Leading the Evolution™

DevPartner Java Edition

User's Guide

Release 4.5

20091014135434

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 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, "Introduction to DevPartner Java Edition" — Overview of DevPartner Java Edition architecture, system requirements, and licensing.
- Chapter 2, "Getting Started" — High-level descriptions of DevPartner Java Edition features and interface.
- Chapter 3, "Command Line Utilities" — Descriptions of and syntax for the command line utilities.
- Chapter 4, "Configurations" — Details of the configuration options for profiling.
- Chapter 5, "Sessions" — Information for using the Session Control pages and session files.
- Chapter 6, "Memory Analysis" — Details for running a Memory analysis and viewing the results.
- Chapter 7, "Performance Analysis" — Details for running a Performance analysis and viewing the results.
- Chapter 8, "Coverage Analysis" — Details for running Code Coverage analysis and using the session files.

- Chapter 9, “IDE Integration” — Instructions for integrating DevPartner Java Edition into supported IDEs.
- Chapter 10, “Sample Applications” — Instructions for running sample applications that demonstrate DevPartner Java Edition features.

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 of your product 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

Introduction to DevPartner Java Edition

DevPartner Java Edition is a comprehensive suite of software development productivity features that help developers build reliable, high-performance applications and components using Java technology. DevPartner Java Edition adapts easily to multi-tier, e-business solutions which combine diverse technologies that could easily sacrifice run-time performance, optimized memory utilization, and adequate code coverage. Developers can more quickly resolve a wide variety of issues, producing more robust Java applications.

DevPartner Java Edition can perform:

- Memory analysis — Detect memory leaks, overall RAM footprint, and use of temporary objects.
- Performance analysis — Identify bottlenecks in your code.
- Coverage analysis — Ensure that your code is thoroughly tested.

Use DevPartner Java Edition to gain an understanding of the memory usage, performance, and test coverage of your:

- Java 2 Standard Edition (J2SE) programs, including applications, 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 Architecture

DevPartner Java Edition consists of:

- A small control service (NCSP) that coordinates communication between the monitored program, the DevPartner Java Edition Web server, and the application under test.
- A Web server, which creates and maintains session files and configuration files. You access the Web server with a Web browser.
- A command line interface, which lets you begin a profiling session by running your Java programs from the command shell.

The Web server, the control service, and all applications being tested must be on the same computer.

If DevPartner Java Edition is installed using concurrent licensing, multiple users can access the Web server from browsers on computers other than the DevPartner Java Edition Web Server computer.

DevPartner Java Edition can be integrated with some Java IDEs, so you can begin a profiling session by running your Java program from your IDE.

How DevPartner Java Edition Profiles Code

To profile your code, DevPartner Java Edition modifies the byte code for each class as it is loaded by the Java Virtual Machine (JVM). This enables DevPartner Java Edition to collect profiling data while your application is running. It does not change the actual **`.class`** file on disk; it only modifies the in-memory representation that the JVM holds.

When your Java applet or application runs with DevPartner Java Edition, each class is instrumented as it is loaded into the JVM. As methods of the class are executed, DevPartner Java Edition collects profiling information. In addition, the Java Virtual Machine Profiling Interface (JVMPI) is used to collect profiling information.

Improving Program Understanding

After you have run your code using DevPartner Java Edition, created a session file, and viewed the Results Summary, your attention will be drawn to the areas of your code that might consume excess memory, might be bottlenecks, or might not be covered in your tests.

You can go directly to the source to understand the problem (clicking on an object generally provides the **View Source** option), but it is often useful to understand the relationship of an object to other objects before attempting to correct a problem.

DevPartner Java Edition enables you to better understand your program in several ways, including the following three useful tools:

- Call Graph — Shows the parent/child relationships of method calls
- Allocation Trace Graph — Shows the chain of calls leading to allocation of memory for an object
- Object Reference Graph — Shows which objects are holding a reference to an object

Multiple Processes

An application can run in the context of multiple processes in a variety of situations such as the following:

- A programmer explicitly creates several processes.
- An application is distributed (such as a Java client exchanging data with a Java server).
- An application server creates multiple processes automatically.

It is often useful to create a single session file containing data from all processes to get a complete picture of your application's performance or test coverage.

DevPartner Java Edition enables you to gather the data from multiple processes into one session file with a single setting in the Configuration file.

Consider the following when running multiple processes:

- You cannot mix Coverage and Performance processes. You can only correlate multiple processes running the same profiling type and configuration. Each analysis type generates separate session files.
- The name of a session file that is generated from multiple processes is `Correlated_`***`number`***, where ***`_number`*** is incremented each time a session file is generated.

About Entry Points and Transactions

Some code in your application is beyond your control. The packages and classes that comprise an application server or an IDE, low-level database drivers, and graphics libraries are all examples of code that becomes part of your application, but that you will generally not want to profile. DevPartner Java Edition lets you control which packages and classes are included in (or excluded from) analysis.

By default, DevPartner Java Edition excludes Java, application server, and IDE classes from monitoring. (You can modify this list in the configuration file.)

An *entry point* is a profiled method that is called by excluded code. For example, each Java Server Page (JSP) would be considered an entry point. A small program might have only a single entry point (**`Main`**). Methods that are called only by other profiled methods, however, are not entry points.

When your program runs, monitoring begins with the first call to a method not on your exclusion list. This is the *user-code entry point.* All calls made from that point until the return to excluded code are part of that entry point.

When you analyze your data, DevPartner Java Edition organizes the data by entry point so you immediately know where to focus your tuning efforts.

Beyond Entry Points

To enable you to focus your analysis on a specific area of your code, DevPartner Java Edition provides three ways for you to manually determine when monitoring should begin:

- For Memory Leak analysis, you manually control the point at which monitoring begins without regard for entry points by using the **Start Tracking** feature on the Session Control page.
- Using Session Control Rules, you can begin monitoring at a specific point.
- In the **Configurations** tab, select **API Categorization and Transaction**, then select the option to use entry point tracking.

Sharing the DevPartner Java Edition Server

With the DevPartner Java Edition client, you can access remote computers running the server, to start and stop application servers, and you can profile applications on application servers. For more information, see “Starting Through an Application Server” on page 24.

Several users can use the same DevPartner Java Edition server at the same time. The following information describes how this can happen:

- Several users can profile simultaneously their applications on the remote server.
- Several users can act on the same active session simultaneously. All changes made by one user will automatically be reflected to all other users.
- Only one application can be profiled at a time per application server. Users can check the **Application Server Testing** tab on the Start page. For more information, see "Application Server Status" on page 25.

Each computer running DevPartner Java Edition must have a Java plug-in installed. Optionally, administrators of the server can provide these required plug-ins for remote users who need them. For more information, see "Providing Java Plug-ins for Remote Users" on page 45.

Understanding Java Platform Performance

The DevPartner Java Edition help system includes links to the online version of the book *Java Platform Performance Strategies and Tactics* by Steve Wilson and Jeff Kesselman. This online book is available through the Sun Microsystems Web site (http://java.sun.com).

Java Platform Performance Strategies and Tactics provides the reader with a solid understanding of performance tuning, including both high-level strategies and code-level performance tuning tactics.

Licensing

DevPartner Java Edition can be used locally on a single computer, locally on multiple computers, or remotely across networks. It can be installed for use by a single user or by multiple users using concurrent licensing.

License Types and Features

A node-locked license enables use of DevPartner Java Edition on a single computer.

A server license enables a user with remote access to the DevPartner Java Edition server computer (through a browser) to create or view session files, and to create or use configuration files on the DevPartner Java Edition server. The server license may be for a single server, or concurrent.

Note: If using DevPartner Java Edition through remote access, you must have both a `DevPartnerJava` license and a `DevPartnerJavaServer` license. If your server platform is AIX or HP-UX, you must use remote access, because these platforms do not support the DevPartner Java Edition web-based interface.

Concurrent licensing requires Distributed License Manager.

For more information, see the *Distributed License Management License Installation Guide* (**`LicInst4.pdf`**), installed by default in **`C:\Program Files\Common Files\Compuware`**.

Activating a License for DevPartner Java Edition

When you install DevPartner Java Edition, you have the option to install a valid user license and server license or use an evaluation license. The evaluation license is active for 14 days.

After DevPartner Java Edition is installed, you can update your license as needed.

If you want remote access to the DevPartner Java Edition server, you must install a DevPartner Java Edition server license. See the *DevPartner Java Edition Installation Guide* for information on using server licensing.

Note: The server evaluation license is automatically installed as part of the DevPartner Java Edition installation process, even if you select a user license during the installation procedure.

Using a Server License

To access DevPartner Java Edition from a remote Web browser, enter

`http://`***`DPmachinename`***`:21578/ui`

where ***`DPmachinename`*** is the name of the computer on which DevPartner Java Edition resides.

Changing the DevPartner Java Edition Web Server Port

The DevPartner Java Edition control service (NCSP), which controls communications between the Web server, the application under test, and your browser, takes requests on port **21578** and passes them to the Web server on port **21580** (the Web server also uses port **21579** for auxiliary communications).

You must open ports 21578 and 21580 in your firewall if you want to use DevPartner Java Edition through the firewall.

If you need to use different ports, you must make changes to certain files that reside in the DevPartner installation folder.

Note: You should select port numbers that are *greater than* 21000. Although not required, you should allocate three consecutive ports for NCSP, the Web Server, and the Web Server auxiliary port.

To illustrate each step below, the following example port numbers are used:

- The NCSP port, defined in the `NCSP_PORT` field = `41000`
- The WebServer (Tomcat in this example) port, defined in the `TOMCAT_PORT` field = `41002`
- The WebServer (Tomcat in this example) auxiliary port, defined in the `TOMCAT_AUX_Port` field = `41001`

The location of the **`var`** folder depends on the operating system:

- Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\conf`**

Note: 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**.

- Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\conf`**

- UNIX — ***DPJ_dir*/var/conf**
  where ***DPJ_dir*** is the path of the DevPartner Java Edition product folder

Change the DevPartner Java Edition configurations as needed:

1. In the file **/var/conf/DPJ.conf**, edit the values for `NCSP_PORT`, `TOMCAT_PORT`, and `TOMCAT_AUX_PORT`. For example:

   ```
   #base port number for NCS communications
   NCSP_PORT=41000
   # DPJ UI server ports must match the port designations
   # in the server.xml file
   TOMCAT_PORT=41002
   TOMCAT_AUX_PORT=41001
   ```

2. In the file **/var/conf/DPJServer.cmd**, change the line containing the `com.compuware.dpj.ncsPort` definition to the new port definition for NCSP. For example:

   ```
   -Dcom.compuware.dpj.ncsPort=41000
   ```

3. In the file ***DPJ_dir*/tomcat/conf/server.xml**, the port designation for the Web server port (`TOMCAT_PORT` in this example) is defined in the `<Connector...>` tag. Replace the old port designation for `TOMCAT_PORT` with the new port designation. For example:

   ```
   <Connector className="org.apache.catalina.connector.http.HttpCon-
   nector" port="41002" minProcessors="5" maxProcessors="75"
   inet="localhost" enableLookups="true" redirectPort="8443" accept-
   Count="10" debug="0" connectionTimeout="60000"/>
   ```

   The port designation for the Web server auxiliary port (`TOMCAT_AUX_PORT` in this example) is defined in the `<Server...>` tag before the service description. Replace the old port designation for `TOMCAT_AUX_PORT` with the new port designation. For example:

   ```
   <Server port="41001" shutdown="SHUTDOWN" debug="0">
   ```

When you have completed these steps, stop all profiling and restart the NCSP service for the changes to take effect.

Chapter 2

Getting Started

This chapter contains high-level descriptions of DevPartner Java Edition features and its user interface.

Configuring Application Servers

Before DevPartner Java Edition can monitor code executing on an application server, you must configure the application server.

Using the Administration Console

If your application server is the Administration Console, use the console to modify the list of application servers that DevPartner Java Edition is able to monitor. You can add application servers to the list, remove application servers, and modify the configuration details for each application server entry.

Once an application server has been added to the list (and its configuration details properly set), you can start that application server from the **Application Server Testing** tab on the Start page to monitor your code as it executes on that application server. You can also use the **`nmserver`** command from a command line to execute the application server directly for monitoring your code with DevPartner Java Edition.

You can define application server configurations for more than one of a particular type of application server within the Administration Console. The console appends an incremental integer at the end of the name of additional application servers of identical type. For example, if your user name is **rg** and you have more than one Tomcat installation, the first name assigned by the Administration Console might be `rg:Tomcat`, the next would be `rg:Tomcat_1`, then `rg:Tomcat_2`, and so on.

For additional information, refer to the online help in the DevPartner Java Edition Administration Console.

Invoking the Profiler Through Your Application Server

You can invoke the profiler through your application server by adding an argument to the server's JVM settings. For more information, see “Invoking the Profiler Through the JVM Settings” on page 21.

Up and Running in 60 Seconds

If you can run your program, you can use DevPartner Java Edition to profile your code.

Determine how your code will start:

- From a command line or batch file — See “Starting from a Command Line” on page 20.
- Through an application that is already running — See “Application Testing” on page 23.
- Through an application server — See “Starting Through an Application Server” on page 24.
- Through an Integrated Development Environment (IDE) — See Chapter 9, “IDE Integration”.

You can profile code using the DevPartner Java Edition API or Session Control Rules. You can also profile applets using the **`nmappletviewer`** command, or directly in the browser.

Starting from a Command Line

DevPartner Java Edition includes four utilities that enable you to profile your program from the command line. For example, **`nmjava`** can be used as a standalone replacement of **`java.exe`**.

If you normally type:

```
$ java com.mycompany.Main args
```

instead, type:

```
$ nmjava option [-batch] com.mycompany.Main args
```

where ***`option`*** is with one of the following:

Option	Description
-perf	Performs computational Performance analysis (the default).
-mem	Performs Memory analysis: Memory Leak, Object Retention, Temporary Object, and RAM Footprint.
-cov	Performs Coverage analysis.

If you specify `-batch` in the command line, DevPartner Java Edition runs in batch mode and does not invoke the Web interface.

For a detailed description of the command line utilities, see Chapter 3, “Command Line Utilities”.

The following is an overview of the profiling process when you use one of the command line utilities to launch your program from a command line:

1. Specify a Configuration in the command line if you want to use a predefined configuration.

   A *configuration* specifies the level of detail to be gathered, rules to be executed during the session, and so on.

   If you do not specify a configuration, a new configuration is created for you. This new configuration uses the default configuration settings.

   To specify a configuration, include a line similar to the following in the list of parameters for your command line, where ***`Configuration_name`*** is the name of your configuration:

   ```
   -config Configuration_name
   ```

2. Specify an analysis type; or use the default, which is Performance analysis.

3. Include `-batch` to profile in batch mode (that is, without displaying the DevPartner Java Edition interface).

4. When your program starts, DevPartner Java Edition begins a session and displays a Session Control page while the data is being collected (unless you are running in batch mode). When the program terminates or when you take a snapshot, DevPartner Java Edition saves data to a session file and displays analysis results.

Specify all Java options after all DevPartner Java Edition options on the command line.

Note: A separate utility, `dpj`, is also available to start the DevPartner Java Edition Web interface. You cannot start the user interface on AIX with the `dpj` utility. Access the DevPartner Java Edition server through one of the supported browsers on a computer that is running one of the other supported platforms.

You can also test your application from a supported IDE.

Manually Invoking the Profiler

Invoking the Profiler Through the JVM Settings

You can invoke the profiler by adding an argument to the JVM settings.

◆ If you are using JDK 5.0 or below with JVMPI, use the `-Xrun` parameter. For more information, see “Using -Xrun to Invoke the Profiler” on page 22.

◆ If you are using JDK 6.0 or above with JVMTI, use the `-agentlib` parameter. For more information, see “Using -agentlib to Invoke the Profiler” on page 22.

Note: JVMPI is deprecated as of JDK 5.0. JDK 6.0 and above will have only JVMTI.

You can specify the default profiler as JVMPI or JVMTI by changing the setting in the **`DPJ.conf`** file. For more information, see “Specifying the Default Profiler” on page 23.

Using -Xrun to Invoke the Profiler

Note: If you are using DevPartner Java Edition with JVMTI (JDK 6.0 and above), use the `-agentlib` parameter. For more information, see "Using -agentlib to Invoke the Profiler" on page 22. As of JDK 5.0, JVMPI is deprecated. JDK 6.0 and above will have only JVMTI. You can specify the default profiler as JVMPI or JVMTI by changing the setting in the **`DPJ.conf`** file. For more information, see "Specifying the Default Profiler" on page 23.

If you are using DevPartner Java Edition with JVMPI, you can invoke the profiler by adding the following argument (as one line) to the JVM settings:

```
-XrundpjCore:NM_ANALYSIS_TYPE={coverage,perfor-
mance,memory}:NM_CONFIG_NAME={name of DPJ Configuration}:NM_BATCH={1}
```

- `NM_ANALYSIS_TYPE` is the type of profiling session to launch. This field is mandatory. One of the three values — `coverage`, `performance`, or `memory` — must be supplied. You can change the analysis type after launch by using the detach/reattach feature in the **Application Testing** tab of the DevPartner Java Edition Start page.
- `NM_CONFIG_NAME` is the name of the profiling configuration to use in the launched session. This field is mandatory. If a configuration with the specified name does not already exist, a new configuration will be created using that name. You can change the configuration after launch by using the detach/reattach feature in the **Application Testing** tab of the DevPartner Java Edition Start page.
- `NM_BATCH` controls whether DevPartner Java Edition opens the Session Control page in your browser upon launch. This field is optional. Including `NM_BATCH=1` prevents the browser window from opening. Omitting the field or setting `NM_BATCH` to any value other than one (1) allows the browser to open the Session Control page.

For instructions for using this argument with JDeveloper, see "Oracle JDeveloper" on page 132. For other IDEs, see the product documentation.

For instructions for using this argument with application servers, see Chapter 7 of the *DevPartner Java Edition Installation Guide*.

Using -agentlib to Invoke the Profiler

Note: If using DevPartner Java Edition with JVMPI (JDK 5.0 or below), use `-Xrun` to invoke the profiler. For more information, see "Using -Xrun to Invoke the Profiler" on page 22. You can specify the default profiler as JVMPI or JVMTI by changing the setting in the **`DPJ.conf`** file. For more information, see "Specifying the Default Profiler" on page 23.

If you are using DevPartner Java Edition with JVMTI, you can invoke the profiler by adding the following argument (as one line) to the JVM settings:

```
-agentlib:dpjJvmtiCore=NM_ANALYSIS_TYPE={coverage,perfor-
mance,memory},NM_CONFIG_NAME={name of DPJ Configuration},NM_BATCH={1}
```

- `NM_ANALYSIS_TYPE` is the type of profiling session to launch. This field is mandatory. One of the three values — `coverage`, `performance`, or `memory` — must be supplied. You can change the analysis type after launch by using the detach/reattach feature in the **Application Testing** tab of the DevPartner Java Edition Start page.

- `NM_CONFIG_NAME` is the name of the profiling configuration to use in the launched session. This field is mandatory. If a configuration with the specified name does not already exist, a new configuration will be created using that name. You can change the configuration after launch by using the detach/reattach feature in the **Application Testing** tab of the DevPartner Java Edition Start page.
- `NM_BATCH` controls whether DevPartner Java Edition opens the Session Control page in your browser upon launch. This field is optional. Including `NM_BATCH=1` prevents the browser window from opening. Omitting the field or setting `NM_BATCH` to any value other than one (1) allows the browser to open the Session Control page.

For instructions for using this argument with JDeveloper, see “Oracle JDeveloper” on page 132. For other IDEs, see the product documentation.

For instructions for using this argument with application servers, see Chapter 7, “Configuring Application Servers”, in the *DevPartner Java Edition Installation Guide*.

Specifying the Default Profiler

You can specify the default profiler as JVMPI or JVMTI by commenting out the appropriate `DPJ_CORE` setting in the **`DPJ.conf`** file. This file is located in the folder **`/var/conf`**. The location of this folder depends on the operating system:

- Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\conf`**

Note: 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**.

- Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\conf`**
- UNIX — ***`DPJ_dir`*`/var/conf`**
  where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder

The `DPJ_CORE` value for JVMPI is `dpjCore`. For JVMTI, it is `dpjJvmtiCore`.

For example, to make JVMTI the default profiler, comment/uncomment the settings as follows:

```
# DPJ core base name
DPJ_CORE=dpjJvmtiCore

# DPJ core base name
# DPJ_CORE=dpjCore
```

Application Testing

To profile an application, start the application using one of the following command line utilities:

- **`nmshell`** — Java programs run in the shell
- **`nmjava`** — Standalone Java programs

- **`nmjavaw`** — Standalone Java programs (hides the output messages in the command window)
- **`nmappletviewer`** — Java applets

Note: Information about applications started under **`nmserver`** is displayed in the **Application Server Testing** tab.

While the application is running, the **Application Testing** tab of the Start page displays the following information about it:

- Application — The path of the application.
- PID — The process ID for the application
- State
  - Running under session *analysis type* — Currently being profiled; the analysis type can be Memory, Performance, or Coverage. Use **View** to display the Session Control page, or **Detach** to stop profiling.
  - Detaching — In the process of detaching from the application while the application continues to run.
  - Available — Not currently being profiled, but running and able to be reattached. Use **Attach** to begin another profiling session.
  - Attaching — In the process of reattaching to the running application.

The default refresh interval for the tab is 15 seconds. To view changes as soon as you make them, click **Refresh**.

When the application is detached, you can select a different **Configuration** or **Analysis Type** before reattaching it.

Starting Through an Application Server

Note: Before you can profile your code through an application server, you must configure the server for DevPartner Java Edition. For more information, see "Configuring Application Servers" on page 19.

If the code to be profiled runs through an application server:

1. Start the DevPartner Java Edition Web interface. You can start the Web interface with the `dpj` utility or, on Windows systems, from **Start>Programs>Micro Focus>DevPartner Java Edition>DevPartner Java Edition**.

Note: You cannot start the user interface on AIX or HP-UX with the `dpj` utility. Access the server by running one of the supported browsers from a Windows, Solaris, or Linux computer. Enter the URL **`http://`*`my_box`*`:21578/ui`**, where ***`my_box`*** is the name of your AIX or HP-UX computer.

2. From the **Application Server Testing** tab, select the application server to be profiled. (If the server is not listed, use the Administration Console to configure the server.)

3 Select the default configuration. A configuration specifies how much data is to be gathered, rules to be executed during the session, and so on. The default configuration is appropriate for general use. You can create configurations specific to your needs.

4 Select the analysis type: Performance, Memory, or Coverage.

5 Click **Start** to start the session, and then exercise your code.

  Any code run through the application server (except code excluded by the configuration) will be monitored. DevPartner Java Edition displays a Session control screen while the data is being collected.

6 When the program terminates or when you stop collection, DevPartner Java Edition saves data to a session file and displays analysis results.

For more information, see “Application Server Status” on page 25.

Notes:

- If your application server is already running, you must stop and restart that application server with DevPartner Java Edition for DevPartner Java Edition to properly hook into the application server and profile your application.
- WebLogic 9.*x* uses the JVM JRockit 5.0 by default. Because of an issue with this version of JRockit, Memory analysis cannot be performed on Java applications running under it. To perform Memory analysis, point the JVM to Sun in the WebLogic script (default location **`BEA_domain\bin\SetDomainEnv.cmd`**).
- 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 **`AbnormalTermination_`*`number`*** (where ***`_number`*** 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.

Application Server Status

To test your application, use DevPartner Java Edition to start your application server by selecting the configuration and analysis type and then clicking **Start**.

The **Application Server Testing** tab of the Start page displays the server state.

Table 2-1. Server States in the Start Page Application Tab

State	Description
Uninitialized	The application server properties are not set.
Starting	The application server is in the process of starting.
Available	The application server is running and available for profiling.

Table 2-1. (Continued)Server States in the Start Page Application Tab

State	Description
Running not available	The application server has been started outside of DevPartner Java Edition and is not available for profiling.
Running under session	The application server is being profiled.
Stopping	The application server is stopping.
Stopped	The application server is stopped and available for profiling.
Initialization error	The application server agent cannot be properly initialized.
Configured Application Server Not Found	DevPartner Java Edition was unable to find a configured application server.
Command Not Found	The application server start or stop script cannot be found.
Inappropriate status	The application server returns a status that indicates it is in an inappropriate state to execute the most recent command.

If your server state is **Running Under Session**, then DevPartner Java Edition is collecting data.

Table 2-2. Actions for Running Under Session Server State

Use this button...	To perform this action...
View	Go to the active session where you can monitor the current test or stop it.
Detach	Stop a test and leave the server running. This option is the normal way to stop a test, because it leaves the application server running and waiting for your next test. It is fairly quick to detach from a server and later reconnect to the server as you start a new test (by clicking **Start**). When the detach is complete, the server state is **Available**.
Stop	Stop a test and stop the server. Use this option when you want to stop an application server completely. This option is useful when you have completed all tests and you want to start your application server normally (without any DevPartner Java Edition instrumentation). You also need to stop and restart a server when you change configuration settings. Stopping a server might take some time.

If your server state is **Available**, then the server is already running with DevPartner Java Edition instrumentation, although no test is currently active.

Table 2-3. Actions for Available Server State

Use this button...	To perform this action...
Start	Start a new test. This option reconnects to the server and starts a new test session.

Table 2-3. (Continued)Actions for Available Server State

Use this button...	To perform this action...
Stop	Stop the server completely. Use this option when you want to stop an application server completely. This option is useful when you have completed all tests and you want to start your application server normally (without any DevPartner Java Edition instrumentation). You also need to stop and restart a server when you change configuration settings. Stopping a server might take some time.

If your server state is **Stopped**, then the server is not running.

Table 2-4. Actions for Stopped Server State

Use this button...	To perform this action...
Start	Start a new test. This option reconnects to the server and starts a new test session.

If your server state is **Running not available,** then the server is running, but without any DevPartner Java Edition instrumentation. To start a test, click **Stop** to stop the server, then click **Start** to restart the server and start a test.

If your application server is not listed, configure the server for use with DevPartner Java Edition. For more information, see "Configuring Application Servers" on page 19.

DevPartner Java Edition User Interface

This topic describes features of the Web interface for DevPartner Java Edition.

Opening the DevPartner Java Edition User Interface

The DevPartner Java Edition Web interface is automatically started when you use a command line utility or your IDE to begin profiling your code, unless you are profiling in batch mode.

Alternately, you can start the Web interface on Windows or UNIX by entering **dpj** at a Windows command prompt or UNIX shell prompt.

Note: You cannot start the Web interface on AIX or HP-UX with the `dpj` utility. Start the DevPartner Java Edition server by running one of the supported browsers from a Windows, Solaris, or Linux computer. Enter the URL **`http://`*`my_box`*`:21578/ui`**, where ***`my_box`*** is the name of the AIX or HP-UX computer.

On Windows operating systems, you can start the Web interface from the **Start** menu by choosing **Program Files>Micro Focus>DevPartner Java Edition>DevPartner Java Edition**.

DevPartner Java Edition Start Page

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

The Start page contains these tabs for the DevPartner Java Edition user interface:

- **Welcome** — Provides overview information and provides links to help text.
- **Application Testing** — Lets you detach and reattach an application started by a DevPartner Java Edition command-line utility.
- **Application Server Testing** — Lets you start a session to collect profiling data on code run through an application server.
- **Session Files** — Lists all session files grouped by configuration, whether created by running your code through the command line, through an IDE, or through an application server.
- **Active Sessions** — Lists all sessions that are currently being profiled. For more information, see Chapter 5, "Sessions".
- **Configurations** — Lets you review the options that DevPartner Java Edition uses to control data collection from your code; you can modify or delete a configuration, or create new configuration based on an existing file.

To release the DevPartner Java Edition license:

- Internet Explorer — If you navigate to another Web site in your browser or close the current browser instance, Internet Explorer automatically releases the license.

Note: Before DevPartner Java Edition can profile code run through an application server, you must configure the application server. For more information, see "Configuring Application Servers" on page 19.

