory Problems" on page 23 to learn more about applications that have memory usage problems.

Application Stability

While it can be important to test an application as you write it, tests can be invalidated by a code base that changes frequently. Calculating the changes to your code base, and identifying the parts that have been tested or not tested, helps you decide when an application is stable enough to deploy.

Testing strategy should provide answers to questions such as:

- How much of the application's code was actually tested, or more importantly, not tested?
- Are current tests adequate for exercising all of the application's functionality?
- Were tests revised to exercise newly added code?

See "Ensuring Testing Consistency" on page 51 for more information about adequately testing Java applications.

Finding Problem Code with DevPartner Java Edition

DevPartner Java Edition helps programmers track down the root causes of the anomalous symptoms that produce unwanted application behavior. Whether a program performs well or has problems, programmers will likely benefit from studying the results of running their Java applications with DevPartner Java Edition.

With DevPartner Java Edition, product development teams can track down specific run-time errors, memory anomalies, performance bottlenecks, and code instability problems across all tiers of a Java application environment. Without DevPartner Java Edition, teams rely more heavily on guesswork, wasting valuable time and sacrificing product quality and end-user confidence.

DevPartner Java Edition is an asset to all members of any Java development organization, not just to a few of its experts. Both developers and quality assurance engineers can use the comprehensive statistics and graphics to prioritize the results and focus on solving the complex quality issues associated with Java development.

DevPartner Java Edition provides an end-to-end understanding of problems associated with run-time performance, memory use, performance, multi-threading, and the test coverage of your:

- Java 2 Standard Edition (J2SE) programs, including applets and Java Web Start applications
- Java 2 Enterprise Edition (J2EE) programs, including servlets, Java Server Pages (JSP) and Enterprise Java Beans (EJBs)
- Third-party pure Java components

DevPartner Java Edition Diagnostic Capability

DevPartner Java Edition delivers these analytical capabilities:

- Computational and wait-time Performance analysis – Uncovering application slowdowns and bottlenecks
- Memory analysis – Detecting potential memory leaks, overall RAM footprint, and use of temporary objects
- Coverage analysis – Ensuring that code is thoroughly tested

Computational Performance Analysis

Performance analysis identifies performance bottlenecks in the entire multi-tier Java environment. DevPartner Java Edition delivers top-to-bottom profiling capability, including Java interpreted code, JIT-compiled code, third-party components, Java Server Pages (JSPs), servlets, and Enterprise Java Beans (EJBs), as well as Java Virtual Machine (JVM) and underlying system code that makes use of JVM APIs. See “Finding Performance Problems” on page 65 for more information.

Memory Analysis

Memory analysis identifies memory-intensive methods and inefficient lines of code. It monitors objects, references, and garbage collection, producing an accurate profile of the program's memory use based on RAM footprint, temporary objects, and memory leaks. With this data, developers can streamline the run-time performance and resource utilization of their code by optimizing methods that consume the most memory. Memory analysis helps to locate inefficient code that would otherwise take hours or days to find manually. See “Finding Memory Problems” on page 23 for more information.

Coverage Analysis

Coverage analysis helps developers and testers quickly identify untested code in Java applications, components, and Web pages. DevPartner Java Edition takes the guesswork out of development milestones such as code check-in, unit or integration testing, and final release. It reduces testing time and improves overall code stability by measuring and tracking code that was previously tested but is still undergoing development work, as well as code that might not have been thoroughly tested. It helps teams avoid redundant testing so they can focus on the effects of ongoing code design and development. See “Ensuring Testing Consistency” on page 51 for more information.

DevPartner Java Edition incorporates multi-thread analysis as part of Coverage analysis. It helps developers identify run-time threading problems that can lead to performance and reliability issues in the application. It pinpoints sections in code where Java threads synchronize out of order. Such sections can sometimes — but not always — result in a deadlock condition, and such intermittent deadlocks are notoriously difficult to diagnose. See "Tracking Code Execution and Code Base Stability and Reliability" on page 64 for more information.

When to Use DevPartner Java Edition

Table 1-1 provides a starting point for using DevPartner Java Edition to diagnose performance and memory problems in Java applications.

Table 1-1. Symptoms and Analysis Tools

Symptom	Analysis Tool
Performance degrades over time; improves after restarting the application, but degrades again.	Memory Leaks
Scalability problems; temporary performance degradation.	Object-Lifetime Analysis Memory Leaks Performance
Sluggish performance, does not improve after restarting the application.	Performance RAM Footprint
Application is slow to start	Performance RAM Footprint
Specific parts of the application are sluggish	Performance
Application hangs intermittently	Coverage (Out of order thread synchronization)

Java programmers can minimize performance and scalability problems in their applications by routinely running all DevPartner Java Edition analyses as a regular part of the development cycle. Duplicating those test runs with Coverage analysis identifies how much of the application is being tested and provides a running record of test coverage.

DevPartner Java Edition User Interface

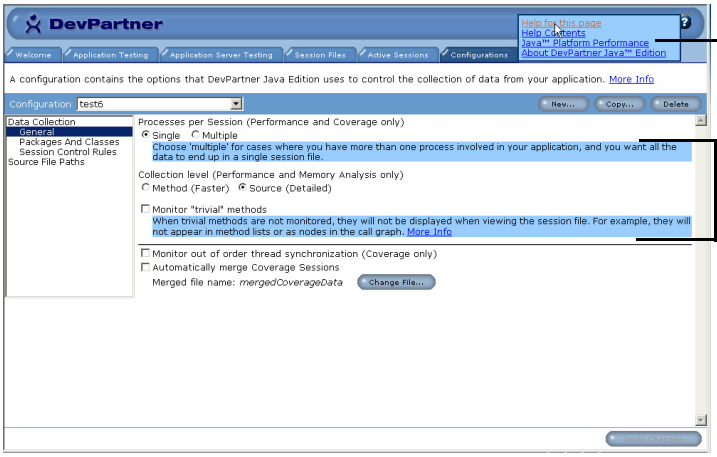
The DevPartner Java Edition user interface has been designed for the cross-platform, multi-browser Java environment. It is identical on a Linux, Solaris, or Windows platform.

Figure 1-1. DevPartner Java Edition Start Page

Required: The DevPartner Java Edition interface displays in a Web browser. If your browser includes a popup blocker, configure it to disable blocking for the DevPartner Java Edition window. If the popup blocker is enabled, DevPartner Java Edition will not operate correctly.

The session, results, and settings pages in DevPartner Java Edition include tabs, options, and graphics to guide you toward the desired result. The DevPartner Java Edition user interface also includes supplemental inline help text and a comprehensive online help system.

Figure 1-2. DevPartner Java Edition User Interface



Inline Help

The DevPartner Java Edition user interface contains concise inline help placed adjacent to controls and other visual items on a page. The inline text leads you to informed choices based on the current information that is displayed. You have the option to turn off the inline help on the results summary pages by clicking **Preferences**.

Integrated Online Help System

A help icon appears at the upper right corner of every page in the DevPartner Java Edition user interface. Click this icon to view:

- **Help for this page** – The online help topic that pertains to the current page. The topic provides details about the options, settings, and/or results that are displayed on that page.
- **Help Contents** – The online help system, with Table of Contents, Index, and Search tabs. In this window, you can select a particular topic of interest.
- **Java Platform Performance** – The online version of *Java Platform Performance — Strategies and Tactics* by Steve Wilson and Jeff Kesselman. This book provides information about performance tuning, including both high-level strategies and code-level performance tuning tactics. It is available through the Sun Microsystems Web site (http://java.sun.com).
- **About DevPartner Java Edition** – A dialog box containing product and release information.

Ways to View DevPartner Java Edition Diagnostics

DevPartner Java Edition performs intensive analysis on your Java programs. When results are generated, it displays the data in graphical displays, such as bar graphs, call trees, and tables. It simplifies the task of trying to comprehend the scope of the analysis results by identifying the top five or fewer items having the largest impact on your program (depending on the kind of analysis performed). From this high-level view, you can drill down into the complete details of the current analysis.

You can also manipulate the data that you want to see in various ways. You can select specific column headings for various results views. Through the **Preferences** dialog box for a Session Control page or Results Summary, you can specify the increments used to display the data:

- Precision – Zero, one, two, three, or four decimal places
- Time – Microseconds, milliseconds, or seconds
- Memory – Bytes, kilobytes, or megabytes

Consult the DevPartner Java Edition online help system for more information.

Command Line Utilities

Note: This section provides a general description of command line utilities. See the online help or Linux/Solaris man pages for complete syntax. Other capabilities, such as setting up a configuration or starting through an application server, are available from the DevPartner Java Edition browser interface. Refer to the online help to learn how to perform these functions.

The DevPartner Java Edition command line utilities enable you to launch and monitor a Java program directly from the command line, either interactively or in batch mode:

- `nmappletviewer` – Monitors your Java applets outside a Web browser. Using **`nmappletviewer`** is essentially equivalent to **`appletviewer.exe`**.
- `nmextract` – Exports data from DevPartner Java Edition session files to an ASCII file as comma-separated values, to an HTML file, or to an XML file.
- `nmjava` – Monitors your standalone Java programs. Using **`nmjava`** is essentially equivalent to using **`java.exe`**.
- `nmserver` – Monitors your Java code run in an application server (this capability is also available from the DevPartner Java Edition browser interface).
- `nmshell` – Monitors all Java programs run in the shell, whether the Java program is executed directly or indirectly (such as via a batch file).

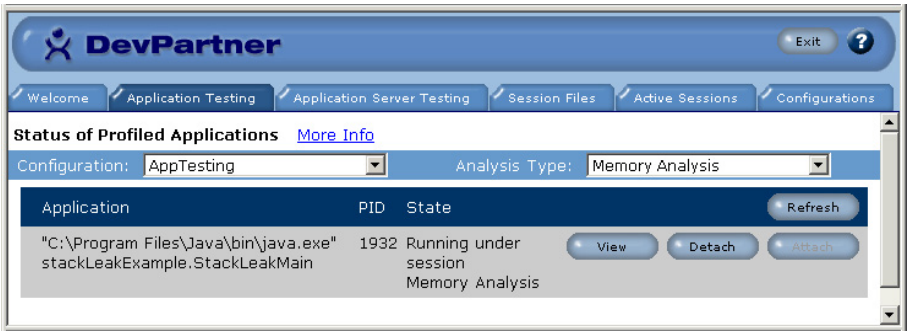
When a program is launched from the command line, a DevPartner Java Edition session starts. The Session Control page appears, unless you launch the application in batch mode, or you are running on a UNIX system. If the Session Control page does not appear automatically, you can start the DevPartner Java Edition user interface and navigate to the active session's control page.

Detaching and Reattaching a Profiled Application

After you start an application through a DevPartner Java Edition command line utility, the application information is displayed in the **Application Testing** tab of the Start page. Through this page, you can detach the application from the profiling session without stopping the application; you can reattach the running application to create another profile, optionally selecting a different configuration and/or analysis type for the new session.

While the application is being profiled, the session is listed in the **Active Sessions** tab. You can review the session data by clicking **View** on the **Application Testing** tab to display the Session Control page.

Figure 1-3. Application Testing



Controlling Data Collection Results

DevPartner Java Edition provides three ways to control when data is collected during the use of your application, each of which gives you progressively finer granularity of control:

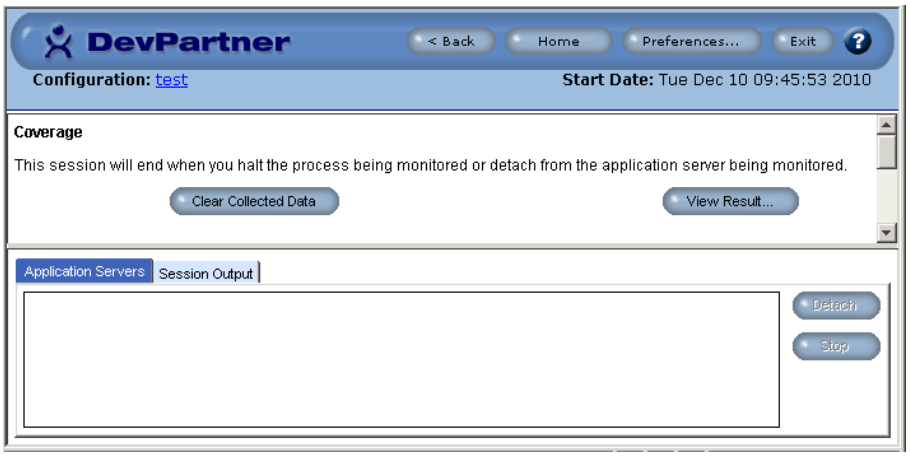
- Use the Session Control page for the active session to interactively control data collection as your program runs.
- Use the Session Control rules to assign Session Control actions to specific methods in your application modules.
- Use the Session Control API to control data collection in your program.

Using Session Controls

You can use the session controls on the Session Control page to control when DevPartner Java Edition collects data.

While your program is running, the Session Control page is displayed (unless you are running on UNIX or started the session in batch mode, as noted earlier). The **Clear Collected Data** and **View Results** buttons let you focus data collection on the portions of your code that are significant to you.

Figure 1-4. Session Control Page for Coverage Analysis



Using Session Control Rules

Session control rules let you specify actions to occur at fixed places in your application code. DevPartner Java Edition adds Session Control Rules that you create to the configuration file. The session control rules enable you to:

- Specify Session Control actions prior to beginning a session
- Specify Session Control actions without altering the module's source code

Using the Session Control API

The Session Control API lets your program code start and stop data collection anywhere within a method.

Table 1-2. Session Control API

Session Control API	Description
TakeSnapshot()	Requests that the profiler take a snapshot.
ClearData()	Requests that the profiler clear all collected data. If you are interested in the data that was collected, do not call **`ClearData()`** until after you call **`TakeSnapshot()`**. Note that RAM Footprint data is not cleared by **`ClearData()`**.
RequestGarbageCollection()	Requests that the VM perform a garbage collection. This is essentially the same as calling **`System.gc(*)`** and **`Runtime.runFinalization()`** a few times
Mark()	Informs the profiler that all objects that are allocated from this point forward are of interest, not just objects allocated within non-excluded code
UnMark()	Reverses the effect of the **`Mark()`** call. If you are interested in the data that was collected, do not call **`UnMark()`** until after you call **`TakeSnapshot()`**.

Configuring Data Collection Options

To get the results you want from DevPartner Java Edition, you may need to modify the session configuration options.

You can control the classes that are included in any DevPartner Java Edition analysis session by changing the exclusion list in the configuration file for the session. As an alternative, you can create an inclusion list to collect data only for specific classes.

See the online help for more information about using session controls, and configuring data collection options.

Comparing Profiling Sessions

When you generate two or more session files using the same configuration, you can compare any two files for Performance analysis or Coverage analysis. For example, you can analyze performance, change the program code, and run another analysis, then compare the two profiles to see how performance is improved by the change.

The **Session Files** tab of the Start page includes a link to the Compare Sessions screen, through which you select the two files to compare.

The Comparison Results Summary displays the results of the two files side by side. Each side displays all the information provided when you display the Results Summary for an individual file. The graphical display makes it easy to see the differences between the two results files.

What's Next

The remaining chapters in *the DevPartner Java Edition Getting Started Guide* provide insight and guidance to help you get the most out of DevPartner Java Edition. Each chapter presents a more detailed explanation of each DevPartner Java Edition feature, along with sample applications, usage scenarios, and analysis of the results.

Chapter 2

Finding Memory Problems

When running a Java program under Memory analysis, DevPartner Java Edition can:

- Show the amount of memory consumed by an object or class
- Track the references that are holding an object in memory
- Identify the lines of source code within a method responsible for allocating the memory.

More important, DevPartner Java Edition presents memory data in context, letting you navigate object reference chains and calling sequences of the methods in your code. This provides both an in-depth understanding of how a program uses memory and the critical information needed to optimize memory use.

This chapter describes potential memory issues in Java programs and shows how to use Memory analysis to improve Java application performance.

Memory Problems in Java Applications

Java frees developers from much of the effort of explicitly handling memory in applications. However, memory allocation and use in Java applications can still cause performance bottlenecks and resource depletion.

Any of these symptoms may indicate a performance problem:

- Slows down over time.
- Runs slowly, or slows down noticeably when you perform certain operations.
- Performs poorly under load conditions.
- Performs poorly when other applications are running.

But how do you know if the problem is memory-related?

A given number of a Java application's classes must be loaded before the program can execute a particular function. Is your program tying up memory resources by immediately loading classes that will not be needed unless a particular task is performed? How many instances of a particular class does your application create? How many do you really need?

Every program must create and allocate objects to do anything useful. Object allocation always incurs memory costs. How do you know if your program is allocating too many objects, or allocating them efficiently? Are the objects your program allocates being cleared by the garbage collector? Are they being collected when you expect them to, or are they remaining in memory long after their usefulness has passed?

Memory analysis provides a comprehensive view of the way your Java application uses memory. DevPartner Java Edition provides three different types of Memory analysis, designed to help you isolate different kinds of memory-related problems. Regardless of which type of analysis you use, all include the following features:

- Real-time graph – DevPartner Java Edition presents a live view of your application's memory use as it runs. You can see how much memory is being used by your application code (profiled code), system and other application code (excluded code), and how memory consumption compares to the memory reserved by the Java Virtual Machine (JVM).
- Dynamic list of classes – DevPartner Java Edition updates the list of profiled classes in real time, showing you the number of objects allocated and number of bytes used by each class, as your application runs.
- Static views – You can capture a static view of the heap at any time during program execution. DevPartner Java Edition saves this data in a session file that you can then use to analyze memory problems in depth. The interface provides multiple ways to drill down into the session data, so you can see in detail how your application uses memory and ultimately identify the methods or lines of code responsible for the most memory use.

Running a Memory Analysis Session

DevPartner Java Edition can help you quickly determine the way your application uses memory resources, revealing current or potential problem areas. To run a standalone Java application under DevPartner Java Edition, open a console window and use the `nmjava` command

```
nmjava -mem ApplicationName
```

Running `nmjava` with the `-mem` option starts DevPartner Java Edition (if it is not already running) and launches your application in a Memory analysis session.

Note: For information on running different types of Java applications under DevPartner Java Edition, including command line options, see the online help.

When you run a Memory analysis session, you can choose to examine one of these important potential problem areas:

- Memory leaks, including object retention
- Temporary object creation
- Overall RAM footprint

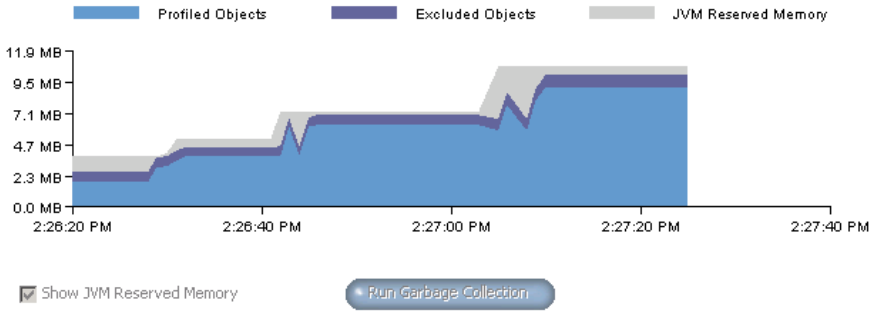
Table 2-1. Performance Symptoms and Memory Analysis Tools

Symptom	Analysis Tool
Performance degrades over time; recovers on restart. Performance improves after restarting the application, but degrades again.	Memory Leaks
Scalability problems; temporary performance degradation.	Object-Lifetime Analysis Memory Leaks
Sluggish performance, does not improve after restarting the application.	RAM Footprint

The first thing you will see when running any Memory analysis session is the real-time graph on the Session Control page. The real-time graph provides a visual representation of how your application is using memory resources. Observe the pattern the graph makes as you exercise your application. Different memory problems create characteristic patterns, so the real-time graph provides the first clue to the existence and nature of a memory problem.

For example, if the graph shows a rising pattern that never returns to baseline, as in Figure 2-1, your application is probably leaking memory. You may suspect that the progressive slowdown of your application noticed by your QA team is consistent with a memory leak; the real-time graph will confirm that diagnosis.

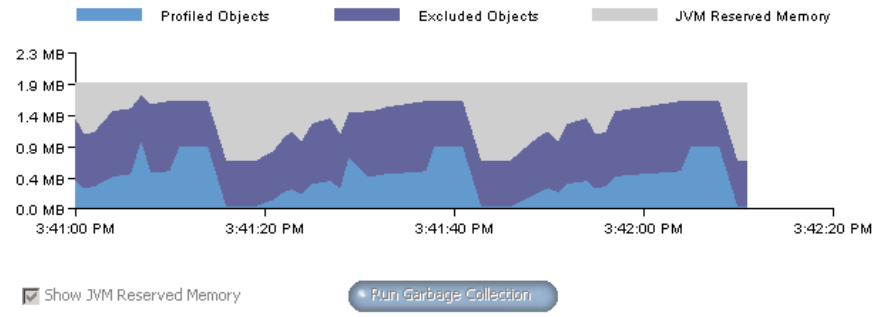
Figure 2-1. Memory Analysis Session Control Page



Among the possible causes of memory leaks are objects that are allocated in memory when they are accessed by the program, but not freed after they are no longer needed by the program. Running an Object-Lifetime analysis can identify the objects that survive garbage collection even though they are not being used.

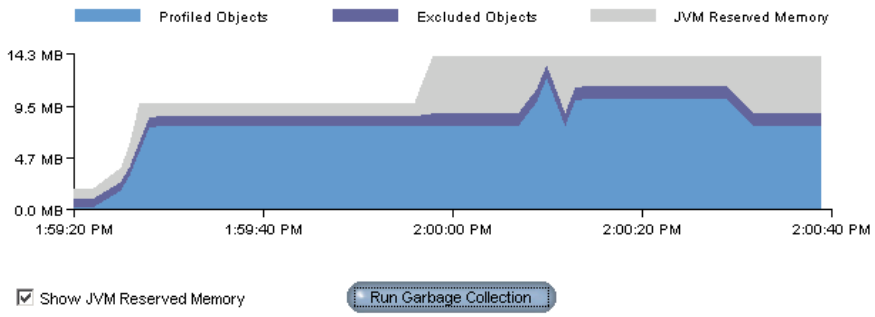
If the graph does return to baseline, but is characterized by periodic spikes in memory use, your application is creating large numbers of objects as it runs. Granted, the memory allocated is being freed, but such an application may not scale well under load. If your application slowdown occurs in response to an increase in users or inputs, it could indicate a scalability issue. Again, the real-time graph will indicate of the nature of the problem, enabling you to immediately point your diagnostic efforts in the right direction.

Figure 2-2. Real-Time Graph — Memory Spikes



Even in the absence of a suggestive pattern, the real-time graph can provide important information. For example, if your application consistently consumes nearly all the JVM reserved memory, and that amount is large relative to the anticipated resources of your target users' systems, you may want to reduce the overall memory footprint of your application. These cases and their implications for application performance are detailed in this chapter.

Figure 2-3. Real-Time Graph — Reserved Memory Usage



Locating Memory Leaks

The amount of memory consumed by your application can have a major impact on how well the application performs. The larger the amount of memory consumed (RAM footprint), the more likely it is that the application will run slowly and scale poorly. Memory leaks — the allocation of memory that is never reclaimed — do occur, even in the garbage-collected Java environment; they can bloat your application's RAM footprint. Java's automatic garbage collection relieves you of the responsibility to explicitly free the objects you create. But it is still possible to retain references to objects that the program will never use again. Java's automatic garbage collection makes leaks more subtle, and therefore more difficult to locate. Java memory leaks are, in a real sense, *forgotten objects*.

As long as a reference to an object exists, that object is considered to be a *live object* by the garbage collector, and so it cannot be collected. Such references can be difficult to track down. That is where Memory analysis helps you.

DevPartner Java Edition provides sample applications to help you understand how to run Memory analysis sessions and interpret the results.

Running a Memory Leak Analysis

The sample application for memory leak analysis is called **`stackLeakExample`**.[1] It is located in the DevPartner Java Edition installation path.

Tip: The file paths should be typed as a single line with no breaks.