Session Control Page

When you start a session interactively, either from the Web interface, through an IDE, or through a command line utility (unless in batch mode), DevPartner Java Edition displays a Session Control page. The options on this page enable you to focus data collection on the portions of your code that are significant to you.

The options on the Session Control page vary depending on the type of analysis being performed:

- Memory analysis, including Object Retention and Temporary Object analysis
- Performance
- Coverage

Note: For Internet Explorer users, when running multiple profiling sessions simultaneously, you can select whether the Session Control page should reuse the existing browser window. By default, the browser window is reused. To change this option, open the **Internet Options** dialog box from the from the **Tools** menu in Internet Explorer, select the **Advanced** tab, and clear the **Reuse windows for launching shortcuts** option.

Results Summary Page

A Results Summary displays the contents of a session file.

The Results Summary graphically displays the most significant data gathered in your profiling session. From this page, you can drill down into specific areas to analyze performance bottlenecks, memory allocation problems, or gaps in test coverage.

The content of the Results Summary varies depending on analysis type. For information specific to each, visit the following:

- Memory analysis: Object-Lifetime, Temporary Objects, Memory Leaks, or RAM Footprint Results Summary
- Performance Results Summary
- Coverage Results Summary

To specify the precision of your data, click **Preferences** in the title bar of a Session Control page or Results Summary. Use the **Preferences** dialog box to set the following parameters:

- **Precision** — Zero, one, two, three, or four decimal place precision
- **Time** — Microseconds, milliseconds, or seconds
- **Memory** — Bytes, kilobytes or megabytes

The selections made here will affect the session detail displayed in the analysis pages, as they pertain to units of time or units of memory (size).

You can also specify whether to show the inline help in the Results Summaries for the various types of Memory analysis. The default is to show the inline help in all results summaries. If you want to use more of your screen space for displaying results of the Memory analysis, clear the selection for the types of Memory analysis that will not display inline help in their results summaries.

Call Graph

When you are tracking down a problem in performance or memory allocation, it is useful to know the chain of calls leading to a particular method call, and the methods that are subsequently called by that method. DevPartner Java Edition presents this information in the *Call Graph*.

Note: The Call Graph is one of several graphical features in DevPartner Java Edition. Although it provides unique benefits as you analyze your application, it also shares common attributes with the Allocation Trace Graph and object reference path.

To view a Call Graph for a method, click a method and select **View Call Graph** from the Details window. The **Call Graph** option is available whenever you view a list of methods, such as in a Results Summary, Method List, Entry Point List, or Source View.

The Call Graph provides useful information about the selected method, such as percent in parent and in child for Performance analysis. For Memory analysis, the Call Graph presents a graphical representation of temporary objects in a method, memory leaks, and RAM footprint statistics. The nodes are displayed from left to right in the order in which they were called.

Recursive calls are displayed as a series of sequential calls, until the maximum number of displayable nodes is reached. To view another series of sequential calls, click a different base node and select **View call graph for this method**.

The critical path is computed and displayed by default, but DevPartner Java Edition enables you to navigate the Call Graph based on criteria of your choosing, building your own path.

Click a node in the Detail pane to display complete details, go to the source code, or view a Call Graph for this method. (The Source View is not available if the session file only includes method level data.) Clicking **View call graph for this method** changes the baseline node to the selected method; a **Back** button appears in the **Graph** toolbar so you can revert to the original base node.

To limit the amount of data shown, clear **Show All Nodes**. When this option is unselected, DevPartner Java Edition groups all methods with less than 0.5% with children (or in parents) except for the first node in a single-method node. When it is selected, each method with less than 0.5% is displayed in separate method nodes. (Selecting this option can make the overall Call Graph large and hard to comprehend, so by default it is disabled.)

The active configuration file determines whether DevPartner Java Edition displays trivial methods in an Allocation Trace Graph, Call Graph, or Method List.

For the Temporary Objects Call Graph, click **Node Data Selection** to specify which of the following details appear on each node in the Call Graph:

- % Temporary Objects in Method
- Temporary Bytes
- Temporary Objects
- Temporary Bytes including Children
- Temporary Objects Count including Children

If **Assign Categories to Classes and Packages** was included in the configuration for the profile, you can click **Node API Pie Chart** to display the results grouped by assigned categories.

The Call Graph shares some features with the Allocation Trace Graph and Object Reference Path. For more information, see “Call Graph, Allocation Trace Graph, and Object Reference Path Common Features” on page 42

Critical Paths in a Call Graph

When you display a Call Graph, DevPartner Java Edition computes the critical path for the selected method by combining the data for each method and all of its children.

- **Performance analysis** displays the critical path by determining the sequence of child method calls that resulted in the largest cumulative consumption of CPU clock time for the selected method.

- **Memory analysis** displays the critical path by determining the sequence of child method calls that resulted in the largest cumulative memory allocation for the selected method.

Note: The percentage that appears at the left of each node in the critical path is local to that path: It represents the time or bytes allocated by the child method (and its children) as a percentage of the time or bytes allocated in the execution of that path. On the other hand, the percentage that appears inside the node or in the Details window is global: it is computed as percentage of time or bytes allocated by all methods during the session. For more information on calculation of the percentages for different types of analysis, see "Parent and Child Percentages Provided in Call Graphs" on page 31.

For ease of understanding, DevPartner Java Edition displays the critical path for the selected method. This critical path includes up to fifteen levels to the left and/or right of the baseline node, spanning a maximum of thirty levels from end to end.

Locating the Critical Node

After the critical path for a method is displayed, the next step is to locate the critical node.

- Use **Node Data Selection** to display additional data in each node of the graph. In addition to percentage data, you can compare quantitative data, such as execution count and average/actual time values (performance), or number of bytes allocated and number of objects allocated (memory). These will help you focus your troubleshooting efforts.
- Use the Details window to view even more data for the selected method. For example, in a temporary object analysis Details window, DevPartner Java Edition breaks down the temporary object data into short, medium, and long-lived objects, so you can see what kinds of objects the method is allocating.
- Explore the Call Graphs for different nodes in the critical path. For example, displaying the Call Graph for a child node will reveal methods other than those in the critical path that call the selected child method.

Displaying a Different Path in a Call Graph

DevPartner Java Edition provides a browsing capability that enables you to navigate the Call Graph based on criteria of your choosing.

Change the current baseline node, if necessary, by clicking on a method and selecting **View call graph for this method**.

Click the **Expand** control (the arrow-shaped control on the left or right side of a node) to collapse either the parents or the children in the critical path; an **Expand** control then appears on all other nodes that can be expanded. Because only one path can be displayed at a time, collapse the critical path before displaying a different path.

Click the **Expand** control for the node you are interested in to display its parents or children.

Continue expanding the path to browse the Call Graph.

Parent and Child Percentages Provided in Call Graphs

In the Call Graph, DevPartner Java Edition displays percentages outside each node in the call path for the selected method. These percentages are computed relative to the call path displayed, as described below.

Percent in Parent

Percent in Parent is displayed next to the nodes that appear to the left of the base (selected method) node. This percent is relative to the total amount of time or memory consumed by this method (including all other methods called from this method) when called from all its parents.

- ◆ Performance

  This value represents the percentage of time to execute the method (and all methods it calls), when it is called from a particular parent.

- ◆ Memory

  - ◇ For temporary objects, this value represents the percentage of temporary objects for this method and its children when called by this parent.
  - ◇ For RAM footprint, this value represents the percentage of memory consumed by this method and its children when called by this parent.
  - ◇ For memory leaks, this value represents the percentage of memory that was leaked for this method and its children when called by this parent.

Percent in Child

Percent in Child is displayed next to the nodes that appear to the right of the base (selected method) node.

- ◆ Performance

  This value represents the percentage of time spent in a child method and its child methods, relative to the total amount of time spent in the child methods called by this method.

- ◆ Memory

  - ◇ For temporary objects, this value represents the percentage of temporary object bytes allocated by a child method and all its children, relative to the total temporary object bytes allocated by the child methods called by this method.
  - ◇ For RAM footprint, this value represents the percentage of memory consumed by a child method and all its children, relative to the total memory consumed by the child methods called by this method.
  - ◇ For memory leaks, this value represents the percentage of memory that was leaked by a child method and all its children, relative to the total memory leaked by the child methods called by this method.

Entry Points Page

The Entry Points page displays a table of all entry points in the session file. This list is available when viewing Performance or Temporary Object data.

To sort the list, click the column head for the column you want to use as the sort criterion. A white arrow indicates the current sort criterion. You can sort by ascending or descending order.

Click an entry point to display the Detail window from which you can view a Call Graph or view source code for the selected entry point.

For lists containing numerous entry points, use **Previous**, **Next**, and **Show All** to display sections of the list. Click **Column Selection** to choose the columns to be displayed for performance or for temporary objects.

Entry Points Columns — Performance

You can choose to display any of the following columns in the Entry Points page for Performance analysis:

- **Class** — Name of the class.
- **Package** — Package in which the class resides.
- **% Thread Time in Class** — The thread time spent in this method (excluding profiled children) as a proportion of the thread time spent in the other methods of this class.
- **% Thread Time in Method** — The thread time spent in this method (excluding profiled children) as a proportion of the total amount of thread time seen during this profiling run.
- **Average Clock Time including Children** — Amount of clock time spent executing, including children, divided by the execution count.
- **Clock Time including Children** — Amount of time spent executing, including children.
- **% Thread Time including Children** — The thread time spent executing this method and all the methods it called, as a percentage of the total thread time collected during this run.
- **Thread Time including Children** — The thread time spent in this method and the methods it calls, seen during this profiling run.
- **Execution Count** — Number of times this method was executed.
- **Average Thread Time including Children** — Average amount of thread time used by this method and any children it called.

Entry Points Columns — Temporary Objects

You can choose to display any of the following columns in the Entry Points page for temporary objects:

- **Class** — Name of the class.
- **Package** — Package in which the class resides.
- **Execution Count** — Number of times this method was called.
- **Average Temporary Bytes** — Average amount of temporary space (short and medium) used by this method when called (not including its profiled child methods).
- **Average Temporary Bytes including Children** — Average amount of temporary space (short and medium) used by this method and its child methods when called.
- **Temporary Bytes including Children** — Amount of accumulated temporary space (short-lived and medium-lived) allocated by this method and its child methods.
- **% 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 it calls.
- **Short-lived Bytes including Children** — Amount of accumulated short-lived space allocated by this method and the child profiled methods that it called.
- **Short-lived Objects including Children** — Number of accumulated short-lived objects allocated by this method and the child methods it calls.
- **Medium-lived Bytes including Children** — Amount of accumulated medium-lived space allocated by this method and the child profiled methods that it called.
- **Medium-lived Object Count including Children** — Number of accumulated medium-lived objects allocated by this method and the child methods it calls.
- **Long-lived Bytes including Children** — Amount of accumulated long-lived space allocated by this method and the child profiled methods that it called.
- **Long-lived Objects including Children** — Number of accumulated long-lived objects allocated by this method and the child methods it calls.

Note: DevPartner does not consider long-lived space as temporary. Long-lived space is generally not of great consequence when looking for potential problems in your application. However, it is included as data in the product to help you understand the behavior of your program. A long-lived object is one that remains reachable after the entry point that allocated it completes execution.

Entry Points with Retained Instances

You can choose to display any of the following columns in the Entry Points page for retained instances:

- **Entry Point** — The name of the entry point (externally called method).
- **Execution Count** — The number of times the method was executed.
- **# of Retained Objects** — Total number of objects created and retained in memory for the method, not including child objects.
- **# of Retained Objects Including Children**

You can sort the list in ascending or descending order by clicking a column head.

To view the Call Graph or the instances for an entry point, click the entry point to display the Details window, then click the appropriate link.

Method List

The Method List displays methods executed during analysis. By default, all methods are listed. Use the tree control to display a subset.

Click a method to display details and to view source code; add a Session Control Rule; or, for Performance analysis and Memory analysis, view the Call Graph.

Click **Preferences** to select precision (one, two, three, or four decimal positions), and to select the units in which data is presented.

Click **Column Selection** to select the columns to be displayed for each method. You can sort the Method List by clicking a column heading.

To sort the tree control, choose from the **Sort By** menu. The selections vary depending on analysis type:

- For Performance analysis, you can sort the tree alphanumerically or by the percent of the total Execution Time.
- For Coverage analysis, you can sort the tree alphanumerically or by the number of lines not executed.
  - For Memory analysis, sorting options depend on the data being viewed:
  - For Temporary Objects, you can sort the tree by temporary bytes, short-lived bytes, medium-lived bytes, or alphanumerically.
  - For Memory Leaks, you can sort the tree by the number of leaked bytes or alphanumerically.
  - For RAM Footprint, the tree is sorted by percent of total footprint. You can sort any of the columns by ascending or descending order.

For lists containing numerous methods, use the **Previous**, **Next**, and **Show All** to traverse the list.

Note: The active Configuration 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.

Method List Columns

The columns available from the Method List can be accessed from various places in DevPartner Java Edition, and its contents will vary, depending on the type of analysis that is performed and the choices made by the user. The column options are grouped below by category:

- Common to All
- Performance
- Coverage
- Temporary Objects
- Leaked Objects
- RAM Footprint

Common to All

The following columns are available for all profiles:

- **Method** — Name of the method.
- **Class** — Name of the class in which the method resides.
- **Package** — Name of the package in which the method resides.

Performance

The following columns are available for Performance profiles:

- **% Thread Time in Class** — The thread time spent in this method (excluding profiled children) as a proportion of the CPU time spent in the other methods of this class.
- **% Thread Time in Method** — The thread time spent in this method (excluding profiled children) as a proportion of the total amount of CPU time seen during this profiling run.
- **%Thread Time including Children** — The thread time spent executing this method and all the methods it called as a percentage of the total thread time collected during this run.
- **Thread Time including Children** — The thread time spent in this method and the methods it calls seen during this profiling run.
- **Average Clock Time including Children** — Amount of clock time spent executing, including children, divided by the execution count.
- **Clock Time including Children** — Amount of time spent executing, including children.
- **Execution Count** — Number of times this method was executed.
- **Average Thread Time** — Average time used by this method for each time it executed.
- **First Execution Thread Time** — Amount of thread time this method used the very first time it was called. This number is reported separately because Java methods often incur additional overhead the first time they are called due to the lazy initialization of other classes that they might call into (either to create a new object or invoke a static method).
- **Minimum Thread Time** — Least amount of thread time DevPartner Java Edition has ever seen this method take (excluding time spent in child profiled methods).
- **Maximum Thread Time** — Greatest amount of thread time DevPartner Java Edition has ever seen this method take (excluding time spent in child profiled methods).
- **Average Thread Time including Children** — Average amount of thread time used by this method and any children it called.
- **Thread Time** — Accumulated amount of thread time DevPartner Java Edition has recorded for this method, not including time spent in child profiled methods.
- **Clock Time** — Accumulated amount of wall-clock time DevPartner Java Edition has recorded for this method, not including time spent in child profiled methods.
- **Wait Time** — Accumulated time spent by a method waiting for some other event to process.

  This time does not include time spent in profiled methods that these methods called. For example, a method might wait for the I/O to complete, a contended synchronization, or for the CPU to finish executing another thread or process. In the latter case, this time is usually inconsequential, and because it is due to relatively small thread time slices, can be easily ignored.

Coverage

The following columns are available for Code Coverage:

- **% Lines Covered** — For the number of executable lines in this method, the percentage that have actually been executed.
- **Execution Count** — Number of times this method was called.
- **Lines not Executed** — Number of executable lines in this method that have not been executed.
- **Lines Executed** — Number of executable lines in this method that have been executed.
- **Lines** — Number of executable lines in this method.
- **State** — Description of the last observed coverage state of this method.

Temporary Objects

The following columns are available for temporary objects:

- **Execution Count** — Number of times this method was called.
- **Average Temporary Bytes** — Average amount of temporary space (short and medium) used by this method when called (not including its profiled child methods).
- **Average Temporary Bytes including children** — Average amount of temporary space (short and medium) used by this method and its child methods when called.
- **Temporary Bytes including Children** — Amount of accumulated temporary space (short and medium) allocated by this method and its child methods.
- **% 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 it calls.
- **Short lived Bytes including Children** — Amount of accumulated short lived space allocated by this method and the child profiled methods that it called.
- **Short lived Objects including Children** — Number of accumulated short lived objects allocated by this method and the child methods it calls.
- **Medium lived Bytes including Children** — Amount of accumulated medium lived space allocated by this method and the child profiled methods that it called.
- **Medium lived Objects including Children** — Number of accumulated medium lived objects allocated by this method and the child methods it calls.
- **Long lived Bytes including Children** — Amount of accumulated long lived space allocated by this method and the child profiled methods that it called.
- **Long lived Objects including Children** — Number of accumulated long lived objects allocated by this method and the child methods it calls.

Leaked Objects

The following columns are available for leaked objects:

- **Execution Count** — Number of times this method was called.
- **Leaked Bytes** — Size of the objects that were leaked.
- **% Leaked Bytes** — Size of the objects that were leaked as a percentage of the total size of all leaks.
- **Leaked Objects** — Number of objects that were leaked.
- **Leaked Bytes including Childre**n — Sum of the sizes of leaked objects allocated by this method and all of the methods it called, including child profiled methods.
- **% Leaked Bytes including Children** — Leaked Bytes with Children represented as a percentage of the total amount of leaked space recorded in this profiling run.
- **Leaked Objects including Children** — Number of leaked objects allocated by this method, including child profiled methods.
- **Total Allocation Bytes** — Size of all the objects ever allocated by this method.
- **Total Allocations** — Number of objects ever allocated by this method.

RAM Footprint

The following columns are available for RAM Footprints:

- **Live Bytes** — Sum of the sizes of the objects that this method (not including child profiled methods) allocated that were still alive when this snapshot was taken.
- **% Live Bytes** — Represented as a percentage of the total size of the profiled objects in the heap when this snapshot was taken.
- **Live Objects** — Number of live objects that this method (not including child profiled methods) allocated that were still alive when this snapshot was taken.
- **Live Bytes including Children** — Sum of the sizes of the objects that this method and the methods that it called allocated that were still alive when this snapshot was taken.
- **% Live Bytes including Children** — Represented as a percentage of the total size of the profiled objects in the heap when this snapshot was taken.
- **Live Objects including Children** — Number of objects that this method and the methods that it called allocated that were still alive when this snapshot was taken.

Computing Total Bytes Including Children

The Method List for Memory Leak or RAM Footprint analysis includes a column for Leaked/ Live Instance Bytes Including Children. (If this column is not visible, click **Column Selection**.)

The value for Instance Bytes Including Children is the sum of the bytes allocated by all instances of the method plus all bytes allocated for all child methods. This value highlights methods that are responsible for the largest amounts of allocated memory.

Source View

To help you isolate the cause of your performance, memory, or coverage problems, DevPartner Java Edition enables you to drill down into the source code for each object.

To view the source code for an object, select **View Source Code** from the Details window for that object. If the configuration used to create this session file does not include the source file paths for your code, you are prompted to enter a path. The path is then entered automatically into the configuration file.

The source code view opens in a new browser window. Click **Column Selection** to select the columns to be displayed; the columns you select are displayed at the left panes and are right-aligned. The source code is displayed in the right pane and is left-aligned with tabs. Tooltips provide additional information about some nodes in the tree view. Statistics are displayed for every line of code called during program execution.

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

- Yellow highlighting identifies the first line of the selected method.
- For Performance analysis, blue highlighting identifies the slowest line in each method.
- For Coverage analysis, the following color coding is used:
  - Green — Lines that were executed
  - Red — Lines that are executable but were not executed

Lines that are comments or are non-executable are not highlighted. Note that, since line statistics are shown only for the selected class, lines in a different class also will not be highlighted.

Click an executed line of code to display its details. You can view instances, go to the declaring method, go to called methods, or update the source path.

Click **Printer Friendly Version** to display the source code view in a format better suited for printing than the initial Source View.

Notes:

- DevPartner Java Edition displays the current source code for the selected file. If you change the source code in a project (for example, by adding or removing lines) and subsequently open an old session file, the information in the data columns in the **Source** tab may not match the changed source code.
- Class files that were compiled without debugging information will not show up in Coverage analysis session results. To get coverage information on these classes, recompile them with debugging information and profile again.

You can sort the tree control by selecting a sort criterion from the **Sort By** menu. The criteria vary depending on analysis type:

- Performance — Percentage of total Execution Time, or No Filter.
- Coverage — Number of lines not executed, or No Filter.
- Memory:
  - Temporary Objects — % average temporary byes, % temporary bytes, % short-lived bytes, % medium-lived bytes, or No Filter.

- ◇ Memory Leaks — Number of leaked bytes, or No Filter.
- ◇ RAM Footprint — Live bytes, or No Filter.

Note: When **No Filter** is selected, all packages and classes are listed in alphanumeric order.

Path Selection

To view source code, you must provide the source file path.

1. Select the path by browsing or designate the path directly in the **Path** field.

   On Windows only, you can also specify UNC paths.

   - ◇ Enter \\server to display a list of all shared folders for that UNC server.
   - ◇ Enter \\server\share\dir\dir\dir to specify a specific path.

2. Click **OK** to enter the path.

   The path is automatically added to the configuration.

For information on setting up NCSP to allow UNC browsing, see “Accessing Source Code on Remote Computers” on page 44.

Source View Columns

The Source View can be displayed from various places in DevPartner Java Edition. Its contents depend on the type of analysis performed and the choices you make for those analyses. The column options are grouped below by category:

- Common to All
- Performance
- Coverage
- Temporary Objects
- Leaked Objects
- RAM Footprint

Common to All

The following columns are available for all profiles:

- **Execution Count** — Number of times this line has been executed.
- **Line Number** — Source file line number.
- **Source / Path** — Name of the source file and the complete path to it (shown in the blue area at the top of the columns in the right pane).
- **Source code** — Source code (right pane).

Performance

The following columns are available for Performance profiles:

- **Child Methods** — Number of child methods called by the method on this line.
- **Thread Time** — CPU time spent on this line (including time spent in any methods this line calls).
- **% Thread Time in Method** — Thread Time spent on this line (including time spent in any methods this line calls) as a proportion of the total amount of Thread Time spent on all the lines of this method.
- **Clock Time** — Elapsed time between the start of this line's execution and the end, including time spent in any methods this line calls.
- **Wait Time** — Non-CPU time spent on this line (Clock Time minus Thread Time), including time spent in any methods this line calls.

Coverage

The following column is available for Code Coverage:

- **Child Methods** — Number of child methods called by the method on this line.

Temporary Objects

The following columns are available for temporary objects. These definitions apply, to short-, medium-, or long-lived objects.

- **Child Methods** — Number of child methods called by the method on this line.
- **Temporary Bytes including Children** — Amount of accumulated temporary space (short-lived and medium-lived) allocated by this method and its child methods.
- **% Temporary Bytes including Children** — Temporary Bytes with Children as a proportion of the total amount of Temporary Bytes allocated by all the lines of this method.

  This statistic is the same as the preceding item, 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 it calls.
- **Short-lived Bytes including Children** — Amount of accumulated short-lived space allocated by this method and the child profiled methods that it called.
- **% Short-lived Bytes** — Percentage of bytes used by short-lived space.
- **Short-lived Objects including Children** — Number of accumulated short-lived objects allocated by this method and the child methods it calls.
- **% Medium-lived Bytes** — Percentage of bytes used by medium-lived space.
- **Medium-lived Bytes including Children** — Amount of accumulated medium-lived space allocated by this method and the child profiled methods that it called.

- **Medium-lived Object(s) Count including Children** — Number of accumulated medium-lived objects allocated by this method and the child methods it calls.
- **Long-lived Bytes including Children** — Amount of accumulated long-lived space allocated by this method and the child profiled methods that it called.
- **% Long-lived Bytes** — Percentage of bytes used by long-lived space.
- **Long-lived Objects including Children** — Number of accumulated long-lived objects allocated by this method and the child methods it calls.

Leaked Objects

The following columns are available for leaked objects:

- **Child Methods** — Number of child methods called by the method on this line.
- **Leaked Bytes including children** — Sum of the sizes of leaked objects allocated by this line, including the child methods it calls.
- **% Leaked Bytes including children** — Same as the preceding item, but represented as a percentage of the total amount of leaked space recorded in this profiling run.
- **Leaked Objects including children** — Number of leaked objects allocated by this line, including child methods it calls.

RAM Footprint

The following columns are available for RAM Footprints:

- **Child Methods** — Number of child methods called by the method on this line.
- **Live Bytes including children** — Sum of the sizes of the objects that this line (including child methods) allocated that were still alive when this snapshot was taken.
- **Live Objects including children** — Number of objects that this line (including child methods) allocated that were still alive when this snapshot was taken.
- **% Live Bytes including children** — Represented as a percentage of the total size of the profiled objects in the heap when this snapshot was taken.

Call Graph, Allocation Trace Graph, and Object Reference Path Common Features

The Call Graph, Allocation Trace Graph and Object Reference Path have these common features:

- Overview pane
- Detail pane
- Information about selected node
- Node Data selection
- Method of changing the layout
- Splitter bar

Overview Pane

- Call Graph — The **Overview** pane displays a thumbnail view of the entire graph for the selected method. Drag the rectangle or click a node to display in the **Detail** pane.
- Object Reference Graph — The **Overview** pane displays the entire path to the root object for the selected instance. Drag the rectangle to select the nodes to be displayed in the **Detail** pane.

Detail Pane

- Call Graph — The **Detail** pane shows the nodes that are visible in the rectangle. You can drag the rectangle to display specific nodes in the **Detail** pane. The legend identifies the colors used to show the baseline node, the critical path to the method, caller methods (parent) to the left of the baseline node, and called methods (children) to the right of the baseline node.
- Object Reference Graph — The **Detail** pane shows the nodes selected in the **Overview** pane. The links are labeled with the data member that refers to the next class.

Information About Selected Node

- Click a node to display details about a given method. Hover the mouse cursor over a node for additional information about that node shown in a tooltip.

Node Data Selection

- Click **Node Data Selection** to select which details will be displayed for each node in the Call Graph and Allocation Trace Graph only (not available for Object Reference Graph).

Changing Layout

- You can drag a node or a group of nodes (using standard Windows click and drag) over different nodes to rearrange the view in the **Detail** pane. Click **Restore Layout** to resume the default layout in the **Detail** pane.
- If the graph overlaps the legend, you can click-and-drag the legend to a different position.

Splitter Bar

- Use the splitter bar to resize the **Overview** pane to increase the **Detail** pane's visible area.
- If the size of the graph exceeds the scale of the **Overview** pane, vertical and/or horizontal scroll bars will appear, enabling you to navigate to other parts of the path. Click **Restore Layout** in the toolbar to return to the default layout.

Printing Profiling Data

Most of the pages displayed by DevPartner Java Edition can be printed. For best printing results:

- Set your browser to print background images. For Internet Explorer, select **Tools>Internet Options>Advanced>Printing>Print background colors and images**. If this option is not set, the appearance of printed pages might vary from their online appearance.
- In Internet Explorer, select **File>Print>Options>As laid out on the screen** to print all frames as displayed on the screen.
- When printing, if black rectangles appear instead of graphics and data, make sure you have the latest version of the Java plug-in for your browser.
- The width of table columns can be adjusted to fit as much of the table on the page as possible. Use the Landscape print setting for best results.
- Pages with dynamic information, such as the Session Control page, cannot be printed. In Windows, you can use the Print Screen key to capture the screen and paste it into an application such as Microsoft Word or Microsoft Paint to print.

Accessing Source Code on Remote Computers

Note: This topic applies to Windows environments only.

The DevPartner Java Edition Control Service accesses source code. By default, this service is installed to log on as the local system. This means the local system cannot see computers on the network; it can only see what is on the local system. By default, DevPartner Java Edition cannot access source code that is on another system.