- Windows — ***`DPJ_dir`*`\samples\stackLeakExample`**

  where ***`DPJ_dir`*** is the DevPartner Java Edition product folder

- UNIX — **`/opt/Micro Focus/DPJ/samples/stackLeakExample`**

This application demonstrates memory leaks. It uses a stack with two operations, push and pop. The stack is implemented by an array of stack entries. A push results in a new stack entry, which is assigned to top of the stack. A pop discards the top stack entry.

The leak, a bug in the pop method, occurs because the array element holding the popped stack entry is not zeroed out. Hence, the garbage collector cannot collect the discarded stack entry.

1.This example was inspired by Joshua Block, *Effective Java Programming Language Guide.* Addison-Wesley, Boston, MA, 2001. ISBN:0-201-31005-8.

When a subsequent push assigns a new stack entry to the array element, the old stack entry is collected, but this condition cannot be guaranteed.

At any point in time the stack can have a number of leaks, which can be substantial if the size of the stack entry is large.

To run this example and see the leaks, follow these instructions:

1. At the command prompt, change the folder to

   DPJ_dir\samples

   where ***DPJ_dir*** is the full path of the DevPartner Java Edition product folder.

2. Enter the command

   ```
   nmjava -mem stackLeakExample.StackLeakMain
   ```

   DevPartner Java Edition starts (if it is not already running) with the Memory analysis Session Control page active.

3. In the Session Control page, select **Memory Leaks** from the list of available analysis types.

4. Enter `40` in the console window to simulate the push/pop operations. This warms up the program to ensure that all classes are loaded and one-time intializations are run.

5. In the Session Control page, use a filter to show only your classes. In the **Filter By** field on the Session Control page, type the package name `stackLeakExample`, then click **Apply**.

6. Click **Start Tracking** to begin tracking memory allocations.

7. To exercise the program and check for memory leaks, enter `40` at the console prompt.

   You see a spike in the real-time graph, and the number of tracked objects for **`StackEntry`** in the list of profiled classes increases to 40.

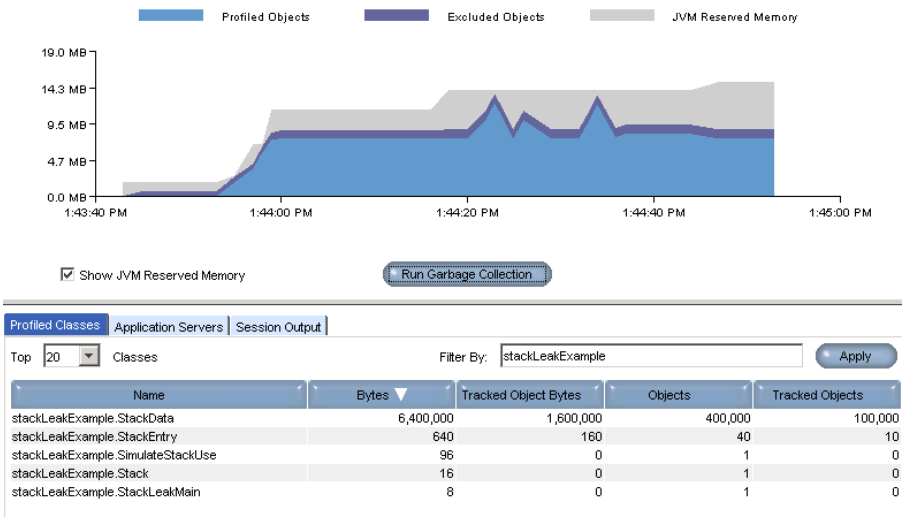
8. Click **Stop Tracking**.

9. Exercise the program section again: Enter `30` at the console prompt for the number of push/pops. The real-time graph spikes again.

10. Force a garbage collection: Click **Run Garbage Collection** in the Session Control page.

    The count of tracked objects for **`StackEntry`** drops to 10. If all the memory allocated while tracking had been freed, however, it would be zero. Also note that the memory display in the real-time graph did not return to the pre-exercise level.

Figure 2-4. Session Control Page Data Display



It appears that ten instances of **`StackEntry`** and 100,000 instances of **`StackData`** have leaked.

Click **View Memory Leaks** to create a session file that you can use to locate the source of the leaks. DevPartner Java Edition displays the session data in the Memory Leak Results Summary.

Memory Leak Analysis Results

DevPartner Java Edition defines a *memory leak* as any object that is allocated during a specified period of time, and has not been freed at the point at which you view memory data. The aim of Memory Leak analysis is to reveal cases in which memory is not being freed when it should be.

If memory use consistently rises and does not decrease (or does not decrease as you would expect it to) in response to garbage collection, your application is probably leaking memory. For an example, see Figure 2-4 (above). The real-time graph for the example you just ran should look similar to the one in the figure (assuming you ran the example only once), and will show that the rise in memory use that occurred when you ran the example did not return to baseline after garbage collection.

In the data for the classes that belong to your application on the **Profiled Classes** list, note that some tracked objects are not being collected by garbage collection. The number of uncollected instances of these objects can be seen in the **Tracked Objects** column in Figure 2-4 on page 28.

The Memory Leaks Results Summary includes the following graphs:

- **Classes with the Most Average Leaked Instance Bytes Including Children**
- **Objects that Refer to the Most Leaked Bytes**
- **Classes with the Most Leaked Bytes**
- **Methods with the Most Leaked Bytes**

Each graph shows the top five classes, objects, or methods that are associated with leaked memory. To see more classes, objects, or methods, click **More Details** for that graph.

Which starting point you choose depends on the problem you are analyzing as well as how you want to approach that problem. For example, if you notice an increase in memory use on the real-time graph, and a corresponding increase in tracked bytes for a particular class on the real-time list of profiled classes at the bottom of the Session Control page, you can create a session file and use the links associated with the **Classes with the Most Average Leaked Instance Bytes Including Children** graph to go directly to a list of all the instances for that class. If you notice that a limited set of specific objects are being leaked, you can use the **Objects that Refer to the Most Leaked Bytes** graph to quickly see which objects hold references to the leaked objects. You may, however, want to start with the **Methods with the Most Leaked Bytes** graph if you are familiar with the source code for the allocating method and can tell by examining that source whether the leaked object should have been cleared. The **Classes with the Most Leaked Bytes** graph presents a simple list of the classes responsible for the most leaked memory.

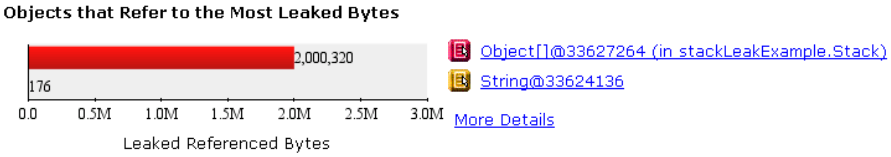
Regardless of the graph you choose to start drilling into the data, you can quickly switch to a view that shows another aspect of your data, for example, a chain of memory allocations, or a Call Graph of method calls, or go directly to source code.

Objects that Refer to the Most Leaked Bytes

In the following example, the **Objects that Refer to the Most Leaked Bytes** graph is the starting point for identifying memory leaks, because a limited set of objects responsible for the leaked memory exist.

If the garbage collector cannot clear an object, it means that there is at least one existing reference to that object. When your application runs, it creates objects. Some objects are needed for as long as the program runs. These are permanent objects. Many if not most objects, however, should become eligible for garbage collection when they are no longer referenced by another object.

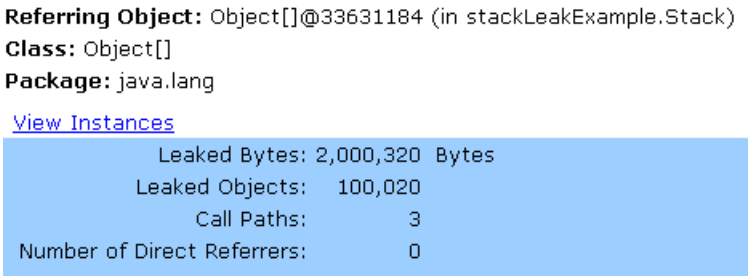
Figure 2-5. Objects that Refer to the Most Leaked Bytes Graph



This graph shows the objects that hold references to the leaked objects, and are thus responsible for preventing the leaked objects from being freed. Referring objects that account for the biggest memory hit appear at the top of the graph. Since the data indicates a particular set of objects is responsible for the leaked memory, use this graph to drill into the session data and locate the source of the leak.

Clicking a bar in the graph or the corresponding object name, for example **`Object[]`**, opens the Details window, from which you can open an Instance List.

Figure 2-6. Details Window



Note that this object refers to over 100,000 instances of leaked objects. Click **View Instances** to display a list of all the leaked objects referenced by **`Object[]`**.

To help you see patterns in the data, you can sort the Instance List by clicking any column head. When there are many instances, it is useful to sort the **Referenced Bytes** column in descending order to see the instances that are consuming the most memory. You may also find it useful to sort on the **Allocation Trace** column, to reveal instances allocated in repetitive patterns, for example, instances allocated in a loop.

With the Instance List sorted by **Referenced Bytes**, notice that the ten instances of **`StackEntry`** are the most expensive of the objects referenced by **`Object[]`**. The **Allocation Trace** column indicates that the reference to **`StackEntry`** was created on line 11 of **`StackEntry.java`**.

Figure 2-7. Instance List — Leaked Objects Referenced by **`Object[]`**

Leaked Objects Referenced from Object[]@33627264 (in stackLeakExample.Stack)

Instance	Class	Package	Referenced Bytes (Bytes)	Allocation Trace
StackEntry@33617920	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33617080	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33619600	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33618760	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33617836	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33616996	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33619516	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33618676	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33617752	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
StackEntry@33616912	StackEntry	stackLeakExample	200,032	StackEntry java - 11 (#12848)
Object[]@33621784 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33620440 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33623464 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33622624 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33621280 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33620356 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33623380 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33622540 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33621112 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
Object[]@33620272 (in stackLeakExample.StackEntry)	Object[]	java.lang	200,016	StackEntry java - 12 (#12360)
StackData@10806376	StackData	stackLeakExample	16	StackEntry java - 14 (#12672)
StackData@10805620	StackData	stackLeakExample	16	StackEntry java - 14 (#12672)

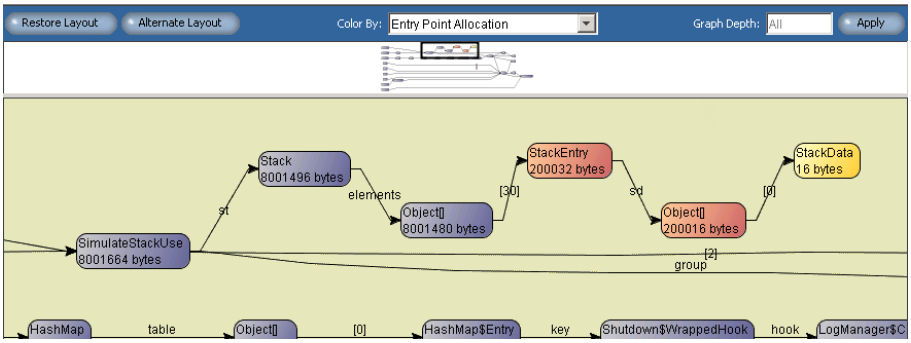
Previous Next

Click the first instance of **`StackEntry`** to open the Referring Object window. Click **View Object Reference Path** to display the Object Reference Graph. This graph shows the chain of references that led to the reference to the selected object in an Instance List. It shows why the selected object, **`StackEntry`**, is still in memory, by showing the objects that hold references to it.

In the Instance List, scroll down and notice that there are also ten instances of **`Object[]`** (in **`stackLeakExample.StackEntry`**). Select an instance of **`Object[]`** and display the Object Reference Graph. Note that the chain of referrers is more extensive. Scroll further down the list of instances to **`StackData`**. As you notice in the **Referenced Bytes** column, each instance of **`StackData`** is small (16 bytes) but there are many instances. If you look again at the real-time list of profiled classes (see Figure 2-4 on page 28), you can see that 100,000 tracked **`StackData`** objects (i.e., objects allocated while you were tracking memory allocations) remained in memory after garbage collection.

Display the Object Reference Graph for **`StackData`**. The chain of referring objects is now more extensive. The next step is to determine the point in the chain of referring objects where it is most efficient to eliminate the leak.

Figure 2-8. Object Reference Graph for an Instance of **`StackData`**.



In this case, the chain of referring objects shown in the Object Reference Graph is relatively simple. However, in many cases, there are multiple referrers and the graph can become very complex. Drag the rectangle or click a node in the overview pane to change the nodes displayed in the detail pane. If you are presented with a complex graph, you can simplify the view by doing one or more of the following:

- Use **Alternate Layout** to arrange the nodes of the graph differently.
- Change the depth of the graph. In a complex graph you may have to increase depth to follow an object reference to the root.
- Drag nodes into position. If the graph is so confusing that even clicking **Alternate Layout** doesn't simplify it, you can click and drag nodes into whatever arrangement you find most suitable.

The labels such as **st** or **elements** on the connecting arrows represent the referring data member in the next class in the graph. Bracketed numbers identify arrays. If you know your code very well, these labels can speed the process of zeroing in on potential problem areas.

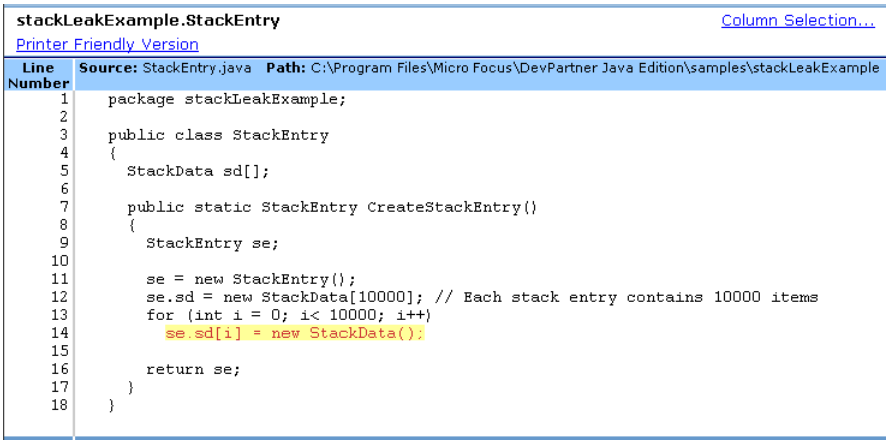
Clicking any node in the graph displays source code, if available. When you view the source code for **`StackData`**, or for **`Object[]`**, you can see that DevPartner Java Edition highlights the line in the method that allocated the object in the graph. To increase program understanding, you can view the source for each node in the graph sequentially and see the events that led to the allocation of the memory that leaked. DevPartner Java Edition offers alternate ways to view these program events. For example, the Allocation Trace Graph shows who called each method that allocated the selected object.

Note: The active configuration file determines whether DevPartner Java Edition displays trivial methods in an Allocation Trace Graph, Call Graph, or Method List. If trivial methods are not monitored (the default), they do not appear in these views. For more information on monitoring trivial methods, see "Trivial Methods" on page 68.

Or, you can go directly from the object in the Instance List to the source code. Since the current example involves a limited set of objects, continue with the Object Reference Graph. However remember that in real-world problem solving, you should drill down using methods suitable to the problem being resolved or corresponding to the way you think about your code.

In the Object Reference Graph, click the **StackData** node to view the source code for **StackEntry.java**. Backtrack through the graph, viewing the source code for the instance of **Object[]** that refers to **StackData**, and for **StackEntry**, which refers to **Object[]**.

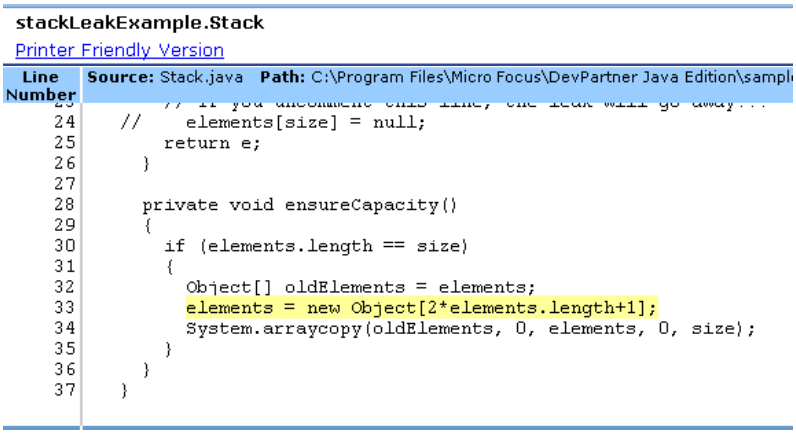
Figure 2-9. Source Code for **StackData**



All three objects were allocated in the **CreateStackEntry()** method. Since there were 10 leaked instances of **StackEntry**, and each entry on the stack refers to 10,000 **StackData** objects, you can see where the 100,000 leaked instances of **StackData** came from. But this does not explain why they are still in memory.

In the Object Reference Graph, click the **Object[]** node that refers to **StackEntry** to view the source code for **Stack.java**. DevPartner Java Edition highlights line 33, which is the line on which this **Object[]** array was allocated in the **pop()** method.

Figure 2-10. Source Code for **Stack.Java**



Examine the code for this method. In our example, you will notice the comment on line 24, but if this were your code you would realize that if you had set

```
elements[size] = null
```

the application would not be leaking memory.

In the preceding example, the Object Reference Path locates the source of the leak. However, the problem can be approached in several other ways. For example, you could view the Allocation Trace Graph to determine who called the method that allocated the object, and from there, view the source code. Or you could view the source code directly from the Instance List.

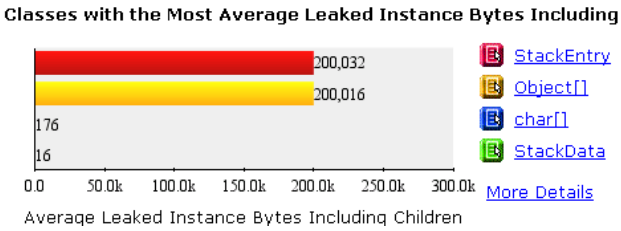
DevPartner Java Edition presents four different views of data on the Memory Leak Results Summary. Depending on the complexity of the data or on your own preferences you could have examined the problem from any of the other graphs on the Results Summary.

Classes with the Most Average Leaked Instance Bytes Including Children

If your application is leaking memory, you want to locate the classes, objects, or methods that are responsible for the leaked memory. More important, you want to focus your attention on those parts of your application that, when fixed, will have the greatest impact on memory use, and therefore, on application performance.

The **Classes with the Most Average Leaked Instance Bytes Including Children** graph shows the top five classes that, when averaged across all instances of the class, are responsible for the largest number of leaked bytes, including bytes attributable to children. For Memory analysis, DevPartner Java Edition computes average instance bytes including children by summing the bytes allocated for each live instance of the class and all bytes allocated for children of each instance, then dividing the total by the number of live instances of the class. The classes at the top of the list will be the classes for which a smaller number of instances is associated with a larger number of leaked bytes. In other words, the classes at the top of the list have the highest ratio of leaked bytes per instance of the class.

Figure 2-11. Classes with the Most Average Leaked Instance Bytes Including Children

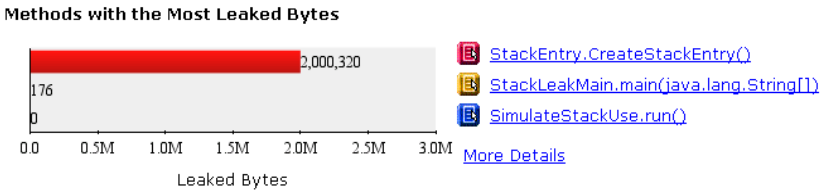


When a number of classes are associated with leaked memory, this graph enables you to immediately zero in on those classes where fixing a relatively small number of instances will have the greatest impact on the amount of memory your application uses.

Methods with the Most Leaked Bytes

This graph shows the top five methods that allocated objects that were leaked. Clicking a method in this graph enables you to see the instances allocated from the method that were leaked, or to view the source code for the method, showing the lines that allocated the leaked objects. Click a line that allocated objects to view an Instance List of objects allocated on that line.

Figure 2-12. Methods with the Most Leaked Bytes



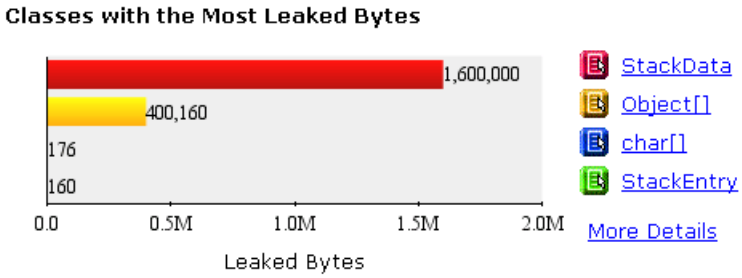
To drill down, do one of the following:

- Choose the source view.
- From one of the source lines that allocated objects, go to an Instance List of the allocated objects.
- From the object instance, view the referring objects or allocation trace.
- From a node in the Object Reference Graph or Allocation Trace Graph, view source for that node and troubleshoot

Classes with the Most Leaked Bytes

If many objects are leaked, you can use this graph to gain an understanding of the kinds of objects that were leaked. From this graph, you can view the instances of the class that were leaked. From any instance, you can view an Object Reference Graph or Allocation Trace Graph, or view the source code.

Figure 2-13. Classes with the Most Leaked Bytes



As an alternative, you can view the methods that allocated the leaked instances. From the Method List, you can view the source code for the method. The line that allocated the object provides a link to the instances allocated on that line. As with any Instance List, you can continue to drill into your data via the Object Reference Graph or Allocation Trace Graph. Once you understand the progression that led to the leaked bytes, you can view the source code to troubleshoot the problem.

Identifying Retained Objects

Among the reasons for memory leaks are objects that are retained in memory after they are no longer needed by the program. Running an Object-Lifetime analysis can identify the objects that persist longest in memory.

Normally, a session file includes only objects that are still live when the session ends. To include objects that have been garbage collected but that were retained a long time, select **Enable Object Retention** for the session configuration.

Note: Including retained objects in the session file incurs overhead and can make the session file very large; select this option only if you are specifically interested in retained objects.

While the analysis is running, select **Object-Lifetime Analysis** in the Session Control page. When the session ends, the results are displayed in the **Object-Lifetimes Results Summary** tab of the Start page.

When you collect data on retained objects, do not use **Run Garbage Collection** or **View Object Lifetimes** (which forces a garbage collection before displaying a snapshot of the session) during the session. Some objects may persist throughout the session for valid reasons; for example, an object may be used both at the start and at the end of the program's run. If you force a garbage collection, you in effect create a "false positive" for such objects, identifying them as retained when they are not wasting memory. To get accurate data on retained objects, allow the program to run unimpeded until it stops, then view the session file for the entire program run.

Running an Object-Lifetime Analysis

The sample application for object retention analysis is called **`objectRetentionExample`**. It is located in the following folders:

- Windows — ***`DPJ_dir`*`\samples\objectRetentionExample`**
  where ***`DPJ_dir`*** is the DevPartner Java Edition product folder
- UNIX — **`/opt/Micro Focus/DPJ/samples/objectRetentionExample`**

The application demonstrates how a memory leak can be caused by retaining objects in memory after they should have been released. It consists of a server and a client. The server waits for a client to connect to it and replies to queries from the client.

To exercise the client, you enter the following information:

- The hostname of the server
- The number of times to query the server

Using this information, the client requests an instance of `DataConnection` from `ConnectionPool` the specified number of times. `ConnectionPool` is initialized with a single `DataConnection` instance and keeps track of two collections:

- Connections that are currently unused and available
- Connections currently in use by the client

When the client returns a connection, the pool checks whether it is a connection that the pool created. If it is, the connection is returned to the "available" collection to be reused for a subsequent request.

In this example, however, the connection is not released back to the pool when the client is finished with it, thus creating a memory leak.

Using the sample application involves three tasks:

1. Create a configuration that has object retention enabled.
2. Run the server.
3. Run the client.

Create the Configuration

Note: If you have previously run this sample application and **`ObjRetConfig`** is included in the list in the **Configurations** tab, you do not need to create a new configuration. You can use the existing **`ObjRetConfig`** configuration.

1. Open the DevPartner Java Edition Start page and select the **Configurations** tab.
2. Click **New** to display the Explorer User Prompt. Type `ObjRetConfig` for the configuration name, then click **OK** to create the new configuration.
3. In the left pane, select **Object Retention**. The object retention options are displayed in the tab.
4. Select **Enable Object Retention**. You do not need to select or change any other options for this example.

The new configuration is saved automatically.

Run the Server

1. At the command prompt, change the folder to

   `DPJ_dir``\samples`

   where ***`DPJ_dir`*** is the full path of the DevPartner Java Edition product folder.

2. Enter the command

   ```
   java objectRetentionExample.PurchasesServerMain
   ```

Note: The server continues to run until you press Ctrl + C.

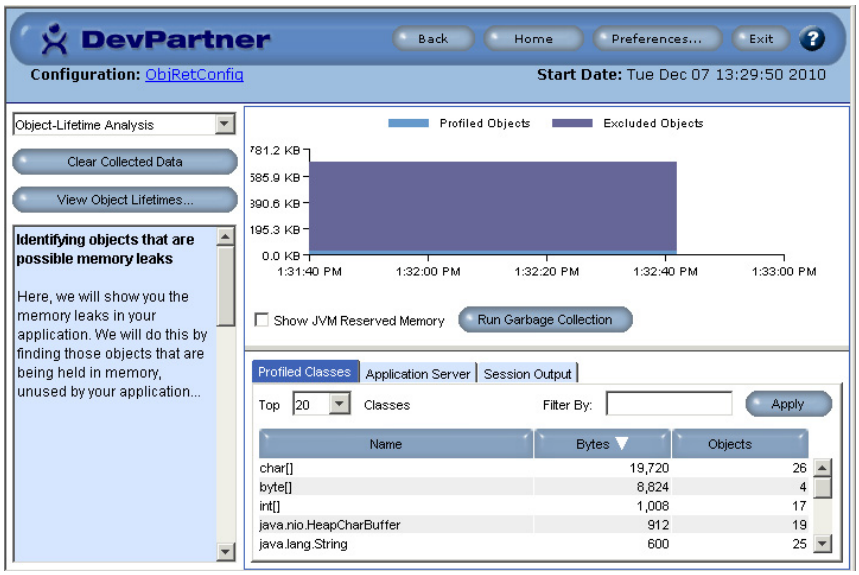
Run the Client

1. Open another command console and change to the **`samples`** folder.
2. Enter the command (as one line)

   ```
   nmjava -mem -config ObjRetConfig objectRetentionExample.Purchases-
   ClientMain
   ```

3. The Session Control page opens. The real-time graph at the top of the window shows the memory usage as the application runs. Select **Object-Lifetime Analysis** from the list.

Figure 2-14. Session Control Page for Object-Lifetime Analysis

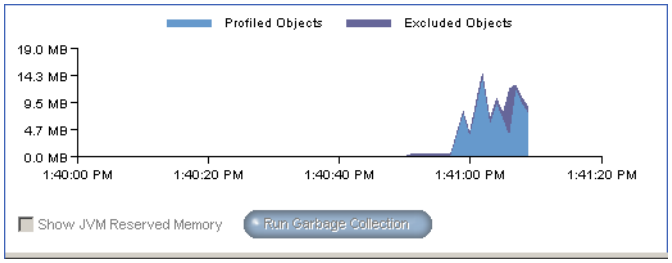


4 Return to the console in which you executed the client. It displays the prompt **Enter Purchases server hostname**. Enter the IP address of the machine on which the server is running. My IP address is 10.20.16.215.

5 The console displays the prompt **Enter # of times to query the server**. Enter 4.

6 Return to the Session Control page. The real-time graph spikes as the queries are executed.

Figure 2-15. Real-Time Graph for objectRetentionExample.PurchasesClientMain



7 After the fourth query, the client quits. You are prompted to view the last session file that was created for the session. Click **Yes** to display the Object-Lifetimes Results Summary.

Note: As the client executes, the console in which you executed it displays a series of messages, ending with **Completed all requests...** The server console displays messages confirming the queries. If you do not want to exercise the client again, press Ctrl + C in the server console to stop the server and close both consoles.

Object-Lifetime Analysis Results

When you run an Object-Lifetime analysis, the results are displayed in two tabs:

- **Object-Lifetimes Results Summary**
- **Temporary Objects Results Summary**

The data on the **Temporary Objects Results Summary** tab can help you resolve scalability problems (see “Solving Scalability Problems” on page 39).

The **Object-Lifetimes Results Summary** tab displays three graphs that provide information about retained objects:

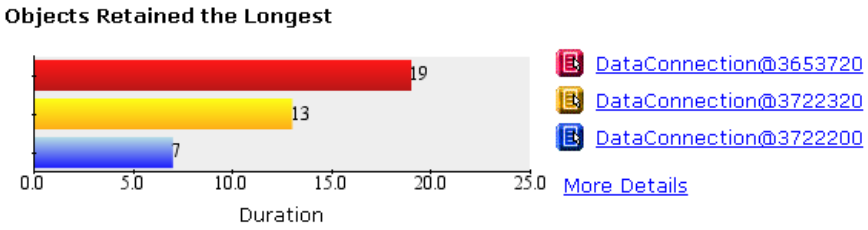
- **Objects Retained the Longest**
- **Classes with the Longest Average Retention Duration**
- **Entry Points with the Most Retained Instances**

For the sample application, the **Objects Retained the Longest** graph provides the most useful information for identifying memory leaks.

Objects Retained the Longest

The **Objects Retained the Longest** graph shows that the `DataConnection` instances were retained. The number at the end of each bar (instance) in the graph shows the number of garbage collections that occurred since the instance was last used.

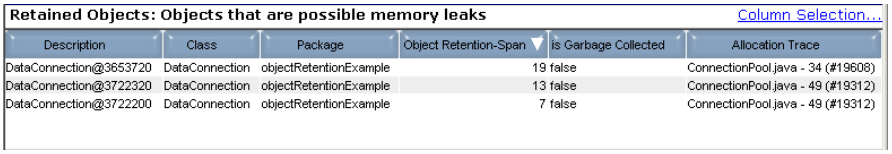
Figure 2-16. Objects Retained the Longest Graph



Note: For more complex applications than this sample, the graph displays five bars, for the five objects that were retained in memory for the longest time.

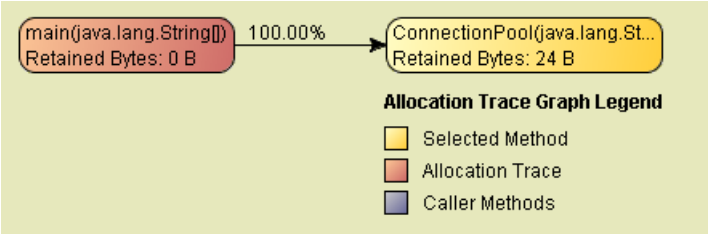
Click **More Details** to display the list of all instances. This list shows that all the instances are not yet garbage collected (the **is Garbage Collected** column lists `false` for each instance).

Figure 2-17. Retained Objects List



By default, the Instance List is sorted by the **Object-Retention Span** column, from largest to smallest value. Click the first instance to display its Referring Object window, then click **View Allocation Trace Graph**. The graph appears below the Instance List. The nodes represent the method calls that led to allocation of memory for the instance.

Figure 2-18. Allocation Trace Graph



The Allocation Trace Graph shows that the instance of `DataConnection` created during the initialization of `ConnectionPool` was retained the longest. This should not have been the case.

Because the server is queried more than once, the connection should have been released, then reused for another query. That the connection was never reused is a clue that it was retained in memory. It is a possible cause of a memory leak.

Classes with the Longest Average Retention Duration

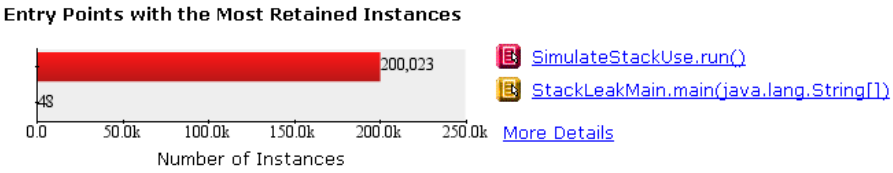
Identifying individual objects that are retained the longest time does not always zero in on the source of memory leaks; the problem may be the combined effect of retained instances associated with a specific class. This graph identifies the classes with instances that are retained the longest average time.

You can click a class to open the Details window, through which you can display a list of the retained instances for the class. Clicking **More Details** displays a longer list of classes of retained object; this list specifies the number of retained objects for each class and the average retention span (the average number of garbage collections the objects survived).

Entry Points with the Most Retained Instances

Like the previous graph, this graph focuses on the combined effect of retained objects to identify the cause of memory leaks.It identifies entry points (or the methods invoked by the entry point) that have allocated most number of retained instances.

Figure 2-19. Entry Points with the Most Retained Instances



Clicking an entry point displays a Details window through which you can display the Call Graph or a list of the retained instances allocated either by the entry point or by a method that was invoked by the entry point. Clicking **More Details** displays the complete list of entry points with retained objects; this list includes the number of instances retained by the entry point and the instances including children).

Solving Scalability Problems

When performing Memory analysis with DevPartner Java Edition, you can use the **Temporary Objects Results Summary** tab to diagnose and correct scalability problems.

Scalability problems surface when an application runs well until users begin to work with it more intensively. For a client-server application, this might happen when the number of users increases. For a standalone application, this might happen after numerous text manipulations or mathematical computations. These can be labeled as scalability problems. As the scale of the work done by the application increases, performance degrades.

One possible cause of scalability problems is the creation of too many temporary objects. In this case, object creation can become performance bottleneck — a problem that requires correction.

The creation and use of objects is important in Java programs. Unfortunately, some coding techniques have the side-effect of creating many objects.

Part of the problem is the creation of immutable objects, such as those created with the **`String`** class. *Immutable objects* cannot be changed. It takes processing cycles to create objects and later destroy them. If you can reduce the number of objects created, you can generally expect better performance.

DevPartner Java Edition tracks the objects allocated by your code and categorizes them based on how long it takes for them to be collected. There are three categories:

- Short-lived – Collected at the next garbage collection
- Medium-lived – Survives at least one garbage collection
- Long-lived – Survives across many (or all) garbage collections during the run of the program

DevPartner Java Edition combines short-lived and medium-lived object allocations in a temporary category.

Individual short-lived objects have less impact on newer garbage collectors although there is still a performance penalty for calling the object's constructor. However, creating large numbers of short-lived objects can cause bottlenecks and memory shortages.

Medium-lived objects cause the garbage collector to work harder than necessary. It consumes CPU cycles in a way that typical performance profilers have difficulty identifying the particular method causing the problem. These objects can cause your program to fail with an `OutOfMemoryError` under heavy-load situations.

If you think your code has scalability issues, you can use DevPartner Java Edition to monitor your code as it executes. If the real-time view suggests that too many temporary objects are the problem, you can use DevPartner Java Edition to analyze Call Graphs, entry points, and methods. You can also identify specific lines of code that allocate temporary objects, and you can see how much memory they consume.

To help you understand how to use these capabilities, DevPartner Java Edition includes a sample application that demonstrates excessive creation of temporary objects. The next few sections describe how to run the sample application and interpret the results.

Running a Profile for Temporary Objects

To run the temporary object sample file, follow these instructions:

1. At the command prompt, change the folder to

   `DPJ_dir``\samples`

   where ***`DPJ_dir`*** is the full path of the DevPartner Java Edition product folder.

2. Enter the command

   ```
   nmjava -mem tempObjExample.TempObjMain
   ```

3. In the Session Control page, select **Object-Lifetime Analysis** from the list of available analysis types.

4. In the console window, enter `1` to simulate creation of a temporary object. This warms up the program to ensure that all classes are loaded and one-time intializations are run.

5 Click **Clear Collected Data** to clear what has been collected so far.

6 Click **Run Garbage Collection**.

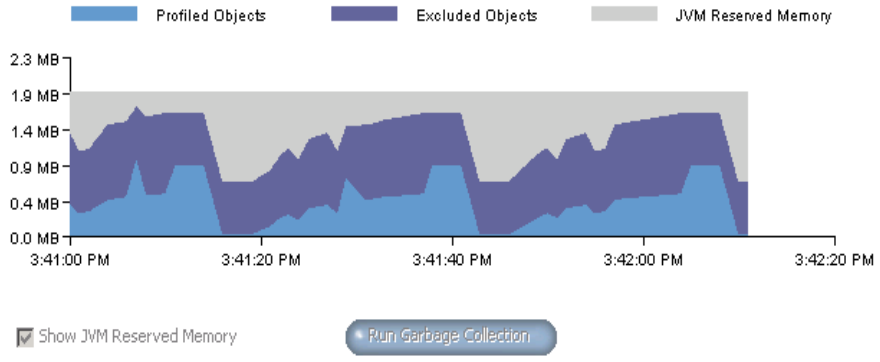
7 In the console window, enter `1000` to simulate a creation of many temporary objects. Note the spike in memory use.

DevPartner Java Edition enables you to locate potential troubles spots and then drill down into your program's use of temporary objects to identify problems and improve the overall quality of your code.

The real-time graph provides a high-level view that enables you to identify areas that might be causing problems.

Figure 2-20 shows a real-time graph that suggests excessive temporary object creation. Spikes in the graph of temporary object creation show where your application is creating more objects. Excessive object creation can create major performance or scalability issues in a Java application. Methods that allocate many short-lived objects often indicate easy-to-fix performance problems.

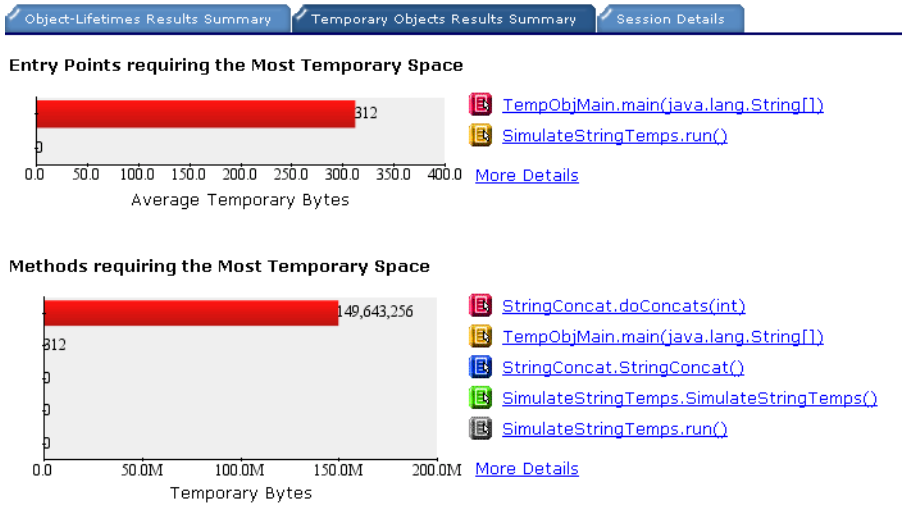
Figure 2-20. Excessive Temporary Object Creation



Temporary Object Analysis Results

Click **View Object Lifetimes** to collect data at a specific point in your program (this action forces a garbage collection). This option displays the **Object-Lifetimes Results Summary** tab; select the **Temporary Objects Results Summary** tab, as shown in Figure 2-21, to see the entry points and methods that are creating the most temporary objects. From this view, you can drill deeper back into your code, to the point where you can identify precisely which lines of code are generating objects, and how much memory theses objects are consuming.

Figure 2-21. Temporary Objects Results Summary Tab

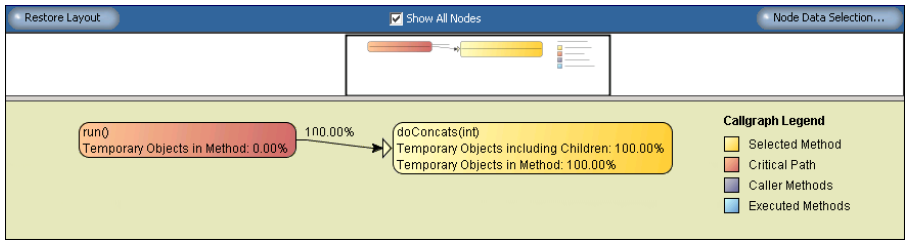


Clicking a method name to the right of the **Methods requiring the Most Temporary Space** graph displays the Details window, from which you can display a Call Graph or source code. Clicking **More Details** displays the complete list of methods called by your program.

The Details window also lists the number of short-lived, medium-lived, and long-lived objects and the temporary space they consume.

By default, the Call Graph shows the critical path. This is the sequence of child method calls that resulted in the largest cumulative memory allocation for the selected method. The Call Graph also displays caller and executed methods.

Figure 2-22. Call Graph



Methods appear as nodes on the Call Graph. Each node can display data about temporary objects in a method, including the number of bytes consumed and the number of objects created by a method. Of special note are the display percentages of temporary objects.

The Call Graph shows the percentage of temporary bytes allocated in two ways:

- The percentage allocated by the child method (and its children) as a percentage of the bytes allocated to execute that path
- The percentage allocated by a method computed as percentage of bytes allocated by all methods during the session

Note: The active configuration file determines whether DevPartner Java Edition displays trivial methods in an Allocation Trace Graph, Call Graph, or Method List. If trivial methods are not monitored (default), they do not appear in Call Graphs, Allocation Trace Graphs, or in the Method List view. For more information on monitoring trivial methods, see “Trivial Methods” on page 68.

When you view the source code for an entry point or method listed in the **Temporary Objects Results Summary** tab**, the following** columns are displayed by default to the left of the source code:

- **Execution Count** – Number of times this method was called.
- **Temporary Bytes including Children (Bytes)** – Amount of temporary space (short and medium) used by this method and its child methods when called.
- **% Temporary Bytes including Children** – Same as above, but expressed as a percentage of the total amount of accumulated temporary space seen in this profiling run.
- **Temporary Objects including Children** – Number of accumulated temporary objects allocated by this method and the child methods that it called
- **Source** – The complete path to the source code appears in the top of the view. The source code appears in the column below the heading.

Highlighting draws your attention to significant lines in the source code:

- Yellow highlighting identifies the first line of the selected method.
- For tracking temporary object creation, the line that consumes the most memory is noted in red type.

Click **Column Selection** to display the following optional columns:

- **Child Methods** – Number of child methods called by this method.
- **Short-lived Bytes including Children (Bytes)** – Amount of short-lived temporary space used by this method and its child methods when called.
- **% Short-lived Bytes** – Same as above, but expressed as a percentage of the total amount of short-lived temporary space seen in this profiling run.
- **Short-lived Objects including Children** – Number of short-lived temporary objects allocated by this method and the child methods that it called.
- **Medium-lived Bytes including Children** – Amount of medium-lived temporary space used by this method and its child methods when called.
- **% Medium-lived Bytes** – Same as above, but expressed as a percentage of the total amount of medium-lived temporary space seen in this profiling run.
- **Medium-lived Objects including Children** – Number of medium-lived temporary objects allocated by this method and the child methods that it called.
- **Long-lived Bytes including Children** – Amount of long-lived temporary space used by this method and its child methods when called.
- **% Long-lived Bytes** – Same as above, but expressed as a percentage of the total amount of long-lived temporary space seen in this profiling run.
- **Long-lived Objects including Children** – Number of long-lived temporary objects allocated by this method and the child methods that it called.
- **Line Number** – The line number in the source code.

The following are some of the ways to interpret Memory analysis results to fix memory-related scalability problems:

- Review the **Temporary Objects Results Summary** tab and modify the code for the entry point or method that requires the most temporary space.
- Review the source code and modify the method noted in red type, which identifies the line that consumes the most memory.)
- View the Details window and compare the number of short- and medium-lived objects, as well as the amount of temporary space they consume. Use this information to determine which parts of your code to modify.
- If the Details window shows that both short-lived and medium-lived objects consume similar amounts of temporary space, run a Performance analysis to find out how much time the constructor uses to create the temp object.
- Use the Call Graph to evaluate methods. Examine characteristics of different methods: local and global percentages of memory consumed; actual bytes used; numbers of temporary objects created. Use this information to identify which method to modify.
- Use the **Object-Lifetimes Results Summary** tab to identify code that can be modified so objects will not be retained in memory after they are no longer needed by the program.

Managing Memory for Better Performance

Specific memory problems are examined in this chapter, such as memory leaks, which can cause your application to consume more and more memory as it runs until it eventually exhausts the heap. Periodic spikes in memory use caused by excessive temporary object creation, which can lead to scalability issues.

These problems affect your application's memory use in negative ways. They also contribute to your application's memory footprint. What do you do, however, if your application is well-behaved with respect to these errors, but it still feels slow, especially when run on the kinds of hardware your target users are likely to have?

One cause of sluggish performance is that your application may be using excessive amounts of memory as it runs. What is excessive? That depends on the hardware and software environment in which your application is used. You may have a pretty good idea of what that environment is, but you cannot know for certain that your target users will not try to run several other applications at the same time as they run yours. Nor can you force hardware upgrades on your users every time you release a new version of your application. All of this makes a strong argument for keeping your application's memory footprint small.

RAM footprint is not the same as overall memory use. The worst thing you can do to ruin application performance — and your end-users' perception of your application — is to force it to rely on the operating system's virtual memory system.

What can you do to optimize your application's use of RAM resources? DevPartner Java Edition provides RAM Footprint analysis as part of its Memory analysis capability. Run this analysis regularly as you develop your application. The way your application uses RAM resources is most likely a result of application design and architecture. It is much easier to redesign a feature early in the development process than to wait until the application is ready for beta release.

Measuring RAM Footprint

What contributes to footprint? in *Java Platform Performance*, Steve Wilson and Jeff Kesselman provide this list:

- Objects
- Classes
- Threads
- Native data structures
- Native code

The objects your program allocates and the classes it loads typically account for the largest part of an application's RAM footprint. DevPartner Java Edition helps you focus your tuning efforts on the areas that will have the greatest impact on RAM consumption.

When you run your application under RAM Footprint analysis, DevPartner Java Edition enables you to:

- View the real-time graph of your application's RAM consumption and the real-time list of profiled classes associated with the most bytes of memory.
- Take a snapshot of the heap, which you use to examine the objects or classes responsible for the most memory use.

Tip: You can collect RAM Footprint data at the same time you analyze memory leaks by checking the **Include RAM Footprint Information** option on the Memory Leaks Session Control page.

To run your application under RAM Footprint analysis:

1. Start your application under a Memory analysis session.

   See the online help for information on starting a DevPartner Java Edition session from the command line, an application server, or an IDE.

2. In the Session Control page, select **RAM Footprint**.

3. Exercise the application to allow classes to load and one-time initializations to run.

   As you run the application, observe the real-time graph. Be alert for patterns that may indicate a specific problem, such as a leak or excessive spiking due to temporary object creation.

   Notice the amount of RAM consumed. The y-axis of the real-time graph indicates the total in megabytes. To see the amount of RAM being used in relation to the memory reserved by the JVM, select **Show JVM Reserved Memory**.

4. Get the application into a steady (idle) state.

5. Click **View RAM Footprint** to generate a session file and display the **RAM Footprint Results Summary** tab, which reflects the current state of the heap.

Note: If, when you click **View RAM Footprint**, you see the message **"Details: java.lang.OutOfMemoryError"**, you need to increase the memory allotment. Quit DevPartner Java Edition, then open the file **`DPJServer.args`**. Change `-Xmx128m` to `-Xmx1024m`, then save and close the file. In Windows XP and 2003 Server, this file is located in **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\conf`**. (By default, the **`Application Data`** folder is hidden. To display the **`conf`** folder and its contents, type the path in the Address bar of Windows Explorer and press **Enter**.) In other supported Windows operating systems, it is in **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\conf`**. In UNIX, it is in ***`DPJ_dir`*`/var/conf`**, where ***`DPJ_dir`*** is the product installation folder.