To set up DevPartner Java Edition to access source code on remote computers:

1. Select **Start>Settings>Control Panel>Administrative Tools>Services**.
2. In the Services window, double-click **Micro Focus DevPartner Java Edition Control Service** to display the **Properties** dialog box.
3. Select the **Log On** tab.
4. Under **Log on as**, select **This account**.
5. In the **This account** field, enter the domain and user on the DevPartner Java Edition server, or browse to select a domain and user. The user must be a member of the administrators group on the server. (The server is the computer where DevPartner Java Edition is installed and NCSP is running.)
6. Enter a valid password for the selected user and confirm it.
7. Click **OK** to save the information and close the dialog box.
8. In the Services window, right-click **Micro Focus DevPartner Java Edition Control Service** and select **Restart** from the menu.

The user or group running NCSP on the remote DevPartner Java Edition server must have the following privileges to start NCSP on the remote computer:

- Debug programs
- Replace a process level token

To set these rights on the remote server:

1. Select **Start>Settings>Control Panel>Administrative Tools>Local Security Policy**.
2. In the tree, expand the **Local Policies** node to display the local policies.
3. Select **User Rights Assignment** to display the policies in the right pane.
4. Right-click **Debug Programs** and select **Security** from the menu to display the **Local Security Policy Setting** dialog box.
5. Click **Add** to display the **Select Users or Groups** dialog box; select a user or group to add access, and click **OK** to save the change and close the dialog box.
6. In the **Local Security Setting** dialog box, right-click **Replace a process level token** and select **Security** from the menu to display the **Local Security Policy Setting** dialog box.
7. Click **Add** to display the **Select Users or Groups** dialog box; select a user or group to add access, and click **OK** to save the change and close the dialog box.

Restart the Micro Focus DevPartner Java Edition service on the local user's computer.

To view source code in DevPartner Java Edition, enter the UNC style path to the source in the **Path** field at the bottom of the **Path Selection** dialog box. For example:

```
\\remotemachine\source\code-directory
```

You can also set this UNC path in the configuration for DevPartner Java Edition by selecting **Source File Paths** on the **Configurations** tab. Before you set this configuration, you must have already enabled **Debug Programs** and **Replace a process level** token rights in the Micro Focus DevPartner Java Edition Control Service properties.

On Windows, you will not be able to use mapped drives, because they belong only to the login session in which they were created.

Providing Java Plug-ins for Remote Users

If you are a remote user and you start the DevPartner Java Edition user interface on your computer, the program detects whether you have the required Java plug-in installed. If the plug-in is not there, DevPartner Java Edition redirects you to the Sun Web site, where you can download the latest plug-in.

To host the Java plug-ins on the DevPartner Java Edition server, copy the installation files to the computer. For more information, see “Hosting Java Plug-ins for Remote Users” on page 47.

Internet Explorer

1. DevPartner Java Edition first looks to see if a plug-in exists on the server. If so, it attempts to download and install the plug-in on the user's system.
2. If there are no plug-ins on the DevPartner Java Edition server, the browser attempts to automatically install the plug-in from Sun Corporation.
3. When a security warning is displayed, click **OK**.
4. Internet Explorer installs the plug-in.

Firefox

DevPartner Java Edition displays a page in which you can select the plug-in to download or use.

1. Select to download and install the plug-in by following the instructions at the Sun Corporation Web site.
2. Close all browser windows.
3. Restart DevPartner Java Edition in a new Browser.

Firefox on Windows

Follow the steps in the Installation Instructions section of the Sun installation instructions.

Firefox on Linux and Solaris

Note: On Linux, use Firefox 3 with JDK 1.6.0_10 or higher to ensure the symlink works properly.

Follow the steps in these sections of the Sun installation instructions:

- Installation of Self-Extracting Binary
- Java Plug-in Installation Instructions

If the plug-in does not work after you follow Sun's instructions, you might need to execute these additional commands:

```
cd <firefox-install-directory>/plugins

ln -s <install-directory-for-jre>/plugin/i386/ns7/
libjavaplugin_oji.so libjavaplugin_oji.so .
```

Hosting Java Plug-ins for Remote Users

To host the Java plug-ins on the DevPartner Java Edition server, copy the installation files to the computer.

Windows

Download the “Windows Offline Installation” file from Sun Corporation and copy it to ***`DPJ_dir`*****`\tomcat\webapps\DPJ\plugin\`**, where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder.

UNIX

Download the appropriate “Self-extracting Installation” file from Sun Corporation and copy it to **`/opt/Micro Focus/DPJ/tomcat/webapps/DPJ/plugin/`**.

Using DevPartner Java Edition with Distributed Application Analysis

The distributed application analysis feature in DevPartner Studio monitors Internet Explorer sessions by default. You can run the distributed application analysis feature and DevPartner Java Edition at the same time. Micro Focus does not support monitoring of DevPartner Java Edition code by the distributed application analysis feature. Here is a scenario in which the distributed application analysis feature will monitor DevPartner Java Edition code.

Problem Scenario

If you are running the distributed application analysis feature and then start DevPartner Java Edition in an existing Internet Explorer window, the distributed application analysis feature automatically starts monitoring the DevPartner Java Edition code. This occurs because the distributed application analysis feature starts monitoring programs running in an Internet Explorer window automatically once a distributed application analysis session starts.

To check for this, examine the Actions Pane of the distributed application analysis feature. If you see information about DevPartner browser pages, then the distributed application analysis feature is monitoring DevPartner Java Edition.

Solution

1. Close the browser in which DevPartner Java Edition is running. If Internet Explorer is your default browser, close all browser instances. You do not need to stop the distributed application analysis feature.
2. Restart DevPartner Java Edition. The program will start but will not be profiled by the distributed application analysis feature.

Profiling Applets

Applets are Java programs that are written to be run in a special container. This container might be either an appletviewer, provided in the Java Development Kit (JDK), or a Web browser.

DevPartner Java Edition provides two ways to profile applets.

- Profiling in the appletviewer — Use the command line **`nmappletviewer`**.
- Profiling directly in the browser — In a browser, use the command line **`nmshell`** and then execute the browser in that new shell. All applets loaded in this browser will subsequently be profiled.

Exit all instances of your browser before starting a new instance through **`nmshell`**.

You can profile an applet to analyze memory leaks. There are two possible work flows when performing memory leak analysis:

- Start tracking, exercise the program, and then view the results.
- Start tracking, exercise the program, stop tracking, exercise the program again, and then view results.

Typically, applet testers would use the second approach for optimum results.

You can profile any applet on any page. It is not necessary that you own the code or even the class files.

You can call a Session Control API from within an applet.

Example Using nmshell in Windows

3. Start a DOS command window.
4. Type the command `nmshell`.
5. To find the name of the Web browser executable to run in **`nmshell`**, right-click the browser's icon and select **Properties**. From the dialog box that appears, select and copy the text in the **Target** field. You might see, for example:
   - For Internet Explorer: **`"C:\Program Files\Internet Explorer\IEXPLORE.EXE"`**
6. Enter the name of the Web browser executable at the **`nmshell`** prompt.
7. Access the HTML page where your applet resides.

Chapter 3

Command Line Utilities

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

Other capabilities, such as setting up a configuration, are available only from the Web interface for DevPartner Java Edition.

When a program is launched from the command line, a session starts and the Session Control page appears, unless the program is running in batch mode.

DevPartner Java Edition provides the following command line utilities:

- **`nmappletviewer`** — Monitors your Java applets.
- **`nmextract`** — Exports data from DevPartner Java Edition session files to either an ASCII file or an XML file.
- **`nmjava`** — Monitors your standalone Java programs.
- **`nmshell`** — Monitors Java applets in browsers or any Java program run in the shell, such as a batch file or executable file.
- **`nmserver`** — Monitors your Java code run in an application server (this functionality is also available through DevPartner Java Edition's Web interface).
- **`pubmetrics`** — Publish code coverage metrics to Optimal Delivery Manager. For more information, see **`PubMetrics.pdf`**, in the DevPartner Java Edition installation folder.

nmappletviewer

The **`nmappletviewer`** utility enables you to monitor Java applets. The DevPartner Java Edition Web interface is launched, unless the applet is run in batch mode.

Syntax

```
nmappletviewer [-config Configuration-name] {-perf, -mem, -cov} [-
batch] [-nmv] [-javahome <path>] [-help] <normal Java command-line
options>
```

Table 3-1. nmappletviewer Syntax

Switch	Description
-config *Configuration-name*	Specifies the name of the configuration. If ***Configuration-name*** does not exist, a configuration with the specified name and the default configuration settings is created. If `-config` is omitted, DevPartner Java Edition assumes `-config` *`JavaAppletName`*
-perf	Monitor for Performance analysis, which is the default if no analysis type is specified.
-mem	Monitor for Memory analysis.
-cov	Monitor for Coverage analysis.
-batch	Run in batch mode, i.e., do not invoke the Web interface.
-nmv	Verbose operation.
-javahome *path*	Select the Java virtual machine to use. By default, DevPartner Java Edition uses the Java virtual machine specified in your Path environment variable, (the same way it would typically determine which **`java.exe`** to run). If you specify a Java home location, for example **`C:\jdk1.5.0_12`**, then DevPartner Java Edition runs the **`java.exe`** found in that path instead of the Java path specified in your Path environment variable.
-help -h -?	Displays the help text for the specified command line utility.

nmextract

The **`nmextract`** utility exports data from a specified session file for a Performance analysis, Memory analysis, or Code Coverage analysis session. Use this utility at the command prompt, first changing to the folder containing the file from which you want to extract data.

By default, the utility creates a text file (**`.csv`** extension) containing comma-delimited data that can be imported into a spreadsheet. You can also export the data to an HTML file; or to an XML file that can be merged with other XML files, for example, to view data from different reports in one file. For more information, see “Exported Data File Contents” on page 76.

When the session information is extracted, a confirming message appears in the command window.

Syntax

```
nmextract {-perf, -mem, -cov} [-xml {-all [-method, -line], -summary,
-metrics}, -html] <filename.ext> [-out <filename>]
```

Required: If the full path of the filename contains spaces, enclose the filepath in double quotation marks, e.g. **`"C:\Program Data\Micro Focus\DevPartner Java Edition\var\conf\DPJ.conf"`**.

The default command, `nmextract` ***`filename.ext`***, detects the analysis type automatically and exports the data to a **`.csv`** file. If you are exporting data to XML or HTML format, you must specify the analysis type.

The switches used with the **`nmextract`** command are described in the following table.

Table 3-2. nmextract Syntax

Switch	Description
filename.ext	The name of the session file to extract. You must specify the complete file name with the extension. **Caution:** The session file must be in the default location (***`config`*** is the folder for the configuration used in the profiling session): • Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\sessionfiles\`*`config`*** (By default, the **`\Application Data`** folder is hidden. To display the **`\sessionfiles`** folder and its contents, type the path in the Address bar of Windows Explorer and press **Enter**.) • Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\sessionfiles\`*`config`*** • Other operating systems — ***`DPJ_dir`*`/var/sessionfiles/config`**, where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder Data can be extracted from the following types of session files: • **`.tcs`** — Code coverage • **`.tts`** — Performance • **`.mps`** — Temporary objects, leaked objects, or RAM footprint • **`.tcm`** — Merged Code Coverage sessions **Note:** Make sure you use the appropriate switch for the file (analysis) type.
-perf	Exports data from a Performance analysis session, by default to a comma-delimited ASCII text file named *`filename`***`.csv`** (where *`filename`* is the same as the source file). If you are exporting the data to a **`.csv`** file, this switch is optional; it is required for other export formats.
-mem	Exports data from a Memory analysis session, by default to a comma-delimited ASCII text file named *`filename`***`.csv`** (where *`filename`* is the same as the source file). If you are exporting the data to a **`.csv`** file, this switch is optional; it is required for other export formats.

Table 3-2. (Continued)nmextract Syntax

Switch	Description
-cov	Exports data from a Code Coverage analysis session, by default to a comma-delimited ASCII text file named *`filename`***`.csv`** (where *`filename`* is the same as the source file). If you are exporting the data to a **`.csv`** file, this switch is optional; it is required for other export formats. **Note:** When you include **`-cov`** in the command line when exporting to a **`.csv`** file, the file contains coverage details for each object analyzed; without this switch, the file contains summary information.
-html	Exports the data in HTML format to a file named *`filename`***`.html`**, where *`filename`* is the original session file. If you use this switch, you must also specify the analysis type (`-perf`, `-mem`, or `-cov`).
-xml	Exports the data in XML format to a file named ***`filename`*****`.xml`**, where ***`filename`*** is the original session file. If you use this switch, you must specify the analysis type (`-perf`, `-mem`, or `-cov`) and you must use the `-summary` `-all`, `-method`, or `-line` switch.
-summary	Exports the same data as for the **`.csv`** and **`.html`** format. If you use this switch, you must also use `-xml`.
-all	Exports detailed information depending on the analysis type (see details in Table 5-4 on page 76). If you use this switch, you must also use `-xml`.
-method	Includes method-level Code Coverage information in the exported data. It can be used only with `-cov -xml`.
-line	Includes line-level Code Coverage information in the exported data. It can be used only with `-cov -xml`.
-metrics	Exports Metrics data from a Coverage session file to an XML file that can be published to Optimal Delivery Manager. If you use this switch, you must also use `-cov -xml`.
-out	Exports data to the specified output file name. Do not include the full filepath or an extension; the file will be created in the default location as described below, with the extension **`.xml`**.
-help -h -?	Displays the help text for the specified command line utility.

The **`nmextract`** utility processes one session file at a time and generates one output file for each session file, with the same name (changing the extension as appropriate). The files are generated in the folder **`/var/exports/`*****`config`***, where ***`config`*** is the folder for the configuration used in the profiling session.

Numbers are expressed in the lowest unit (for either microseconds or bytes). The number precision will always be the maximum.

Note: A sample application is provided for viewing exported line-level Code Coverage data in HTML format. The HTML display is similar to the Source View within DevPartner Java Edition. For more information see "Exporting and Viewing Line-Level Code Coverage Data" on page 146.

nmjava

The **`nmjava`** utility enables you to monitor Java programs that are not run through an application server. The DevPartner Java Edition Web interface is launched, unless the program is running in batch mode.

Syntax

```
nmjava [–config Configuration-name] {-perf, -mem, -cov} [-batch] [-
nmv] [-javahome <path>]
[-help] <normal Java command-line options>
```

Table 3-3. nmjava Syntax

Switch	Description
-config *Configuration-name*	Specifies the name of the configuration. If ***`Configuration-name`*** does not exist, a configuration with the specified name and the default configuration settings will be created. If `-config` is omitted, DevPartner Java Edition assumes `-config` *`JavaAppName`*
-perf	Monitor for Performance analysis, which is the default if no analysis type is specified.
-mem	Monitor for Memory analysis.
-cov	Monitor for Coverage analysis.
-batch	Run in batch mode, i.e., do not invoke the Web interface.
-nmv	Verbose operation.
-javahome *path*	Select the Java virtual machine to use. By default, DevPartner Java Edition uses the Java virtual machine specified in your Path environment variable, (the same way it would typically determine which **`java.exe`** to run). If you specify a Java home path, for example **`C:\jdk1.5.0_12`**, then DevPartner Java Edition runs the **`java.exe`** found in that path instead of the Java path specified in your Path environment variable.
-help -h -?	Displays the help text for the specified command line utility.

nmserver

The **`nmserver`** utility enables you to profile programs run through an application server without launching the DevPartner Java Edition Web interface.

Note: If the application server is running as a service under supported Windows operating systems other than Windows XP or 2003 Server, the Session Control page will not open automatically when you begin profiling the application server. To view the Session Control page, open the DevPartner Java Edition Start page and select the **Application Server Testing** tab.

Syntax

```
nmserver [-config Configuration-name] {-attach, -detach, -kill} {-
perf, -mem, -cov} [-timeout seconds] [-batch] [-nmv] [-help]
<appserver_name>
```

Table 3-4. nmserver Syntax

Switch	Description
-config *Configuration-name*	Specifies the name of the configuration. If ***Configuration-name*** does not exist, a configuration with the specified name and the default configuration settings will be created. If `-config` is omitted, DevPartner Java Edition assumes `-config JavaAppName`
-attach	If the application server is running and is available, a new session is created and the application server is attached to this session; if the application server is not running, DevPartner Java Edition first starts it and then hooks itself to the application server. If a session is already running under the specified configuration, and that session allows multiple processes per session, the application server is attached to the existing session. To attach an application server that is already executing, that application server must have been started by DevPartner Java Edition.
-detach	Stops monitoring the specified application server and the application server becomes available. It continues running. If the application server does not exist, an error is output to standard error, and no other action is taken.
-kill	Stops the specified application server.
-perf	Monitor for Performance analysis, which is the default if no analysis type is specified.
-mem	Monitor for Memory analysis.
-cov	Monitor for Coverage analysis.
-timeout *seconds*	Timeout for server operations; the default is 600 seconds.
-batch	Run in batch mode, i.e., do not invoke the Web interface.
-nmv	Verbose operation.
appserver_name	Specifies the name of the configured application server to run (for example, WebLogic or WebSphere).
-help -h -?	Displays the help text for the specified command line utility.

nmshell

The **`nmshell`** utility invokes a new console (shell) in which all Java programs are monitored. This utility could be used, for example, to launch a browser and profile an applet in that browser, or to launch a batch file or executable.

To terminate monitoring, exit the shell.

While this command is active, invoking additional `nmshell` commands in the same console will result in an error. You can, however, invoke the `nmjava` or `nmserver` commands in the same console; these commands will behave normally, as if they were invoked outside of this console.

Note: To use **`nmshell`** with an unsupported application server, first configure the server, then use **`nmshell`** as usual. For more information, see "Invoking the Profiler Through the JVM Settings" on page 21.

The **`nmshell`** utility cannot be used to profile OC4J Integrated.

Syntax

```
nmshell -config Configuration-name {-perf, -mem, -cov} [-batch] [-nmv]
[-help]
```

```
nmshell -config Configuration-name {-perf, -mem, -cov} [-batch] [-nmv]
[-help] –exec <command>
```

Table 3-5. nmshell Syntax

	Description
-config *Configuration-name*	Specifies the name of the configuration. If ***`Configuration-name`*** does not exist, a configuration with the specified name and the default configuration settings is created. This option is required.
-perf	Monitor for Performance analysis, which is the default if no analysis type is specified.
-mem	Monitor for Memory analysis.
-cov	Monitor for Coverage analysis.
-batch	Run in batch mode (do not launch the DevPartner Java Edition Start page).
-nmv	Verbose operation.
-help -h -?	Displays the help text for the specified command line utility.
-exec *command*	Execute this command under this configuration; e.g., `-exec` could be followed by the name of a batch file, or by `java` followed by an executable name. This switch is essentially a shortcut to calling **`nmshell`**, then running the command, and then exiting the shell.

Metrics Publishing Utility

DevPartner Java Edition calculates two code coverage metrics, Coverage and Volatility, that can be published to the Optimal Delivery Manager (ODM) using the Metrics Publishing utility. For complete information about these metrics and the utility, see the PDF file **`PubMetrics.pdf`**, which is provided in the DevPartner Java Edition installation folder.

Before publishing the metrics, you must export them from DevPartner Java Edition. For more information, see "Exporting Session Data" on page 75.

Session Control API

DevPartner Java Edition provides an application programming interface (API) that enables you to programmatically control profiling. The API provides a more focused analysis of the target code.

The Session Control API can be most useful when performing a session control action precisely at some arbitrary point in your program, such as method begin or method end. It is also useful on a conditional basis, when a method is passed certain parameters. For example, if you must clear the collected data at the beginning of a method, it is probably impossible to click **Clear** at exactly that precise point in time. Calling the "clear session control" API, however, ensures that this action is taken at exactly the right point in your program.

Using a Session Control API

1 Include the **`DPJSessionControls.jar`** file in your classpath. The file is in the following locations:

- ◇ Windows: ***`DPJ_dir`*****`\SessionControlAPI`**

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

- ◇ Unix: **`/opt/Micro Focus/DPJ/SessionControlAPI`**

2 Add the following `import` statement to your code:

```
import com.compuware.dpj.SessionControls;
```

Methods in Class Session Controls

The **`com.compuware.dpj.SessionControls`** class includes the following methods.

Table 3-6. com.compuware.dpj.SessionControls Class Methods

Method	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()`**.

Table 3-6. com.compuware.dpj.SessionControls Class Methods

Method	Description
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()	For memory leaks only, informs the profiler that all objects that are allocated from this point forward are of interest, as opposed to just objects allocated within non-excluded code.
UnMark()	Reverses the effect of the **`Mark()`** call.

Note: The **`Mark()`** and **`UnMark()`** calls are the programmatic equivalents of the **Start Tracking** and **Stop Tracking** buttons on the Memory analysis Session Control page. Whether you call **`UnMark()`** before or after you call **`TakeSnapshot()`** depends on the data collection scenario appropriate for your application. See "Memory Leaks" on page 93 for more information on tracking allocated objects during Memory Leak analysis.

To access the JavaDoc description for each method, select the Session Control API Reference:

- Windows — **Start>Programs>Micro Focus>DevPartner Java Edition>Utilities>Session Control API Reference**
- UNIX — From the DevPartner Java Edition folder

Calling a Session Control API When Profiling an Applet

1. Add an import statement:

   ```
   import com.compuware.dpj.SessionControls;
   ```

2. Code the API call into your applet.
3. Copy the file **`DPJSessionControls.jar`** from the DevPartner Java Edition product folder to the Web location where your applet resides.
4. Include the **`DPJSessionControls.jar`** file in your classpath. The file is in the following locations:
   - Windows: **`\`*`DPJ_dir`*`\SessionControlAPI`**

     where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder
   - UNIX: **`/opt/Micro Focus/DPJ/SessionControlAPI`**
5. Modify the HTML page or Java Server Page that references the applet to include a reference to this jar file. For example, using **`Test.jar`**:
   - Before: `<PARAM name="java_archive" value="Test.jar">`
   - After: `<PARAM name="java_archive" value="Test.jar,DPJSessionControls.jar">`
6. Run the applet under the web browser under **`nmshell`**.

Chapter 4

Configurations

A *configuration* is a set of parameters that control how a profiling session will run. Each configuration may contain the following information:

- Various general parameters, such as whether single or multiple processes are to be monitored per session, the collection level to be used, whether trivial methods are monitored, whether out-of-order thread synchronization is monitored, and whether Coverage Sessions are automatically merged.
- Whether to profile specific entry points, not the entire program.
- Packages and classes assigned to categories for analyzing results.
- Objects that are retained in memory after they are no longer needed by the program.
- The packages and classes to include or exclude from profiling.
- Session Control Rules which specify the action that DevPartner Java Edition is to take when your code enters or exits a method.
- The length of time to display thread activity in the Thread Viewer during Performance profiling.
- The source file paths for your code (required if you want to view source code during analysis).

A configuration file is owned by the DevPartner Java Edition server. It is accessible and can be modified by any user who has access to the DevPartner server. You should not edit a configuration file outside of the Edition server.

More than one session file can be associated with a given configuration.

Use the **Configurations** tab of the Start page to create configurations, or use the Default configuration. The Default configuration includes standard configuration settings.

To specify a configuration in a command line, include `-config` ***`Configuration_name`*** in the list of parameters for your command line, where ***`Configuration_name`*** is the name of your configuration.

Configuration files are stored in the following folders:

- Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\configurations`**

Note: By default, the **`Application Data`** folder is hidden. To display the **`configurations`** folder and its contents, type the path in the Address bar of Windows Explorer and press **Enter**.

- Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\configurations`**
- UNIX — ***`DPJ_dir`*****`/var/configurations`**
  where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder

The session files associated with a configuration are stored in the following folders, in a folder with the same name as the configuration file that generated the session files:

- Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Appli-cation Data\Micro Focus\DevPartner Java Edition\var\sessionfiles`**
- Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\sessionfiles`**
- UNIX — **`/opt/Micro Focus/DPJ/var/sessionfiles`**

You can copy your configuration and session files from one computer to another outside of the DevPartner Java Edition client. Keep in mind that the content of a session file is controlled by its configuration file; be careful when moving configuration and/or session files.

Also be careful to keep the folder structures intact when moving configurations and session files outside of the client. Otherwise, DevPartner Java Edition might not detect configuration/ session file(s) you imported. Or, if you move a session file to a folder associated with another configuration file, the session file might not behave as you expect.

Creating and Managing Configurations

To create a configuration, select the **Configurations** tab on the Start page. DevPartner Java Edition displays the values in the current configuration.

Note: You cannot modify or delete the default configuration.

To view a configuration, select it from the list. To create or manage configurations, use the following options.

Table 4-1. Options to Create or Manage Configurations

Option	Description
New	You are prompted for the name of the new configuration. After you specify the name, DevPartner Java Edition displays the General Parameters page in which you can adjust the parameters of the configuration.
Copy	Create a copy of the configuration. This option is useful if you have already created a configuration and specified numerous file names or session control rules, and you want to create a similar configuration without having to redo all that work. You are prompted for the name of the copy and can then change any of its configuration parameters.
Delete	Remove the selected configuration from the server. **Caution:** This option also deletes any session files created under this configu-ration.

A configuration file name can include alphanumeric characters, underscores, hyphens, and periods. A configuration is accessible by and can be modified by any user who has access to the server.

To modify a configuration, select the configuration from the list and make the desired changes.

DevPartner Java Edition automatically saves all changes to a configuration as you make them.

To cancel your changes, click **Undo Changes**.

Viewing Configurations

There are three scenarios in which DevPartner Java Edition displays configurations as read-only, so that you cannot change the configuration settings.

- **Default** — DevPartner Java Edition provides a Default configuration file that includes configuration settings typical for most simple memory and coverage scenarios. These settings can be viewed by selecting **Default** from the **Configuration** list in the **Configurations** tab of the Start page.
- **Active Session** — The configuration associated with the currently active session cannot be modified.
- **Results Session** — The configuration associated with a session file that is being viewed (i.e., the contents are currently displayed in any Results Summary) cannot be modified.

Configuration — General Parameters

You can specify the following general parameters in a configuration file.

Table 4-2. Configuration — General Parameters

Parameter	Description
Processes per Session	Select whether to create a new session for every process that is run, or to allow multiple processes to run in a single session. Note that Memory analysis always generates a new session for each process.
Collection level	For Performance analysis and Memory analysis, select whether to collect only method-level data (which is faster), or to collect source-level data (which produces a more detailed analysis). • If you collect method level data, information about each method called during analysis is displayed. • If you collect source level data, more detailed information about each line of source code executed during analysis displayed. Note that for Coverage analysis, only source-level data collection is supported. If source level data is not collected for Memory analysis, source code will not be accessible during analysis. If you change this setting, you will need to restart your application server for the change to take effect.

Table 4-2. (Continued)Configuration — General Parameters

Parameter	Description
Monitor “trivial” methods (See Note.)	When selected, all methods are monitored including trivial methods. If not selected, DevPartner Java Edition profiles only those methods that are most important to your analysis.
Monitor out-of-order thread synchronization	When selected, and when this configuration is used for Coverage analysis, DevPartner Java Edition tracks whether threads are synchronized out of order, (which can cause deadlocks).
Automatically merge Coverage Sessions	When selected, and when the analysis type is Coverage, merges all session files created with the current configuration. You must enter a file name for the Merge File that will be created.

Trivial Methods

A *trivial method* 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.

These characteristics describe approximately 2,400 functions out of the standard Java library and javax.swing. Some examples of trivial methods are:

```
public String getName() { return name; }

public void setName( String s ) { name = s; }

public String toString() { return getName(); }
```

By default, trivial methods are not monitored; omitting them from the profiling session significantly reduces data collection overhead. To monitor all methods, select **General** in the **Configurations** tab of the Start page, then select **Monitor “trivial” methods**.

Configuration — API Categorization and Transaction

You can focus Performance analysis on just the parts of your program that concern you by selecting entry points to profile and by assigning categories to classes and packages.

Using Entry Points

By default, DevPartner Java Edition profiles all transactions except for those involving packages and classes in the exclude list for the configuration. The session file may, therefore, include much data that is of no particular interest to you. You can use the Entry Point Tracking list to include only specified objects in the session file. This feature enables you to focus on the objects that may be causing performance problems, without the distraction of irrelevant details.

The same profiling information is displayed in the Performance Results Summary as for an unfiltered session.

To track specific entry points, select the **Use Entry Point Tracking to Only Profile Transactions** option, then click **Add** to add entry points to the list.

Assigning Categories

Assigning objects to categories enables you to group related objects so the data for those objects can be displayed together instead of being scattered among the overall results. For example, if you change one area of code, you can assign all the affected objects to the same category.

The profile information is displayed in a pie chart on the Performance Results Summary. It can also be displayed through the Call Graph for any object listed in the results.

Assign Categories to Classes and Packages is enabled by default. The configuration includes a list of objects assigned to categories and a list of the categories to which objects can be assigned. You can create your own categories as needed.

Creating a New Category

To create a new category, click **Add** next to the **Categories** list. The first character of the category name must be a letter. The name can contain only alphanumeric characters and underscores, without spaces.

Configuration — Object Retention

Normally, objects that were garbage-collected are not included in session files. The session file for a memory analysis, therefore, may not include all objects that affect program performance by being retained in memory longer than necessary. You can use the Object Retention configuration to detect the objects that are retained in memory for the longest time after their last use by the program.

When you enable Object Retention, you can also choose whether to retain information for classes that would otherwise be excluded from the analysis.

You can specify the number of objects to retain in the session file after they are garbage collected; the default is 50. The more objects you track, the larger the session file will be. DevPartner Java Edition ranks retained object by amount of overhead, calculating the overhead by multiplying the size of the object by the retention span (number of garbage collections it survives).

When you enable object retention, session results are displayed in the Object-Lifetimes Results Summary tab.

Notes:

- Using this feature requires large overhead, so by default, Object Retention is disabled.
- If you are using a generic application server configured through the DevPartner Java Edition Administration Console and you selected the option **Detach session before terminating app server** for the server, the results of object retention analysis may be incomplete or incorrect because the session may end before all objects are profiled.
- Because an instance is marked as retained depending on when the instance is used via a method invocation, primitive data types and arrays are not tracked for retention. If an instance is used via a public member *variable*, it is not tracked as used because it relies only on method invocation on the instances.

Configuration — Packages and Classes

You can simplify your analysis and improve the speed of monitoring by decreasing the amount of analysis data collected. You can reduce the amount of data collected by specifying the packages and classes to be monitored. All other packages and classes are not analyzed.

By default, DevPartner Java Edition does not monitor the Java Runtime Environment, application server, and IDE classes.

To specify the packages and classes to be monitored, display the Packages And Classes options in the **Configurations** tab of the Start page.

- ◆ To exclude specific packages and classes from analysis, select **Collect data for everything except**.
- ◆ To limit the data collected to a specific set of packages and classes, select **Collect data only for**.

Note: These options are mutually exclusive.

Use **Add**, **Modify**, and **Remove** to change the excluded or included packages and classes.

If you change exclusions/inclusions between detaching from and attaching to an application server, you need to redeploy the application being tested for the changes to exclusions/inclusions to take effect.

Exclusions and inclusions can take two forms: simple wildcard and regular expression.

- ◆ You can enter **one** wildcard character (*) anywhere in a package or class name. Any subsequent wildcard characters are interpreted as actual asterisk characters, not wildcard characters, in the package or class name. Use standard dotted notation for packages or classes.
- ◆ You can use a tag with syntax `@REGEX(<pattern>)` to specify a set of packages or classes that follow the naming pattern expressed in a regular expression. For more information, see “Regular Expressions” on page 65.

Note: The regular expression `@REGEX(^com\.ibm\.(?!_jsp))` is provided in the Exclusion list by default.

To restore the default exclusion list, click **Restore Defaults**.

Default Exclusions

The default exclusions change from release to release, based on the packages associated with supported application servers. Reinstalling the same release or installing a newer release into the existing installation folder **does not overwrite** the existing default or user-created configurations. Therefore, all your existing settings will be preserved. If you do not need to preserve your settings and you want to adopt the product defaults when reinstalling, do one of the following:

- Remove the existing configuration files from the configurations folder to reuse the same installation folder:
  - Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\configurations`**

Note: By default, the **`Application Data`** folder is hidden. To display the **`configurations`** folder and its contents, type the path in the Address bar of Windows Explorer and press **Enter**.

  - Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\configurations`**
  - UNIX — ***`DPJ_dir`*`/var/configurations`**
    where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder
- Install DevPartner Java Edition into a different installation folder.

Note: If no configuration files are detected during installation, the default configuration will be provided automatically.

Regular Expressions

You can exclude or include subsets of packages and classes in a profiling session by using Perl-compatible regular expressions in the Exclusion or Inclusion list in the Packages and Classes configuration. The regular expression pattern will be expanded against the names of any packages or classes profiled by DevPartner Java Edition.

To add a regular expression, click **Add** for either the Exclusion or Inclusion section, as appropriate, to display the **Package or Class** dialog box. To add a regular expression to an item, select the item and click **Modify**; the dialog box displays the selected item in the editing field.

A regular expression has the syntax

```
@REGEX(<pattern>)
```

where `<pattern>` defines the subset of packages or classes to be excluded or included.

For example:

- In an Inclusion pattern, to include all **`com.compuware.vj2ee`** classes in which the subpackage starts with **s**, enter the following in the **Collect data only for** field:

  ```
  @REGEX(^com\.compuware\.vj2ee\.s+).
  ```

Subpackages with names like **`com.compuware.vj2ee.serer`** and **`com.compuware.vj2ee.snmp`** will be included, while all other **`com.compuware.vj2ee.*`** packages will not be.

- In an Exclusion pattern, to exclude all **`com.ibm`** packages, *except* where the subpackage name is **`_jsp`**, enter the following in the **Collect data for everything except** field:

  ```
  @REGEX(^com\.ibm\.(?!_jsp))
  ```

  In this example, all **`com.ibm.*`** packages will be excluded except for those starting with **`com.ibm._jsp`**. This example would include the JSP-compiled servlets under WebSphere 6.*x*.

Note: This expression is provided in the Exclusion list by default.

- To exclude all **`org.apache.catalina`** packages, except where the subpackage name is **`servlets`**, enter the following in the **Collect data for everything except** field:

  ```
  @REGEX(^org\.apache\.catalina\.(?!servlets))
  ```

The syntax for regular expressions includes the following tokens:

- **`^`** (caret)
  The match must start at the beginning of a string.
- **`+`** (plus sign)
  This token specifies one or more matches.
- **`\.`** (backslash followed by a period, without a space between)
  This escape sequence explicitly specifies the period.
- **`?!`** (question mark followed by exclamation point, no space between)
  The "except for" token defines exceptions to the exclusion or inclusion.

For full details regarding Perl-compatible regular expression usage, review the **perlre** (Perl Programmers Reference Guide) man page. Active drafts are available from a number of sites, including the following:

- Perl.com: The Source for Perl
- Emerson Center at Emory University
- LinuxCommand.org

Configuration — Source File Paths

DevPartner Java Edition does not require source file paths to monitor your code, but if you want to view the source file during analysis, you must provide the path.

You can provide the source file path in two ways:

- Use the **Source File Paths** option in the **Configurations** tab of the Start page to identify the paths to your source code. You can use this technique to establish paths before you run a session against this configuration.

- If you try to view a source file for which you have not specified a path, DevPartner Java Edition prompts you for the path. It adds the specified path to the configuration.

Use **Add**, **Modify**, and **Remove** to change source file paths.

Configuration — Thread Viewer

The Thread Viewer displays thread activity during Performance profiling. It is enabled by default.

Thread activity is displayed in the Thread Viewer for the number of seconds specified in the **Thread Viewer History in The Live View** option. The default value is 30 seconds; you can specify a minimum of 5 seconds through a maximum of 120 seconds.

Note: A higher value for **Thread Viewer History in The Live View** requires higher overhead for retaining the data, and may degrade performance. If Performance profiling proceeds unacceptably slowly even at lower values, you may want disable the Thread Viewer.

Configuration — Session Control Rules

Session Control Rules enable you to control data collection based on a call to a specific method, so you can get a snapshot at a precise point in the program execution even when running in batch mode.

For example, you might have a rule that takes a snapshot when the **`beginSpellCheck()`** method ends. When running interactively, you could just run a spell check and then click **Snapshot** in the interface. But when running in batch mode, the program is running unattended, in which case you obviously cannot click a button in the interface.

Session Control Rules enable you to specify that upon entry or exit from a method, DevPartner Java Edition will do one of the following:

- View Results (i.e., take a snapshot of the data collected to that point).
- Clear all collected data (note that RAM Footprint data is not cleared by this action).
- Start Leak Analysis.
- Perform Garbage Collection.

Because method names are not known by DevPartner Java Edition when a configuration is first created, you can only add a Session Control Rule after the configuration has been used to generate a session file. For more information, see “Adding a Session Control Rule” on page 68.

To modify or copy a session control rule:

1. Select a configuration in the **Configurations** tab, then select **Session Control Rules** in the left pane. A list of session control rules is displayed.
2. Select a rule in the list.

3 Click the appropriate button for the action you want to perform. The **Configuration** dialog box appears.

4 Select the desired options for **Event** and **Action**, then click **OK**.

Note: When creating a new session control rule by copying an existing rule, make all desired changes to the new rule at the time you create it, before closing the Configuration dialog box. If you modify a copy of a rule, the rule from which it was copied will also be modified.

To delete a session control rule, select the rule and click **Delete**.

Note: Deleting a rule is irreversible. When you click **Delete**, you will not be prompted to confirm that you want to delete.

If you clear the check box for **Enable Session Control Rules**, Session Control Rules will be ignored when a profile runs under the configuration.

If you change a Session Control Rule in a configuration that is to be used on an application server that you have already begun profiling, you must do one of the following:

- Restart the application server.
- Redeploy the application of interest, if the application server provides this feature (for example, WebLogic or WebSphere).

Adding a Session Control Rule

Because method names are not known by DevPartner Java Edition when a configuration is first created, you can only add a Session Control Rule after the configuration has been used to generate a session file.

To add a Session Control Rule:

1 Open a session file to display the Results Summary for the file.

2 For any graph of *methods*, click **More Details** to display the Method List.

3 In the Method List, click the method for which you want to create a rule. The Details window appears.

4 Click **Add Session Control Rule** to display the rule configuration window.

5 Select the desired options for the rule, then click **OK** to create the rule and close the window.

After a rule has been added, you can modify it through the **Configurations** tab of the Start page. For more information, see “Configuration — Session Control Rules” on page 67.

Chapter 5

Sessions

Analysis of your Java program occurs in the context of a *session*. A session begins when you start monitoring a process; it ends when all processes running in the session terminate either because the program ends or because you use the Session Control page or Session Control Rules to stop data collection.

A session is a property of the DevPartner Java Edition server on which the program was started. You can begin a session from DevPartner Java Edition, exit the user interface, start another instance of the user interface (locally or remotely), and connect to the session you started previously.

In the **Active Sessions** tab of the Start page, you can view, select, and control all Active (running) sessions.

You can have more than one user interface connected and controlling the same session at the same time. All users see the same session information simultaneously.

The data collected in a session is determined by both of the following:

- The configuration under which the session was started — The configuration determines what will be monitored, how much data will be collected, and so on.
- The analysis type specified — Memory, Performance, or Coverage.

When a session ends, or when you take a snapshot of data collected to that point, the data is stored in a session file.

Viewing Active Sessions

On the **Active Sessions** tab of the Start page, DevPartner Java Edition lists the following information:

- Sessions that are currently running for the selected configuration
- The date and time the session began
- Analysis type being performed

Click **Open** to display the Session Control page for the selected session.

Multiple browsers (multiple users) can view active sessions simultaneously. Actions taken on that session (e.g., stopping collection) are reflected in all browsers.

Session Results — Session Details Tab

The **Session Details** tab lists the precise conditions under which the session file was created. This information can be helpful when tracing obscure problems. The data is presented in a report format that can be printed or copied. The tab includes information in one or more of the following categories:

- General
- Machine
- Java Virtual Machine
- Performance Analysis
- Coverage Analysis
- Merged Coverage Files
- Memory Analysis

General Session Details

Table 5-1. General Session Details

Details	Description
Session File	Name of session file
Started	Date and time stamp when the session was started
Ended	Date and time stamp when the session was saved (ended)
Analysis Performed	Performance, Coverage (merged or not), or Memory
Configuration	Name of configuration
Session File Location	Path to session file

If there are one or more hosts in a given session:

Host: <Machine Name>

Application Server:
Application Process: <List of all Java programs>
Exit Code:

Host: <Machine Name>

Application Server:
Application Process:
Exit Code:

Application Server:
Application Process:
Exit Code:

Host: <Machine Name>

Application Server:
Application Process:
Exit Code:

Machine Session Details

Host: <Machine Name>

Processor:
# of Processors:
OS Version:

Host: <Machine Name>

Processor:
# of Processors:
OS Version:

Host: <Machine Name>

Processor:
# of Processors:
OS Version:

Java Virtual Machine Session Details

Process:

Java VM:
ClassPath:
Command Args:
java.runtime.name:
java.vm.version:
java.vm.name:
java.runtime.version:
java.class.version:
java.library.path:
java.class.path:
java.home:
os.arch:
os.name:
os.version:
user.name:
user.language:
sun.cpu.endian:
sun.os.patch.level:

Performance Analysis Session Details

# of Called Methods (with thread starts):
# of Calls:
Total Timing (as user-defined: seconds, milliseconds, microseconds):

Coverage Session Details

Percent of Lines Executed:
Number of Lines:
Number of Lines Executed:

Number of Lines Not Executed:
Percent of Methods Called:
Number of Methods:
Number of Methods Executed:
Number of Methods Not Executed:

Merged Coverage File Session Details

The current merged session file includes a current list of all session files in this format.

Table 5-2. Merged Coverage File Session Details

Unsaved Session	11/14/2004	1:39:48 PM	Username	None
Unsaved Session	11/14/2004	1:40:06 PM	Username	None
Session1.tcs	11/14/2004	1:41:05 PM	Username	None

Memory Analysis Session Details

# of Called Methods:
# of Calls:
Total Memory Allocated: xxx Bytes
Total size of user-allocated objects (as bytes and % of total VM memory):
Total size of system objects (as bytes and % of total VM memory):
Total size of available memory (as bytes and % of total VM memory):
Total # of temporary objects:
Total bytes leaked:
Total # of objects with leaks:

About Session Files

A session file is created each time you view profiling results (take a snapshot). Multiple session files can be created during one session. Data is written to a session file:

- When you take a snapshot (e.g., click **View Results**) on a Session Control page.
- When indicated by a session control rule.
- Programmatically, through execution of a method from the Session Control API.
- When the program ends.

Naming Conventions

DevPartner Java Edition automatically names session files based on the analysis type. For example, a session file might be named **`ProgramEndTemp4`**, indicating that this file contains temporary object data captured at the end of a session, and that it is the fourth such file for this configuration. A session file named **`InteractiveLeak3`** contains memory leak data captured manually while a session was ongoing, and it is the third such file for this configuration.

Note: When you analyze memory leaks or RAM footprint, DevPartner Java Edition displays the analysis results in real time. If you stop your application, DevPartner Java Edition asks whether you want to view the session file. When you click **Yes**, a temporary object session file is displayed (e.g.,**`ProgramEndTemp`**), because DevPartner Java Edition does not store historical data on memory leaks or RAM footprint, but it does store temporary object data.

Using Session Files

From the **Session Files** tab of the Start page, you can view the session files for each configuration.

You can export session file data for any analysis type. For more information, see "Exporting Session Data" on page 75.

File Errors

DevPartner Java Edition detects problems with a session file such as the following, and displays an appropriate error message:

- A session file does not contain any data.
- A session file was not created for the session.
- An attempt is made to process a second snapshot request while the first snapshot request is in process. (This error would only occur if you are calling the DevPartner Java Edition profiling API).

File Locations

A session file is automatically stored on the same computer as the application server that ran the generating session. The session file can be viewed at a later time from that computer, and can be accessed by other developers. The session files associated with a configuration are stored in the following folders, under a folder with the same name as the configuration file that generated the session files:

- Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\sessionfiles`**

Note: By default, the **`Application Data`** folder is hidden. To display the **`sessionfiles`** folder and its contents, type the path in the Address bar of Windows Explorer and press **Enter**.

- Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\sessionfiles`**
- UNIX — ***`/opt/Micro Focus/DPJ/var/sessionfiles`***

You can copy your configuration and session files from one computer to another outside of the DevPartner Java Edition client. Keep in mind that the content of a session file is controlled by its configuration file, so you must be careful when moving configuration and/or session files.

Also be careful to keep the folder structures intact when moving configurations and session files outside of the client. Otherwise, DevPartner Java Edition might not detect configuration/ session file(s) you imported; or if you move a session file to a folder associated with another configuration file, the session file might not behave as you expect.

Viewing Session Files

The **Session Files** tab of the Start page displays the following information:

- A list of all session files for the selected configuration
- The date and time the session file was created
- The analysis type

Displaying Session Files

To display the session files for another configuration, select the configuration from the list.

Options in the Session Files Tab

The following table lists the available options.

Table 5-3. Session Files Tab Options

Option	Description
Delete	Remove the selected session file from the configuration.
Merge	Display the Merge Coverage Files screen, which enables you to merge coverage data contained in multiple session files.
Open	Open the selected session file and displays its data in the Results Summary.
Refresh List	Update the list of session files for the configuration.
Export	Export the data from the selected session file to a text, XML, or HTML file.
Rename	Change the name of a session file.
Compare	Display the Compare Session Files screen, so you can select two performance or code-coverage session files to display side by side. This option is not available for Memory analysis.

Deleting a Configuration

To delete a configuration from the **Session files for** list, delete the corresponding configuration in the **Configurations** tab.

1. Select the **Configurations** tab on the DevPartner Java Edition Start page.
2. From the **Configuration** list, select the configuration.
3. Click **Delete**.

The corresponding configuration and all associated files are deleted from the **Session Files** tab.

Exporting Session Data

You can export session data to a file in text, HTML, or XML format from the command line or through the **Session Files** tab of the Start page.

Data can be exported from one session file at a time. The output file is created in the same folder as the session file and with the same name (changing the extension as appropriate).

Note: For descriptions of the exported data, see "Exported Data File Contents" on page 76.

Exporting Data from the Command Line

For information on exporting data from the command line, see "nmextract" on page 50.

Exporting Data Through the Session Files Tab

1. Select the desired configuration, then select the session file.
2. Click **Export** to display the **Export** dialog box.
3. From the **Export into** list, select the desired format.
4. The **Data** list is available for:
   - ◇ XML format for any type of analysis — Select **All** or **Summary**, depending on how much detail you want exported. For Code Coverage analysis, **Method** and **Line** options are also available to provide method-level or line-level detail as needed. The **All** option for Code Coverage includes both method-level and line-level details.

     To export metrics for Optimal Delivery Manager, select Metrics. For more information, see **`PubMetrics.pdf`** in the DevPartner Java Edition installation folder.

   Note: A sample application is provided for viewing exported line-level Code Coverage data in HTML format. The HTML display is similar to the Source View within DevPartner Java Edition. For more information about this sample application, see "Exporting and Viewing Line-Level Code Coverage Data" on page 146.

   - ◇ Text format for Code Coverage analysis — Select **All** or **Summary**, depending on how much detail you want exported.
5. By default, the session file name is entered in the **File Name** field. You can change the name if desired.
6. Click **OK** to create the file and close the dialog box. A confirming message displays the file location.

Exported Data File Contents

This topic describes the data exported by specifying different options in the **`nmextract`** utility or the **Session Files** tab of the Start page.

- Table 5-4 — All analysis types: summary data
- Table 5-5 — Performance analysis and Memory analysis: all data
- Table 5-6 — Code Coverage only: all data, method-level data, and line-level data

Summary Data for All Analysis Types

Table 5-4 describes the summary data exported in the specified formats:

- **`nmextract`**
  - A text (**`.csv`**) file (the default)
  - An HTML file (`-html`)
  - An XML file with the summary option (`-xml -summary`)
- **Session Files** tab
  - A text (**`.csv`**) file with the **Summary** option
  - An HTML file
  - An XML file with the **Summary** option

Table 5-4. Exported Summary Data

Type of Analysis	Exported Data
Performance	Top 20 entry points with the slowest average response time: • Method name • Package name • Total thread time including children • Total clock time including children • Average thread time including children • Average clock time including children • Percent of thread time including children • Execution count
Temporary objects	Top 20 entry points requiring the most temporary space: • Method name • Package name • Number of temporary bytes including children • Number of temporary objects including children • Execution count

Table 5-4. (Continued)Exported Summary Data

Type of Analysis	Exported Data
Memory footprint	Object distribution: • Size of profiled objects (in bytes) • Size of excluded objects (in bytes) • Size of JVM Reserved memory (in bytes) Classes with Most Average Live Instance Bytes Including Children: • Class name • Package name • Number of average live instance bytes including children • Number of profiled instance bytes • Count of profiled Instances • Number of total instance bytes • Count of total instances
Memory leaks	Top 20 Classes with Most Average Leaked Instance Bytes Including Children: • Class name • Package name • Number of average leaked instance bytes including children • Number of leaked instance bytes • Number of leaked instances
Code Coverage	**Note:** For **`nmextract`**, this list describes data exported *without* the `-cov` switch; for data exported *with* the `-cov` switch, see Table 5-6. • Overall coverage statistics: – Percent of methods executed – Percent of lines executed • Top 20 methods with the most lines not covered: – Method name – Package name – Number of lines not executed – Number of lines executed – Execution count • Top 20 classes with the most lines not covered: – Method name – Package name – Number of lines not executed – Number of lines executed • If the session file is a merged coverage session file, the exported file also provides the merged session history data: – Number of the merge file – Percent of methods executed – Percent of lines executed – Percent of volatility. • If out-of-order thread deadlocks were detected: – Names of the methods that experienced deadlocks – Number of deadlocks detected

Detailed Data for Performance Analysis and Memory Analysis

Table 5-5 describes the detailed data exported in XML format:

- Using **`nmextract`** with `-xml -all`
- In the **Session Files** tab with the **All** option

Table 5-5. Exported Data for Performance Analysis and Memory Analysis

Type of Analysis	Exported Data
Performance	Package-level data • Package name • Percentage of thread time in class • Percentage of thread time in method • Average clock time including children • Total clock time including children • Percentage of thread time including children • Total thread time including children • Execution count from the generated session file • Average thread time including children Class-level data — All the data provided at the package level, plus: • Class name Method-level data — All the data provided at the package level, plus: • Class name • Method name • First execution thread time • Minimum thread time • Maximum thread time • Thread time • Clock time • Wait time
Memory	• Profiled objects • Excluded objects • JVM reserved memory • Average live instance bytes including children • Total instance bytes

Detailed Code Coverage Data

Table 5-6 describes the detailed data exported to a text or XML file from a Code Coverage session.

Table 5-6. Exported Data for Code Covereage

Export Format	Detail Level and Options	Exported Data
Text (**`.csv`**)	All details • **`nmextract`**: `-cov` • **Session Files** tab: **All**	• Percentage of code covered in each object • Number of times each object was called • Number of lines not executed, for each object
XML	All details • **`nmextract`**: `-xml -all` • **Session Files** tab: **All**	• Packages – Package name – Number of methods in the package – Number of methods called – Total number of lines – Number of lines executed • Classes – Class name – Name of the containing package – Number of methods in the class – Number of methods called – Total number of lines – Number of lines executed • Methods, organized by class – Method name – Number of times called – Percentage of the code covered – Number of lines not executed • Lines, organized by source – Line number – State (4 = covered; 2 = not covered) – Number of times executed – Number of child methods
XML	Method details • **`nmextract`**: `-xml -method` • **Session Files** tab: **Method**	• The same information for packages, classes, and methods is included as listed above for **All**. • Line information is not included.
XML	Line details • **`nmextract`**: `-xml -Line` • **Session Files** tab: **Line**	• The same information for packages, classes, and lines is included as listed above for **All**. • Method information is not included.

Note: A sample application is provided for viewing exported line-level Code Coverage data in HTML format. The HTML display is similar to the Source View in the Web-based interface. For more information about this sample application, see “Exporting and Viewing Line-Level Code Coverage Data” on page 146.

Comparing Two 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.

To compare session files:

1. At the bottom of the **Session Files** tab of the Start page, click **Compare**. The Compare Session Files screen appears.
2. Select the desired configuration.
3. Select the type of analysis performed in the session.
4. Select two session files.
5. Click **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.

Chapter 6

Memory Analysis

Automatic memory management is one of the strong points of Java. Nevertheless, memory mismanagement is still possible with Java; it can degrade the performance, hamper the scalability, and weaken the robustness of your applications. The Memory analysis capabilities of DevPartner Java Edition target three principal memory problems often encountered by Java programs:

- Temporary objects — To help you analyze the scalability and performance of your program, DevPartner Java Edition tracks the use of temporary objects by your program.

  For a description of how DevPartner Java Edition defines temporary objects, see "Temporary Objects" on page 97. For a detailed description of temporary objects and their impact on memory usage and performance, see Chapter 7 of *Java Platform Performance*.

- Memory leaks — Memory leaks do not occur in Java. Programs can, however, retain references to objects that you expect to be handed by garbage collection. Memory Leak analysis helps you identify memory leaks in your program. By default, the session file does not include objects that were garbage-collected, but you have the option to include in the session configuration a specified number of garbage-collected objects that were retained the longest.

  For a description of how DevPartner Java Edition defines memory leaks, see "Memory Leaks" on page 93. For a detailed description of memory leaks, see Section 3.2.3 of *Java Platform Performance*.

- RAM Footprint — Some Java applications can use very large amounts of memory while they are running. Relying on the virtual memory of the operating system will probably degrade the performance of your Java program. RAM Footprint analysis helps you identify the components using the most memory use in your program.

  For a detailed description of RAM footprint, see Section 1.2 of *Java Platform Performance*.

To learn how to begin a Memory analysis session, see "Up and Running in 60 Seconds" on page 20.

Note: If using the Sun 5.0 JVM, Memory analysis may not work correctly if class data sharing is enabled, because of an issue in Sun's release of Java 5.0. (See Sun bug ID 5100404 at http://bugs.sun.com for more information about this issue.) The Win32 release of the Sun J2SE 5.0 enables class data sharing by default and it must be explicitly disabled. To disable class data sharing, pass -Xshare:off as a command line parameter to Java. With DevPartner Java Edition, you can also disable class data sharing by creating a global environment variable named NM_VM_OPTIONS and setting it to -Xshare:off.

If you are using WebLogic 9.*x*, you must use the Sun JDK with it. WebLogic 9.*x* uses JRockit 5.0 by default, but because of a JRockit bug, Memory analysis cannot be performed on Java applications running under this JVM.

Short-, Medium-, and Long-Lived Objects

DevPartner Java Edition groups objects that have been garbage collected into one of three categories, based on how long the particular object was alive. These categories are short-lived, medium-lived, and long-lived objects. The short-lived and medium-lived objects are considered temporary objects.

Short-Lived Objects

An object is *short-lived* if it was freed during the first or second garbage collection after it was allocated.

Short-lived objects do not generally escape generation 0. They are relatively free of performance costs other than time spent initializing them. It is recommended that you track down CPU bottlenecks, such as short-lived objects, by running a Performance analysis. There are cases, however, where it is more convenient to look for extremely short-lived objects to detect common performance errors such as using string concatenation when a `StringBuffer` would be more appropriate.

Medium-Lived Objects

An object is *medium-lived* if it survived at least two garbage collections but was freed prior to or immediately after the end of the entry point in which it was allocated. As with short-lived objects, medium-lived objects are temporary and are confined to a particular facet of the program. They are used for a long enough period of time, however, that they escape generation 0 and cause additional strain on the garbage collector. It is valuable to determine whether you can reduce the amount of time for which they remain alive, perhaps by nulling out some local variable references earlier in the algorithm, by reducing the number of medium-lived objects, or by reducing just their size.