Use the RAM Footprint Results Summary to gain an in-depth understanding of how your application is using memory. This page includes the following graphs to drill down into your data:

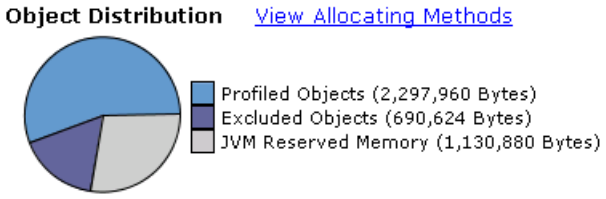
- Object Distribution
- Classes with the Most Average Live Instance Bytes Including Children
- Objects that Refer to the Most Live Bytes
- Classes of Profiled Instances Taking up the Most Space

Which you use first will depend on the data presented, and to some extent, on the way you think about your application.

Object Distribution

DevPartner Java Edition presents the distribution of objects in memory as a pie chart so you can immediately see the proportion of memory used by your application (profiled code) relative to that used by system code (excluded) and the balance of memory reserved by the JVM, but not currently used.

Figure 2-23. Object Distribution Pie Chart



If the **Profiled Objects** section of the pie chart is small, your application is not the main allocator of memory. This is a good thing. If your application (profiled code) is the largest wedge in the pie, and memory use is moderate to high relative to expected resources in the target deployment environment, you should determine which part(s) of your application allocates the most memory. Click **View Allocating Methods** to see the Method List for your profiled objects.

In the Method List, click a method to display the Details window, in which you can select to view the source code for the method, view a Call Graph for any of the profiled methods, or add a Session Control Rule.

In the source code view, DevPartner Java Edition highlights the first line of the selected method in yellow, and the line in the method that allocated the most memory is indicated with red type.

You can navigate from a line that allocated memory to an Instance List of live objects allocated on that line. By default, this Instance List is sorted by **Referenced Bytes**. This enables you to focus on instances of a class or object responsible for the largest amount of memory. The next column, **Allocation Trace**, shows the location of the method that allocated the memory, including the number of the line on which it was allocated.

Click the largest object (referenced bytes) in the Instance List to display the Details window. From the Details window, you can view:

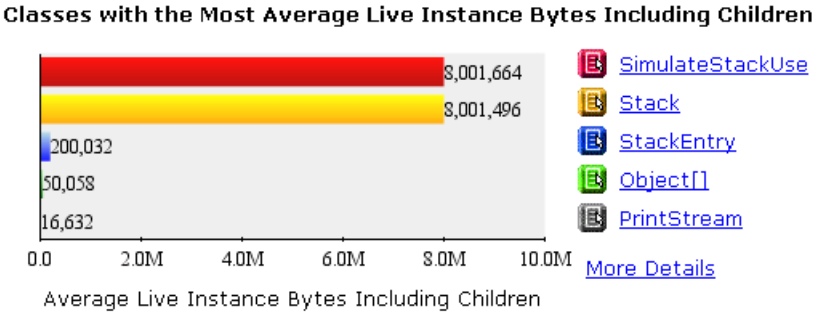
- An Object Reference Graph to see the object's referrers, that is, why the object is still in memory
- An Allocation Trace Graph to see the chain of methods that led to the allocation of the memory

For information on using the Object Reference Graph and Allocation Trace Graph, see page 29.

Classes with the Most Average Live Instance Bytes including Children

If your application is using too much memory, you want to locate the classes, objects, or methods that are responsible for the memory use. You also want to focus your attention on those parts of your application that, when fixed, will have the greatest impact on memory use, and therefore, on application performance.

Figure 2-24. Classes with the Most Average Live Instance Bytes including Children



As described earlier, the **Classes with the Most Average Live Instance Bytes including Children** graph shows the top five classes that, when averaged across all live instances of each class, are responsible for the largest number of bytes in memory (including bytes attributable to children) at the time the snapshot of the heap was taken. The classes at the top of the list have the highest ratio of bytes in memory per instance of the class. Hence, this graph enables you to focus immediately on those classes for which fixing a relatively small number of instances will have the greatest impact on the amount of memory your application uses.

Objects that Refer to the Most Live Bytes

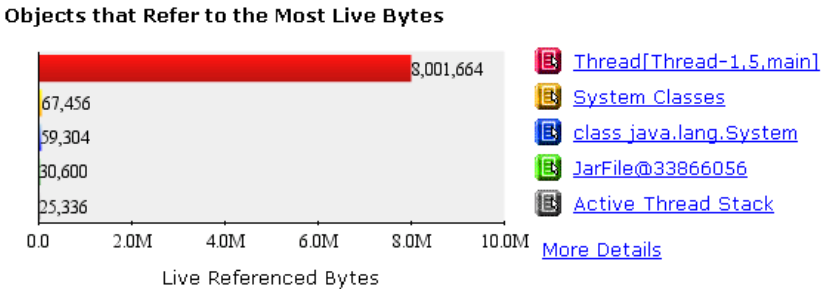
This graph shows the largest objects in memory. Objects in the graph include bytes attributable to all child objects for which that object is the only parent. Considered singly, objects in Java tend to be small. However, an object with several child objects, each of which might also have child objects, plus per-object overhead for parent and child objects, might actually be consum-

ing a large amount of memory. In essence, DevPartner Java Edition uses the Object Reference Path to roll up the bytes associated with child objects and attributes them to the parent object.

From the **Objects that Refer to the Most Live Bytes** graph, you can navigate to:

- A Details window for the object, including the number of bytes allocated for the object and its child objects. This window provides a link to a list of classes of the child objects referenced by the large object. Click a bar in the graph or an object name to display the Details window.
- A detailed list of all objects holding references to live objects at the time the session file was created.

Figure 2-25. Objects that Refer to the Most Live Bytes



When you view source code (available via the Method List), you can jump from any line that allocated memory to a list of all the instances allocated by that line. From the Instance List, you can view:

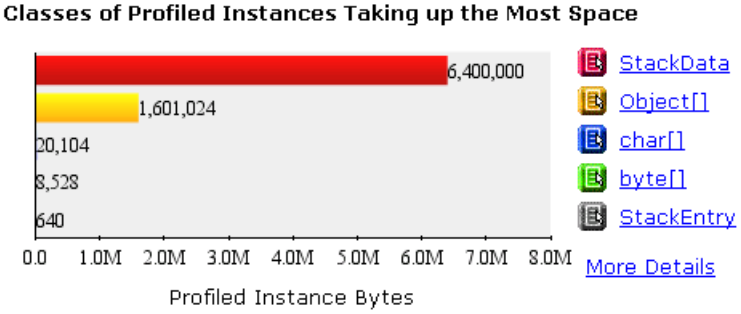
- An Object Reference Graph to see why the object is still in memory
- An Allocation Trace Graph to see the methods that allocated memory

For information on using the Object Reference Graph and Allocation Trace Graph, see page 29.

Classes of Profiled Instances Taking up the Most Space

This graph shows the profiled classes in memory with the largest summed instance sizes at the time the session file (results summary) was created. This graph gives an overview of the kinds of classes that are consuming the most memory. Click a class to view details and drill into the instance data.

Figure 2-26. Classes of Profiled Instances Taking up the Most Space.



Optimizing Memory Use

Once you understand how your application is using memory, you can begin to optimize memory use. Classes and objects are typically the largest memory consumers. Take them as starting points.

Your program probably creates many objects as it runs. Do you simply try to reduce the number objects created? How do you know where to focus your tuning efforts?

DevPartner Java Edition does much of the cost/benefit calculating for you. Remember that individual objects in Java tend to be small; but when you consider objects with their children, some objects are much larger than others. DevPartner Java Edition uses the concept of *large object* to alert you to those objects which, with their child objects, are large consumers of memory. Focusing your tuning efforts on these object allocations promises the most rapid route to a reduced footprint.

When a class is loaded, the following entities consume memory[1]:

- Bytecodes
- Reflective data structures
- Constant pool entries
- JIT compiled code

Each entity entails a memory cost. Various strategies exist for optimizing class loading. One strategy is to avoid loading too many classes before they are needed. Another is to avoid creating too many small classes. But which classes do you focus on?

Knowing which classes are associated with high memory use is a step in the right direction; but if there are hundreds of instances of a class, each responsible for a few bytes, it will be difficult to troubleshoot and maintain the development schedule. DevPartner Java Edition shows you not only which classes are associated with the most memory use, but which classes have the highest average memory allocation. That means fewer instances of these classes are taking up a larger amount of memory. If you use the **Classes of Profiled Instances Taking up the Greatest Average Space** graph as a view into your application's memory use, you can focus your optimizing efforts on those class instances that will have maximum impact.

How Memory Analysis Fits in Your Development Cycle

It is not necessary to wait until you suspect that you have a problem to begin testing. If you run Memory analysis early and often, and know what to look for when you analyze your application, you can correct problems early, when they are both easier to track down and entail less risk to fix.

Memory problems in Java are often the result of larger design and architecture decisions, rather than simple coding errors. For example, one source of memory loss is an object that is not collected because of an out-dated reference to it that is not freed. This can be the result of revisions made in another part of the code. The later these problems are identified in the development cycle, the more difficult and expensive they will be to fix.

1.For details, see page 62 in Wilson, Steve, and Kesselman, Jeff, *Java Platform Performance*. Addison-Wesley, Boston, MA, 2000. ISBN:0-201-70969-4.

It is valuable to use Memory analysis as part of a continuous testing program throughout the development cycle. For example, using it during unit testing provides an understanding of how the individual modules handle memory. Once you identify and fix areas that need improvement, run another Memory analysis to verify the fix. Then, as you integrate the modules into your application, repeat your memory testing to ensure that new memory problems do not appear.

Chapter 3

Ensuring Testing Consistency

DevPartner Java Edition helps development teams save testing time and improve code reliability by measuring and tracking code execution and code base stability during development.

Covering All Your Bases: Code Coverage Consistency

Coverage testing verifies that all possible execution paths in an application have been executed: that there are no missed `if-else` clauses, no missed `case` statement branches, no untested exception handling code, and so on.

Code coverage testing is an important weapon in your development process's testing arsenal. Without adequate coverage testing, sections of your application's code might not be exercised prior to release. These untested sections could hide bugs that will manifest themselves in production.

A good coverage tool must be able to consolidate coverage data gathered over time, and from separate testing sessions. It must be able to do its work even on the changing landscape of an application under development. It must track changes in the application, so that you can adjust coverage testing as application code is modified, added, and removed.

DevPartner Java Edition covers all these bases.

You can employ DevPartner Java Edition during the development and testing phases of the software development cycle. Developers use the Coverage analysis feature in unit testing significant code check-ins.

Software testers can also use the Coverage analysis feature during routine regression and reliability testing to ensure that applications and components have been thoroughly exercised.

Running Coverage Analysis from the Command Line

The command line utilities enable you to launch and monitor a Java application, an applet, or components running in an application server). Monitoring can be performed either interactively or in batch mode. In interactive mode, the user interface is active while the application runs; you can view collected data in real time. In batch mode, the monitoring occurs in the "background" and results are saved to a session file. To view the collected data, you must launch DevPartner Java Edition and open the session file.

Configuring a Session for Code Coverage

DevPartner Java Edition defines extremely short methods as *trivial methods* (see "Trivial Methods" on page 68). Trivial methods are treated in a particular fashion, and if you are not aware of this, coverage results might appear to be erroneous.

Generally, you should enable **Monitor Trivial Methods** in the configuration setup to monitor all methods, so you will see the full execution path of your program. If you do not monitor trivial methods, your results might be confusing. For example, suppose you are convinced that your coverage tests will provide 100% coverage of the application, but running them through DevPartner Java Edition shows that a tiny percentage of the code is still uncovered. This might well be because you have disabled the monitoring of Trivial Methods, in which case DevPartner Java Edition will not have flagged them as having executed.

Sometimes, there are benefits to *not* monitoring trivial methods. Turning off trivial method monitoring significantly reduces data collection overhead. One tack to take would be to disable monitoring of trivial methods after you understand how DevPartner Java Edition monitors code and you are satisfied with the execution path of your program. This will allow you to reduce testing time during the bulk of the development and testing cycle. However, near the end of your testing process, re-enable monitoring of trivial methods as a final check that all of your code is being exercised.

Running the Code Coverage Example

As a quick introduction to collecting coverage data in an interactive session, you can run the **`coverageExample`** sample program as follows:

1. At the command prompt, change the folder to

   `DPJ_dir``\samples\coverageExample`

   where ***`DPJ_dir`*** is the ***`DevPartner Java Edition`*** product folder.

2. Run the command

   ```
   nmjava -cov -cp .. coverageExample.CoverageMain coverageEx-
   ample.Class1
   ```

3. The Session Control page is displayed, prompting you to choose whether to view results or clear collected data. The sample application will terminate immediately. Click **Yes** when prompted to view the final coverage results.

(For further instructions on using the **`coverageExample`** sample file to explore coverage features of DevPartner Java Edition, refer to the online help.)

To perform Coverage analysis on an application, use the `nmjava` command instead of the java command. For example:

`nmjava -cov [-config` ***`configuration-name`***`] [ -batch ] <`***`application-name`***`>`

Note: Optionally, you can specify a configuration that sets parameters on what code is monitored. For information on specifying a configuration, see the online help.

Viewing the Results of a Coverage Session

When your program has finished executing, you can analyze the data gathered by DevPartner Java Edition.

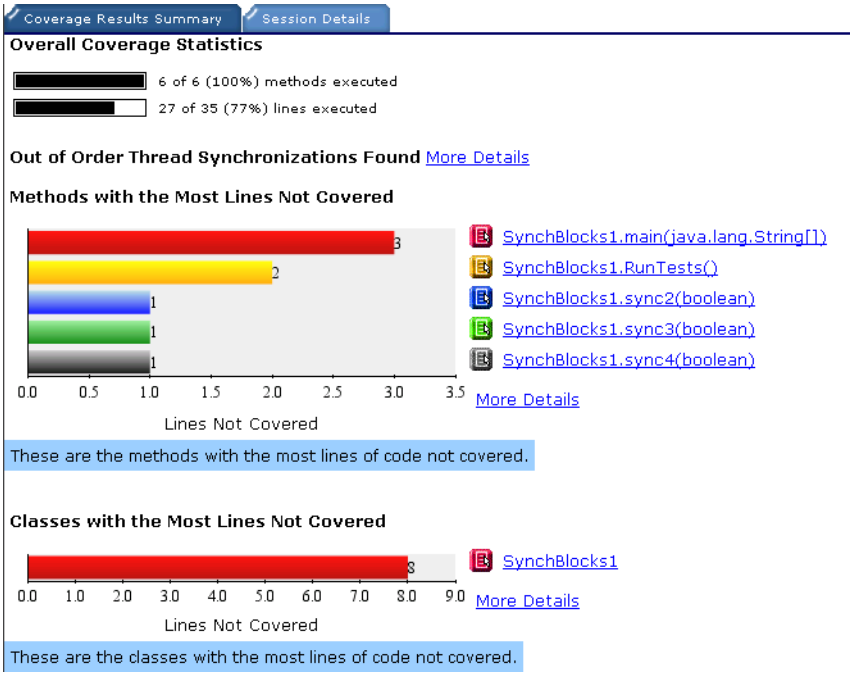
To view your Coverage session results:

1 If you have run the application in interactive mode, select **Yes** when prompted to open the most recent session file. Otherwise, navigate to the **Session Files** tab of the Start page, select the correct configuration from the list, and open the most recent coverage session file.

2 The Coverage Results Summary appears.

The Coverage Results Summary graphically displays Code Coverage data. Figure 3-1 shows a sample Coverage Results Summary.

Figure 3-1. Coverage Results Summary



By default, the **Coverage Results Summary** tab is selected. It includes the following graphs:

- **Overall Coverage Statistics**

  This pair of simple bar graphs shows the number and percentage of methods and lines that have been executed. They give a high-level view of the overall coverage.

  If out of order thread synchronization monitoring is enabled in the session configuration, a line of text below these bar graphs indicates whether out of order thread synchronizations were found. If none are found, the line simply states that. Otherwise, the line provides a **More Details** link to a screen from which you can explore the specifics of the potential deadlock. (For more information on out of order thread synchronization, see “Deadlock: The Deadly Embrace” on page 58.)

- **Methods with the Most Lines Not Covered**

  This graph shows the top five methods with the most uncovered lines.

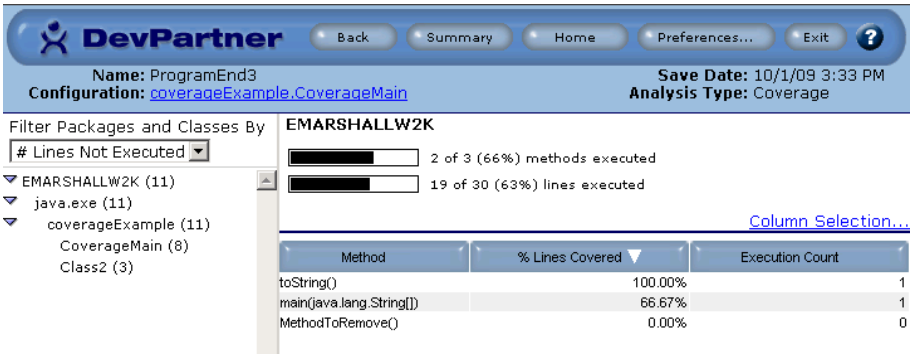
- **Classes with the Most Lines Not Covered**

  The last graph displays the top five classes with the most uncovered lines.

The bottom two graphs provide links that let you drill down into the details of the gathered data. For example, click either a bar or the associated method/class name, and a Details window appears, showing the numeric statistics associated with the item selected. You can also view the source code from the Details window.

Clicking **More Details** takes you to the detailed view, as shown in Figure 3-2.

Figure 3-2. Detailed View



Tracking Code Changes by Merging Session Files

Merging Coverage session files achieves several objectives:

- It assembles the results of individual coverage sessions into a unified whole. It is unlikely that a testing regime will include a single coverage test; more probably there will be many coverage tests, each designed to explore a different area of the application. Merging provides the means of gathering the results of these tests into a single vantage point from which you can survey the coverage landscape to determine which portions of the entire application have been covered, and which have not.

- It tracks the progress of your coverage testing. For any sizeable application, it is a rare test plan that -- on the first try -- produces tests that provide 100% coverage. In the real world, the creation of coverage tests is usually a series of successive approximations. The first tests will coat broad tracts of the source code. But proper testing demands comprehensiveness, so follow-up tests must be created to exercise the missed nooks and crannies. However, missed nooks and crannies can multiply as a project evolves, and new source is added. Tests must be either modified, or new tests continually added.

  All this demands a mechanism whereby you can track testing's progress. Such tracking allows you to answer specific key questions. How quickly are you approaching total coverage? Are you responding with new coverage tests nimbly enough to stay abreast of new source additions? These questions are key because they form the foundation atop which sits the most important question: Will coverage testing be complete in time to meet the shipping deadlines?

- It tracks the rate of change in the code being tested. DevPartner Java Edition charts the volatility of the code on the **Merge Details** graph. The volatility quantifies the amount of change in the target code from session to session, so that you can know if and when it's time to step up coverage testing efforts. It even identifies methods that have been added, altered, or removed since the last session.

Generally, coverage data is accumulated in several sessions and then analyzed to provide the total coverage statistics. Accumulating data in this manner lets you track changes in your code so that you can gauge the stability of your code base.

Merged session files maintain a record of all the classes and methods that were loaded in any of the contributing session files. To create a merged session file, you can either merge existing session files manually or automatically.

Manually Merging Session Files

To manually merge session files:

1. On the **Session Files** tab of the Start page, click **Merge** to display the Merge screen.
2. Select the desired configuration from the list.
3. Select the session files to merge. You can select any number of files.

Note: If you select a range of files to merge, those sessions are merged in chronological order. But, be careful. If at a later time, you merge additional session files into the same merge file, those additional files will be added to the merge file as though they had been created at a point in time later than the original session files, even if they have not. This could have unexpected and erroneous consequences.
The reason for this behavior is complex, but has to do with the fact that the merged file contains information that is calculated at the time it is created. As such, it is impossible (without making the processing time burdensome) to add a new file out of order, and expect the merge process to insert that new session's information at the proper chronological position in the dataset. That would require wholesale recalculation of the merge file whenever a new file was added. Once a merge file included data from a large number of session files, the time consumed to merge a new file would make the entire process impractical.
If such a situation occurs, you can remedy it by simply re-merging all the files into a new merge file (with the understanding that files will be merged based on their creation dates).

4. Specify whether to create a new merge file or add to an existing merge file.
   - ◇ **Create a new merge file** – Type a name in the field.
   - ◇ **Add to an existing merge file** – Select the file from the list.

Automatically Merging Session Files

You can specify that all session files for a configuration are merged into a specified merge file automatically. To automatically merge session files:

1. Select the **Configurations** tab of the Start page.
2. From the **Configuration** list, select the configuration for which you want sessions automatically merged.

Note: If session files already exist for the chosen configuration, they will not be included in the automatic merge. Only session files created after you change the configuration will be merged.

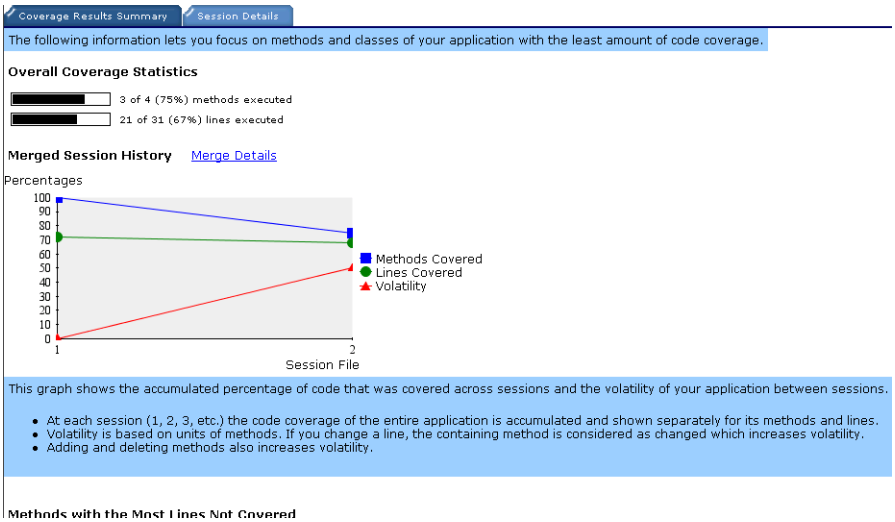
3 Select **Automatically merge Coverage Session**.

4 If you want to use a different name for the merged file, click **Change File** and type the new name in the **Explorer User Prompt** dialog box.

Using Merged Session Files

When you merge session files, the merged session file is listed on the **Session Files** tab and can be opened just like any other coverage session file. The following is the Results Summary from a sample merged session:

Figure 3-3. Coverage Results Summary for Merged Sessions



The **Merged Session History** graph features three lines: **Methods Covered**, **Lines Covered**, and **Volatility**. All three indicate percentage measurements. Data points on the first two graphs represent accumulated data (the graphs are read from left to right). The leftmost data point represents data from the first session file in the merge. The next data point to the right represents data from the second session file, combined with data from the first. The next data point to the right represents data from the third session file, combined with the first two, and so on.

The blue line shows the percentage of methods covered. This percentage is calculated by dividing the number of covered methods by the total number of methods. A method is counted as covered if it has been executed at least once.

The green line shows the percentage of lines covered. Similar to the preceding graph, the percentage is calculated by dividing the number of covered lines by the total number of lines.