Long-Lived Objects

An object is *long-lived* if it survived past the end of the entry point in which it was allocated, but then was eventually freed. Long-lived objects are generally a required part of your application's architecture. For example, if you allocate a JFrame or an Application Scope Session Bean, that object needs to be there for some reason that is intrinsic to your application. The option of trying to get rid of it earlier generally will not apply.

Performance and Scalability Implications

In a generational garbage collector, an object's impact on the performance or scalability of your program is mostly related to how long it lives. By categorizing freed objects as short-lived, medium-live, or long-lived, DevPartner Java Edition enables you to gain additional insights into the performance and scalability costs of the objects your code is allocating and using.

Hotspot's implementation of JVMPI currently interferes with the behavior of the garbage collector. Specifically, turning on certain memory-related events disables the generational garbage collector and falls back to using mark-and-sweep. This means that it is not that useful for DevPartner Java Edition to simply report what happened while it was watching the program under test. Instead, it employs simple heuristics to estimate how each object would have affected a generational garbage collector. Note that objects that are still alive are not reported during temporary object analysis. You can use RAM Footprint analysis to find information about them.

Example Demonstrating Object Differentiation

This example uses a JSP as an entry point to your program to illustrate the object differentiation. In this example, the JSP is implemented in the following sequence:

1. Open the text file.
2. Read all the contents into a Vector of Strings (one per line).
3. Enumerate through the last thousand lines.
4. Format these lines.
5. Write the lines out via the JspWriter `out`.

The JSP then stores the thousand Strings into a `String[]` in the session bean so that it is not necessary to re-read the text file the next time it is called. Some time later, the session bean will time out, and the thousand Strings in this example will be garbage collected.

Recall that short-lived objects are garbage collected the next time garbage collection runs. For example, any temporary Strings that were used, such as in the following line, would be deemed short-lived:

```
String thisFormattedLine = "<b>" + stringVector.get( i ) + "</b>";
```

Assume that the formatting takes long enough that several garbage collections must run while this formatting is processing. The Vector and all the Strings that were not stored into the session bean will be considered medium-lived. In other words, these Strings survived several garbage collections while the JSP was processing; but when the JSP finished, they were garbage collected.

If you wait to take a snapshot for another thirty minutes, such as when the session bean times out, the array of a thousand Strings will be considered long-lived. This means that they were freed but were retained long enough that their duration is probably necessarily the way your program functions, rather than a problem that can be fixed.

If you took a snapshot before the session timed out, those thousand Strings would be considered part of the RAM footprint of your application, although a thousand Strings might be considered relatively small in the grand scheme of things. They might also be hard to find as part of a RAM Footprint analysis. If, however, you turned on memory leak tracking in DevPartner Java Edition before executing the JSP, and then generated a snapshot before the session timed out, the thousand Strings would appear quite emphatically as memory leaks.

Memory Analysis Session Control

While your program is running (unless you are profiling in batch mode), DevPartner Java Edition displays the Session Control page. On this page, you can watch memory allocation in real time and control data collection.

The graph is a live display of the amount of available heap memory, as well as the amount of heap consumed. Color coding reveals what portion of heap consumption is attributed to excluded objects, and what portion is attributed to profiled objects. (Profiled and excluded objects are determined by settings in the configuration file for this session.)

With any of the three Memory analysis options, you can perform the following tasks:

- View the real-time graph and class list to monitor data as it is gathered.
- Force a garbage collection.
- Detach from or stop application servers.
- View session output.
- Set your preferences for the precision and units used in this display.

When you click **View** in the left pane to create a session file, monitoring continues as you view the Results Summary, until the program ends.

Use the list in the top left of the page to select your analysis type. The options and help text in the left pane vary depending on the analysis type selected.

RAM Footprint

Click **View RAM Footprint** to create a snapshot of the session and display the RAM Footprint Results Summary in a new browser window. A session file is added in the **Session Files** tab to the configuration used for this profile.

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 other operating systems, it is in ***`DPJ_dir`*****`/var/conf`**, where ***`DPJ_dir`*** is the product installation folder.

Object-Lifetimes Analysis

Object-Lifetimes analysis including analysis of temporary objects).

End the profiling session to display the **Object-Lifetimes Results Summary** and **Temporary Objects Results Summary** tabs in a new browser window. A session file is added in the **Session Files** tab to the Configuration used for this profile.

Click **Clear Collected Data** to discard accumulated statistics for temporary objects.

Note: If collecting data on retained objects, allow the program you are profiling to run unimpeded until the end of the session. Do not use View Object Lifetimes or Run Garbage Collection, because these actions may cause objects to be incorrectly identified as retained objects.

Memory Leaks

Select **Track Excluded Objects** to include objects allocated from excluded code in the profile.

Click **Start Tracking** to begin collecting data on allocation objects. Click **Stop Tracking** to stop collecting the data.

Click **View Memory Leaks** to create a snapshot of the session, and display the Memory Leaks Results Summary in a new browser window.

Select **Include RAM Footprint Information** to display the RAM Footprint Results Summary as well as the Memory Leaks Summary when you click **View Memory Leaks** or end the session.

Session Control Tabs

The lower pane of the Session Control window contains three tabs:

- Profiled Classes
- Application Servers
- Session Output

Profiled Classes

The **Profiled Classes** tab lists the classes for which information is currently being gathered. The table consists of three columns: **Class Name**, **Bytes**, and **Objects**. The **Bytes** column displays the number of bytes attributed to the class. The **Objects** column lists the number of currently instantiated objects of that class. By default, the table is sorted by the **Bytes** column. You can sort the table in ascending or descending order, by **Bytes** or **Objects**, by clicking the column head.

When you select Memory Leaks as the analysis type and click **Start Tracking**, two columns are added to the **Profiled Classes** table: **Tracked Object Bytes** and **Tracked Objects**.

The **Filter By** field above the table enables you to narrow the list of classes appearing in the table. Type a letter or string in the field and click **Apply** to display only the classes whose names begin with that letter or string.

Application Servers

The **Application Servers** tab is significant only if the current session is attached to a running application server. Controls on this tab enable you to detach or stop the application server.

Session Output

The **Session Output** tab displays a continually updated log of all the activities and commands carried out by the DevPartner Java Edition analysis server for this session.

Memory Analysis Session Control Class List

The Session Control page for a Memory analysis includes a list of classes being called. The list is updated in real time. It enables you to monitor the size and count of each class being called.

You can set the number of classes to be displayed from 10 to 100 in increments of 10.

To focus on specific classes, use the **Filter By** field to filter the class list by package or class. Wildcard characters are accepted (`.*`, applies to all classes or sub-packages).

You can sort the list by size or count, in ascending or descending order, by clicking a column heading. Sorting by class name, however, is not supported.

Memory Analysis Results Summary

The Memory Analysis Results Summary graphically displays the most significant memory consumption data.

This page includes the **Session Details** tab plus additional tabs, depending on the type of analysis you ran:

- Memory Leaks
- Object-Lifetime and Temporary Objects
- Ram Footprint

Each tab includes graphs that show the objects consuming the most memory or creating the most temporary objects.

From any of the graphs on these tabs, you can drill down for more information by clicking on one of the classes or methods, or elect to display all classes or methods.

Hints for Analyzing Data

When viewing an Instance List, you may notice that one instance seems unusually large compared to other instances of the same object. Typically, all instances of an object would have approximately the same number of bytes. This unusually large size might occur if the value was not constrained when it was initialized and an abnormally large value was passed. View an Allocation Trace Graph to determine where this object was allocated. Then display the Source View to further analyze the problem.

When viewing the Call Stack ID, you may notice that many instances are allocated in the same stack. View an Allocation Trace Graph to see where it was allocated. Then, display the Source View to see whether you can optimize it.

Classes of Leaked Objects

From the Memory Leaks Results Summary page, you can display a list of all classes of leaked objects.

Clicking on a class displays the Class window, which shows the instances of that class or the methods in that class that were leaked.

The following columns are included in this table:

- **Class** — Name of the class.
- **Package** — Package in which this class resides.
- Average Leaked Instance Bytes including Children (Bytes)—Average amount of leaked space referred to by instances of this class, expressed in bytes.
- **Leaked Instances** — Number of instances of this class that were leaked.
- **Leaked Instance Bytes (Bytes)** — Sum of the sizes of the Leaked Instances, expressed in bytes.
- **Call Paths** — Number of unique Allocation Traces (or call stacks) for the Leaked Instances of this class.

  If Call Path equals 1, all instances of this class were allocated by the same call path.

  If Call Path equals 0, there was no user method on the stack when any of the instances of this class were allocated. This can only happen in a memory leak analysis if you had selected **Track Excluded Objects** in the Session Control page during profiling.
- **Number of Direct Referrers** — Number of direct references to these leaked objects.

  If this number is high, there are many references to the leaked objects. As a result, it might be challenging to get the objects freed. If the number is low, it indicates that there are only a few references holding these leaked objects in memory, and it might be easy to get them freed.

For large lists, click **Next** or **Previous** to view additional classes.

Click **View Instances** to see all instances of these leaked objects.

Classes of Live Objects

From the RAM Footprint Results Summary, you can display a list of all classes that were live when the session file was created. Clicking on a class displays the Details window through which you can display the instances of that class or the methods in that class.

Columns in this table include:

- **Class** — Name of the class.
- **Package** — Package in which this class resides.

- **Profiled Instance Bytes (Bytes)** — Number of bytes pertaining to this class, which represents what was in memory when the session file snapshot was taken.
- **Profiled Instances** — Number of instances of this class that were in memory when the session file snapshot was taken.
- **Total Instance Bytes (Bytes)** — Sum of the sizes pertaining to the instances of this class that were in memory when the session file snapshot was taken.
- **Total Instances** — Number of profiled instances pertaining to this class that were in memory when the session file snapshot was taken.
- **Average Live Instance Bytes including Childre**n — Average amount of space referred to by instances of this class.

The current sort order is indicated by a white arrow in the header. Click a column header to change the sort order.

Note: When you sort by class name, the list expands to reveal all classes, whether or not there are live instances of a class in the program. This feature enables you to find a specific class regardless of whether or not there were any live instances of that class when the session file was created.

For large lists, click **Next** or **Previous** to view additional classes.

Classes of Retained Objects

You can display a list of the classes for the retained objects that have the longest average retention span by clicking **More Details** for the **Classes with the Longest Average Retention Duration** graph in the Object-Lifetimes Results Summary. Clicking a class in this list displays the Details window, through which you can display the instances for that class.

You can display the following columns for this table:

- **Class** — Name of the class.
- **Package** — The package in which the class resides.
- **# of Retained Objects** — The number of objects for the class that were retained in memory during the session.
- **# of Destroyed Objects** — The number of objects for the class that were freed from memory during the session.
- **Total Instances** — The total number of instances for the class that still existed when the session ended. This column is the difference between the **# of Retained Objects** and **# of Destroyed Objects** columns.
- **Average Object Retention-Span** — The average number of garbage collection cycles the instances survived.
- **Total Retention-Span** — The aggregate of all garbage collection cycles of all instances.

To view a list of instances for a class, click the class to display the Details window, then click **View Instances**.

The current sort order is indicated by a white arrow in the header. Click a column header to change the sort order.

Allocation Trace Graph

To analyze memory leaks, RAM footprint, and object retention, it is useful to understand the chain of calls that led to memory being allocated for an object. DevPartner Java Edition displays this information in the Allocation Trace Graph.

While viewing an Instance List, click an instance and select **View Allocation Trace Graph** from the Details window. Below the Instance List a graph is displayed, the nodes of which represent the method calls that led to allocation of memory for the selected instance.

The Allocation Trace Graph includes two panes:

- **Overview** — The entire path to the root object for the selected instance. Drag the rectangle to select the nodes to be displayed in the **Detail** pane.
- **Detail** — A close-up of the nodes selected in the **Overview** pane. The legend identifies the colors that identify the base node, the path leading to this allocation, and caller methods. (Caller methods do not directly lead to this allocation, but are shown for more complete program understanding.)

The nodes are displayed from left to right in the order in which they were called. Circular calls are not represented as such; each call is represented by its own node regardless of whether that method was previously called.

Each node displays details about one method. Click **Node Data Selection** to choose the details to display on each node.

Long method names are displayed as ToolTips when you hold the mouse pointer over a node.

You can drag the nodes to rearrange the graph, which is sometimes useful in complex graphs. Click **Restore Layout** to restore the default graph layout.

Use the splitter bars to resize the Instance List and **Overview** pane, for example, to increase the **Detail** pane's visible area.

Click a node to display complete details, go to the Source View, or view a Call Graph for this method. (The Source View is not available if the session file only includes method level data.)

The active Configuration file determines whether DevPartner Java Edition displays trivial methods in an Allocation Trace Graph, Call Graph, or Method List.

Note: The Allocation Trace Graph shares common attributes with the Call Graph and Object Reference Path. For more information, see "Call Graph, Allocation Trace Graph, and Object Reference Path Common Features" on page 42

Instance List

An Instance List provides information about instances related to a selected class in the Results Summary.

- For Memory Leak and RAM Footprint analyses, the list includes all live instances. A *live object* is one on which methods can be invoked. Garbage collection identifies which objects have valid references (live objects) and which do not (dead objects). After the objects are marked as live or dead, they are then compacted by the garbage collector.
- For Object-Lifetime analysis, the list includes all retained objects in the session file. The Instance List is not available for Temporary Objects.

To display all instances of a selected class, click a class in a Results Summary graph, Class List, or Source View; then, in the Details window, click **View Instances**.

To drill down, click an instance and select one of the following options (not all options are available for all Instance Lists):

- **View Object Reference Path** — Shows why the object is in memory by showing its referring objects.
- **View Allocation Trace Graph** — Shows the object that called the method that allocated the object.
- **View Call Graph** — Shows the chain of calls leading to a method call, and the methods that are subsequently called by it.
- **View Referenced Objects** — Shows a list of instances referenced from the selected object.
- **View Source Code** — Shows where the instance is allocated in the code. (The Source View is not available if the session file only includes method level data.)

Click **Column Selection** to select the columns to be displayed for each method. You can sort the Method List by clicking a column heading.

Instance List Columns

You can specify the columns to display in an Instance List by clicking **Column Selection**. The available columns vary depending on the type of analysis that is performed.

All Instance Lists

These columns are included in all lists:

- **Instance** (*or* **Description**) — An identifier for the object.

Note: In the **Referrers of Leaked Objects** list, this column is called **Referring Object**.

- **Class** — The class with which the object is associated.
- **Package** — The package containing the class.
- **Allocation Trace** — An identifier for the stack trace that is generated when the object was allocated

Note: This column is not included in the **Referrers of Leaked Objects** list.

Memory Leak and RAM Footprint

This column is available for Memory Leak and RAM Footprint analyses:

- **Referenced Bytes** — The total size of the object plus the objects for which is it solely responsible

Referrers of Leaked Objects

These columns are available for Referrers of Leaked Objects:

- **Leaked Bytes** — The total memory usage of the leaked objects referred by this instance.
- **Leaked Objects** — The total number of objects that were leaked.
- **Call Paths** — The number of unique allocation traces for the leaked objects.
- **Number of Direct Referrers** — The number of objects that hold direct references to the leaked objects.

Retained Objects

These columns are available for **Retained Objects** lists:

- **Object Retention-Span** — The number of garbage collection cycles for which the object remained in memory.
- **is Garbage Collected** — Whether the object was still alive when the profiling session ended.
- **Garbage-Collection Cycle Allocated** — The garbage collection cycle during which the object was created.
- **Garbage-Collection Cycle Last-Used** — The cycle during which the object was last accessed by the program.
- **Garbage-Collection Cycle Destroyed** — For a garbage-collected object, the cycle during which the object was removed from memory; for a live object, the garbage collection cycle that was current when the profiling session ended.

The list can be sorted by any column, in ascending or descending order, by clicking the column heading.

Object Reference Path

When analyzing memory leaks and RAM footprint, to understand why an object has not been garbage collected, you need to know which objects are holding a reference to that object. DevPartner Java Edition displays this information in the Object Reference Path.

The Object Reference Path shows you the chain of objects, leading back to a garbage collected root, that are preventing the selected object from being garbage collected. In the presence of certain collection implementations and Java inner classes, it is often the case that simply displaying the transitive closure of incoming references to an object would unnecessarily obscure this information. DevPartner Java Edition uses a filtering mechanism (loosely based on a topological sort) that follows only incoming references from objects that do not lead further away from a garbage collected root.

The Object Reference Path is available through the Instance List. Click on an instance to open the Details window, then click **View Object Reference Graph**. Below the Instance List a graph is displayed, the nodes of which represent the objects that led to the reference to the selected instance.

Use the **Color By** list to select how colors display the data:

- **Entry Point Allocation** — Whether the object was allocated during execution of an entry point or the object existed before the entry point.
- **Profiled versus Excluded Objects** — Profiled objects (objects that were instantiated in included code) and excluded objects (objects that were excluded because they were instantiated in excluded code), and the base node for the object in different colors. The list of included/excluded code can be found in the Configuration used for this application.
- **Size (with children)** — The total size used by each object and its associated children.
- **Age** — The relative age of each object. Age categories are Oldest, Old, Young, and Youngest. The legend shows the shading for each category.

For large graphs (containing over 150 links), by default DevPartner Java Edition limits the display to three levels for each branch to speed up the processing of this page. To change this value, enter a number in the **Graph Depth** field and click **Apply**. It is recommended that you increase the value in small increments until you can see all the instances. If you find that it takes too long to process the new value, click **Back**, select **Object Reference Path** again, and DevPartner Java Edition restores the default value of 3.

Click on a node to display the source code where the instance was allocated. The Source View appears in a new browser window. The Source View is not available if the session file only includes method level data.

Note: The Object Reference Path shares common attributes with the Call Graph and Allocation Trace Graph. For more information, see "Call Graph, Allocation Trace Graph, and Object Reference Path Common Features" on page 42.

Freeing Objects with Shared References

When you use the Object Reference Path to view the chain of referrers that are keeping child objects in memory, you may notice a large parent object that, if released, will free a substantial amount of memory. Before you attempt to fix the problem, you should be aware of just how much memory you will be able to free.

In the following example. `I` is a class instance. The size of each object is given in parentheses.

```
I1(40) -> I2(10) -> I3(20)
       -> I4(8)
```

```
I5(50) -> I4(8)
       -> I6(30) -> I7(30)
```

The Total Bytes Including Children reported for `I1` is 40+10+20=70. If you free `I1`, you release 70 bytes of memory. Notice that DevPartner Java Edition does not count the 8 bytes attributable to `I4`. The reason is that `I5` holds a reference to `I4`, which makes `I4` a *shared* object.

Now consider another example:

```
I1(40) -> I2(10) -> I3(20)
       -> I4(8)  -> I3(20)

I5(50) -> I4(8)
       -> I6(30) -> I7(30)
```

This example is almost identical to the first. In this case, however, if you release `I1`, you only free 50 bytes of memory, because `I4` shares a reference to `I3`, meaning that `I3` is not available for garbage collection until both `I1` and `I4` are released.

When examining a complex chain of object references, it is useful to go to the last referred object and use it as the base node to look at the Object Reference Graph. You will be able to move back up the chain of referrers to see if there are shared instances. When you view the Details window for an object, notice the **Number of Direct Referrers**. This will give an idea of how easy — or difficult — it will be to free the memory.

Memory Leaks

In Java, memory is leaked via objects. DevPartner Java Edition defines a *leaked object* as an object that was allocated at time A, but that has not been gathered by the garbage collector by later time B. Determining time A is straightforward in most cases; object allocation is either explicit, or is the result of some method call or operation. Determining time B is less straightforward, mainly because object de-allocation is not explicit in Java. It is virtually impossible for DevPartner Java Edition to deduce whether a given object has been forgotten by an application, or is simply being kept alive for future use.

Consequently, when you use the **View Results** button on the Memory Leaks Session Control page, it is important to realize that you are setting the end time (time B) for determining which objects will appear as potentially leaked. **View Results** causes DevPartner Java Edition to force a garbage collection and create a session file that shows objects that remain in memory. You can also force a garbage collection at any time as your application runs.

For more information on memory leaks, see Section 3.2.3 in *Java Platform Performance: Strategies and Tactics*.

There are two possible scenarios for locating memory leaks.

Scenario 1

The following sequence of steps represents the simplest case. For example, if your application sequentially loads JSP pages, and you expect the operation performed on the final page to result in the clearing of previously allocated objects, you can use the steps below to see if any objects were left in memory.

1 Start your application or applet using the appropriate command line utility with the `-mem` option. If the code you want to profile runs on an application server, you can use the Application Server Testing tab on the Start page to start the session.

2 On the Session Control page, select **Memory Leaks**.

3 Warm up your application by exercising the functions you plan to test.

4 Click **Start Tracking** to begin analysis. All objects allocated beginning from that point will be tracked.

5 Exercise your application and perform whatever functions should clear previously allocated memory.

6 Click **View Results**. Objects still in memory will be reported as leaked.

Scenario 2

If you click **View Leaks** before **Stop Tracking** as described above, all memory allocated and not freed will be reported as a leak. The following sequence of steps is appropriate for any Java application or applet for which you normally expect memory to be freed by allocating new sets of objects. In this scenario, you track object allocation as you exercise the features you want to test, then stop tracking and exercise the application again. Because you are no longer tracking memory allocation, none of the second set of objects will be included in the session data when you view the leaks, but any objects allocated while tracking that have not been cleared will show up as leaked objects.

You can also use **Stop Tracking** before viewing the leaks in situations where you expect the next operation performed by your program to clear old objects but also allocate additional objects, thus creating unnecessarily complex data. Similarly, you can use **Stop Tracking** to limit data collection to specific parts of your application in subsequent analysis sessions.

Note: You can focus data collection with greater precision by using the Session Control API.

1 Start your application or applet using the appropriate command line utility with the `-mem` option.

If you are collecting data for an applet, use **`nmappletviewer`** if you normally start the applet using **`appletviewer`**. If your applet requires a browser, use **`nmshell`** to launch a console prompt and start an instance of the browser from the console window. (Close all open browser instances before running **`nmshell`**.)

2 On the Session Control page, select **Memory Leaks**.

3 Warm up your application by exercising the functions you plan to test.

4 Click **Start Tracking** to begin analysis. All objects allocated beginning from that point will be tracked.

5 Exercise your application.

6 Click **Stop Tracking**. Newly allocated objects will no longer be tracked.

7 Exercise your program to perform whatever function should clear previously allocated memory.

8 Click **View Leaks**. Any memory that was allocated between **Start** and **Stop** and has not yet been freed will be reported as a leak.

Memory Leaks Results Summary

The Memory Leaks Results Summary displays memory leak analysis data. DevPartner Java Edition defines a *memory leak* as any object allocated during a specified period of time that has not been freed at the point at which you take a snapshot of the memory data. Memory leak analysis helps to reveal cases where memory has not been freed when you expect it to be freed, or has not been freed at all after garbage collection.

This summary page displays 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. Which graphs are most useful in a given situation will depend upon the nature or complexity of the results data, your knowledge of the source code, and your preference in the way you think about your code.

Classes with the Most Average Leaked Instance Bytes Including Children

This graph shows classes ordered by the greatest average number of leaked bytes per instance of the class, including leaked bytes attributable to child classes. These classes have the highest ratio of leaked bytes per instance of the class. The graph shows the average amount of leaked memory you can expect to reclaim by eliminating the leak in an instance of the class. If Memory Leak analysis shows that a number of classes are associated with leaked memory, this graph enables you to focus on classes for which fixing a relatively small number of instances can potentially have the largest impact on the amount of memory your application is leaking. It is important to note that this graph ranks classes based on the highest average number of leaked bytes. These classes are not necessarily the classes associated with the most leaked bytes.

Click a class to displays its Details window; in this window, click **View Instances** to display a list of Leaked Instances for the class.

Click **More Details** to display a list of all classes that leaked objects.

Objects that Refer to the Most Leaked Bytes

To understand how much memory is being leaked and where in your application the leaks are occurring, it helps to see which objects in your application are holding references to the most leaked bytes. The purpose of the garbage collector is to free the memory used by an object, but an object cannot be garbage collected when there are references to it. Some objects may be necessary as long as the program runs, while others are temporary. For optimized performance, objects should be freed for garbage collection when they are no longer needed by the program.

This graph shows the top five objects that are responsible for holding references to the leaked objects. It is important to identify these types of objects as they cause the greatest memory impact.

Click a referring object to displays its Details window; in this window, click **View Instances** to display a list of Leaked Objects Referenced from the object.

Click **More Details** to display a list of all referrers of leaked objects.

Classes with the Most Leaked Bytes

This graph focuses on classes responsible for the most leaked memory. It shows the top five classes that allocated objects that were never garbage collected, and whose leaked bytes consume the most memory. It provides a different perspective to the same problem: which classes allocated objects that have not been freed for garbage collection. It helps you see the kinds of objects that were leaked, as well as the number of instances of classes that leaked memory. This graph differs from the **Classes with the Most Average Leaked Instance Bytes Including Children** graph at the top of the Results Summary in that it does not include children; and it is based not on class averages, but on the total number of bytes allocated by that class.

Click a class to display the Details window for that class; in this window, click **View Instances** to display the Instance List for the class.

Click **More Details** displays a list of all classes of leaked objects.

Methods with the Most Leaked Bytes

This graph shows the top five methods that allocated objects that were leaked.

Click a method to display the Details window; through this window, you can display the Call Graph or the source code for the method.

Click **More Details** to view the Method List.

Computing Average Bytes Including Children

When you view the **Classes with the Most Average Leaked Instance Bytes Including Children** graph on the Memory Leaks Results Summary, or the **Classes with the Most Average Live Instance Bytes Including Children** graph on the RAM Footprint Results Summary, DevPartner Java Edition displays data in terms of an average of the bytes of memory consumed by leaked (Memory Leak analysis) or live (RAM Footprint analysis) objects. It also displays the averaged data in the corresponding data column in the Class List displayed through the **More Details** link below the graph, and in the Details window.

DevPartner Java Edition computes the *average including children* by calculating the sum of the bytes allocated for all instances of an object or class plus all bytes allocated for all children of the object or class, and dividing that sum by the number of instances of the object or class. Providing this average highlights classes or objects that are responsible for a large amount of allocated memory, relative to the number of instances. It is useful for identifying the places in a program where fixing a small number of instances can have a significant impact on memory consumption.

Averaging does not always indicate the top memory consumers in your program, but it can identify the large consumers that are associated with relatively few instances.

Note: When you are deciding which objects you want to free to save memory, consider the possible effect of shared references to the objects you want to free. For more information, see Freeing Objects with Shared References.

Object Retention

In Java, memory is leaked via objects. When memory is allocated for an object but not released when the object is no longer needed, overhead increases unnecessarily and may affect your program's performance.

Normally, the memory is freed when an object no longer used by the program is garbage collected. Some objects, however, may survive garbage collection. DevPartner Java Edition can identify the *retained objects* that are using the most memory. To include retained objects in the session file, select the **Enable Object Retention** option for the session configuration.

Temporary Objects

Java does not allocate new objects onto the thread stack in the same way as C or C++. Instead, all objects are allocated into a heap. While modern garbage collectors are designed so that this is a relatively cheap operation, excessive object creation has often proven to be the major performance or scalability issue in a Java application. (Even if you have a generational garbage collector, methods allocating many short-lived objects often indicate easy-to-fix performance problems, such as the famous String Concatenation example.)

DevPartner Java Edition tracks object allocations done by your code and categorizes them based on collection times. The three categories are *short-lived*, *medium-lived*, and *long-lived*. Because long-lived objects are rarely a problem, a *temporary* category is also provided; it is the sum of the short-lived and medium-lived object allocations. Thus, *temporary objects* are the short-lived objects and medium-lived objects, considered as a single group.

Short-lived objects are almost free in terms of their impact on garbage collection although there is still a performance penalty for calling the object's constructor.