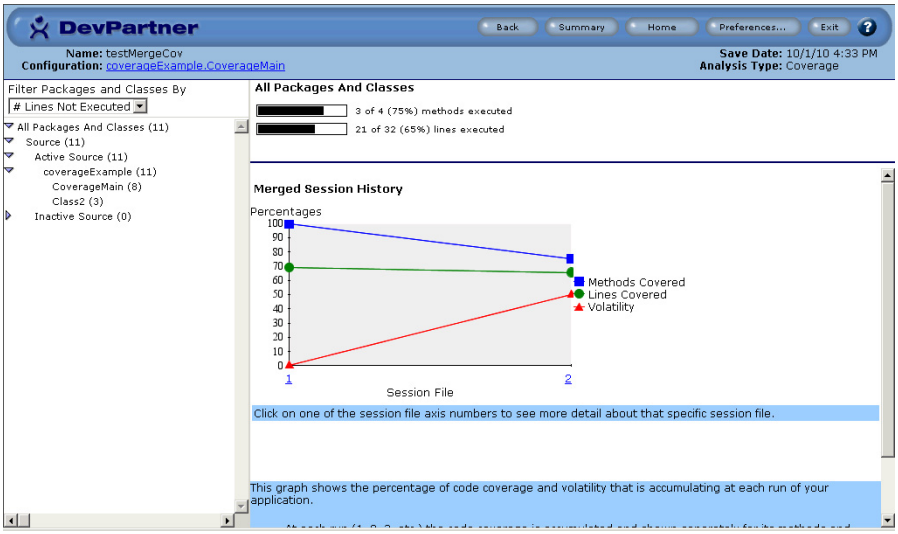
The red line shows volatility. Although volatility is also a percentage, volatility measurements are not accumulated as on the preceding two graphs. The volatility line represents the percentage of methods that changed in your code between sessions. (For example, the third data point on a Volatility graph reflects changes in the code between the second and third session files merged into the merge file. And it is *not* influenced by changes in the source code between the first and second session files merged in.) Volatility characterizes the stability of your code. The higher the volatility, the more the code changed.

Tip: The **Session Files** tab lists all files; both individual sessions and merge files. If you want to see a complete list of the individual session files that have been merged into a particular merge file, select the **Session Details** tab of the Coverage Results Summary for the merge file.

To see more details about a specific session file, click one of the numbers below the **Merged Session History** graph. These numbers are called *session file axis numbers*.

To see more details about the merged session and source code that changed, click the **Merge Details** link to the right of **Merged Session History**. Figure 3-4 shows a sample Merge Details window.

Figure 3-4. Merge Details Window



The frame in the left side of the window contains the following:

- **A view of the class structure** – Items are arranged in a hierarchical tree that reveals the package and class structure. Branches of the tree are indicated by triangles that you can click to expand or collapse, thereby revealing the branch structure. The number in parentheses to the right of each entry indicates the number of lines of code not executed in the session.
- **Inactive Source** – This item on the tree includes those classes that were removed in the most recent session, but were loaded and executed in earlier sessions.
- **Removed Methods** – Similar to **Inactive Source**, this item includes the methods that were deleted in the most recent session, but were executed in earlier sessions.

As you merge session files, code changes are reflected in the **Merge States**. To see the **Merge States,** click the **Method List** tab. If you do not see the **States** column, click **Column Selection** and check the **States** check box.

The status of the code from session to session contributes to the overall volatility of the application. The less change in the code from session to session, the more consistent results from Coverage analysis, and the less volatile the code base. Using these states helps you understand the volatility level of the application. That is, they indicate changes in the application since the last session file was recorded. Merge States that are associated with methods have the following values:

- Added – The method has been added since the last session.
- Removed – The method appeared in one session, but was absent from the next session.
- Changed – The method has been changed from a previous session.
- Unchanged – The method has not changed from a previous session.

Similar merge states apply to classes:

- Added **– This class was added in the most recent session.**
- Inactive – This class was loaded in a previous session, but not in the most recent session.
- Activated – A class is activated if it was previously inactive, but has been loaded in the most recent session.

Using the status definitions, you can easily determine what code has been changed and where additional testing has become necessary.

Automatic Merging and Live Monitoring

DevPartner Java Edition's automatic merge feature can sometimes collide with its live monitoring capability, as in the following example:

1. You have specified in a configuration that files be merged automatically.
2. You are currently viewing the merge file in DevPartner Java Edition while a Coverage session is running.
3. When that Coverage session ends, the session file data is automatically merged into the merge session you are viewing. The data you are looking at is now invalid, because the file that you are looking at just changed.

When this situation occurs, DevPartner Java Edition issues a warning that the session file you are viewing is out of date. To view the updated data, close your view into the merge session file, return to the Start page, and open the new copy of the merge session file.

Deadlock: The Deadly Embrace

Deadlock is a common problem with multithreaded applications. In Java applications, deadlock can be caused by threads calling into a series of synchronized blocks or methods in different order. DevPartner Java Edition can assist in the analysis of applications deadlocked by such "out of order synchronization" calls.

For example, a deadlock can occur if Java thread T1 synchronizes on objects O1, O2, and O3 (in that order) but thread T2 synchronizes on the same objects in a different order (for example, O3, O2, and O1). Deadlocks can be sporadic, making detection difficult.

Note: "Thread T1 synchronizes on object O1", means that thread T1 has entered object O1's monitor. Object O1 might be a class (in the case of class-based synchronization), an implied object instance (in the case of a synchronized method), or a specified object (in the case of a synchronized block).

Throughout this section on out of order synchronization, "sequences of calls into synchronized methods" refers to nested calls. For example, "thread T1 calls synchronized methods M1 and M2 in that order," means that M2 is called while *still in the scope of M1*.

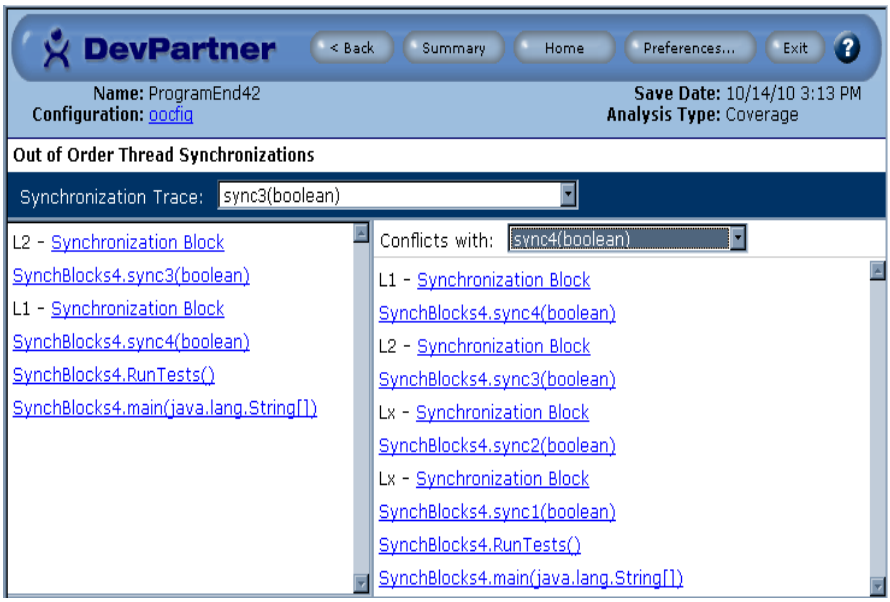
The out of order thread synchronization feature of DevPartner Java Edition is integrated with Coverage analysis. It detects potential deadlock situations by tracing your code's execution.

Note: Apparent out of order synchronization sequences will not trigger deadlock detection if they are "protected." If one thread calls synchronized methods M1, M2, and M3; and another thread calls those methods in the order M1, M3, and M2; that will not trigger deadlock detection. The sub-sequences involving M2 and M3 are protected by M1, so no deadlock can occur.

Analyzing Out of Order Thread Synchronization

The Out of Order Thread Synchronizations window shows the contents of a sample session file that recorded out of order thread synchronization. Figure 3-5 is an example of this window.

Figure 3-5. Out of Order Thread Synchronization Details



The window provides the following information:

- Synchronized blocks and methods involved in the potential deadlock scenario
- Method calls that led up to entry into those sequences

Each frame shows a synchronization trace. A *synchronization trace* is a list of synchronized methods or blocks, depending on the context. (A synchronized block entry is accompanied by its enclosing method name.) A synchronization trace looks like a stack trace; the method names in the list are arranged from bottom to top, in the order they were called.

A synchronization trace can include the following items:

- Synchronized method – A method that can only be executed by one thread at any given time.
- Synchronized block – Similar to a synchronized method in that only one thread can execute a synchronized block at any given time.
- Plain method – Any method that is not a synchronized method. The only plain methods shown in the synchronization traces are those that led up to entry into the conflicting sequences.
- Lock tag – A shorter name that is substituted for a longer method name or a synchronized block. The form is L*x* (where *x* is a number starting with 1 and incremented with each instance) followed by either the text **Synchronization Block** or the name of the method that caused out of order thread synchronization. Using lock tags makes it easier for you to find synchronization traces that occurred out of order, even if the synchronization trace is quite long.

  Out of order traces are always presented in pairs (i.e., one synchronization trace shows that one thread called the methods in one order, and the other synchronization trace shows that a different thread called the methods in a different order). Lock tags are particularly helpful when you are comparing synchronization traces that include synchronized blocks. A block has no associated name; a lock tag supplies a name for the block, so that you can match a block in one trace with its occurrence in another trace.

Note: DevPartner Java Edition labels the synchronization traces with the name of the last method executed by that thread.

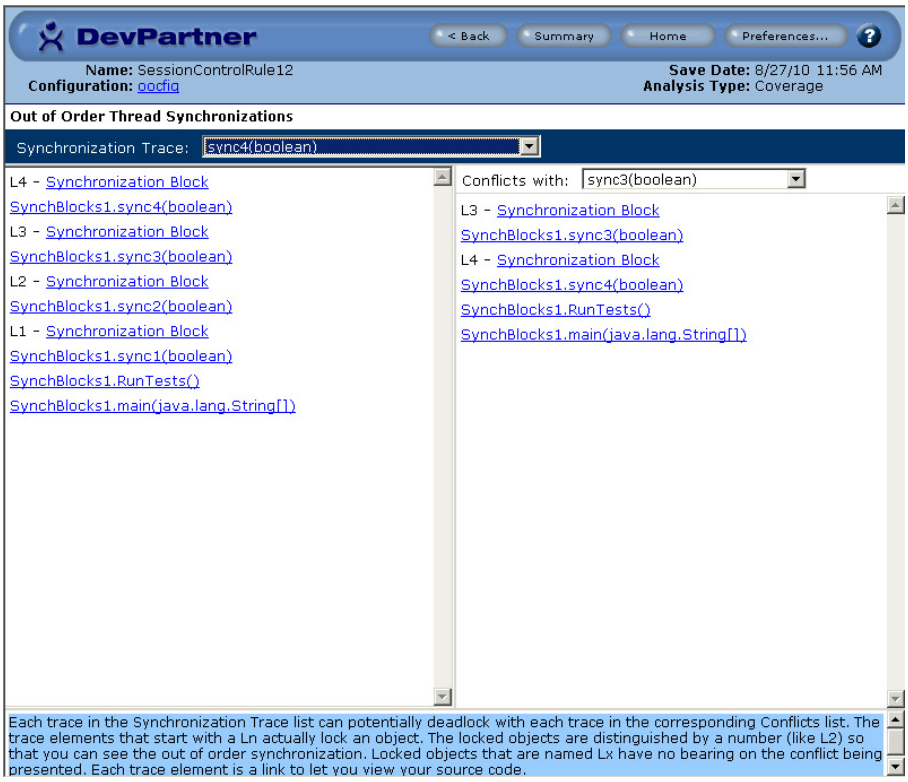
The elements in the synchronization trace lists are hyperlinks that lead you to the appropriate place in the source code (via the Source View) corresponding to the method or block. For example, click a link for a synchronized block, and you are taken to a source-code window. The window is positioned in the source to the location of the block, and the first line of the block is highlighted in yellow.

In Figure 3-5, there are two synchronization traces, one in the left frame (referred to as the “primary sequence”) and another in the right pane (referred to as the “conflicting sequence”). Each synchronization trace shows the order of program execution for a particular thread, reading from the bottom line up.

The **Synchronization Trace** list in the left pane and the **Conflicts with** list in the right pane allow you to select a specific sequence. The **Synchronization Trace** list allows you to select the primary sequence. The **Conflicts with** lists includes all the sequences that conflict with the current primary sequence.

Figure 3-6 shows a sample synchronization trace where deadlock was detected.

Figure 3-6. Out of Order Thread Synchronization Traces



The following describes the execution path of the left pane (primary sequence):

1. The thread begins by executing the **`main()`** method. (Note that the class name is **`SynchBlocks1`**.)
2. From the **`main()`** method, the thread calls the **`RunTests()`** method.
3. The **`RunTests()`** method calls the **`synch1()`** method. While executing the **`synch1()`** method, the thread enters the first synchronized block that participates in this particular conflict. The synchronized block is identified by the lock tag `L1`.
4. The thread enters the **`synch2()`** method, in which there is another synchronized block, identified by the lock tag `L2`. This process repeats for methods **`synch3()`** and **`synch4()`**; these methods are associated with `L3` and `L4`, respectively.

After these steps, the thread exits “pops out” of the nested method calls, and ends execution.

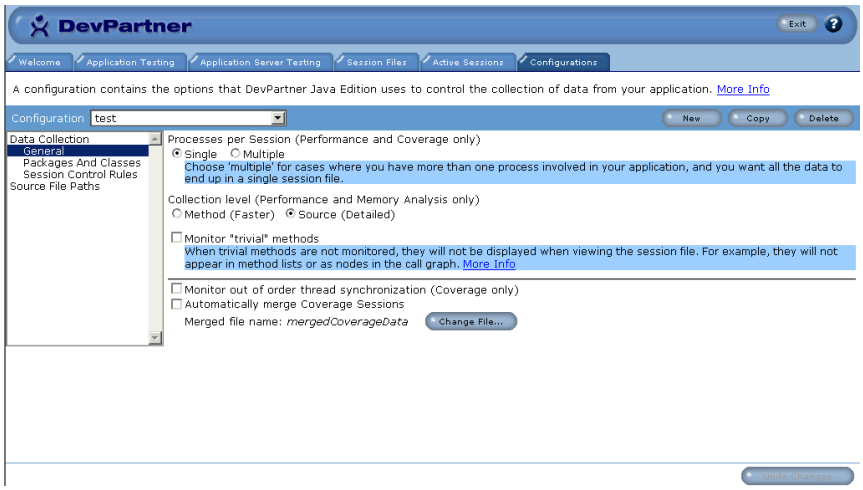
Only one conflict pair is displayed at a time. When you select a primary conflict, DevPartner Java Edition populates the **Conflicts with** list with all the traces that conflict with the selected primary trace. You then select which secondary trace to display.

Detecting Out of Order Thread Synchronization

If you suspect that deadlock is occurring, perform the following steps to analyze and debug the deadlocks:

1. To enable monitoring of out of order thread synchronizations as part of Coverage analysis, select **Monitor out of order thread synchronization (Coverage only)** on the **Configurations** tab of the Start page.

Figure 3-7. Configurations Tab



2. Run DevPartner Java Edition against that program, making sure you specify a configuration that has enabled out of order thread synchronization using a command line similar to the following:

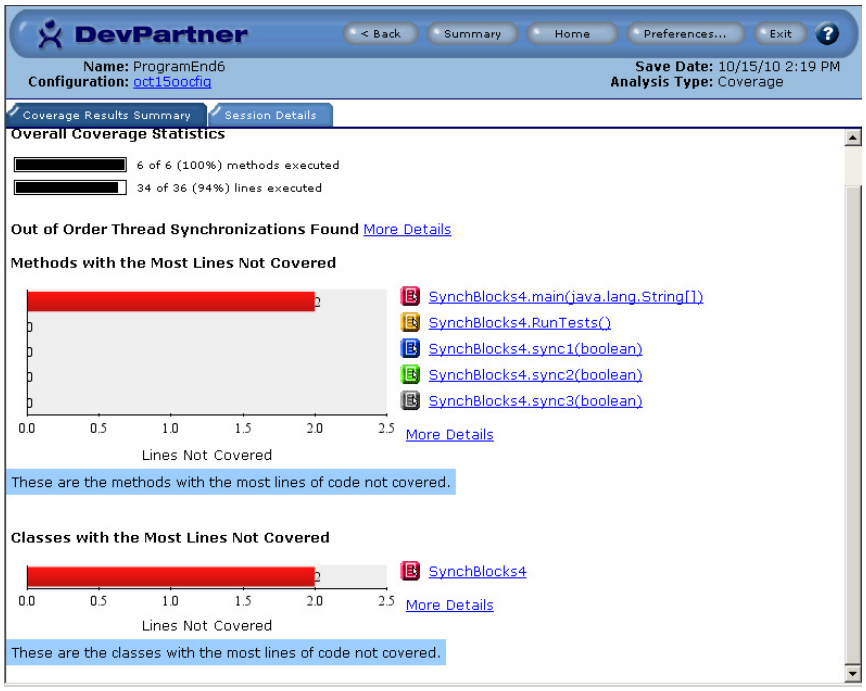
   `nmjava -config` ***`config-name`*** `-cov -batch` ***`Program-name`***

   where ***`config-name`*** is the selected configuration and ***`Program-name`*** is the name of your program.

3. Exercise the target application in such a way that potential deadlocking sequences are executed.

4. On the **Coverage Results Summary** tab, click **More Details** for **Out of Order Thread Synchronizations Found** to display information that will help identify deadlocks in your code.

Figure 3-8. Coverage Results Summary Tab



5 From the **Synchronization Trace** list on the left pane of the Out of Order Thread Synchronizations window, select the primary synchronization trace to examine.

6 From the **Conflicts with** list on the right pane, select one of the conflicting traces. (See Figure 3-6 on page 61 for the output of a sample run.)

7 Examine the sequences to determine where the sequence of Synchronization Blocks differs. In this example, the primary sequences shows that the thread labeled `sync4 method` executes synchronization blocks `L3` and `L4` in that order, while the conflicting sequence shows the thread labeled `sync2 method` has the sequence of `L3` and `L2`. This is the out of order conflict in this example.

8 To determine how this happened, examine the code of the method preceding the point of departure. Look for areas in the code where some global variable, `if` statement, `case` statement, or other code element might have caused the divergence.

9 If you do not find an instance where a decision point might have caused the divergence, examine the code in the next higher parent. Again look for decisions that would cause the thread to take different a path of execution.

Exclusions and Out of Order Synchronization Detection

If a class with the synchronized method or block is excluded, DevPartner Java Edition does not monitor that class. As a result, if synchronized methods or blocks in the excluded code participate in out of order deadlock, DevPartner Java Edition does not detect them.

Excluding a parent class (whose synchronized methods are not overridden in included classes) excludes this class from analysis. Any synchronized methods from the parent class that participate in out of order deadlock will not be detected by DevPartner Java Edition.

Excluding the target class of a synchronized statement does not exclude the class performing the synchronization. For example, if you had an object **A** of class **C**, and a method that looked like the following, excluding class **C** does not cause the method to be excluded from any synchronization traces in which it would otherwise appear.

```
public int method()
  {
     synchronized(A) { ... }
     ....
  }
```

Tracking Code Execution and Code Base Stability and Reliability

To ensure the reliability of your software, you must know how much of your code is being exercised by your tests and how stable your code base is.

DevPartner Java Edition Coverage analysis tracks code execution and code base stability, helping you locate untested code and areas of volatility. With this information, you can minimize testing time while maximizing the productivity of your testing efforts.

- Coverage analysis collects coverage data for Java, down to individual lines of code.
- Session Controls let you focus your Coverage analysis on any phase of your application. Use the Session Controls to take a snapshot of the data currently collected, or clear data collected to that point and then continue recording.
- File merging combines the coverage data from multiple session files into a single file. Accumulating data lets you track changes in your code so that you can gauge the stability of your code base.
- Out of order thread synchronization detects potential thread deadlock conditions. When you enable out of order thread synchronization for a session, all synchronization points in your program that have conflicts requiring examination are recorded in the coverage session file.

Run Coverage analysis even after you have 100% code coverage. Just because your code tests at 100% coverage doesn’t mean that each method or class has the same number of tests. In many situations, multiple tests must be run on the same feature to ensure that the functionality is flawless. Use DevPartner Java Edition to determine which areas of your application need more testing.

Chapter 4

Finding Performance Problems

The Performance analysis capabilities of DevPartner Java Edition help you to track down poorly-performing code. In this chapter, you'll see how to use DevPartner Java Edition to gather performance data on an application, and how to use its intuitive interface to navigate through that data and home in on performance hotspots.

Identifying Performance Problems

Ensuring that an application performs optimally is a top priority for any Java programmer. Recognizing the importance of optimal performance, however, does not alleviate the task of finding and correcting the problem code. Performance issues are often more complex than a developer might at first suspect. Finding the origin of the problem often requires exhaustive testing of the application, followed by a laborious walk-through of the suspicious areas of code.

Suppose testing has revealed that an application is functioning properly. It is doing what it was designed to do. Members of the testing group, however, have reported that the application is slower than users are likely to tolerate. The testers have not quantified their findings; they are simply reporting their subjective experience with the application.

After running the application yourself, you concur with the testing group. The application is unacceptably slow, though you are not sure why. You are faced with two questions:

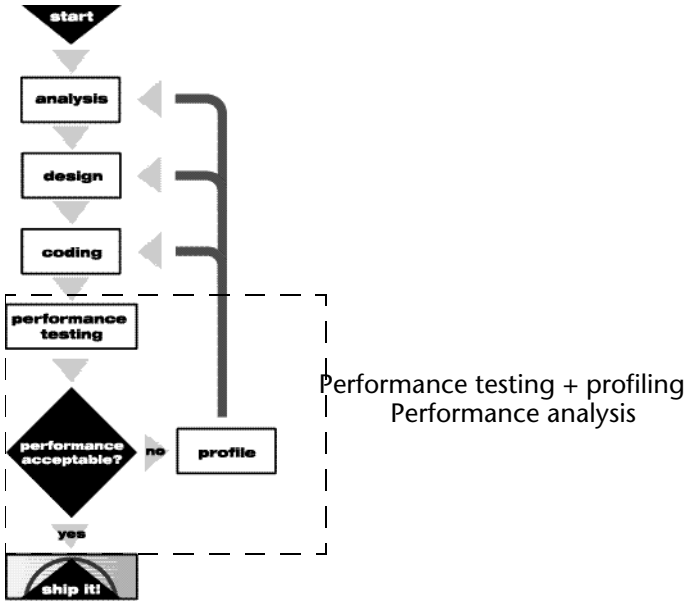
- How can you collect performance data on the application?
- How can you use that data to isolate the offending code?

The answer to both, of course, is to use a Performance analysis tool such as DevPartner Java Edition. However, even with DevPartner Java Edition, the proper cure for performance problems is not to address those problems **after** someone complains of your application's sluggishness. The real solution is more proactive than reactive.

Performance Testing and Profiling in Software Development

Performance problems can be largely eliminated if you include performance testing and profiling into the software development cycle, as illustrated in Figure 4-1.[1]

Figure 4-1. Performance Testing as Part of Software Development



In *Java Platform Performance, Strategies and Tactics*, the authors show how performance testing and profiling fit into the software development life cycle. As they suggest, after you have determined that your code's execution is functionally correct, you employ performance testing to verify that your application's execution time is acceptable.

If your application passes the performance tests, you can move to the next stage of the cycle (which is often deployment). Otherwise, you follow the testing with a performance profiling step, in which you gather data to locate the problem (or problems). Analysis of the results of the profiling step will send you back to a preceding stage in the cycle where the problem can be corrected. Perhaps the slow code was a design flaw; or, perhaps it was a poorly-coded algorithm. In any case, the performance testing and profiling steps produce a cyclic process. You run performance tests and use profiling to determine whether you must loop back to a previous step (analysis, design, coding) in order to fix a problem. Cycles will be repeated until the application meets design standards.

Performance Profiling Terminology

Before you run a Performance analysis session in DevPartner Java Edition, there are concepts and terms used throughout the documentation that you might be unfamiliar with.

Profiled versus Excluded Code

Profiled code is the code for which you want DevPartner Java Edition to gather performance data. Generally, this is the application code you actually wrote. Profiled code is distinguished from *excluded code*, for which DevPartner Java Edition does not collect performance data.

1. *Java Platform Performance, Strategies and Tactics*, Wilson, Kesselman)

Excluded code is usually that part of your application's source over which you have no direct control; for example, system code and third-party libraries.

Note: Occasionally, you may want to explicitly exclude portions of your own code to narrow your area of investigation. By default, DevPartner Java Edition excludes classes in the Java Runtime Environment (JRE), application server classes, and IDE classes.

DevPartner Java Edition lets you select which parts of an application are profiled and which are excluded. You do this in the **Packages and Classes** section of the **Configurations** tab of the DevPartner Java Edition Start page. (See the online help for details.)

Entry Points

When your program runs, profiling begins with the first call to a method that is not on your excluded list. In DevPartner Java Edition, such methods are called *entry points*. (Methods that are called by other profiled methods are not entry points.) The **Entry Points with the Slowest Average Response Time** graph on the Performance Results Summary identifies execution paths that consume the most clock time and, consequently, should be explored for poorly performing code.

Performance Timings

DevPartner Java Edition recognizes timing values: wait time, real time, and thread time. The following equation illustrates the relationship of these timings:

```
wait time = real time – thread time
```

Wait Time

Wait time is the amount of time that a thread spends waiting to execute. A thread can accumulate wait time while it is blocked on an I/O operation, while it is waiting for a synchronization object, or simply when it is preempted by other threads.

Note: Wait time is associated with a method. A method's overall wait time is the sum of the individual times that all the threads that have executed that method have spent waiting. Real time and thread time values (defined below) are also associated with methods.

Real Time

Real time (also called *clock time* or *elapsed time*) is the total time from the moment the profiled method is called to the moment the method exits, summed across all invocations of the method. If a method is called at time T1, exits at time T2, is called later at time T3, and exits at time T4, its real time is (T2 – T1) + (T4 – T3).

Real time includes timings for other events, child methods, and threads executing outside of the context of the profiled portion of the application; it may even include the times of other processes that preempted the application. It may help to think of real time as the amount of time that a user perceives a method as having taken to execute.

Besides reporting the elapsed real time, DevPartner Java Edition also reports a method's average real time, which is simply the method's real-time measurement divided by the number of times the method is called.

Thread Time

Thread time (also called *CPU time*) is the amount of time threads spend executing in a particular method, independent of any other events taking place. It excludes time spent in any child methods. Thread time is the time that pertains only to the method being profiled.

Example: Wait Time, Real Time, and Thread Time

A simple illustration might clarify the relationship of these different time values. Suppose your application consists of a single method, run by a single thread. The instant the method begins executing, an LED is illuminated on the front panel of your system. When the method terminates, the LED turns off. In the midst of executing the method, however, the system preempts your application, runs a background task for several tens of microseconds, then returns to complete your application. The real time would be the length of time the LED was illuminated. The wait time would be the amount of time that the background task executed (the time "stolen" from your application). The thread time would be the real time less the wait time.

Trivial Methods

A *trivial method* is, as its name implies, an extremely brief and simple method. It performs only one of the following operations:

- Returns a constant value, or a member, static, or parameter variable.
- Returns the length of an array.
- Puts a constant value, an array length, or a member, static, or parameter variable into a member or static variable.
- Calls one other function, passing parameters that are constant values, array lengths, or values that can be retrieved from member, static, or parameter variables.

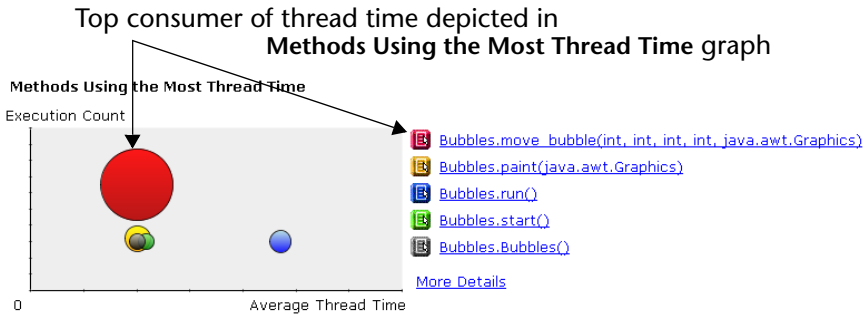
DevPartner Java Edition lets you enable or disable monitoring of trivial methods. You do this in the **Configurations** tab. Generally, the first time that you profile for performance, it is a good idea to enable **Monitor Trivial Methods**. This ensures that DevPartner Java Edition monitors all methods so that you see the entire execution path of your program, rather than only a filtered view. If you elect not to monitor trivial methods, your results might be misleading. For example, if your code executes **`method_a`**, **`method_b`**, and **`method_c`** (in that order), but **`method_b`** is a trivial method, the performance Call Graph (see example in Figure 4-9 on page 75) would show **`method_a`** calling **`method_c`**; **`method_b`** would not even appear in the Call Graph. Because you disabled that setting, you would not know whether **`method_b`** was ever executed. When you enable the monitoring of trivial methods, however, DevPartner Java Edition would accurately include all methods (trivial or not) in its analysis results.

A benefit of not monitoring trivial methods is that you significantly reduce data collection overhead. This, however, may cause you to misinterpret the results. A good approach is to enable the monitoring of trivial methods the first time you profile your code. Once you fully understand the execution path of your program, you can run a subsequent Performance analysis, this time without monitoring trivial methods, to focus your attention on the critical methods in your application that are causing the greatest performance hits.

Bubble Graphs

DevPartner Java Edition uses *bubble graphs* to represent three variables in two dimensions. The **Methods Using the Most Thread Time** graph, for example, shows the top five methods that use the most thread time.

Figure 4-2. Top Consumers of Thread Time



Each method is depicted as a bubble. The size of each bubble represents the accumulated thread time for the associated method. Hence, the larger the bubble, the more thread time that the method has accumulated.

The location of a bubble in the graph helps you identify which factor likely contributes the most to the corresponding method's accumulated thread time. The closer a bubble is to the top of the graph, the more frequently the associated method is called. In that case, you would likely accomplish the largest reduction in the method's overall thread time by reducing the number of times it is called. The farther to the right side of the graph a bubble is situated, the greater its associated method's average thread time. For such a method, the largest reduction in thread time would most likely come from optimizing within the method itself.

Running a Performance Profiling Session

DevPartner Java Edition provides various ways to look at Performance analysis results. Whatever direction you take, you can drill down into your data using one of the graphs provided from a performance profiling session. Choose a graph that focuses on the problem you are trying to solve. Table 4-1 provides some rules of thumb.

Table 4-1. Using DevPartner Java Edition for Performance Analysis

Performance Problem	Performance Analysis Tool	Issues Addressed
Methods with long execution times that are entry points into the application code	**Entry Points with the Slowest Average Response Time graph (see Figure 4-12 on page 76)**	Responsiveness, computational performance, start-up time
Methods that take a long time to execute because of conditions within the method itself (i.e., contains an inefficient algorithm or data structure)	**Methods Using the Most Thread Time graph (see Figure 4-7 on page 74)**	Computational performance, scalability

Table 4-1. Using DevPartner Java Edition for Performance Analysis

Performance Problem	Performance Analysis Tool	Issues Addressed
Methods that spend excessive time waiting because of being blocked, (i.e., waiting on I/O or a shared resource)	**Methods Spending the Most Time Waiting graph (see Figure 4-4 on page 72)**	Responsiveness, computational performance

Regardless of the graph you choose, you can drill down to another view that shows a different aspect of the captured results. For example:

- You can view a Call Graph to look at the chain of calls leading to a particular method call.
- You can view performance and timing details about a method.
- You can view a table listing all the methods in the application to compare performance and timing data at a glance.
- You can go directly to the source code to find slow lines of code.

To illustrate how you can use DevPartner Java Edition to find performance problems, use one of the sample applications provided with the installation. The sample application is called **`waitTimeExample`** and is located in the following path location(s):

- Windows — ***`DPJ_dir`*`\samples\waitTimeExample`**

  where ***`DPJ_dir`*** is the DevPartner Java Edition product folder
- UNIX — **`/opt/Micro Focus/DPJ/samples/waitTimeExample`**

This example explores how to use DevPartner Java Edition to locate excessive wait time in a Java program. This multi-threaded program writes to a simple in-memory log object that consists of a small, circular buffer. You can run this application in one of two modes:

- Global, in which all threads use a single synchronized buffer
- Local, in which each thread uses its own buffer

Each mode illustrates different capabilities of DevPartner Java Edition. Start by profiling the application in *global* mode and focus on computational performance issues. Following that, profile in *local* mode, for a different perspective, and drill down to results that look at computational performance factors as well as program responsiveness.

Profiling in Global Mode

To run this sample application in global mode:

1. At the command prompt, change the folder to

   `DPJ_dir``\samples\waitTimeExample`

   where ***`DPJ_dir`*** is the DevPartner Java Edition product folder.

2 Run the command

```
nmjava -perf -cp .. waitTimeExample.ThreadLogMain global
```

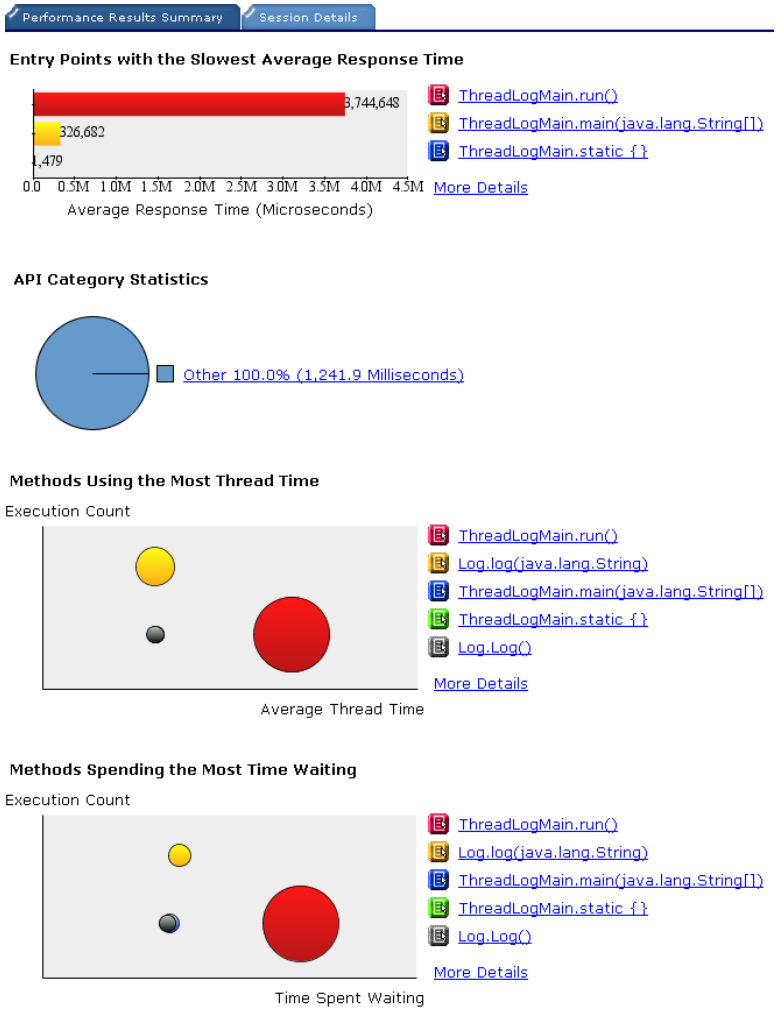
This command profiles the application in global mode. As your application runs, the Session Control page appears and prompts you to view results.

Note: For a demonstration of the Thread Viewer in the Session Control page, see "Using the Thread Viewer to Analyze Performance" on page 77.

3 Click **Yes** at the prompt.

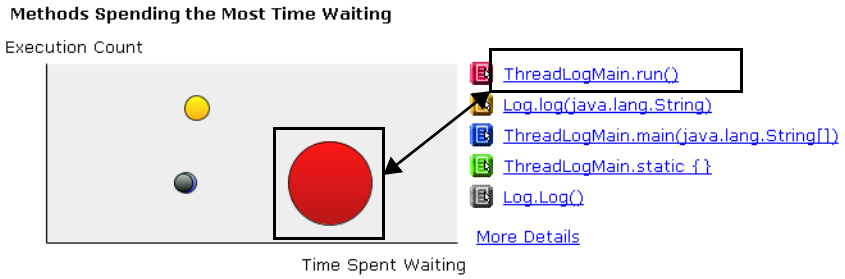
The Performance Results Summary displays the performance graphs, as shown in Figure 4-3.

Figure 4-3. Performance Results Summary — Global Mode



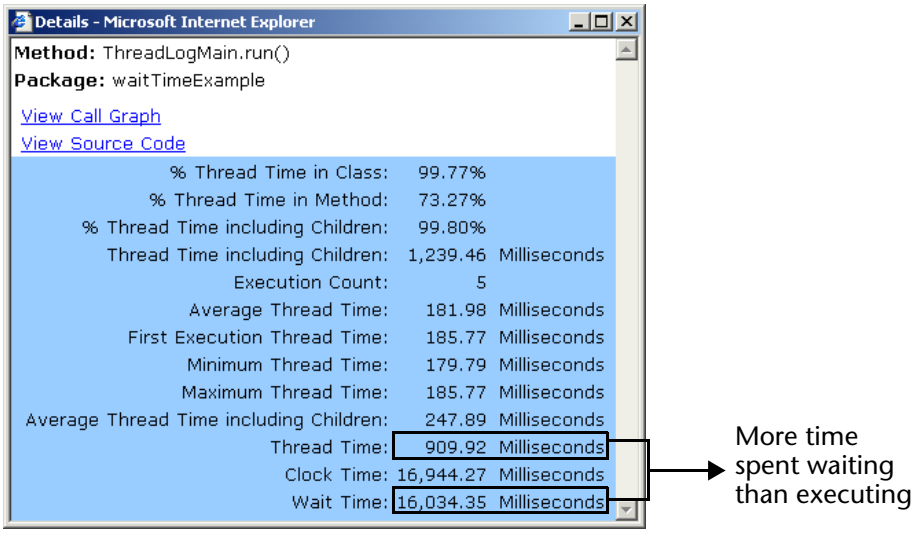
In the **Methods Spending the Most Time Waiting** graph, **`ThreadLogMain.run()`** is at the top of the list because it calls the **`Log.log()`** method. The method **`Log.log()`** executes the writing to the circular buffer, and is synchronized to eliminate the possibility of corruption. Because this application runs several threads — each thread executing **`ThreadLog-Main.run()`** — the threads spend most of their time in **`ThreadLogMain.run()`**, waiting for **`Log.log()`** to become available.

Figure 4-4. Methods Spending the Most Time Waiting to Execute



Click the hyperlink to the right of the graph to display a Details window for **`ThreadLogMain.run()`**, in which you can see the wait time.

Figure 4-5. Details of **`ThreadLogMain.run()`**



Look at the bottom of the window. Notice that the code executing **`ThreadLogMain.run()`** spends far more time waiting (**Wait Time** entry) than actually executing (**Thread Time** entry).

Profiling in Local Mode

An alternative to logging information into a single, shared buffer is to allot a buffer to each thread. This enables each thread to save data to its own local buffer for later merging. This approach is demonstrated using the local mode of the sample application. It dramatically reduces the synchronization overhead of the log call itself.

To run this sample application in local mode:

1 At the command prompt, change the folder to

`DPJ_dir\samples\waitTimeExample`

where ***`DPJ_dir`*** is the DevPartner Java Edition product folder.

2 Run the command

```
nmjava -perf -cp .. waitTimeExample.ThreadLogMain local
```

This command profiles the application in local mode. As your application runs, the Session Control page appears and prompts you to view results.

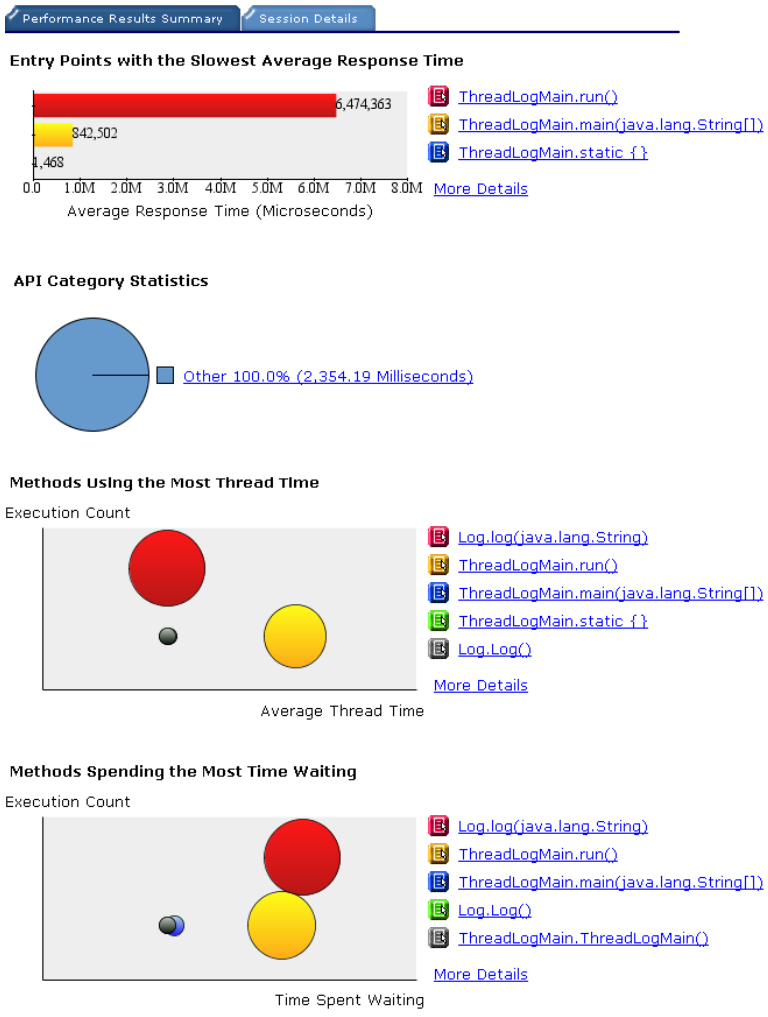
Note: For a demonstration of the Thread Viewer in the Session Control page, see "Using the Thread Viewer to Analyze Performance" on page 77.

3 Click **Yes** at the prompt.

The Performance Results Summary displays results captured in the local mode.

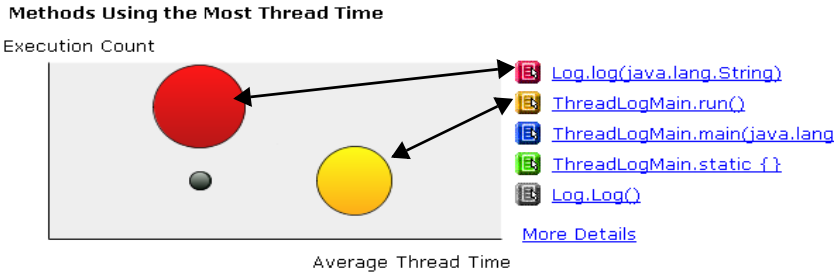
As shown in Figure 4-6, the results in the local mode vary somewhat from the results in the global mode. Instead of sharing a single buffer, each of the five threads now has its own buffer. The wait time has been shifted from the **`ThreadLogMain.run()`** method down into the **`Log.log()`** method. In the global version, wait time was caused by loitering threads waiting to get into the synchronized **`Log.log()`** method. In the local method, virtually all the wait time is associated with threads contending for the CPU. Even though the local mode finishes faster, the total amount of thread time (CPU time) has actually increased (this increase is even more apparent if you are on a single CPU system). This is due, in part, to the fact that this sample application is completely CPU-bound.

Figure 4-6. Performance Results Summary — Local Mode



The **Methods Using the Most Thread Time** graph provides a snapshot of the greatest consumers of pure thread time in the program, when it was profiled in the local mode.

Figure 4-7. Methods Using the Most Thread Time

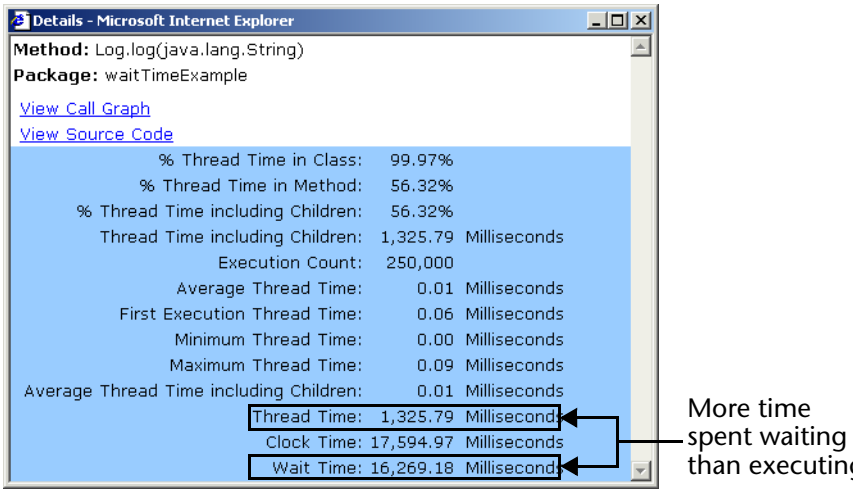


The graph provides several clues. The largest bubble is the method **`Log.log(java.Lang.String)`**, shown in red at the upper part of the graph. Because it is the largest bubble, the associated method has accumulated the most thread time relative to all other methods profiled. And, because the bubble is near the top of the graph, execution count plays the larger role — compared to the method's average thread time — in the accumulated time.

The graph highlights **`ThreadLogMain.run()`**. The position of the yellow bubble closest to the right side of the graph indicates that this method is expensive because, on average, it consumes a lot of thread time.

Next, look at the methods that call into **`Log.log(java.lang.String)`**, and determine if it is feasible to reduce the number of times it is called. Click the link for **`Log.log(java.Lang.String)`** to open a Details window.

Figure 4-8. Details Window — Profiled in Local Mode

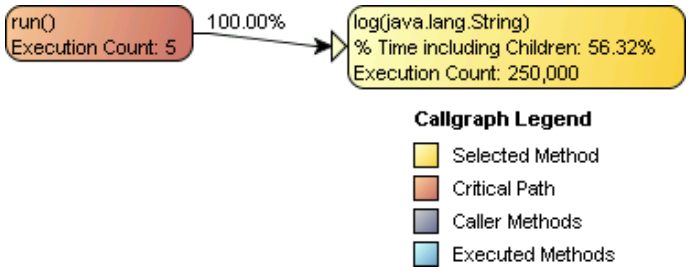


This window lists additional details about **`Log.log(java.lang.String)`**. Like when this program is profiled in the global mode (Figure 4-5 on page 72), more time is spent waiting to execute (wait time) rather than actually executing (thread time). This information quantifies the graphical representation just evaluated.

Analyzing the Call Structure

Look at the call structure for **`Log.log(java.lang.String)`**, to see methods that might be affected if making a change to this method. To view the Call Graph for this method, click **View Call Graph** in the Details window.

Figure 4-9. Call Graph Showing **`run()`** Calling into **`Log.log()`**

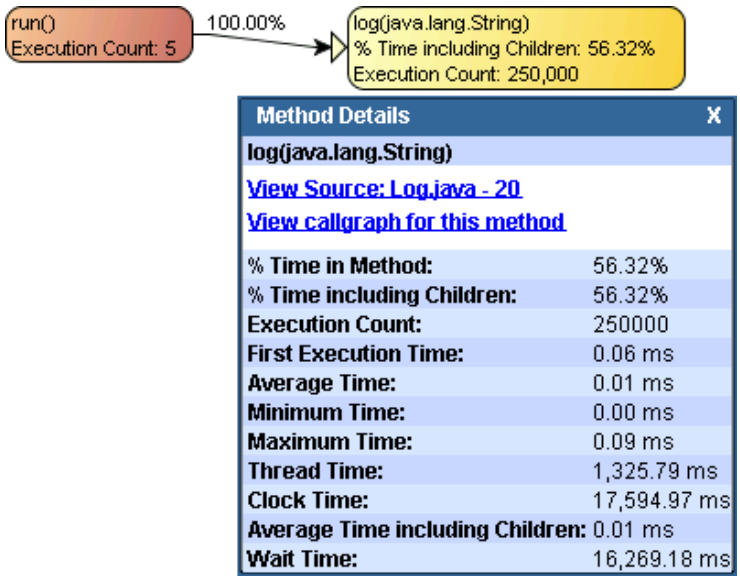


The nodes in the Call Graph are displayed sequentially from left to right in the order in which they were called. DevPartner Java Edition displays the critical path by determining the sequence of child method calls that resulted in the largest cumulative consumption of thread time for the selected method. The percentage that appears to the left of each node in the critical path is local to that path. It represents the time allocated by the child method (and its children) as a percentage of the time allocated in the execution of that path. In contrast, the percentage that appears inside the node or in the Details window is global. This value is computed as a percentage of time allocated by all methods during the profiling session.