Medium-lived objects cause the garbage collector to work harder than necessary. They consume CPU cycles in such 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` in heavy-load situations.

Long-lived objects are in use for a long time, such as user interface widgets or application scope JSP beans.

DevPartner Java Edition enables you to drill into your program’s use of temporary objects (that is, short- and medium-lived objects) to identify problems and improve the overall quality of your code.

For an in-depth description of temporary objects and their impact on memory usage and performance, refer to Chapter 7 of *Java Platform Performance*.

Object-Lifetimes Results Summary

The Object-Lifetimes Results Summary depicts the objects that remain in memory the longest time after they are no longer used by the program.

Note: Because an instance is marked as retained depending on when the instance is used via a method invocation, primitive data types and arrays are not tracked for retention. If an instance is used via a public member variable, it is not tracked as used because it relies only on method invocation on the instances.

This summary page displays three graphs:

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

Objects Retained the Longest

This graph shows the five objects that were retained in memory for the longest time, whether they were freed before the session ended or were still live. The number at the end of each bar indicates the number of garbage collections the object survived. Click an object to open the Details window, through which you can display the Allocation Trace Graph for the object or view its source code.

Click **More Details** to display a more complete list of retained objects.

Classes with the Longest Average Retention Duration

This graph shows the classes whose instances had the longest average duration. Individual instances that are retained the longest may be less of a problem than the combined effect of these averaged instances, so it is helpful to analyze memory usage by class. Click a class to open the Details window, through which you can display the Instance List for the class.

Click **More Details** to display a more complete list of classes.

Entry Points with the Most Retained Instances

This graph shows the entry points that, on average, allocated the largest amount of retained bytes. The associated windows display statistics for the retained objects including child objects, so you can analyze how the entry points manage memory usage. Click an entry point to open the Details window, through which you can display the Call Graph or the Instance List for the entry point.

Click **More Details** to display a more complete list of entry points.

Note: The Temporary Objects Results Summary also includes a graph that provides information about entry points.

Temporary Objects Results Summary

The Temporary Objects Results Summary highlights the code that allocates the most temporary space. This tab is associated with the Object-Lifetimes Results Summary.

This summary includes two graphs:

- Entry Points requiring the Most Temporary Space
- Methods requiring the Most Temporary Space

Entry Points requiring the Most Temporary Space

This graph shows entry points whose bytes occupied the most temporary space since the beginning of the application, or from the last time that data was cleared.

Click an entry point to display the Details window, through which you can display a Call Graph. From the Call Graph, you can select a method and view the source code.

Click **More Details** to display a list of all user-code entry points.

Methods requiring the Most Temporary Space

This graph shows methods whose bytes occupied the most temporary space since the beginning of the application, or from the last time that data was cleared.

Click a method to displays the Details window, through which you can display a Call Graph or source code.

Click **More Details** to display the complete list of methods called by your program.

RAM Footprint

RAM footprint is the total amount of memory your program needs to run. If the program consumes too much memory, it might become necessary for the operating system to alternately depend upon virtual memory to run a program.

Virtual memory is a mechanism that quite literally tricks a program into believing that there is more physical RAM than really exists. But it plays that trick by copying portions of physical RAM (blocks called *pages*) to and from the system's hard disk. As a program uses more virtual memory, more and more pages must be swapped to and from the hard disk. Although modern hard drives are very fast, they are still far slower than dynamic RAM. This swapping impedes a program's performance; the more swapping that takes place, the slower the response time.

To optimize the RAM footprint, you need to examine several factors that affect it. Typical factors include the following:

- Temporary objects that have not been garbage collected
- The number of classes that need to be loaded
- Bytecodes for a class's methods that must also be loaded
- The number and size of objects, and their impact on the program's performance
- Size of data structures created inside the JVM as a result of the class's size

For more information on RAM Footprint, see Chapter 5 in *Java Platform Performance Strategies and Tactics*.

JVM Reserved Memory

JVM Reserved Memory is the amount of memory that the Java Virtual Memory reports as **allocated but unused**. For example, this memory would be allocated by the operating system for use by Java use but not actually currently in use by any objects.

JVM Reserved Memory is set with these Java options:

- `-Xms` — Sets the initial amount of JVM reserved memory
- `-Xmx` — Sets the maximum possible amount of JVM reserved memory

Note: The JVM Reserved Memory value differs from the total memory reported by the Windows Task Manager. The Windows Task Manager includes overhead for Java Virtual Memory, as well as overhead induced by DevPartner Java Edition.

Profiled Instances vs. Total Instances

By default, DevPartner Java Edition excludes Java Runtime Environment, application server, and IDE classes while monitoring your application. When you see references to *profiled* instance bytes as opposed to *total* instance bytes in a column header or in the Details window, *profiled* refers to memory allocations that occurred in your application code. Total bytes includes all allocations, whether from your code or system code.

RAM Footprint Results Summary

The RAM Footprint Results Summary shows the classes and objects that were consuming the most bytes of memory when the session file was created. This summary page includes the following graphs to help you address the parts of the application code that are consuming large amounts of memory, which can degrade application performance:

- 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

Object Distribution

It is important to have a high-level view of where the most memory is being allocated within your application, along with any other overhead. This pie chart depicts where memory was allocated when the results were generated. It shows the relative sizes of profiled objects (i.e., objects allocated in your application code), excluded objects (i.e., objects allocated in JRE, IDE, application server or other system code), and JVM Reserved Memory.

A larger wedge indicates a greater allocation of memory. If the Profiled Objects wedge is the largest in the chart, your application code allocates the most memory used by your program; examine the methods that allocated the profiled objects. If the Profiled Objects wedge is small, your application code is not the main allocator of memory.

Click **View Allocating Methods** to display the Method List, which shows statistics for every method in your user code.

Note: You can control the code that DevPartner Java Edition profiles by including or excluding packages and classes in your Configuration.

Classes with the Most Average Live Instance Bytes Including Children

To reduce memory consumption, it helps to identify the classes in your application whose instances, on average, use the most memory. This graph focuses on the classes that, with their children, are responsible for the greatest amount of memory consumed when averaged across all instances of the class.

The classes in the chart have the highest ratio of live bytes per instance of the class, that is, the classes for which a smaller number of instances is associated with a larger amount of memory. This chart enables you to focus on classes for which fixing a relatively small number of instances can potentially have the largest impact on the amount of memory your application is leaking. Click a class to display the Details window, through which you can display a list of the live instances of the class.

Objects that Refer to the Most Live Bytes

When you have a picture of how much and where your application is using the most memory, it helps to see which objects in your application are holding references to the most live objects. While a typical Java object might be small, it becomes much larger when you include memory consumed by objects it refers to, plus other overhead associated with allocation of the parent and child objects.

This graph helps you focus your tuning efforts on large object allocations in order to reduce RAM footprint. The size displayed for each object represents the total of all objects referenced from that particular instance.

Click a referring object to display the Details window, in which you can view the number of child objects for which the referring object is responsible. Through this window, you can display a list of classes that contain the child objects for which this referring object is responsible.

Click **More Details** to display a list of all the objects in memory at the time the snapshot was taken.

Classes of Profiled Instances Taking up the Most Space

This graph shows the sum of the profiled instance sizes for each class in memory when the results snapshot was generated. It provides an overview of the classes in your application code that are the largest memory consumers. For more information, see “Profiled Instances vs. Total Instances” on page 100.

Click a class to display the Details window, through which you can display the instances of this class.

Click **More Details** to display a list of all classes in memory at the time the session file was created.

Chapter 7

Performance Analysis

To produce fast, accurate software that scales to meet your users' needs, you must understand your program's computational performance as well as its memory usage.

DevPartner Java Edition enables you to quickly find performance bottlenecks anywhere in your code, third-party components, or virtual machine, even when the source code is not available. Performance profiling identifies objects that slow your program's performance: entry points with the slowest average response time and methods that use the most thread time or spend the most time waiting (e.g. on I/O or a shared resource).

The Thread Viewer provides a view of mechanical behavior of code running under the JVM. This helps to determine whether code is running as expected, and may help identify where deadlocks occur. While your program is being profiled, the Thread Viewer displays a real-time listing of active threads in a graphical depiction of when the threads are waiting, running, and terminated. The associated table shows how many monitors each thread is holding.

To get more information about deadlocks, run a Coverage analysis session with **Monitor out of order thread synchronization** enabled in the configuration.

For an in-depth description of the concepts involved in performance and memory management, refer to Chapter 1 of *Java Platform Performance, Strategies and Tactics*.

Note: This site is external to Micro Focus, which has no responsibility for the accuracy of its contents. Any questions should be directed to the proprietor of the site.

To begin a Performance profiling session, see "Up and Running in 60 Seconds" on page 20.

Thread Time

Thread time is synonymous with CPU time. For example, if Thread A executes `MethodB()`, then thread time for `MethodB()` is the amount of time the CPU spends running Thread A while that thread is executing `MethodB()`.This statistic is distinct from clock time, which is the total duration from the Thread A entering `MethodB()` through the thread exiting the method. Clock time does not take into account the fact that the system is multitasking. While Thread A is executing `MethodB()`, the CPU might suspend that Thread A, go execute one or more other threads, then return to Thread A to complete `MethodB()`. Thread time measures only the time spent in `MethodB()`.

Thread time gives a more accurate picture of the performance of a given method, because it factors out the execution of code that, although executed while the measured method is running, is not responsible for the target method's final execution duration. Note, however, that wait time — which might (or might not) be the fault of the measured method — is invisible to thread time. DevPartner Java Edition provides a separate **Wait Time** graph on the Performance Results Summary so wait time can be analyzed.

Recursive Calls

A literal profile of an application that uses recursion contains double counts of recursive calls. When DevPartner Java Edition does Performance analysis, it eliminates this duplication by detecting when it is already timing a method. It stops timing for the first method call and starts a new accumulation for the second call.

For example:

1. Method A calls Method B
2. Method B calls Method C
3. Method C calls Method B
4. Method B calls Method D

Method C takes 5 seconds to execute and all other methods take 1 second.

The total time for each method call in this example is as follows.

Called Method	Time with Children (in seconds)
A	9
B (first call)	8
C	7
B (second call)	2
D	1

Without any correction for recursion, DevPartner would calculate the total time spent in each method plus its child methods as follows.

Called Method	Time with Children (in seconds)
A	9
B	10
C	7
D	1

The two paths out of Method B take 8 seconds and 2 seconds, respectively. In these calculations, the time for Method D is counted twice. DevPartner Java Edition detects the recursion and stops timing the first call to Method B when it is called the second time. Using these calculations, the sum of the time spent in each of the two calls to Method B is equal to the total time spent in Method A plus its child methods. The resulting times are as follows.

	Time with Children (in seconds)
A	9
B	8
C	7
D	1

Performance Session Control

While the profiled program is running, DevPartner Java Edition displays the Session Control page (unless you are profiling in batch mode).

When you do Performance profiling, the top of the Session Control page displays the Thread Viewer, session controls, and three tabs that provide information about the session.

Thread Viewer

The Thread Viewer graph 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 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.

Note: The graph refreshes once every second. You may not see a thread's state change 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 thread data is not saved to a file; it persists only as long as the thread is displayed in the graph.

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 through the **Configurations** tab of the Start page. For more information, see "Configuration — Thread Viewer" on page 67.

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

Session Controls

To control data collection during Performance profiling, use the buttons below the Thread Viewer.

- **Clear Collected Data** — Zero out the data collected to that point and begin collection again at the next user-code entry point. For example, you may want to clear data that was gathered during application startup, since an application's performance during startup is often unlike its steady-state performance.
- **View Results** — Create a session file containing the data collected since the beginning of the session (or the last time you clicked **Clear Collected Data**). You can create multiple session files while the profiled program continues to run.

You may find it useful to collect data at different stages of execution. For example, you might want to compare performance of two segments of an application. Clear the collected data at the point you want to start collecting data, view results at the end of the segment, then clear again to reset the data for the next segment.

Tabs

The bottom of the Session Control page contains three tabs:

- Threads
- Application Server
- Session Output

Threads

Note: If the Thread Viewer is disabled, this tab does not appear in the Session Control page.

The table in the **Threads** tab lists the following information about the threads displayed in the Thread Viewer:

- Thread name — The thread names are listed in the same order that the threads are displayed in the live view.
- Unique ID — Each thread is automatically assigned a unique ID because multiple threads in a given process may have the same name; and when long thread names are truncated in the graph, the truncated names may be identical if the names begin with the same character string.
- Monitors Held — The number of synchronized methods or blocks of Java code currently held by the thread.
- State — The current state of the thread; valid values are Running, Waiting or Terminated.

The table is dynamic; information about a thread is displayed only as long as the thread persists in the live view.

Application Server

If the profiled code is running through an application server, the server is listed in this tab. To terminate the session and create a session file, select the server in the list and click either **Detach** or **Stop**. **Detach** ends the session but leaves the application server running. **Stop** both ends the session and terminates the application server. When you detach the application server, the server state changes to *[Available]*. You can reattach the application server through the **Application Server Testing** tab. (When you attach the application server, the Session Control page appears automatically.)

Session Output

The **Session Output** tab displays a continually updated log of all the activities and commands carried out by the DevPartner Java Edition analysis server for this session.

Viewing Session Results

To view session results, do one of the following:

- To view cumulative results without ending the profiling session, click the **View Results** button. While the results appear in the Performance Results Summary, the session information continues to be updated in the Session Control page.
- To end the profiling session without stopping the application, click **Home** to display the Start page, select the **Application Testing** tab, and click **Detach**. Use the browser's **Back** button to return to the Session Control page; a message prompts you to display the session results. Note that when you detach the application, the session information disappears from the Session Control page.

  To reattach the application, click **Home**, select the **Application Testing** tab, and click **Attach**. To view the Session Control page, click **View**; the page appears in the same browser window. (If necessary, click **Refresh** to make the **Attach** or **View** button available.)
- To stop the application and end the profiling session, click **Close** in the Bounce window. In the Session Control page, a message prompts you to display the session results. Note that when you stop the application, the session information remains displayed in the Session Control page.

When you use any of these methods to display session results, the Performance Results Summary appears in a new browser window.

Performance Results Summary

The Performance Results Summary displays the most significant execution time data. From this page, you can drill down into specific areas to analyze performance bottlenecks.

The page includes the following graphs:

- Entry Points with the Slowest Average Response Time
- API Category Statistics
- Methods Using the Most Clock Time
- Methods Spending the Most Time Waiting

The Performance Results Summary also includes a **Session Details** tab, which provides detailed information about the session.

If two or more performance analyses are run using the same configuration, the results summaries from any two session files can be compared side by side.

Entry Points with the Slowest Average Response Time

The **Entry Points with the Slowest Average Response Time** graph shows the top five Entry Points that accumulated the largest average response time (i.e., the longest time to execute). Clicking on an entry point displays the Details window. This window provides a comprehensive table of the timing data collected for this entry point, as well as the links to additional areas of Performance analysis associated with this entry point.

Click **View Call Graph** to display the execution trace for this entry point. In the Call Graph, select a method node and click **View Source Code**.

Click **More Details** to open a table that lists all the Entry Points.

API Category Statistics

If **Assign Categories to Classes and Packages** is included in the configuration used for the profile, the **API Category Statistics** pie chart displays the categories assigned to the objects that were profiled. Click a category to display the Method List for each category.

Note: The **Assign Categories to Classes and Packages** option is enabled by default.

Methods Using the Most Clock Time

The **Methods Using the Most Clock Time** graph shows the methods consuming the largest percentage of time. The size of each bubble is calculated based on the number of times the method is called and the execution time. You will get the most performance improvement by analyzing the larger bubbles.

Click a method to display the Details window, through which you can display a Call Graph or source code.

Click **View Source Code** to display the Source Code and Method List. (Note that DevPartner Java Edition automatically positions the Source Code display to the Entry Point selected.)

Click **More Details** to display the complete list of methods called by your program.

Methods Spending the Most Time Waiting

The **Methods Spending the Most Time Waiting** graph shows the methods that spent the most time not executing (waiting).

Click a method to display the Details window, from which you can display a Call Graph or source code.

Click **More Details** to display the Method List and Source View.

Note: You cannot collect wait times on AIX, HP-UX, or Linux.

Chapter 8

Coverage Analysis

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

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

To learn how to begin a Coverage profiling session, see "Up and Running in 60 Seconds" on page 20.

Note: Class files that were compiled without debugging information will not be included in Coverage analysis session results. To get Coverage information on these classes, recompile them with debugging information and profile again. For example, classes compiled with the **`javac`** compiler should have the `-g` flag enabled. Classes compiled with Ant using the **`javac`** task should have the `debug="yes"` attribute set.

Any classes not compiled with symbols will not have line number tables, therefore DevPartner Java Edition cannot properly instrument the methods in the classes. Those classes will be ignored and will not appear within the Coverage session results.

Volatility

The volatility metric appears on the Merged Session History graph. *Volatility* is the measure of the rate of change, from session to session, of the source code being analyzed. This metric is expressed as a percentage. For a given session, the percentage is calculated by dividing the total number of changed methods by the total number of methods loaded for the last session. The total number of methods does not include any inactive or removed methods.

When volatility is calculated, keep in mind that a method is regarded as *changed* if any of its executable content has been altered. Volatility does not measure the extent of change in a method. Therefore, changing one line in a method has the same effect on the volatility score as changing ten lines in that same method. In addition, a method is considered changed if it has been added or removed since the last session.

The purpose of the volatility metric is to draw attention to those sessions where a large degree of change has taken place in the source code. A high volatility metric for a given session indicates significant change in the source since the preceding session. So, the first volatility metric on the graph will always be zero. Obviously, large changes in the source code make that source code more susceptible to problems As a result, you should more closely examine, test, and analyze that source code. Conversely, low volatility scores indicate few changes in source code, which suggests that the code in question has stabilized.

Out-of-Order Thread Synchronization

Detecting conditions that can cause deadlocks in your program is critical. The most typical, and hardest to find, deadlock condition occurs when threads enter synchronization regions out of order. For example, 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), a deadlock can occur. This deadlock can be sporadic, making detection difficult.

DevPartner Java Edition detects out-of-order thread synchronization in conjunction with Coverage analysis to provide a workable solution to this problem.

Note: DevPartner Java Edition does not keep track of synchronization on a *per-thread basis.* It tracks a single thread that enters a set of synchronization objects out of order and will flag it as a problem. DevPartner Java Edition also enables you to specify which threads to monitor, based on the inclusion/exclusion list.

Java programmers who are writing multi-threaded applications use various techniques to address deadlocks caused by entering synchronized objects out of order, such as the following:

- Conducting a rigorous manual code analysis to ensure that out-of-order synchronization does not exist in the code (this approach is tedious and error-prone).
- Attempting to detect a deadlock after it occurs in the code, by testing the code.

When activated, the out-of-order thread synchronization option points you to areas in your code where Java threads synchronize out of order, whether or not they result in a deadlock condition. DevPartner Java Edition detects out-of-order synchronization conditions by watching the execution paths of your application. You can use the resulting data to focus on areas in your code that could cause deadlocks.

To properly locate potential synchronization problems, you should execute as many paths in your code as possible, to ensure that no out-of-order synchronization events occur.

Monitoring Out-of-Order Thread Synchronization

By default, detecting out-of-order thread synchronization is turned off, because monitoring for this condition may significantly slow your program. To turn it on, do the following:

1. On the **Configurations** tab of the Start page, select the configuration to be used to detect out-of-order thread synchronization.
2. In the left pane, select **General**.
3. Select **Monitor out of order thread synchronization (Coverage only)**.

   When selected, out-of-order thread synchronization will be monitored in conjunction with Coverage analysis, and these conditions will be reported on the Results Summary.
4. Use this configuration to run your program under a Coverage session.

When you view a session file from this session, in addition to other Coverage analysis, you can see out-of-order thread synchronization results. Only out-of-order synchronization conditions in the paths executed in your program are reported. To accurately assess out-of-order thread synchronization, use the Coverage analysis information to ensure that the majority of your code has been tested.

Out-of-Order Thread Synchronization Analysis Results

When you run Coverage analysis with out-of-order thread synchronization enabled, the resulting Coverage session file will include out-of-order thread synchronization results. All synchronization traces that have conflicts are available for viewing in a separate window. In addition, you can quickly navigate to the suspicious areas of the source code.

The Out of Order Thread Synchronizations window shows the following:

- 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 methods or blocks, depending on the context. The trace looks like a stack trace; method names are arranged in a list, 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.
- 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.
- Synchronized block — Similar to a synchronized method in that only one thread can execute a synchronized block at any given time. The entire block of code is considered as synchronized, not just a specified method.
- 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 an integer, 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: one shows that one thread called the methods in one order, and the other shows that a different thread called the methods in a different order.

Lock tags also help when you are comparing synchronization traces of synchronization blocks. A lock tag associates a name with a synchronization block. If a synchronization block did not have an associated symbolic name, it would be harder to compare two synchronization traces.

Note: DevPartner Java Edition labels the synchronization traces with the name of the last method executed by that thread. Click an element in either the left or right pane to display the source code for that element on the **Source Code** pane.

The **Synchronization Trace** list in the left pane and the **Conflicts with** list in the right pane enable you to select a specific sequence.

Only one conflict pair can display at a time. Select the primary sequence in the **Synchronization Trace** list, and the **Conflicts with** list is populated with all the secondary sequences that conflict with the current primary sequence so you can select which secondary trace to display.

You can determine what out-of-order thread synchronization has occurred by visually comparing the trace elements of the single trace against the trace elements of each conflicting trace. You can also select a trace element and go to its corresponding source line in the source file.

Any protected sequences are not reported. In a *protected sequence*, each sequence first synchronizes on the same object before committing an out-of-order thread synchronization. These sequences can never cause a deadlock.

Coverage Session Control

The Session Control page is available while your program is running and you are *not* profiling in batch mode. The options on this page enable you to focus data collection on the portions of your code that are significant to you.

On this page, you can control data collection in these ways:

- Clearing collected data will zero out the data collected to that point and will begin collection again at the next user-code entry point. This is useful, for example, to avoid collecting data on application startup.
- Viewing results creates a session file containing the data collected from the beginning of the session (or the last Clear command) to that point. You can create multiple session files during program execution. If merging coverage data is enabled in the configuration file, the session data will be merged with existing coverage data.

  You might collect data at different stages of execution, for example, to compare code coverage of two segments of an application. Clear collected data at the point you want to start collecting data, view results at the end of the segment, then clear again to reset the data for the next segment.
- Through the **Application Testing** tab of the Start page, you can detach from and reattach to the application to create different session files without stopping the application itself.
- If you are running code through an application server, selecting the server in the **Application Server** list and detaching will terminate the session and create a session file.

When you want to analyze the collected data, click **View Results**. A new browser window opens, displaying the Results Summary. Monitoring continues in the background until the program ends.

In the Session Control page you can also view the session output, set your preferences for the precision and units used in the results, or click **Home** to return to the Start page.

Coverage Results Summary

The Coverage Results Summary displays code coverage data in graphs. You can drill down into specific areas to determine how to improve your tests to increase code coverage.

The graphs on this tab show the methods and classes with the most untested code.

- The **Overall Coverage Statistics** section displays two sets of statistics:

  - ◇ The number of methods executed, the number of methods in the program, and the percentage of methods in the program that were executed.
  - ◇ The number of lines executed, the number of lines in the program, and the percentage of lines in the program that were executed.

- ◆ The **Method with the Most Lines Not Covered** graph shows the methods with the largest number of untested lines of code.
  - ◇ Click a method to display the Details window, from which you can display the source code for that method.
  - ◇ Click **More Details** to display the complete list of methods called by your program.
- ◆ The **Classes with the Most Lines Not Covered** graph shows the classes with the most untested methods.
  - ◇ Click a class to display the Details window, from which you can display the source code for that class.
  - ◇ Click **More Details** to display the complete list of methods called by your program.
- ◆ If you are viewing a merged session file, a **Merged Session History** graph is also displayed. This graph shows you the methods and lines covered for each merged file and the volatility (amount of change from one session to the next) of the code base.

The Coverage Results Summary includes a **Session Details** tab, which provides detailed information about the session.

If two or more coverage analyses are run using the same configuration, the results summaries from any two session files can be compared side by side.

Merging Session Files

When you are testing your application, it is unlikely that you will execute all of your code in one session. You will generally gather coverage data over several sessions and then analyze your total coverage statistics. *Merging* is the process of accumulating data from multiple sessions into a single file.

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

Performing a Merge

When you merge session files, they are listed on the **Session Files** tab and can be opened just like any other Coverage session file. DevPartner Java Edition performs the following tasks when you merge session files.

Computes the percentage of covered values for lines and methods

These values are calculated so that, as session files are merged in, the coverage reported is the union of the coverage contributed by the individual files. So, if a session file with 20% coverage is merged with a session file with 30% coverage, and if the areas covered are independent of one another, the resulting coverage calculated will be 50%. If there is a 10% overlap between the two session files, the resulting coverage will be 40%.

Each session file added to the merge file contributes data points to the Merged Session History graph on the Results Summary (one data point each for lines covered, methods covered, and volatility). If multiple session files are merged simultaneously, they are added in chronological order. Otherwise, they are added in the order you merge them in. Consequently, you can read the **Merged Session History** graph from left to right, and see how each addition of a session file has contributed to the change in net coverage.

The **Merged Session History** graph shows the aggregate merge session data for the entire target application. Similar **Merged Session History** graphs whose contents are limited to the data gathered from individual packages and classes are available from the **Merge Details** tab on the Details page. You can go to the Details page by clicking **More Details** just above the **Merged Session History** graph on the Results Summary. In the Details page, select the package or class in the left pane, then select the **Merge Details** tab in the right pane.

Calculates the % Volatility values for each source file and image

The percentage of volatility represents the percentage of methods that changed in your code between sessions. This value characterizes the stability of your code.

When Merging Causes Differences Between Files

If the code has changed between sessions, DevPartner Java Edition tracks those changes and adjusts the data accordingly, reporting the volatility of the code as well as the coverage statistics in the merged file.

There are instances when you merge files that differences might exist, for example:

- When methods have been removed from one file but not from the other. Click **Removed Methods** on the **Methods List** source tree for a list of methods that have been removed since the files were merged.
- When you merge two coverage files, and then merge again. The original merge file might have been changed or updated in the background, causing a discrepancy. DevPartner Java Edition detects the differences and indicates that the session file has changed since it was last accessed. Click **Open the updated session file** to use the most recent merged data.

Merging Existing Session Files

To merge existing session files:

1. Select the **Session Files** tab.
2. Click **Merge** to display the Merge Files page.
3. Select the Configuration used to create the session files to be merged. You cannot merge files created under different configurations.

4 Hold down the Ctrl key, then select the session files to be merged.

5 Select either **Create a new merge file** or **Add to an existing merge file**.

6 Enter or select the name for the merge file, as appropriate. If you are entering a new name, do not include a file extension; the extension `.tcm` is added automatically when the file is created.

7 Click **Merge** to add the selected session files to the specified merge file.

Automatically Merging Session Files

To automatically merge session files, select that option for the configuration:

1 Select the **Configurations** tab of the Start page.

2 In the left pane, select **General**; then select **Automatically Merge Coverage Sessions**.

3 By default, the merged file will be named `mergedCoverageData`. To specify another name, click **Change File**, then enter a name in the dialog box and click **OK** to save the name.

Whenever this Configuration is used for Coverage analysis, multiple session files will automatically be merged into one file with the specified name.

Merged Session History Graph

The **Merged Session History** graph displays the progression of values for statistics for the current merge file. Each point on the graph represents the statistical values after a merge. The **Merge History** tab displays data for the item currently selected in the tree pane:

- **% Methods Covered** — Percentage of methods that were executed from the selected item.
- **% Lines Covered** — Percentage of source code lines that were executed from the selected item.
- **% Volatility** — Percentage of methods that have changed since the last merge was performed in the current merge file. This value represents the stability of the class. If the value stays high, it will be difficult to significantly increase the percentage of code that is covered.