Our sample application is a simple console program with a single call into a child node, and this is confirmed by the Call Graph. It shows **`ThreadLogMain.run()`**, appearing as **`run()`**, calling into **`Log.log(java.lang.String)`**. It further tells us that **`run()`** would be affected by a change to the code in **`Log.log`**.

To go to the Source View, click the node for **`Log.log(java.lang.String)`**. This displays the Method Details window, as shown in Figure 4-10. Click the hyperlink **View Source** appearing in this window.

Figure 4-10. Details of **`Log.log(java.lang.String)`**



Finding Slow Code

The following example shows the Source View for **`Log.log(java.lang.String)`**. DevPartner Java Edition highlights the slow lines of code in red. For both slow lines of code, more time is spent waiting (**Wait Time** column) than executing (**Thread Time** column).

Another factor in performance is how many times a line executes; For example, the second highlighted line of code executes 250,000 times (**Execution Count** column).

Figure 4-11. Source View for **`Log.log`**

waitTimeExample.Log

Printer Friendly Version

Execution Count	% Thread Time in Method	Thread Time (Milliseconds)	Clock Time (Milliseconds)	Wait Time (Milliseconds)	Source: Log.java Path: C:\Program Files\Micro Focus\DevPartner Java Edition\samp
					package waitTimeExample;
					import java.util.LinkedList;
					/**
					* A simple circular log class
					*/
					public class Log
					{
					private final static int DEFAULT_LOG_CAPACITY = 50;
					private final LinkedList log;
					public Log()
					{
6	97.32%	0.34	0.35	0.01	log = new LinkedList();
6	2.68%	0.01	2.57	2.56	}
					public synchronized void log(String message)
					{
250,000	19.55%	259.23	3,380.01	3,120.78	String threadName = Thread.currentThread().getName(
250,000	29.54%	391.66	4,302.17	3,910.51	log.addLast(threadName + ": " + message);
250,000	16.50%	218.71	3,350.63	3,131.91	if(log.size() >= DEFAULT_LOG_CAPACITY)
249,755	18.00%	238.69	3,243.03	3,004.34	log.removeFirst();
250,000	16.41%	217.50	3,319.14	3,101.64	}
					}

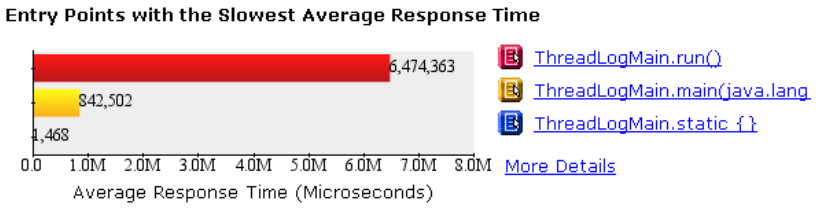
Reducing the number of calls is an important way to help improve overall performance.

DevPartner Java Edition presents another way to understand program performance and responsiveness, through the analysis of the **Entry Points with the Slowest Average Response Time** graph on the Performance Results Summary.

Looking at Program Responsiveness

In the preceding solution track, after profiling in local mode, methods that were the greatest consumers of thread time and executed most often were analyzed. In addition, the amount of time that the method spent waiting to execute was analyzed. A program's responsiveness can also be evaluated by looking at the **Entry Points with the Slowest Average Response Time** graph on the Performance Results Summary, when profiled in local mode Figure 4-6 on page 73).

Figure 4-12. Entry Points with Slowest Average Response Time — Local Mode



As described earlier, the **Entry Points with the Slowest Average Response Time** graph shows those entry points with the longest average clock time. In the example above, the bulk of the execution time is in **`ThreadLogMain.run()`**, so it is at the top of the list. In addition, there are few entry points in this sample application because it runs as a command line application. Results from this graph coincide with other data about expensive methods. Remember that **`ThreadLogMain.run()`** is also one of the top consumers of thread time in the program, and that this method calls into the method that consumes the most thread time, **`Log.log(java.lang.String)`** (Figure 4-9 on page 75).

Using the Thread Viewer to Analyze Performance

The Thread Viewer in the Session Control page provides a live view of thread states. It lists all the threads that are currently running or recently terminated. The graph shows how long each thread is running or waiting (possibly blocked on the monitor), and when the thread is terminated.

The Thread Viewer is enabled by default; you can disable it in the session configuration. (When it is disabled, the Thread Viewer is not displayed in the Session Control page.)

To see a demonstration of the Thread Viewer, use the sample application **`threadExample`**.

This demonstration uses the default settings in the session configuration.

1. At the command prompt, change the folder to

   `DPJ_dir``\samples\threadExample`

   where *`DPJ_dir`* is the DevPartner Java Edition product folder.

2. Run the command

   ```
   nmjava -perf -cp .. threadExample.BounceThread
   ```

   The DevPartner Java Edition interface and the Bounce application window open.

3. To start the Bounce application, click **Start**. A ball appears; it bounces randomly around the screen 1,000 times, then disappears.

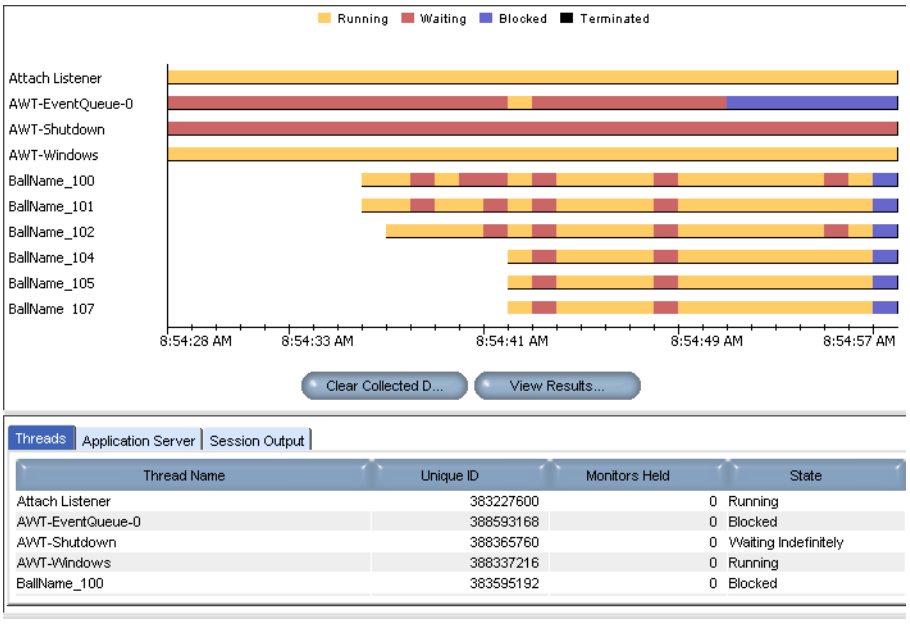
   To add more bouncing balls to the screen, click **Start** again; each new ball is a separate thread. Use the **Clear** button to stop all the ball threads, which removes the balls from the Bounce window.

   To increase or decrease the bouncing speed, click the **+** (plus) or **-** (minus) button.

   To stop the application and close the Bounce window, click **Close**.

As the Bounce application runs, its threads are displayed in the “live view” graph in the Session Control page.

Figure 4-13. Thread Viewer



The thread names are listed down the Y-axis of the graph; long thread names may be truncated. The X-axis shows the time intervals. Thread status is identified by color: yellow for running, red for waiting, purple for blocked, and black for terminated. Each time you click Start and a new ball appears in the Bounce window, a new thread begins running and appears in the graph.

The graph is refreshed once every second. You may not see a thread's state change between Running and Waiting if the state changes too quickly to be captured by the refresh. If a thread is never listed in the graph, either it runs and terminates too quickly to be captured (i.e. less than a second), or it was never executed.

The duration for displaying thread states ranges from a minimum of 5 seconds through a maximum of 120 seconds. The default is 30 seconds.

Note: A higher duration requires higher overhead for retaining the data, and may degrade performance. If the profiling proceeds unacceptably slowly, disable the Thread Viewer.

A terminated thread continues to be listed only for the number of seconds specified for Thread Viewer History in the Live View in the session configuration (for example, for another 30 seconds, if the default duration is used).

The thread data is not saved to a file; it persists only as long as the thread is displayed in the graph.

While the application runs, you use the buttons below the graph to open a Performance Results Summary window to view the data collected thus far, or to clear the collected data.

The **Threads** tab at the bottom of the Session Control page displays a table that provides four columns of information about the currently running threads:

- **Thread Name** — The threads are listed in the same order as in the graph. A terminated thread disappears from the table when it disappears from the graph.

- **Unique ID** — DevPartner Java Edition assigns a unique identifier to each thread because multiple threads may have the same name, or thread names may be truncated to identical strings.
- **Monitors Held** — As each thread runs, this column displays the number of synchronized methods or blocks of Java code currently held by the thread. You may see the number change as the thread holds and releases the monitors. A ball thread can hold up to 5 monitors.
- **State** — This column displays the current state of the thread: Running, Waiting, or Terminated.

Note: For descriptions of the other two tabs, see the “Session Control – Performance” topic in the online help.

When you close the Bounce window, you are prompted to display the session results.

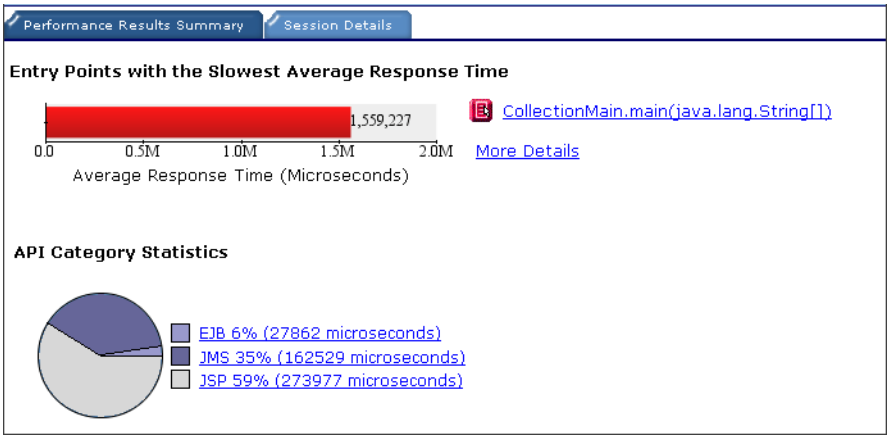
Analyzing Performance by Object Category

The **API Categorization and Transaction** configuration option focuses Performance analysis on specific parts of your program. For example, if you change one area of code, you can assign all the affected objects to the same category so performance data for the objects will be grouped together for analysis. The option is enabled by default.

The **Configurations** tab lists default categories. You can create your own categories as needed. You can use regular expressions for the categories (see the online help for details).

The category statistics are displayed in a pie chart on the Performance Results Summary, as shown in the following example.

Figure 4-14. Performance Results by Category



Performance Analysis Pointers

To recap:

- Performance analysis should be a planned step (not an afterthought) in the development process. Its point of inclusion is suggested in *Java Platform Performance, Strategies and Tactics, which gives other information regarding the development process, and the role that Performance analysis plays in it.*
- The ability of DevPartner Java Edition to drill down into session data is critical in uncovering the cause of a performance bottleneck. When you have identified a problem, locating it requires the ability to easily shift your view from charts to Call Graphs to numeric data to source code and so on. Pinpointing the source of the problem is like calculating a parallax, and that is most easily done when the problem can be viewed from different angles, and by different means. DevPartner Java Edition's interconnected user interface lets you jump quickly from one view to another.
- Although there is no formal recipe for tracking the problem to its source, a good rule-of-thumb approach is to begin with the **Entry Points with the Slowest Average Response Time** graph, and tunnel into the session data along a route charted by the Call Graph and guided by the **Methods Using the Most Thread Time** and **Methods Spending the Most Time Waiting** graphs. The latter two graphs not only will suggest the destination, but will indicate the nature of the problem (whether it is processing that must be optimized, or wait time that must be reduced). Experience with this process will enable you to improve it over time.
- Finally, keep in mind that, in Java applications, many performance problems have their root causes in improper memory management. Locating such problems will require you to employ DevPartner Java Edition’s Memory analysis capability, as described in Chapter 2.

Chapter 5

Working with Integrated Development Environments

In the preceding chapters, you saw how Java applications, applets, and components could be profiled either by using DevPartner Java Edition command line utilities, or by executing the applications interactively through the DevPartner Java Edition user interface. There is yet another way to incorporate DevPartner Java Edition in your development process. DevPartner Java Edition integrates with several of the most popular Java integrated development environments (IDEs).

After DevPartner Java Edition has been incorporated into an IDE, its presence is virtually unnoticeable unless you specifically activate it. In most cases, controls for activating DevPartner Java Edition appear either as a new menu on the menubar, or as a new entry in an existing menu. You can run a target project under DevPartner Java Edition as easily as if you had run the target from the IDE without using DevPartner Java Edition.

For example, as soon as a particular module is ready to run, you can launch the module under DevPartner Java Edition to analyze its memory consumption (or its execution performance), return to the IDE to make any changes, rerun through DevPartner Java Edition to test the modifications, and so on.

IDE integration is available only in Windows environments.

For the most part, profiling an application (or applet or Java component) with DevPartner Java Edition from within an IDE is as easy as launching that application with the IDE's **Run** command. After DevPartner Java Edition is installed in the IDE through the Add-in Manager, no additional work is necessary to make a new or existing project in the IDE compatible with DevPartner Java Edition.

Nevertheless, there are details of which you should be aware:

- Almost all IDEs have built-in debugging facilities. Some have various profiling capabilities. These facilities are disabled when you run a target under DevPartner Java Edition.
- Many of the IDEs are accompanied by "test-bed" application servers and EJB containers. You can launch servlets, JSPs, and EJBs from within the IDE to execute in the application server, and profile those components under DevPartner Java Edition as easily as launching them directly from the IDE without involving DevPartner Java Edition. In such a case, however, the behavior of DevPartner Java Edition will be slightly different than were the components executed in the application server (or EJB container) by the DevPartner Java Edition `nmserver` command or from the **Application Server Testing** tab of the DevPartner Java Edition Start page.

  When DevPartner Java Edition launches the application server, an entry for it appears on the **Application Server Testing** tab. If, however, the application server is launched as part of a project from within an IDE, no entry appears in the DevPartner Java Edition interface. The behavior of the tested components, the application server, and DevPartner Java Edition are not otherwise affected.

Installing and Uninstalling IDE Integration

DevPartner Java Edition includes utilities for integrating it into IDEs and for removing the integration.

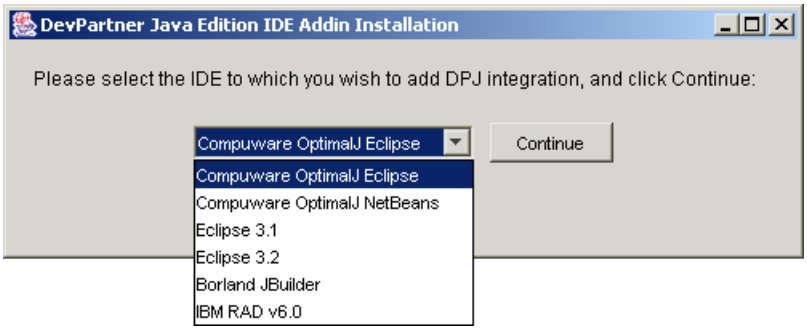
Note: You can also invoke the profiler through your IDE by including the `-Xrun` argument in the JVM settings. For details, see the online help topic “Using -Xrun to Invoke the Profiler”.

Using the Add-in Manager

You can integrate DevPartner Java Edition into your IDE at the time you install DevPartner Java Edition. If you select this option, the installer automatically launches the DevPartner Java Edition IDE Add-in Manager (see Figure 5-1). If you do not integrate DevPartner Java Edition as part of installation, you can perform this task at any time afterward. The procedure is the same during or after installation.

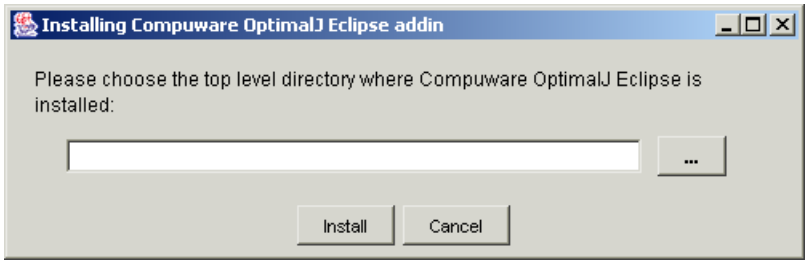
If the Add-In Manager is not opened by the installation wizard, select it from the Windows **Start** menu: **Programs>Micro Focus>DevPartner Java Edition>Utilities>Java IDE Add-in Manager**.

Figure 5-1. IDE Add-in Manager



Within the utility, perform the following steps:

1. From the list, select the IDE into which you want to integrate DevPartner Java Edition.
2. Click **Continue** to display the **Installing addin** dialog box.

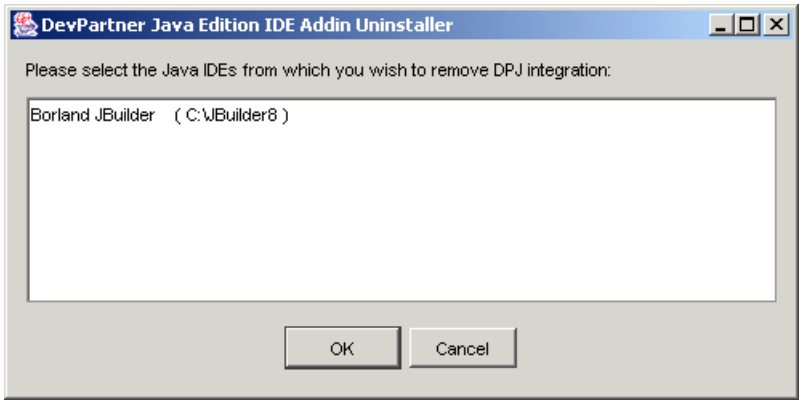


3. In the field, type the root folder in which the IDE has been installed; or click the browse button to navigate to and select the location.
4. Click **Install**. A message confirms that the integration was successful.

Uninstalling IDE Integration

Use the Java IDE Add-in Uninstallation utility to remove DevPartner Java Edition integration from an IDE.

1 From the Windows **Start** menu, select **Programs>Micro Focus>DevPartner Java Edition>Utilities>Java IDE Add-in Uninstallation**.



2 In the window, select the IDE and click **OK**.

3 The DevPartner Java Edition integration is removed from the selected IDE. A confirmation message appears.

IDE integration is uninstalled automatically when you uninstall DevPartner Java Edition.

Running DevPartner Java Edition from Within an IDE

The Java IDE Add-in Manager can integrate DevPartner Java Edition with the following IDEs:

- Compuware OptimalJ (page 83)
- Borland JBuilder (page 86)
- Eclipse (page 88)
- IBM Rational Application Developer (page 90)

Compuware OptimalJ

You can access DevPartner Java Edition features from within Compuware OptimalJ. OptimalJ enables you to profile Java applications, applets, JSPs, servlets, and EJBs.

When you profile an application with DevPartner Java Edition, OptimalJ automatically creates a DevPartner Java Edition configuration file for the module or class that you are profiling. You can edit these configuration files in DevPartner Java Edition by clicking the **Configurations** tab of the DevPartner Java Edition Start page.

You can use OptimalJ with DevPartner Java Edition to profile:

- An OptimalJ project

- A specific Java class

Profiling an OptimalJ Application

When you run DevPartner Java Edition profiling on an application from within OptimalJ, the results are displayed in the DevPartner Java Edition interface after the integrated test environment starts. All OptimalJ integrated test environment features are available while profiling your application with DevPartner Java Edition.

Note: Users of Internet Explorer should clear the option **Reuse windows for launching shortcuts** on the **Advanced** tab of the **Internet Options** dialog box, displayed through the browser's **Tools** menu. Otherwise, you might not be aware of the multiple sessions started.

DevPartner Java Edition profiling messages are displayed with the application server messages in the **Application Server** tab of the OptimalJ Output window.

OptimalJ powered by NetBeans

To profile an application in OptimalJ powered by NetBeans, do one of the following:

- From the OptimalJ menu bar, choose **Debug>Start Application Server under DevPartner**; then select the desired analysis type from the submenu:
  - **Analyze Memory Usage**
  - **Analyze Performance**
  - **Analyze Coverage**
- In the Code Model explorer, locate the file for your application. Right-click the file to display the popup menu, choose **Tools>Start Application Server under DevPartner**, and select the desired analysis from the submenu.

OptimalJ built on Eclipse

To profile an application in OptimalJ built on Eclipse, you must be in the Application perspective. To begin the profiling, choose **Test>Start Application Server under DevPartner**; then select the desired analysis type from the submenu:

- **Analyze Memory Usage**
- **Analyze Performance**
- **Analyze Coverage**

Profiling a Specific Application Module

It is not possible to deploy both the EJB and Web modules in the integrated test environment and restrict the DevPartner Java Edition profiling to an individual application module. DevPartner Java Edition profiles the entire process running your test application server and any application modules deployed to it during that test session. For example, if you start the integrated test environment with DevPartner Java Edition profiling enabled and deploy just your application's EJB module, the profile information covers the running application server and the deployed EJB module only. If you subsequently deploy the application's Web module to the existing test session, it will be added to the profiling information from that moment forward.

Note: If you are testing on JBoss and configured OptimalJ to use the Tomcat installation provided with the NetBeans IDE, you will not be able to profile the Web module for your application because it runs in a separate process.

To profile a specific application module:

1. In the Code Model view, locate the archive file for the application module you want to profile.

   Examples of application archives include the following:

   ejbModuleName`Ejb.jar`

   webModuleName`Web.war`

2. Right-click the file to display the popup menu, then choose the command:

   - ◇ OptimalJ powered by NetBeans — **Tools>Start Application Server under DevPartner**
   - ◇ OptimalJ built under Eclipse — **Test>Start Application Server under DevPartner**

   Then select the desired analysis from the submenu.

The integrated test environment starts and the DevPartner Java Edition profile window appears. All OptimalJ integrated test environment features are available while profiling your application with DevPartner Java Edition.

DevPartner Java Edition profiling messages appear with the application server messages in the Output window.

Profiling a Specific Java Class

If your application contains a Java class that requires only the Java Runtime Environment to execute, you can enable DevPartner Java Edition profiling in the class. OptimalJ displays DevPartner Java Edition profiling status in the **DevPartner** tab of the Output window. The message “Class is profiled” indicates that the class has been executed and profiling is enabled.

OptimalJ powered by NetBeans

To profile a class in OptimalJ powered by NetBeans:

1. Compile the code for your application.
2. In the Code Model explorer, select the Java class to profile.
3. From the menu bar, choose **Debug>DevPartner**, then select the desired analysis type from the submenu:

   - ◇ **Analyze Memory Usage**
   - ◇ **Analyze Performance**
   - ◇ **Analyze Coverage**.

4. OptimalJ enables the DevPartner Java Edition profiling and executes the selected class in the Java Runtime Environment.

OptimalJ built on Eclipse

To profile a class in OptimalJ built on Eclipse, right-click the class in the Code Model view, then do one of the following:

- Choose **Run As** from the context menu, then select the desired analysis type from the submenu:
  - **DevPartner Coverage Analysis**
  - **DevPartner Memory Analysis**
  - **DevPartner Performance Analysis**
- Choose DevPartner Java from the context menu, then select the desired analysis type from the submenu:
  - **Coverage Analysis**
  - **Memory Analysis**
  - **Performance Analysis**

Borland JBuilder

DevPartner Java Edition does not support JBuilder in the IDE Add-in Manager. To use JBbuilder, you must manually configure the integration.

Manual integration with JBuilder 2008

Add the -Xrun or -agentlib argument to the JVM Arguments section in JBuilder in order to invoke DevPartner Java Edition from within JBuilder.

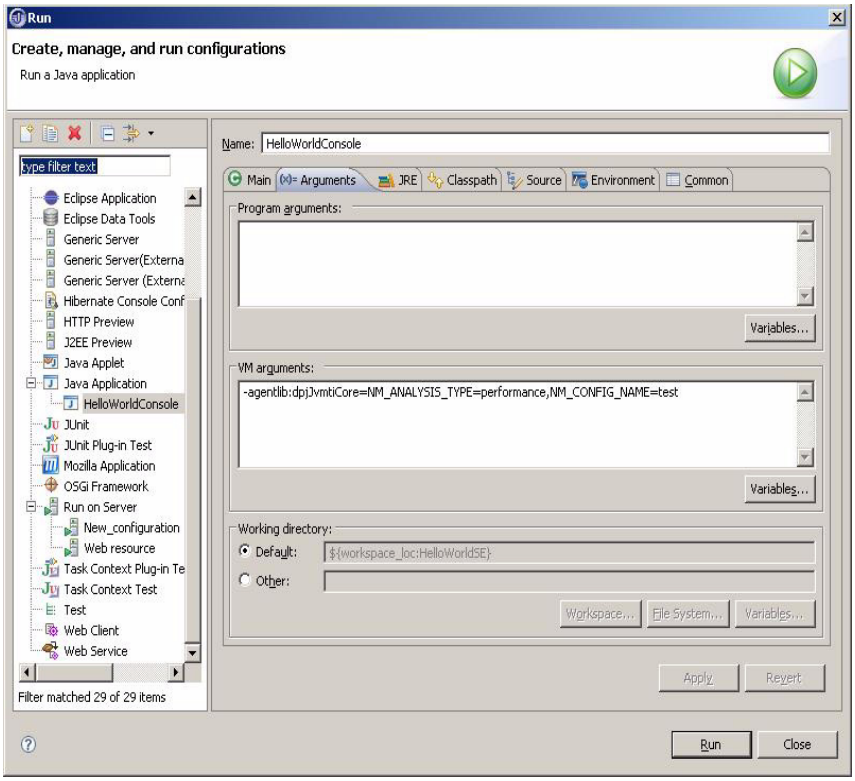
Configuring the Application

To configure the application:

1. Select Run>Open Run Dialog… from the menu.
2. In the Create, manage and run configurations dialog box, create a new launch configuration.
3. Click the Arguments tab.
4. Do one of the following:
   - If you are using JVMPI (JDK 5.0 or below), add the -Xrun argument to the VM Arguments field, as described in **Using -Xrun to Invoke the Profiler** in the online help. A sample string is listed below:

     -XrundpjCore:NM_ANALYSIS_TYPE=performance:NM_CONFIG_NAME=test
   - If you are using JVMTI (JDK 6.0 or above), add the -agentlib argument to the VM Arguments field, as described in **Using -agentlib to Invoke the Profiler** in the online help. A sample string is listed below:

```
-
agentlib:dpjJvmtiCore=NM_ANALYSIS_TYPE=performance,N
M_CONFIG_NAME=test
```



Configuring the Server

To configure the server:

You will need to modify the server configuration profile used when launching the Application server or create a configuration profile especially designed for this purpose.

1. Select Run>Open Run Dialog… from the drop down menu.
2. Select the target server configuration created for this application.
3. Click the Arguments tab.
4. In the dialog box do one of the following:

   ◇ If you are using JVMPI (JDK 5.0 or below), add the -Xrun argument to the VM parameters text box, as described in Using -Xrun to Invoke the Profiler in the online help. A sample string is listed below.

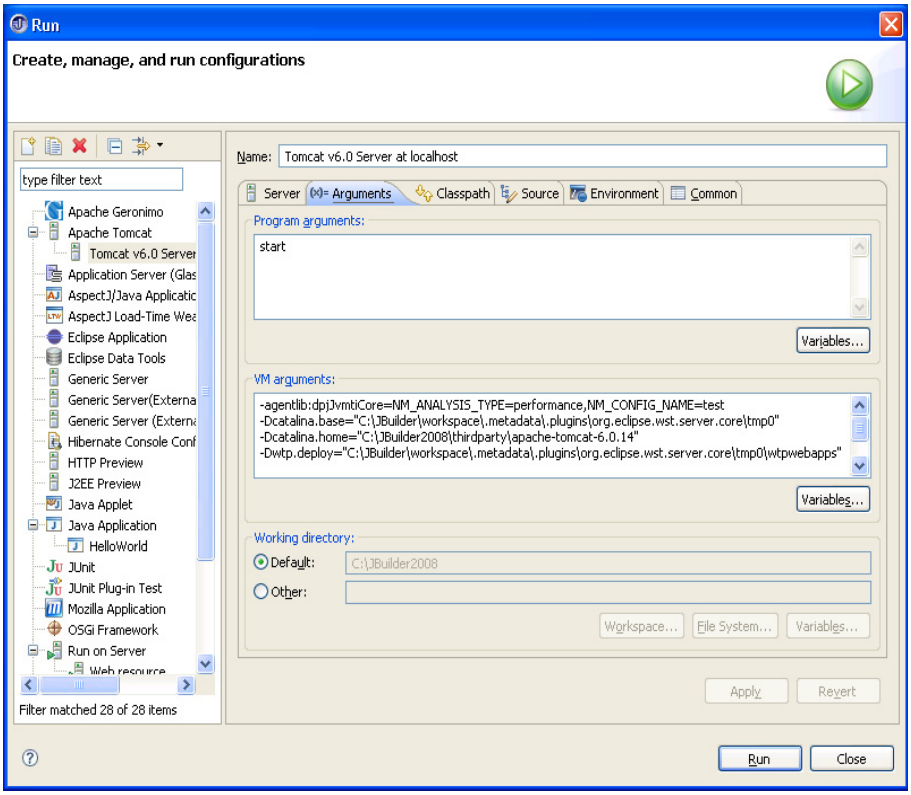
   ```
   -XrundpjCore:NM_ANALYSIS_TYPE=performance:NM_CONFIG_NAME=test
   ```

Note: This is one string without a newline character inserted. This string must be inserted before any other arguments in this box.

   ◇ If you are using JVMTI (JDK 6.0 or above), add the -agentlib argument to the VM parameters, as described in Using -agentlib to Invoke the Profiler in the online help. A sample string is listed below.

```
-
agentlib:dpjJvmtiCore=NM_ANALYSIS_TYPE=performance,NM_CONFIG_NA
ME=test
```

Note: This is one string without a newline character inserted. Also note the comma delimiter prior to NM_CONFIG_NAME.This string must be inserted before any other arguments in this box.



Eclipse

You can run coverage, memory, and performance analyses from within Eclipse.

From the Package Explorer

In the Package Explorer, you can select an object to analyze in DevPartner Java Edition.

1. In the Package Explorer, right-click the object you want to analyze and select **DevPartner Java** from the menu.
2. From the submenu, select the type of analysis you want to perform.

The selected object runs under DevPartner Java Edition, and the results are displayed in the DevPartner Java Edition interface.

From the Run Menu or Toolbar

Run Command

You can execute the **Run** command from the menu or toolbar to run a DevPartner Java Edition analysis.

1. In the Package Explorer, select the object you want to analyze.
2. Select **Run** from the **Run** menu, or click **Run**. The **Run Configuration** dialog box appears.
3. In the **Configurations** list, select **DevPartner Java Applet**, **DevPartner Java Application**, or **DevPartner Java Eclipse Application**, as appropriate, then click **New**. The dialog box changes to display the **Name** field and tabs containing configuration options. In addition to the standard Eclipse configuration tabs, the dialog box includes the **Analysis** tab in which you select the type of DevPartner Java Edition analysis you want to run.
4. In the tabs, select options as needed to define the configuration.
5. If desired, change the default configuration name in the **Name** field. The name you specify will also be the name of the configuration in DevPartner Java Edition.
6. After defining the configuration, click **Run**.

When the analysis runs, the results are displayed in the DevPartner Java Edition interface.

Run As Command

From the **Run** menu, you can also select the **Run As** command, and then select a DevPartner Java Edition analysis from the submenu.

Starting JBoss, WebLogic, and Tomcat with Eclipse WTP

There are three ways to start JBoss, WebLogic, and Tomcat using Eclipse WTP. For first time use, the Select Preferred Launcher dialog box appears. Select **Override workspace settings** and **DPJ Launcher** and click **OK**.

From WTP Server View:

1. From the Servers tab, Right-click JBoss, WebLogic, or Tomcat and choose Profile. If this is a first time run, the Select Preferred Launcher dialog box appears. Otherwise, the DevPartner Java Plug-in dialog box appears.
2. Select the desired analysis type and click **OK**.

From Profile on Server:

1. From the Package Explorer tab, right-click the package and choose **Profile As>Profile on Server**. The Profile on Server dialog box appears.
2. Choose JBoss, WebLogic, or Tomcat from the server list and click Next. The Add and Remove Projects dialog box appears.
3. Add or remove projects as necessary and click Finish. The DevPartner Java Plug-in dialog box appears.
4. Select the desired analysis type and click **OK**.

From the Profile Launch Configuration:

1. From the Package Explorer tab, right-click the package and choose P**rofile As>Open Profile Dialog**. The Profile dialog box appears.
2. Choose JBoss, WebLogic, or Tomcat from the server list and click **Profile**. If this is a first time run, the Select Preferred Launcher dialog box appears. Otherwise, the DevPartner Java Plug-in dialog box appears.
3. Select the desired analysis type and click **OK**.

Uninstalling DevPartner Java Edition from Eclipse

If you need to uninstall Eclipse, make sure you uninstall the DevPartner Java Edition plug-in before uninstalling Eclipse. Use the Java IDE Uninstallation utility to uninstall DevPartner Java Edition.

IBM Rational Application Developer

When you integrate DevPartner Java Edition into IBM Rational Application Developer (RAD), you can profile Java applications, Java Beans, applets, JSPs, servlets, and EJBs from within the RAD interface.

When DevPartner Java Edition is integrated with Rational Application Developer, it is attached to the **Run** menu, which is accessible from multiple perspectives.

As shown in Figure 5-2, the Rational Application Developer **Run** menu is opened from the **Run** menu item. When you select that menu option, a **Run** dialog box appears. The **DevPartner** choices, identified by the DevPartner Java Edition product icon, appear at the top of the list. Choose one of the analysis types to launch the currently selected target under DevPartner Java Edition.

Figure 5-2. DevPartner Java Edition in Rational Application Developer

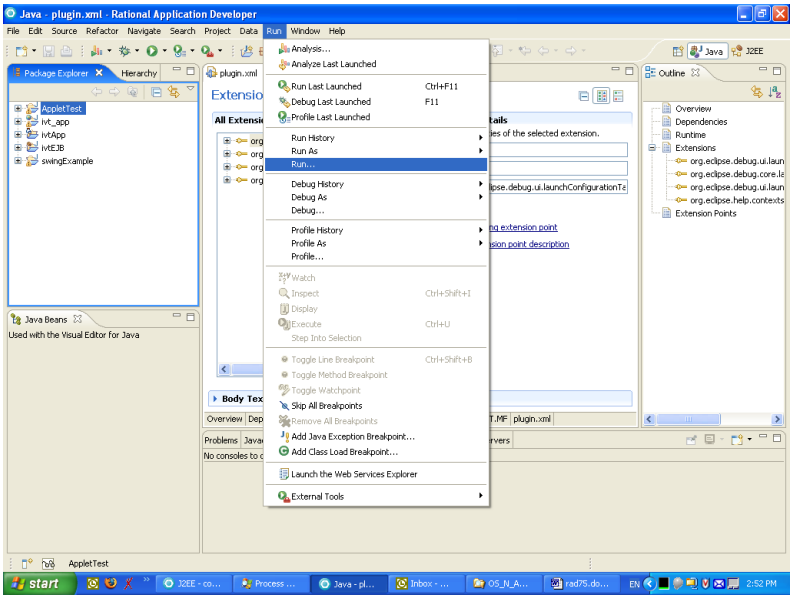
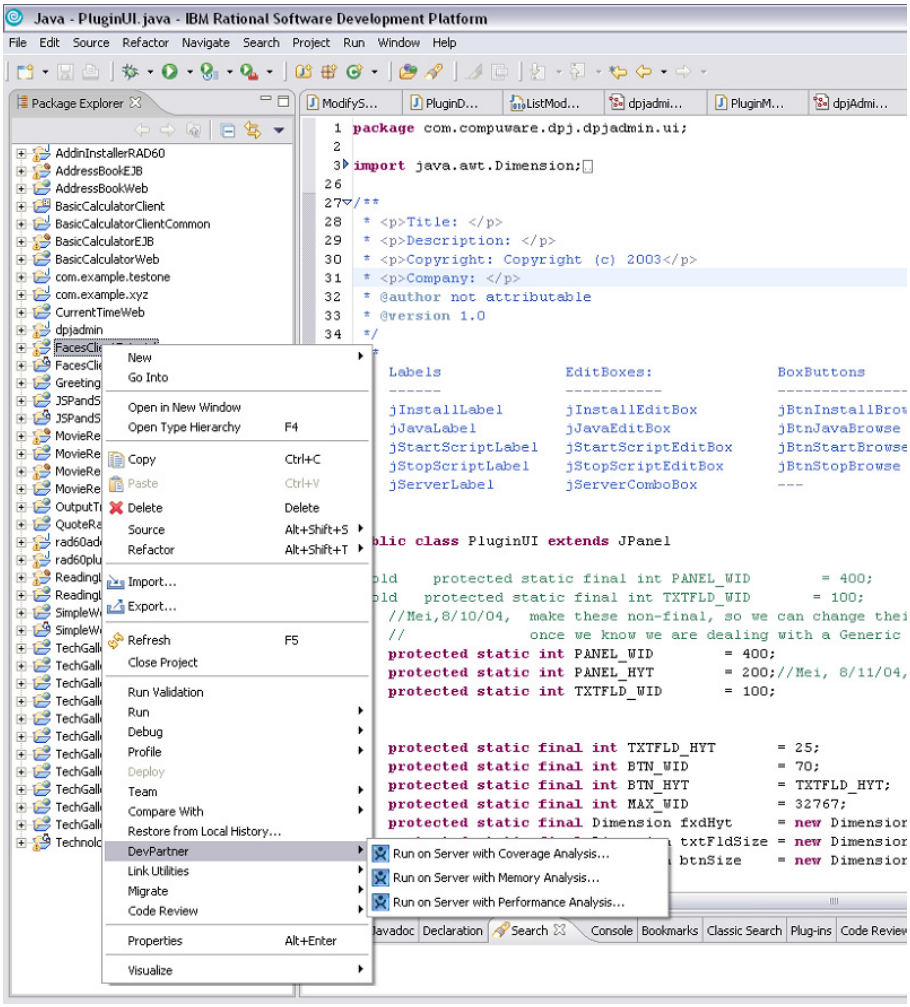


Figure 5-3 shows another means by which an application can be launched under DevPartner Java Edition: open the Web or J2EE perspective, select the index HTML page of a web application by right-clicking it, and scroll down to choose the **DevPartner** item from the shortcut menu. Choosing one of the analysis types launches the Web application. The application runs in the integrated Rational Application Server, with DevPartner Java Edition monitoring using the selected form of analysis.

Figure 5-3. Launching DevPartner Java Edition from Within RAD



For more information about using DevPartner Java Edition from within Rational Application Developer, see the online help.

Uninstalling DevPartner Java Edition from Rational Application Developer

Before you uninstall DevPartner Java Edition from Rational Application Developer, you must first delete all related Launch Configurations by doing the following:

1. Select **Run>Run...**
2. Select a DevPartner Java Edition-related Launch Configuration and click **Delete**.
3. Click **Yes** in the **Confirm Launch Configuration Deletion** message box.
4. Repeat steps 2 and 3 for each DevPartner Java Edition-related Launch Configuration.

Chapter 6

Working with Application Servers

DevPartner Java Edition can monitor your Java code as it executes in an application server as easily as it can profile a standalone Java application. It is compatible with a number of the most popular application servers, and configuring DevPartner Java Edition for a server is quick and easy. As in other profiling scenarios, you do not have to modify your code to profile it with DevPartner Java Edition.

For the list of supported application servers, refer to the “Supported Environments” topic in the online help.

If you did not create a configuration for your application server when you installed DevPartner Java Edition, you can use the Administration Console to perform this task at any time afterward. If your application server is not listed as supported, you can create a generic configuration for it. Note, however, that support is not guaranteed if the application server is not in the list.

For more information, see the online help in the DevPartner Java Edition Administration Console.

Running Application Servers Through DevPartner Java Edition

When you have created a configuration for an application server, you can control the profiling of Java code within that application server through either the DevPartner Java Edition command line utilities or the user interface.

From the Command Line

The DevPartner Java Edition command line utility **`nmserver`** enables you to start, attach, kill, and detach from an application server, using the appropriate parameters. To see a list of the utility's parameters, execute `nmserver -help`.

Many of the command line parameters are self-explanatory. For example, `nmserver -kill` terminates an executing application server. Executing `nmserver -detach` detaches the current profiling session from the application server, but leaves the server running.

Executing `nmserver -attach` attaches a DevPartner Java Edition profiling session to an application server. If the application server is not executing, DevPartner Java Edition starts it, then hooks itself to it. If the application server is already executing, the `-attach` parameter “hooks” DevPartner Java Edition to the application server.

Note: You can use the `-attach` parameter only with an application server that was started through DevPartner Java Edition.

In addition to controlling the execution state of the application server, you can define which analysis type DevPartner Java Edition performs, as well as which configuration to use for the session. To specify the analysis type, use the appropriate switch: `-mem`, `-cov`, or `-perf`. Specify the configuration with the `-config` switch, which has the format `-config` ***`configuration-name`*** (where ***`configuration-name`*** is the name of an existing or new configuration).

For example, the command line

```
nmserver -attach -config test -mem rg:Tomcat
```

launches the Apache Tomcat server under a DevPartner Java Edition Memory analysis session with a configuration named `test`.

Note: If you do not specify a configuration with the `-config` switch, DevPartner Java Edition creates a configuration with a name based on the name of the specified application server. For example, the command line `nmserver -attach -mem rg:Tomcat` creates a configuration named **`rg_Tomcat`**, with default parameters.

The full range of command line options for the **`nmserver`** utility are described in detail in the online help.

From the DevPartner Java Edition Start Page

You can also control application servers via the **Application Server Testing** tab of the DevPartner Java Edition Start page.

When you create an application server configuration through the Administration Console, an entry for that application server appears on the **Application Server Testing** tab (see Figure 6-1.) In this window is a table that displays the status of all application servers for which DevPartner Java Edition configurations have been created. The table includes buttons for viewing, detaching, and stopping each application server.

Figure 6-1. DevPartner Java Edition Application Server Testing Tab

Above the table are lists from which you select the session configuration and the analysis type.

The help text below the table describes the steps for starting an application server:

1. From the **Configuration** list, select a configuration.
2. From the **Analysis Type** list, select **Performance**, **Memory Analysis**, or **Coverage**.
3. Select the application server by clicking the corresponding **Start** button.

DevPartner Java Edition launches the chosen application server with the specified parameters, and displays the Session Control page. The format of the Session Control screen varies depending on the analysis type. (For details, see the online help.)

While a server is executing under DevPartner Java Edition, you can view intermediate results, clear the data collected so far in the session, detach from and reattach to the server, and so on.

In the **Application Server Testing** tab, the buttons for those application servers that you have started are active:

- Click **View** to return to the Session Control page for that particular application server.
- Click **Detach** to detach DevPartner Java Edition data collection from the application server. The application server continues running.
- When you click **Detach**, the **Attach** button becomes available. Click this button to start a new profiling session for the application server.
- Click **Stop** to terminate the application server, which also terminates the current session.

Selecting **Detach** or **Stop** for an application server running under DevPartner Java Edition analysis generates a session file.

Note: If your application server is BEA WebLogic or Oracle OC4J Standalone and you started the server through the **Application Server Testing** tab of the Start page, then stopping the application server by using the **Stop** button causes an abnormal termination and a session file named **`AbnormalTerminationx`** (where *x* is an incremental number) is created.

You can generate an accurate session file for these application servers by doing either of the following:

- In the **Application Server Testing** tab, use **Detach** rather than **Stop** to end the profiling session. When you use **Detach**, the application server continues to run after you end the session.
- Stop the server from outside DevPartner Java Edition by using the server console or running a script.

There is no functional difference between starting an application server using `nmserver` and starting the server from the **Application Server Testing** tab. Both methods use the same techniques for controlling the application server.

If you start an application server from the command line, using `nmserver` with the `-batch` command line switch, then launch the DevPartner Java Edition interface, you can manage the session as though you had launched it from the **Application Server Testing** tab. If you launched the application server from the **Application Server Testing** tab, you can detach from it by executing `nmserver -detach`.

Including and Excluding Code for Profiling

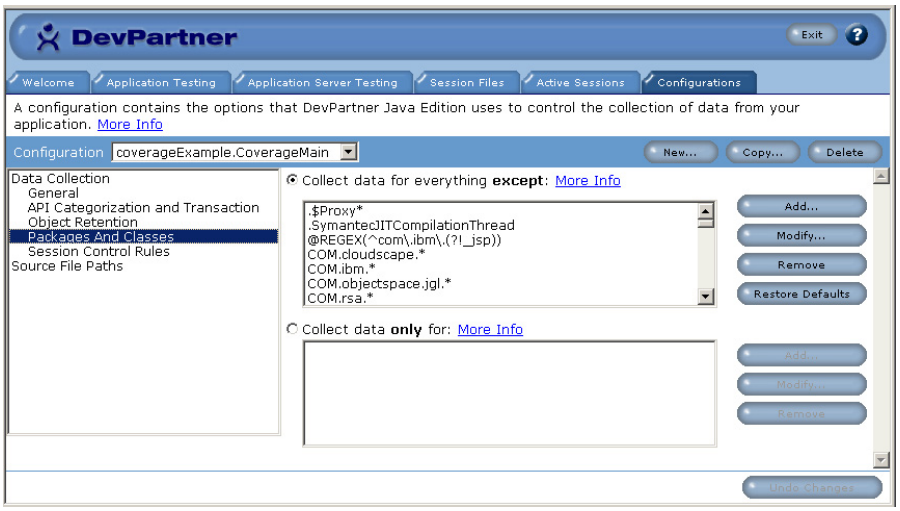
Because application servers are largely written in Java, when you run a servlet, JSP, or EJB in an application server, Java code executes that is not part of your application. The JVM, however, does not distinguish code that is part of the application server from code that is part of your application.

DevPartner Java Edition enables you to exclude the server code from the analysis through the **Packages and Classes** section of the **Configurations** tab.

As shown in Figure 6-2, this section offers two methods of specifying which code to profile:

- **Collect data for everything except** – Specify packages to *exclude* from profiling, in which case all other packages and their constituent classes are profiled.
- **Collect data only for** – Specify the packages and classes to profile, in which case all other packages and classes are excluded.

Figure 6-2. Package and Classes Section of the Configurations Tab



When you create a new configuration, that configuration's exclusion list is automatically populated with a default set of packages. The packages have been chosen so that the collected session data will exclude the most likely packages and classes that comprise the runtime of most frequently used libraries, application servers, and IDEs.

You can modify the contents of the inclusion and exclusion lists using the **Add**, **Modify**, and **Remove** buttons that appear to the right of each list box. The **Restore Defaults** button returns both lists to their states at the time the configuration was created. For more details, see the online help.

Note: You can use Perl-compatible regular expressions in the Exclusion and Inclusion lists. See the online help for details.

Flexible Profiling

Profiling Java components in an application server is extremely flexible. You can, for example, simultaneously profile multiple application servers from a single instance of DevPartner Java Edition — and each profiling session can be executing a different analysis type.

You can profile Java components executing in an application server running on a remote machine. This capability is useful if you need to profile an enterprise application in an application server running on a staging system that is separate from your development system.

Note: DevPartner Java Edition's remote capabilities are not limited to working with application servers. For more information about remote profiling, see the online help.

Index

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W