To see more details about a specific session file, click on one of the numbers below the **Merged Session History** graph. These numbers are the *session file axis numbers*. Clicking on a session file axis number displays the following columns below the **Merged Session History** graph:

- **File** — The axis number representing the session whose information is displayed.
- **Merge Date** — The date the session file was added to the merged session file.
- **Save Date** — The date the original session file was saved.
- **Session File** — The full path to the session file and the name of the session file that was added to the merged session file.

Merge States for Methods and Classes

When you merge sessions, DevPartner tracks the state of your methods and classes. For example, DevPartner knows when you have changed, added, or removed a method. DevPartner displays information about these states in the State column on the Method List and using filters in the Filter pane.

Methods

DevPartner Java Edition uses the following states for methods.

When you...	DevPartner marks the method as...
Create a new method	Added
Make a change to a method	Changed
Delete a method	Removed

Notes:

- Removed methods are displayed in the Removed Methods filter. They are not used to calculate coverage statistics.
- DevPartner Java Edition marks a method as changed when you make any change that affects generated code. For example, adding or removing comments will not cause DevPartner Java Edition to mark a method as changed, but splitting a line of code into two lines will result in the method being marked as changed. DevPartner Java Edition cannot distinguish between major and minor code changes.
- If a method is marked as changed, all previously accumulated line execution counts are discarded.

Classes

Classes can be loaded in one session and not in another. If a class is not loaded, DevPartner Java Edition cannot determine which methods are in the class, and cannot compare the class and its methods to find changes in relation to another session.

DevPartner Java Edition uses the following states for classes.

When you...	DevPartner marks the class as...
Add a new class	Added
Re-load a class that was present in another session in the merge file	Activated
Remove a class	Inactive

Notes:

- DevPartner Java Edition marks a class as inactive any time it is not loaded. For example, if your application uses a class but you do not load it during a session, then when you merge that session with an earlier session that did load the class, the class is marked as Inactive.
- Inactive classes are displayed in the Inactive Source filter. They are not used to calculate coverage statistics.

Merging Coverage Analysis Results with JUnit Reports

From the command line, you can generate a report that merges DevPartner Java Edition Coverage analysis results with a JUnit report. This feature uses the DevPartner Java Edition utility **`nmextract`** and the Java utility Ant.

Note: Use only JUnit test cases, not JUnit test suites. Using a JUnit test suite for your merged report may cause incorrect data.

Prerequisites

- JUnit must be installed (refer to the JUnit documentation for installation instructions), and `junit.jar` must be included in the Java classpath.
- The Ant utility must be installed (refer to the Ant documentation for installation instructions). The variable `ANT_HOME` must be set, and **`ANT_HOME/bin`** must be included in your `path` environment variable.
- Java JDK 1.4 or above must be installed, and the environment variable `JAVA_HOME` must be set to the location of your JDK.
- The Micro Focus DevPartner Java Edition Control Service (Windows) or daemon (UNIX) must be running. (If DevPartner Java Edition is installed, the service/daemon starts automatically upon computer startup.)
- The version of your Ant installation must match the version you set for `ANT_HOME` and the folder in your `PATH`. To check the Ant installation, enter `ant -version` at a command prompt.
- Your Java version must match the version you set for `JAVA_HOME` and the folder in your `PATH`. To check the Java version, enter `java -version` at a command prompt.

Note: Run the sample project to learn the steps required to generate the merged report and to confirm that the process works correctly.

Creating a Merged Report

Before generating the merged report, you must add code to your Ant script. You can copy this code from the **`build.xml`** file for the sample project into the **`build.xml`** file for your project. For more information, see "Ant Script Code for Merging JUnit and Coverage Analysis Reports" on page 120.

After editing the Ant script, perform the following steps to run the report:

1 Specify the configuration for the Coverage analysis:

a Open the DevPartner Java Edition Start page and select the **Configurations** tab.

b Click **New** to create a new configuration with the name defined by the `projectName` property that you added to the **`build.xml`** file above.

Note: You can use an existing configuration by selecting it from the **Configuration** list; the value of the `projectName` property must be identical to the selected configuration's name.

   c. In the **General** section of the **Configurations** tab, make sure **Automatically merge Coverage Sessions** is selected. No other configuration option is specifically required for the merged report.

   d. Exit the Start page.

2. Compile your Java code.

3. Compile the JUnit test code.

4. Open a command prompt and change to your project folder.

5. Execute the command

   ```
   ant report
   ```

   The Ant script (**`build.xml`**) performs the following actions:

   a. Runs the JUnit tests and DevPartner Java Edition Coverage analysis in batch mode.

   b. Creates a folder structure for the results files (***`project_dir`*** is the path of your project folder):

      `project_dir\report`
      `\dpj` (for Coverage results)
      `\xml` (for XML files of Coverage analysis and JUnit test results)
      `\html` (for the merged report)

   c. Generates XML files for the Coverage analysis and JUnit test results.

   d. From the XML files, creates the HTML file **`junit-dpj-report.html`** containing the merged report.

You can view **`junit-dpj-report.html`** in your Web browser.

The XML files in **`\dpj`** and **`\xml`** are not needed after **`junit-dpj-report.html`** is created. You can safely delete the folders and all their contents.

Sample Project — Merging Coverage Analysis Results with JUnit Reports

To learn the procedure for generating a merged report for Coverage analysis and JUnit testing, use the sample project. It is located in

`DPJ_dir``/junit-dpj-report/JUnit-DPJ-Report-Sample`

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

Prerequisites

- JUnit must be installed (refer to the JUnit documentation for installation instructions), and `junit.jar` must be included in the Java classpath.

- The Ant utility must be installed (refer to the Ant documentation for installation instructions). The variable `ANT_HOME` must be set, and **`ANT_HOME/bin`** must be included in your `path` environment variable.

- Java JDK 1.4 or above must be installed, and the environment variable `JAVA_HOME` must be set to the location of your JDK.
- The Micro Focus DevPartner Java Edition Control Service (Windows) or daemon (UNIX) must be running. (If DevPartner Java Edition is installed, the service/daemon starts automatically upon computer startup.)
- The version of your Ant installation must match the version you set for `ANT_HOME` and the folder in your `PATH`. To check the Ant installation, enter `ant -version` at a command prompt.
- Your Java version must match the version you set for `JAVA_HOME` and the folder in your `PATH`. To check the Java version, enter `java  -version` at a command prompt.

Running the Sample Project

1 Open the file **`build.xml`** (in the **`JUnit-DPJ-Report-Sample`** folder) into a text editor.

2 In the `<!-- SETUP VARIABLES -->` section, change the default values of the properties to the values for your system.

```
<property name="dpj.dir" value="DPJ_dir" />
```

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

```
<property name="junit.jar.path" location="file_location" />
```

where ***`file_location`*** is the actual location of **`junit.jar`** within the JUnit product folder

```
<property name="projectName" value="junit-DPJ-report" />
```

The value `junit-DPJ-report` is the default name of the configuration for the merged report. You can keep the default or change it, but the configuration created in the next step must have the name defined by this property.

3 Specify the configuration for the Coverage analysis:

a Open the DevPartner Java Edition Start page and select the **Configurations** tab.

b Click **New** to create a new configuration with the name defined by the `projectName` property in the **`build.xml`** file above.

Note: You can use an existing configuration by selecting it from the **Configuration** list; the value of the `projectName` property must be identical to the selected configuration's name.

c In the **General** section of the **Configurations** tab, make sure **Automatically merge Coverage Sessions** is selected. No other configuration option is specifically required for the merged report.

d Exit the Start page.

4 Open a command prompt and change to the following folder.

```
DPJ_dir/junit-dpj-report/JUnit-DPJ-Report-Sample
```

5 Execute the command

```
ant report
```

The Ant script (**`build.xml`**) performs the following actions:

a Compiles the sample Java code.

b Compiles the JUnit test code.

c Runs the JUnit tests and DevPartner Java Edition Coverage analysis in batch mode.

Note: Some of the sample JUnit tests will result in failure because one of the test samples is an example of a failed test case.

d Creates a folder structure for the results files:

`DPJ_dir`**`\junit-dpj-report/JUnit-DPJ-Report-Sample\report`**

`\dpj` (for Coverage analysis results)

`\xml` (for XML files of Coverage analysis and JUnit test results)

`\html` (for the merged report)

e Generates XML files for the Coverage analysis and JUnit test results.

f From the XML files, creates the HTML file **`junit-dpj-report.html`** containing the merged report.

You can view **`junit-dpj-report.html`** in your Web browser.

The XML files in **`\dpj`** and **`\xml`** are not needed after **`junit-dpj-report.html`** is created. You can safely delete these foldersfolders and all their contents.

Ant Script Code for Merging JUnit and Coverage Analysis Reports

To generate a merged JUnit and Code Coverage report, add the following code to your Ant script (**`build.xml`**).

```
<!-- SETUP VARIABLES -->
<property name="dpj.dir" value="DPJ_dir" />
<property name="junit.jar.path" location="file_location" />
<property name="projectName" value="junit-DPJ-report" />
```

The value ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder, and ***`file_location`*** is the actual location of **`junit.jar`** within the JUnit product folder. The value ***`junit-DPJ-report`*** is the name of the configuration you will use for the Coverage analysis. You can keep this default name or change it; make sure the value is identical to the name of the configuration you create or select for the merged report.

You can copy the following code from the **`build.xml`** file provided with the sample project. (It doesn't matter exactly where in the file you put it.) The file is located in

`DPJ_dir``/junit-dpj-report/JUnit-DPJ-Report-Sample`

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

```
<!--  RUN THE JUNIT TEST AND MAKE  _XML_ REPORT -->
<target name="test.java" depends="compile.java, compile.test">
```

```
<!-- Setup the Junit xml report folder before running junit -->
<!-- This is done through a call to Junit-DPJ Report Ant library code
(the antcall will call the general do-Junit-DPJ-Report target to do the
actual ant call into the library) -->
<antcall target="do-Junit-DPJ-Report">
  <param name="target" value="target.junit.xml.report.dir.setup"/>
</antcall>
    <junit jvm="${dpj}" printsummary="yes" haltonerror="no" halton
    failure="no" fork="true">
      <formatter type="plain" usefile="false"/>
<!-- for the junit report -->
  <formatter type="xml"/>
    <jvmarg value="-cov"/>
  <jvmarg value="-batch"/>
  <jvmarg value="-config"/>
  <jvmarg value="${projectName}"/>
<!-- These commands actually run the tests. Notice, the 'todir'
parameter has been added. This will output the xml data of the junit
results in the report dir!  -->
    <test name="testCalcSuite" todir="${target.junit.xml.report.dir}"
    />
  <test name="testCalc"   todir="${target.junit.xml.report.dir}"  />
  <test name="testCalc2"   todir="${target.junit.xml.report.dir}"  />
    <classpath refid="project.classpath">
  </classpath>
</junit>
</target>

<!--  Utility code   -->
<target name="do-Junit-DPJ-Report">
  <ant dir="${junit-dpj-report.src.dir}" inheritAll="false"
  target="${target}">
    <property name="target.report.dir"
    location="${target.report.dir}"/>
    <property name="target.junit.xml.report.dir"
    location="${target.junit.xml.report.dir}"/>
    <property name="dpj.session.dir" location="${dpj.session.dir}"/>
    <property name="dpj.configuration.name" value="${projectName}"/>
</ant>
</target>
```

Chapter 9

IDE Integration

DevPartner Java Edition integrates into integrated development environments (IDEs) for Java program development, enabling you to monitor your standalone programs from inside the IDE.

Note: DevPartner Java Edition does not profile application servers running as a service within IDEs.

Use the Java IDE Add-in Manager to integrate DevPartner Java Edition with supported IDEs.

To remove the integration from an IDE, use the Java IDE Add-in Uninstallation utility.

You can also invoke the profiler through your IDE by adding an argument to the JVM settings. For more information, see "Invoking the Profiler Through the JVM Settings" on page 21.

Using the Java IDE Add-in Manager

The following table lists the supported IDEs and the entry that appears in the Java IDE Add-in Manager.

Table 9-1. Supported IDEs in the Add-in Manager

IDE	Entry in Add-in Manager
Borland JBuilder X and 2005	Borland JBuilder
Compuware OptimalJ 4.1 and 4.2	Compuware OptimalJ Net-Beans Compuware OptimalJ Eclipse
Eclipse 3.3	Eclipse 3.3
Eclipse 3.4	Eclipse 3.4
Eclipse 3.5	Eclipse 3.5
IBM Rational Application Developer v7.0	IBM RAD v7.0
IBM Rational Application Developer v6.0	IBM RAD v7.5

To integrate any of these IDEs with DevPartner Java Edition, run the Java IDE Add-in Manager:

1 Click **Start>Programs>Micro Focus>DevPartner Java Edition>Utilities>Java IDE Add-in Manager** to open the utility.

2 From the list, select the IDE; then click **Continue**. The **Installing** dialog box appears.

3. In the field, type the path of the product folder for the IDE; or click the browse button to navigate to the folder.

4. Click **Install**. A confirmation message appears. DevPartner Java Edition is now integrated into the selected IDE.

Note: If your IDE is not listed in the Java IDE Add-in Manager, you can invoke the profiler through your IDE by adding an argument to the JVM settings. For more information, see “Invoking the Profiler Through the JVM Settings” on page 21.

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=perfor-
   mance,NM_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: 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=perfor-
mance,NM_CONFIG_NAME=test
```

Note: 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.

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.

To integrate DevPartner Java Edition with this IDE, use the Java IDE Add-in Manager. For more information, see "Using the Java IDE Add-in Manager" on page 123.

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 by selecting the **Configurations** tab of the DevPartner Java Edition Start page.

You can use OptimalJ with DevPartner Java Edition to profile the following:

- An OptimalJ application
- A specific application module
- 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 OptimalJ application, do one of the following:

- On the OptimalJ menu bar, select **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, select **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 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 Explorer [Code Model], 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 **Application Server** tab of 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 an application containing a Java class that requires only the JRE to execute:

1 Compile the code for your application.

2 In the Explorer [Code Model], select the Java class to profile.

3 On the OptimalJ menu bar, select **Debug>DevPartner**, then select the desired analysis type from the submenu: **Analyze Memory Usage**, **Analyze Performance**, or **Analyze Coverage**. OptimalJ enables the DevPartner Java Edition profiling and executes the selected class in the Java Runtime Environment.

4 OptimalJ displays the 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 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**

Eclipse

You can access DevPartner Java Edition features from within the Eclipse IDE to profile Java applications, applets, JSPs, servlets, EJBs, and Eclipse Application in PDE (Plug-in Development Environment).

To integrate DevPartner Java Edition with this IDE, use the Java IDE Add-in Manager. For more information, see "Using the Java IDE Add-in Manager" on page 123.

Profiling Within Eclipse

To run DevPartner Java Edition profiles from within Eclipse:

1 In the Eclipse Package Explorer, select the object you want to profile.

2 Do one of the following:

  - Right-click the object and select **DevPartner Java** from the menu, then select the type of analysis from the submenu.
  - Select **Run Configuration** from the **Run** menu.

  a Select **DevPartner Java Applet**, **DevPartner Java Application**, or **DevPartner Java Eclipse Application**, as appropriate.

  b In the **Name** field, specify a name for the configuration.

  c Use the tabs in the dialog box to define the configuration. In addition to the standard Eclipse tabs, the dialog box contains an **Analysis** tab to specify the type of DevPartner Java Edition analysis to run.

  d Click **Run** to begin profiling.

The program compiles and starts under Eclipse, but with DevPartner Java Edition profiling enabled. The DevPartner Java Edition user interface displays as the program runs.

Note: If your IDE test server is already running, you must stop and restart that IDE test server with DevPartner Java Edition in order for DevPartner Java Edition to properly hook into the IDE test server and profile your application.

Starting JBoss and Tomcat with Eclipse WTP

There are three ways to start JBoss and Tomcat using Eclipse WTP. From WTP Server View:

1 From the **Servers** tab, Right-click **JBoss** or **Tomcat** and choose **Profile**. 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** 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 **Profile As>Open Profile Dialog**. The Profile dialog box appears.
2. Choose **JBoss** or **Tomcat** from the server list and click **Profile**. The DevPartner Java Plug-in dialog box appears.
3. Select the desired analysis type and click **OK**.

Starting WebLogic with Eclipse WTP

After you have added your Web server application to Ecipse WTP, you can use DevPartner Java Edition to profile it.

1. In the Package Explorer pane, right-click on the project you want to profile.
2. Select **DevPartner** and select one of the following the types of analysis.
   - ◇ "DevPartner > Run on server with Performance analysis
   - ◇ "DevPartner > Run on server with Memory analysis
   - ◇ "DevPartner > Run on server with Coverage analysis

   The Server Selection dialog box appears.
3. Select an existing server or define a new server. Refer to Eclipse documentation for more information on defining a new server.
4. To end, exit and close DevPartner Java Edition.
5. Stop the server.
   - a. In Eclipse, select the **Servers** tab.
   - b. Select the **WebLogic** server entry.
   - c. Click the **stop server** icon in the upper right toolbar of the lower pane.

Note: Due to a third party issue, Eclipse must be restarted whenever trying to profile the Weblogic server on another analysis type or when starting the server without DevPartner Java Edition.

IBM Rational Application Developer

You can access DevPartner Java Edition features from within IBM Rational Application Developer.

To integrate DevPartner Java Edition with this IDE, use the Java IDE Add-in Manager. For more information, see "Using the Java IDE Add-in Manager" on page 123.

You can profile Java applications, Java Beans, applets, JSPs, servlets, and EJBs.

Note: IBM Rational Application Developer 7.5 does not support Java Beans.

- Applications or applets
- Java Beans
- Web server applications

Profiling an Application or Applet

In the **Packages** pane, select the project to be profiled; then perform one of the following procedures:

- Click the **Run** toolbar button or the **Run** menu, and select **Run**.

  a From the submenu, select **DevPartner Java Applet Analysis** or **DevPartner Java Application Analysis**, as appropriate. The **Configuration** dialog box appears.

  b In the left pane of the dialog box, locate the project; either expand it and select an existing configuration, or right-click the project to create a new configuration.

  c If you are creating a new configuration, use the **Main** tab to define it.

  d On the configuration's **Analysis** tab, select the type of analysis to perform.

  e As needed, use the other tabs to provide required information.

  f Click **Apply**, then click **Run** to run the configuration.

  When you define a configuration for profiling, the configuration is added to the **Run** menu, and you can run the profile again by selecting it from the menu.

- If a configuration is already defined for the object, click the **Run** toolbar button or the **Run** menu, then click **Run As**.

  a From the submenu, select **DevPartner Java Applet Analysis** or **DevPartner Java Application Analysis**, as appropriate, to display a list of configurations.

  b Select a configuration and click **OK** to display the **Configuration** dialog box.

  c To change the type of analysis to perform, select the type of analysis to perform from the **Analysis** tab.

  d Click **Apply** if needed, then click **Run** to run the selected configuration.

Profiling Java Beans

Note: IBM Rational Application Developer 7.5 does not support Java Beans.

In the **Packages** pane, select the project to be profiled; then click the **Run** toolbar button and perform one of the following procedures:

- From the **Run** menu, select **DevPartner Java Bean/Applet Analysis**.
  - **a** Either select an existing configuration for your application, or right-click to create a new configuration.
  - **b** On the configuration's **Analysis** tab, select the type of analysis to perform.
  - **c** Click **Apply**, then click **Run** to run the selected configuration.
- From the **Run As** menu, select **DevPartner Java Bean/Applet Analysis**. By default, DevPartner Java Edition does Performance profiling.
- Select an existing configuration from the list shown in the menu.
  - **a** To change the type of analysis to perform, select the type of analysis to perform from the configuration's **Analysis** tab.
  - **b** Click **Run** to run the selected configuration.

Profiling a Web Server Application

After you have added your Web server application to Rational Application developer, you can use DevPartner Java Edition to profile it:

1. In the **Package Explorer** pane, right-click the project to run.
2. Select **DevPartner** and select the type of analysis to run:
   - ◇ DevPartner>Run on server with Performance analysis
   - ◇ DevPartner>Run on server with Memory analysis
   - ◇ DevPartner>Run on server with Coverage analysis
3. The **Server Selection** dialog box appears. Select an existing server or define a new server. (For details, see the documentation for Rational Application Developer.)
4. To end, exit and close DevPartner Java Edition.
5. To stop the server:
   - **a** Select the **Servers** tab at the bottom of the lower right pane in Rational Application Developer.
   - **b** Select the WebSphere server entry.
   - **c** Click the **stop server** icon in the upper right toolbar of the lower pane.

You can also run Web server applications by performing the following steps:

1. Open the Web Content folder.
2. Right-click on the **`.jsp`** file to run the application.
3. Select **DevPartner** and select the type of analysis to run:
   - ◇ DevPartner>Run on server with Performance analysis
   - ◇ DevPartner>Run on server with Memory analysis
   - ◇ DevPartner>Run on server with Coverage analysis
4. The **Server Selection** dialog box appears. Select an existing server or define a new server. (For details, see the documentation for Rational Application Developer.)
5. To end, exit and close DevPartner Java Edition.
6. To stop the server:
   - a Select the **Servers** tab at the bottom of the lower right pane in Rational Application Developer.
   - b Select the WebSphere server entry.
   - c Click the **stop server** icon in the upper right toolbar of the lower pane.

Note: If your IDE test server is already running, you must stop and restart that IDE test server within DevPartner Java Edition for DevPartner Java Edition to properly hook into the IDE test server and profile your application.

Oracle JDeveloper

The Java IDE Add-in Manager cannot be used to integrate DevPartner Java Edition with Oracle JDeveloper. You can, however, invoke DevPartner Java Edition from within JDeveloper by adding the `-Xrun` or `-agentlib` argument to the JVM Arguments section in JDeveloper.

In JDeveloper 10.1.2:

1. In the **Project Properties** dialog box, create a new Profile.
2. In the left pane, select **Runner** under the Profile.
3. Do one of the following:
   - ◇ If you are using JVMPI (JDK 5.0 or below), add the `-Xrun` argument to the **Java Options** field, as described in “Using -Xrun to Invoke the Profiler” on page 22.
   - ◇ If you are using JVMTI (JDK 6.0 or above), add the -agentlib argument to the **Java Options** field, as described in “Using -agentlib to Invoke the Profiler” on page 22.
4. Click **OK** to save the change and close the dialog box.

In JDeveloper 10.1.3:

1 In the **Project Properties** dialog box, create a new Run Configuration.

2 Select the new Run Configuration and click **Edit** to open the **Edit Run Configuration** dialog box.

3 In the left pane of the dialog box, select **Launch Settings**.

4 Do one of the following:

- ◇ If you are using JVMPI (JDK 5.0 or below), add the `-Xrun` argument to the **Java Options** field, as described in "Using -Xrun to Invoke the Profiler" on page 22.
- ◇ If you are using JVMTI (JDK 6.0 or above), add the -agentlib argument to the **Java Options** field, as described in "Using -agentlib to Invoke the Profiler" on page 22.

5 Click **OK** to save the change and close the dialog box.

When you include the `-Xrun` argument in the settings, the program is compiled and run under JDeveloper, but with DevPartner Java Edition profiling enabled.

Notes:

- DevPartner Java Edition does not support the Oracle Java VM, which JDeveloper runs by default; and does not profile applications run in `-minimal` or `-vanilla` VMs.
- JSPs, servlets, and EJBs run in an embedded version of OC4J. To make sure the application terminates normally and the DevPartner Java Edition profile is not interrupted before completion, select **Embedded OC4J Server Preferences** from the JDeveloper **Tools** menu.
- If your IDE test server is already running, you must stop and restart that IDE test server with DevPartner Java Edition in order for DevPartner Java Edition to properly hook into the IDE test server and profile your application.

Using the Java IDE Add-in Uninstallation Utility

You can use the Java IDE Add-in Uninstallation utility to remove the DevPartner Java Edition plug-in for an IDE.

Note: Remove DevPartner Java Edition integration from the IDE before upgrading or uninstalling/reinstalling the IDE.

1 Click **Start>Programs>Micro Focus>DevPartner Java Edition>Utilities>Java IDE Add-in Uninstallation** to open the utility.

2 In the utility window, select the IDE from which you want to remove DevPartner Java Edition integration

3 Click **OK**. A confirmation message appears.

Chapter 10

Sample Applications

DevPartner Java Edition provides sample applications that demonstrate the types of analysis that can be performed.

These applications are located in the following folders:

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

  where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder and ***`app`*** is the folder for the application.

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

  where ***`app`*** is the folder for the application.

The sample applications cover these subject areas:

- Performance analysis
  - Computational
  - Wait time
  - Thread Viewer
- Coverage analysis
  - Part 1 — Generate coverage data
  - Part 2 — Merge coverage files
  - Exporting and Viewing Line-Level Code Coverage Data
- Memory analysis
  - Leaks
  - Retained objects
  - Temporary objects

Performance Analysis

The sample applications for Performance analysis demonstrate how to:

- Find performance bottlenecks.
- Find possible causes of excessive wait times.
- Use the Thread Viewer.

Finding Performance Bottlenecks

In the sample application **`compPerfExample`**, a set of common operations are performed on:

- Simple array
- ArrayList
- LinkedList

You can compare these operations to see which class has the best performance. You can also find the bottleneck in the program, that is, what class and method is using the most thread time. You might get the most performance improvement by changing or improving this code.

This application is located in the following folders:

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

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

- UNIX — /opt/Micro Focus/DPJ/samples/compPerfExample

To run **`compPerfExample`** from the command prompt:

1. Change the folder to

   ```
   DPJ_dir\samples\compPerfExample
   ```

2. Run the appropriate command for your operating system (as one line):

   - Windows

     ```
     nmjava -perf -cp ..;..\..\SessionControlAPI\DPJSessionCon-
     trols.jar compPerfExample.CollectionMain
     ```

   - UNIX

     ```
     nmjava -perf -cp ..:/opt/Micro Focus/DPJ/SessionControlAPI/
     DPJSessionControls.jar compPerfExample.CollectionMain
     ```

   The DevPartner Java Edition user interface opens in your browser, and displays the Session Control window.

3. When the program finishes executing, a message prompts you to display the session file. Click **Yes**. The Performance Results Summary appears.

Note: For a sample application that demonstrates the Thread Viewer in the Session Control page, see "Using the Thread Viewer in Performance Profiling " on page 141.

4. Examine the **Methods Using the Most Thread Time** graph. The bubbles represent the relative amount of thread time used by each method. The list to the right of the graph shows the ranking of the methods in terms of thread time used. In order of most to least thread time used, the ranking is **`doArrayList`**, **`LinkedList`**, and plain array.

   Note that all three bubbles are to the right of the Average Thread Time scale, meaning they have a high value for the average thread time used. But they are low on the Execution Count scale so they are not called many times.

5. Click **CollectionMain.doArrayList** to display the Details window for this method. Note the **Thread Time** and **Clock Time** statistics. They are very close in value.

6. Click **CollectionMain.doLinkedList** and **CollectionMain.doPlainArray**. Note that the **`doLinkedList`** and **`doPlainArray`** methods spend much more time executing than waiting. So they are probably running fairly efficiently.

7. In the Details window for **`doArrayList`**, click **View Source** to display the source code for the method (you may be prompted to navigate to the location).

8. In the Source Code view, click **Column Selection** and select the **Execution Count**, **% Thread Time in Method**, **Thread Time**, **Wait Time**, and **Line Number** columns. Scroll to the bottom of the source code. Note the thread time and percentage of thread time in method values for the three classes: simple array, array list, and linked list. The values are highest for the linked list and array list while the plain array has relatively low values. So, looking at the code for the linked list and array list is likely to have the most benefit in terms of performance.

9. Scan for lines highlighted in red. These are the slowest lines of code in the methods. In this sample application, the following lines are highlighted in red:

   ```
   21 array0.init

   41 nextItem = array().GetNext(tr)

   72 linkedList().populate()

   112 cd.doLinkedList()
   ```

10. Examine the number of times each of the highlighted methods is executed, the percentage of total thread time spent in that method, thread time used, and amount of time the method spent waiting to execute.

    Line 21 is executed only once and has relatively low values in the columns. Line 41 has one of the highest execution counts, and more than 50% of the thread time is spent executing this line. Line 72 is executed 1,000 times, and one third of the thread time is spent executing this line; this is the only line in the linked list code. Line 112 is executed only once, but almost half of the thread time is spent executing this line.

11. Examine the code for the array list; note that almost half the thread time is spent executing the sort method 1,000 times. In the **Thread Time** column, you can see that this method uses the most thread time of any line in the array list code. The biggest performance gain for this code might involve calling the sort method fewer times or using a more efficient sorting algorithm.

12. Examine the code for the linked list. About one third of the thread time is spent in the populate method and one third is spent getting the next item. The **`getNext`** method is called 201,000 times. Both methods take about the same number of time to execute and have the highest thread time values for the code in the linked list. Reducing the number of calls to this method might improve the performance of the linked list code.

13 Examine the code for the plain array. Note that more than 50% of the thread time is spent executing the **`getNext`** method. It is called 201,000 times. The total thread time for this method is much lower than for the methods using the most thread time in the array list and linked list code. Reducing the number of calls to this method might improve the performance of the plain array code. But you probably won't gain much performance for the plain array.

After you have made changes to the code, rerun the sample application and determine whether your changes improved the performance of the application.

Finding Excessive Wait Times

The sample application **`waitTimeExample`** demonstrates how DevPartner Java Edition analyzes wait times. This feature helps you spot system scalability hotspots, such as lock contention and disk I/O before they happen in production. This class writes to a very simple in-memory Log Object, which consists of a small circular buffer. The class runs in two modes:

- Global — All threads use a single synchronized buffer.
- Local — Each thread uses its own buffer.

These modes demonstrate how a program can spend a great deal of time waiting for access to a synchronized resource; in this sample, more time waiting than doing the actual work.

The application is located in the following folders:

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

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

Note: You cannot collect wait times on Linux or AIX.

Global Mode (With Lock Contention)

In the global mode, there is only one Log object. The method **`Log.log()`** is synchronized to prevent data corruption.

To run this sample using a single Log object:

1 At the command prompt, change the folder to

```
DPJ_dir\samples\waitTimeExample
```

2 Run the command (on one line)

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

3 When the Session Control message appears, click **Yes**.

For a sample application that demonstrates the Thread Viewer in the Session Control page, see "Using the Thread Viewer in Performance Profiling " on page 141.

4 In the Performance Results Summary, view the **Methods Spending the Most Time Waiting** graph.

Note that out of the five methods listed, **`Log.log(java.lang.String)`** and **`ThreadLogMain.run()`** spend the most time waiting.

5 Click each of the methods listed to the right of the graph, starting with **`Log.log(java.lang.String)`**. When you click a method, the Details window for that method appears. Note the Thread Time and Wait Time values for each method. The code executing **`ThreadLogMain.run()`** spends more time waiting than executing. The values for Thread Time and Wait Time for the method **`Log.log(java.lang.string)`** are much higher than those values for the last three methods shown in the graph.

6 Click **More Details**.

7 Click **Column Selection**, then select the **Wait time** and **Thread time including Children** columns for viewing.

Note in the **Wait Time** column that the **`run()`** method spends a significant amount of time waiting. The **Thread Time including Children** column shows that it also spends a significant amount of time executing. This is because five different threads are spinning, attempting to acquire the same lock.

In a real-world situation, this problem might be resolved by reducing the number of threads. This would be a good approach if it is simply a tuning parameter to your application server.

Local Mode (No Lock Contention)

Another solution common in high-performance logging applications is for each thread to perform its own logging and, if necessary, merge later. This greatly reduces the synchronization overhead of the log call itself. You could also remove the actual synchronized statements from the code, but this solution is not necessary for the sample because addressing the lock contention improved the performance/scalability of the program. (The synchronizations that did not have lock contention are not a problem.)

1 At the command prompt, change the folder if necessary:

`DPJ_dir``\samples\waitTimeExample`

2 Run the command (as one line)

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

3 When the Session Control message appears, click **Yes**.

4 On the Performance Results Summary, view the **Methods Spending the Most Time Waiting** graph.

Note that **`Log.log(java.lang.String)`** is now the largest bubble. **`ThreadLogMain.run()`** is smaller than in the global mode, but it is the second-largest bubble in local mode.

5 View the **Methods Using the Most Thread Time** graph to see which methods use the most thread time when running the sample application in local mode. The **`Log.log(java.lang.String)`** bubble is the largest in the graph; it uses the most thread time of the methods in the sample application. The position of the **`Log.log(java.lang.String)`** bubble is higher on the Execution Count scale than on the Average Thread Time scale. This tells us that execution count contributes more to the use of thread time by this method than the average thread time used by this method.

In contrast, the second largest bubble, **`ThreadLogMain.run()`**, is higher on the Average Thread Time scale than on the Execution Count scale. This tells us that on average, it uses a lot of thread time.

6 Click **`Log.log(java.lang.string)`** to open the Details window for this method. Note that the Wait Time value is much higher than the Thread Time value. This means that the method spends more time waiting to execute than actually executing.

7 Click **View Call Graph** in the Details window. The Call Graph shows that **`ThreadLog-Main.run()`** [identified in the Call Graph as **`run()`**] calls **`Log.log(java.lang.string)`** [identified in the Call Graph as **`log(java.lang.string)`**]. Since only one line connects these two methods, any changes made to **`Log.log(java.lang.string)`** will affect the **`ThreadLog-Main.run()`**.

8 In the graph, click **`Log.log(java.lang.string)`**. The Details window appears.

9 Click **View Source** to display the source code for **`Log.log(java.lang.string)`**. If prompted, browse to the path to the source file for this application.

10 Click **Column Selection**; select **Thread Time** and **Line Number**, then click **OK**.

In the Source View, DevPartner Java Edition highlights the slow line(s) of code in yellow. Details about each line of code appear in the columns to the left of the line of code. In this example, line 20 is highlighted. Note that the Wait Time value is much higher than the Thread Time value. This means that the method spends more time waiting to execute than actually executing. Also, notice that this line of code is executed 250,000 times in the sample application.

One way to improve the performance of this sample application is to reduce the number of calls to methods that are running slowly. In this sample application, reducing the number of calls to **`Log.log(java.lang.string)`** might improve performance.

Even though the local version finishes faster than the global version, the total amount of CPU time (thread time) is greater, particularly on single CPU systems. This is because this sample program is completely CPU-bound; there is no I/O.

Regardless of how many threads you can run simultaneously, the sample program will not run any faster on a single CPU system. Keep in mind that the differences between profiling on your desktop computer and profiling on a multi-processor server system means that a dual or quad processor computer could nearly double or triple the throughput. Although this example demonstrates scalability, its only bottleneck is the CPU. Therefore, there is no real advantage in having multiple threads that are capable of running concurrently.

Although this behavior occurs because **`waitTimeExample`** is a very simple example for demonstrating wait time, the example can also serve as a performance optimization lesson. In the real world, if the CPU is your bottleneck, adding threads will only slow your program down.

Using the Thread Viewer in Performance Profiling

The sample application **`threadExample`** is located in the following folders:

- Windows

  `DPJ_dir``\samples`

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

- UNIX

  `/opt/Micro Focus/DPJ/samples`

Run the Application

This demonstration uses the default settings in the session configuration. The Thread Viewer is enabled by default, and the live thread data is retained for 30 seconds.

1. At the command prompt, change the folder to

   ```
   DPJ_dir\samples
   ```

2. Run the command (as one line)

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

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

4. 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) buttons.

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

View the Session Control Page

As the Bounce application runs, its threads are displayed in the "live view" graph in the Session Control page. The names of all the currently running threads are listed down the Y-axis of the graph in alphabetical order. 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 every second. If a thread runs and terminates in less than one second, it may not appear in the graph. For example, it is likely that you will not see **`main()`** when you start the Bounce application. You may also not see state changes from waiting to running and back again (or vice versa) if a thread runs or waits for less than one second.

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. In this demonstration, a thread persists for 30 seconds, then disappears from the list.

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. The ball threads are named `BallName_1`, `BallName_2`, etc.; each new thread is designated with the next higher number.
- **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 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 “Performance Session Control” on page 105.

The thread data is available only as long as it is displayed in the Thread Viewer and table. The data is not included in the session file.

View Results

To view session results, do one of the following:

- To view cumulative results without ending the profiling session, click the **View Results** button. While the results appear in the Performance Results Summary, the session information continues to be updated in the Session Control page.
- To end the profiling session without stopping the application, click **Home** to display the Start page, select the **Application Testing** tab, and click **Detach**. Use the browser's **Back** button to return to the Session Control page; a message prompts you to display the session results. Note that when you detach the application, the session information disappears from the Session Control page.

- To reattach the application, click **Home**, select the **Application Testing** tab, and click **Attach**. To view the Session Control page, click **View**; the page appears in the same browser window. (If necessary, click **Refresh** to make the **Attach** or **View** button available.) If you detach and reattach the application, each new ball thread after reattaching is numbered incrementally from the previous session; for example, if the last thread before you detached the application was `BallName_7`, the first ball thread in the new session is `BallName_8`.
- To stop the application and end the profiling session, click **Close** in the Bounce window. In the Session Control page, a message prompts you to display the session results. Note that when you stop the application, the session information remains displayed in the Session Control page.

When you use any of these methods to display session results, the Performance Results Summary appears in a new browser window.

Coverage Analysis

The sample applications for Coverage analysis demonstrate how to:

- Generate Code Coverage data.
- Work with merged Code Coverage session files.
- Export and view line-level Code Coverage data.

Coverage Sample — Part 1: Generating Data

This sample application **`coverageExample`** takes one parameter consisting of the name of a class to load, either **`coverageExample.Class1`** (to run Part 1) or **`coverageExample.Class2`** (to run Part 2).

This application is located in the following folders:

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

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

To run the first part of this sample:

1. Change the folder to the **`coverageExample`** path.
2. Run the command (as one line)

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

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

The Coverage Results Summary presents a high-level view of the application's coverage statistics in the following sequence.

- Two bar graphs depict how many methods and how many lines respectively were covered overall. It shows what percentage of the methods are called, and what percent of lines are executed, along with actual counts.
- The number of out of order thread synchronizations found is listed.
- A graph shows the top five methods with the *least* amount of code coverage. It shows the percentage of called classes containing methods, and the percentage of lines executed, along with the actual count of lines not executed.

To determine which lines of code are not covered, and the methods that contain them:

1. Click **CoverageMain.main(java.lang.String[])**, the method with the most lines of code not covered, to view specific information about the number of lines that are covered.
2. Click **View Source Code** to display the source code for this method in a new browser window.

   Two bar graphs appear at the top of the right pane. The first graph shows the number of methods called, the total number of methods in the class, and the percentage of methods called. The second shows the number of lines of code executed in the called method, the total number of lines of code in the called method, and the percentage of lines executed.

   The right pane also shows the source code for **`CoverageMain.main(java.lang.String[])`**. Scroll down the **Execution Count** column for instances of **0** (zero), which indicates lines of code that were not executed. For ease of reference, the **Line Number** column shows the line numbers for each line of code.

   You can also display this information by clicking the hyperlink for **`CoverageMain`**, the class with the most methods/lines of code not covered, which is adjacent to the second graph on the Coverage Results Summary.

   To get an overview of which methods and lines are executed in which class, click **More Details** to the right of a graph on the Coverage Results Summary to display a window in which the left pane shows a tree structure containing classes in your program, branching each class based upon the package to which it belongs. Fully expand the tree structure if it is closed. The number to the right represents the sum of its classes' total number of lines not covered (**`CoverageMain`** and **`Class1`**). In this case, **`coverageExample`** contains four lines of code not covered, which are in **`CoverageMain`**. **`Class1`** does not have any lines of code that are not covered.

   Two bar graphs appear at the top of the right pane. The first graph shows the number of methods called, the total number of methods in the class, and the percentage of methods called. The second shows the number of lines of code executed in the called method, the total number of lines of code in the called method, and the percentage of lines executed.

   The right pane also shows the source code for the method **`CoverageMain.main(java.lang.String[])`**. Scroll down the **Execution Count** column for instances of **0** (zero), which indicates lines of code that were not executed. For ease of reference, the **Line Number** column shows the line numbers for each line of code.

   You can also display this information by clicking the hyperlink for **`CoverageMain`**, the class with the most methods/lines of code not covered, which is adjacent to the second graph on the Coverage Results Summary.

   - To select which columns to display, click **Column Selection**.

- ◇ To display just the source code window with the execution counts and line number, click **Printer Friendly Version**.

  Evaluate the need for the methods and/or lines of code that are not covered in your program. If they are needed, you should consider adding them to your test cases as your test cases currently do not exercise them adequately, if at all.

In Part 2, you will experiment with the merged file feature of DevPartner Java Edition.

Coverage Sample — Part 2: Working with Merged Files

When you performed Part 1 of the Coverage sample, DevPartner Java Edition created a Configuration file with default values, named **`coverageExample.CoverageMain`**.

To modify this configuration file:

1. In the DevPartner Java Edition interface, click **Home** to display the Start page.
2. Select the **Configurations** tab.
3. From the list of configurations, select **coverageExample.CoverageMain**.
4. Select **Automatically merge Coverage Sessions**.
5. Click **Change File** and enter a new name for the merge file in the field in the **Explorer User Prompt** dialog box. Click **OK** to save the merge file with the new name.
6. At the command prompt, change the folder to

   `DPJ_dir``\samples\coverageExample`

   where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder.
7. Run the command (as one line)

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

   A merge file is created, containing a single session.

Note: When you use `-batch`, the Session Control page is not displayed.

8. To view the merge file, select the **Session Files** tab in the DevPartner Java Edition interface.
9. From the list, select **coverageExample.CoverageMain** if it is not already selected.
10. Double-click the file name you specified as the merge file when you modified the configuration.

    The merge Results Summary resembles the Coverage Results Summary, except that it also has a Merged Session History below the **Overall Coverage Statistics** graphs. The Summary depicts the progression of your coverage statistics over time, as session files from different runs are merged. It also displays volatility, which indicates how much the code changed from run to run. In general, lower volatility is preferred, because there is less code to retest.
11. Click **Merge Details** to view the same graph broken down by class and package.

12 Select an entity in the tree in the left pane to view the corresponding graph in the right pane.

A list of removed methods and Inactive classes also appears in the tree.

13 At the command prompt (***DPJ_dir*\samples\coverageExample**), run the command (as one line)

```
nmjava -cov -batch -cp .. coverageExample.CoverageMain coverageEx-
ample.Class2
```

A second run is added to the merge file.

14 Repeat steps 8 through 12.

15 Click **Source Code**.

In this run, a new class (**`Class2`**) was loaded that was not loaded before. Also, **`Class1`** was not loaded this time.

16 Fully expand the **Inactive Source** tree.

The merge file now displays **`Class1`** as Inactive Source in the Merge Details view. Volatility is increased by the addition of the methods in **`Class2`**.

17 Comment out the method **`MethodToRemove()`** in **`Class2.java`**.

18 Recompile **`Class2.java`** by running the following command:

```
javac Class2.java
```

19 Repeat steps 8 through 12.

20 Click **Removed Methods**.

The merge file now displays **`Class2.MethodToRemove()`** in the Removed Methods node at the bottom of the tree.

21 Select **`Class2`** in the tree on the Merge Details view.

Volatility for **`Class2`** increased because the class changed since the last run.

22 Select **Inactive Source** in the tree.

The overall volatility of the application has in fact decreased, because fewer lines of code have changed between the second and third runs compared to between the first and second runs.

Exporting and Viewing Line-Level Code Coverage Data

You can export *line-level* data from Code Coverage sessions in XML format, using either the utility **`nmextract`** or the **Export** feature in the **Session Files** tab of the Start page.

The sample application **`CardGame`** demonstrates how to use the utility **`convert2HTML.jar`** to display the line-level Code Coverage data in HTML pages. You can adapt this utility to use with your reporting system (see "Adapting the Conversion Utility" on page 149).

For example, you may want to generate detailed Code Coverage reports for application source code as part of a nightly unit test. To create this report, you want to chain together your coverage-enabled test suite, the DevPartner Java Edition export facility to generate XML files of Code Coverage data, and a Web formatting tool to annotate the source files with the Code Coverage details.

Using the Example Application

Perform these steps to create a Code Coverage session file for the **`CardGame`** application, extract the line-level data, and display it in an HTML browser:

1 Run a Code Coverage session.

2 Export the session file to an XML file.

3 Use **`convert2HTML.jar`** to create HTML pages from the exported data.

4 Display the HTML report in your Web browser.

Note: In the procedures below, ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder. The location of the **`\var`** folder depends on the operating system:

- Windows XP or 2003 Server — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var`** (By default, the **`Application Data`** folder is hidden. To display the **`var`** folder and its contents, type the path in the Address bar of Windows Explorer and press **Enter**.)
- Other supported Windows operating systems — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var`**
- UNIX — ***`DPJ_dir`*****`/var/configurations`**

Run the Code Coverage Session

To run the Code Coverage session:

1 Open a command window.

2 Change to the path of the DevPartner Java Edition product folder.

3 To generate the Code Coverage session file, execute (as one line):

```
bin\nmjava -cov -jar "samples\nmExtractEx-
ample\CardGame\dist\CardGame.jar"
```

The session file **`ProgramEndx.tcs`** is generated in the folder **`\var\sessionfiles\CardGame`**. In the filename, ***`x`*** is a number generated automatically to uniquely identify the file. For example, the third time you execute the command, the file name is **`ProgramEnd3.tcs`**.

Note: When **`nmjava`** is executed, the Session Control page opens automatically. When you are prompted to view the session file, click **No**. You can close the Session Control page.

Export the Data to an XML File

Execute the following command as one line, replacing the ***x*** in **ProgramEnd*x*.tcs** with the actual number and ***path*** with the path for your operating system as listed above:

```
bin\nmextract -cov -xml

-all "path\var\sessionfiles\CardGame\ProgramEndx.tcs"

-out "path\var\sessionfiles\CardGame\CoverageData"
```

- `-cov` — Extract Code Coverage data.
- `-xml` — Create an XML file of the data.
- `-all` — Export all data, including line-level.
- `-out` — Create the export file in the specified folder.

The file **CoverageData.xml** is generated in the folder **path\var\session-files\CardGame**.

Note: You can also export line-level data through the **Session Files** tab of the Start page. For details, see “Exporting Session Data” on page 75.

Create the HTML Report

1 Change to the folder ***DPJ_dir*\samples\nmExtractExample**.

2 Execute the following as one line, where ***path*** is the path for your operation system as listed above:

```
java -jar convert2HTML\dist\convert2HTML.jar

-i "path\var\sessionfiles\CardGame\CoverageData.xml"

-p "DPJ_dir\samples\nmExtractExample\CardGame\src"

-o "DPJ_dir\samples\nmExtractExample"
```

- `-i` — Input file.
- `-p` — Path(s) to search for source files; separate multiple paths with commas.
- `-o` — Location of the output folder.

Note: To display help information for this utility, execute the following command from ***DPJ_dir*\samples\nmExtractExample**:

```
java -jar convert2HTML\dist\convert2HTML.jar -h
```

The **convert2HTML.jar** utility generates HTML files for all the source files listed in the XML file specified by the `-i` option, as well as the files it can find in the paths specified by the `-p` option.

A folder named **`\html`** is created inside the folder specified by the `-o` option. Within **`\html`** is a table of contents, (**`TOC.HTML)`** and a folderfolder containing an HTML file for each source file (corresponding to each package in the application).

Note: The **`convert2HTML.jar`** utility may display error messages while generating the HTML files. If the message "Successfully generated HTML files into *destination_directory*" appears, then the conversion was successful and the error messages can be ignored. If an error message appears, followed by information about convert2HTML.jar usage, check the command for syntax errors.

Display the HTML Reports

To display the HTML reports, open **`TOC.HTML`** into your browser. This page displays a table that lists the classes for which Code Coverage data was exported.

Click on a class name in the table to display the source code. Each line that was executed is highlighted in green and identified by a check mark next to the line number; the number to the right of the check mark indicates how many times the line was executed. Each executable line that was not executed is highlighted in red, with an X next to the line number and a zero (0) to the right of the X. This annotation is similar to the Source View in the DevPartner Java Edition Start page.

The HTML report also displays environment information from the Code Coverage session, in a scrolling window above the class table. This information is the same as the **Process** section of the **Session Details** tab, which is displayed with the **Coverage Results Summary** tab when you open the session file within the DevPartner Java Edition Start page.

Adapting the Conversion Utility

The source code for **`convert2HTML.jar`** is provided in **`DPJ_dir\samples\nmExtract-Example\convert2HTML\src\convert2html`**.

You can use the source code to extend or adapt this utility to fit your Web reporting system.

You could also wrap the **`nmjava`**, **`nmextract`**, and **`convert2HTML`** utilities in an Ant script to further automate the Code Coverage reports.

Memory Analysis

The sample applications for Memory analysis demonstrate how to:

- Find memory leaks.
- Find retained objects.
- Identify temporary objects.

Finding Memory Leaks

The sample application **`stackLeakExample`** demonstrates memory leaks that can occur when you push and pop with a stack. Ideally, when you pop each entry, you zero it out. Each entry has a pointer to a stack entry location. If the entry does not get zeroed out, it cannot get garbage collected. The bug that is demonstrated in this sample is that none of the pushes are subsequently popped or zeroed out; therefore, they cannot be garbage collected. This problem causes memory leaks, which in turn can cause performance degradation.

The application can be found in the following folders:

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

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

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

Run the Application

1. Change the folder to

   `DPJ_dir``\samples\stackLeakExample`

2. Run the command (as one line)

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

3. The Session Control window appears. From the list in the top left corner, select **Memory Leaks**.

4. At the command prompt, enter **40** to simulate performing 40 stack pushes/pops.

5. In the Session Control page, select the **Profiled Classes** tab if it is not already displayed.

6. In the **Filter By** field, enter **stackLeakExample;** click **Apply**.

   Only the classes of the class **`stackLeakExample`** are listed.

7. To enable tracking of memory allocations, click **Start Tracking**.

8. To check for memory leaks, enter **40** at the command prompt to simulate performing 40 stack pushes/pops. The class **`stackLeakExample.StackEntry`** shows the number of Tracked Objects to be 40, representing the number of stack pushes/pops. The real-time graph shows a spike.

9. Click **Stop Tracking** to end the tracking of allocated objects.

10. At the command prompt, enter **30** to simulate performing 30 stack pushes/pops.

11. Click **Run Garbage Collection**.

    The count of tracked objects for **`StackEntry`** is 10, when it should be 0 (zero). This indicates that 10 instances of **`StackEntry`** have leaked. The memory display in the real-time graphs did not return to the pre-exercise level. Also, the **`StackData`** object has 100,000 leaked objects.

View the Results

1 Click **View Memory Leaks** to create a session file. The session file data is displayed in the Memory Leaks Results Summary.

The session file records the leaked classes, and helps you locate where the memory leak resides. Note the number of uncollected instances of an object in the **Tracked Objects** column.

2 In the Memory Leaks Results Summary, view the **Objects that Refer to the Most Leaked Bytes** graph. Use this graph because there is only one set of objects causing the leaked memory (the **`Object`** array).

3 Click the **Object** link to the right of the graph to display a Details window. Note that this object refers to over 100,000 leaked objects.

4 Click **View Instances** to display a list of all the **Leaked Objects Referenced from Object** array.

5 If necessary, click the **Referenced Bytes** column head in the Instance List to sort the list is descending order. Note that the 10 instances of **`StackEntry`** contribute the most leaked bytes of the objects referenced by **`Object`**.

6 Scroll down the Instance List. Note that there are also 10 instances of **`Object`** in **`stack-LeakExample.StackEntry`**. Continue scrolling; there are many instances of **`Stack-Data`**, even though each one only leaks 16 bytes. Looking back on the Session Control page, note that there were 100,000 **`StackData`** objects remaining in memory after garbage collection was performed.

7 In the Instance List, click any instance of **`StackEntry`** to display the Referring Object window. Click **View Object Reference Path**. Drag the divider bar towards the top of the Instance List to see the Object Reference Path. Click the **StackEntry** node to display the source code for the method. Note that two of the objects leaking data (**`StackData`** and **`StackEntry`**) were allocated in the method **`CreateStackEntry`**. There are 10 leaked instances of **`StackEntry`** and each entry on the stack refers to 10,000 **`StackData`** objects; this results in the 100,000 leaked instances of **`StackData`** (10 * 10,000 = 100,000).

8 In the Object Reference Path, click the **Object[]** node that refers to **`StackEntry`** to view the source code for **`Stack.java`**. In the Source View, note that line 33 is highlighted; it is the line where this **`Object[]`** array was allocated in the pop method.

Notice a comment in the code on lines 23-24. The problem that causes memory leaks in this application is that the referencing variable, **`elements[size]`**, is not set to `null`. If you uncomment line 24, compile the program, and rerun it, the memory leaks no longer occur.

Most instances of memory leaks in programs occur because a referencing variable either is not set to `null` or is overwriting the referencing variable with a reference to a object that is being used by the program.

Finding Retained Objects

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 four tasks:

1. Create a configuration that has object retention enabled.
2. Run the server.
3. Run the client.
4. Review the results in the Object-Lifetimes Results Summary.

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 Open a command console and change to the **`/samples`** 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.

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 computer on which the server is running.

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. 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.

Review the Results

In the Object-Lifetimes Results Summary, 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.

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).

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 Details 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.

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.

Identifying Temporary Objects

The sample application **`tempObjExample`** is located in the following folders:

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

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

- UNIX — `/opt/Micro Focus/DPJ/samples/tempObjExample`

Run the Application

1. At the command prompt, change the folder to

   `DPJ_dir``\samples\tempObjExample`

2. Run the command (as one line)

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

3. In the Session Control page, select **Object-Lifetime Summary** from the list of available analysis types.
4. At the command prompt, enter **1** to warm up the program (to make sure all classes are loaded and all one-time initializations are run).
5. Click **Clear Collected Data** to clear what has been collected so far.
6. Click **Run Garbage Collection** to clear the temporary objects.
7. At the command prompt, enter **1000** to create a large number of temporary objects. Note the spike in memory use.

View the Results

1. In the Session Control page, click **View Temporary Objects** to display the Object-Lifetime Results Summary, then select the **Temporary Objects Results Summary** tab.
2. View the **Methods Requiring the Most Temporary Space** graph. Note that the method **`StringConcat.doConcats`** uses the most temporary space. The other methods use very little temporary space. So, examine **`StingConcat.doConcats`** to determine which lines of code are using the most temporary space.

3 Click **StringConcat.doConcats** to the right of the graph. Note that this method uses a high number of temporary bytes. The largest number of temporary objects are short-lived objects, with a relatively small number of medium-lived objects.

4 In the Details window, click **View Call Graph**. The Call Graph shows the critical path: the sequence of child method calls that resulted in the largest total allocation of memory. In this example, the method **`doConcats`** causes the largest allocation. The bubble for this method shows that this method is responsible for 100% of the temporary bytes allocated. This method also is responsible for 100% of the temporary objects used by all methods during the session.

5 Click the **doConcats** bubble, then click **View Source**. If prompted, browse to the path to the source file for this application.

6 Click **Column Selection**, and select the following columns: **Line Number**, **Temporary Bytes including Children**, **Temporary Objects including Children**, **Short-lived Bytes including Children**, **Short-lived Objects including Children**, **Medium-lived Bytes including Children**, and **Medium-lived Objects including Children**. Click **OK** to save your selections.

7 Note that line number 9 is highlighted in yellow; this is the first line in the selected method that performs an action. The next line (a `for` loop), is executed 659 times; it creates the large numbers of temporary bytes, temporary objects, short-lived bytes, and short-lived objects, and a lesser number of medium-lived bytes and objects. If you reduce the value of `num` (that is, make the number of strings processed smaller), the number of temporary objects produced by **`doConcats`** will decrease.

Index