# Table of Contents

1. Overview ................................................................. 1

2. Installing SAP JVM Profiler (Eclipse Plug-in) ................................................................. 2
   2.1. Installing the Profiling Front-End ................................................................. 2
   2.2. Notes on Security ................................................................. 5

3. Getting Started Tutorials ................................................................. 7
   3.1. Remote Profiling ................................................................. 7
      3.1.1. Connect to a VM Using the Start Profiling Wizard ........................................... 8
   3.2. Allocation Analysis Using SAP JVM Profiler ................................................................. 11
      3.2.1. Connecting the SAP JVM Profiler to a SAP NetWeaver Application Server Java ............... 11
      3.2.2. Dynamically Enable and Disable the Allocation Analysis ........................................... 13
      3.2.3. Determine the Memory Consumption ................................................................. 15
   3.3. Performance Hotspot Analysis Using SAP JVM Profiler ................................................................. 16
      3.3.1. Starting the Application in Profiling Mode ................................................................. 17
      3.3.2. Create and Manage the Launch Configuration ................................................................. 18
      3.3.3. Run the Application ................................................................. 19
      3.3.4. Self and Total Times ................................................................. 20
      3.3.5. Identify Performance Hotspots Using the Methods (Flat) View ................................................................. 21
      3.3.6. Define a Solution for the Identified Performance Problem ................................................................. 22
      3.3.7. Open the Source Code and Apply the Performance Fix ................................................................. 22
      3.3.8. Validate the Performance Fix ................................................................. 23
      3.3.9. Running the Example ................................................................. 25
   3.4. Method Parameter Analysis Using SAP JVM Profiler ................................................................. 25
      3.4.1. Connecting the SAP JVM Profiler to a SAP NetWeaver Application Server Java ............... 26
      3.4.2. Dynamically Enable and Disable the Method Parameter Analysis ................................................................. 27
      3.4.3. Analyze the Method Invocations ................................................................. 35
   3.5. Synchronization Analysis Using SAP JVM Profiler ................................................................. 38
      3.5.1. Starting the Application in Profiling Mode ................................................................. 39
      3.5.2. Create and Manage the Launch Configuration ................................................................. 40
      3.5.3. Run the Application ................................................................. 40
      3.5.4. Self and Total Blocked Times ................................................................. 41
      3.5.5. Identify Synchronization Issues Using the Methods (Flat) View ................................................................. 41
      3.5.6. Running the Example ................................................................. 43
   3.6. File I/O Analysis Using SAP JVM Profiler ................................................................. 43
      3.6.1. Starting the Application in Profiling Mode ................................................................. 43
      3.6.2. Create and Manage the Launch Configuration ................................................................. 44
      3.6.3. Run the Application ................................................................. 45
      3.6.4. Overview of File I/O Operations Using the Files View ................................................................. 46

4. Allocation Analysis ................................................................. 49
   4.1. Allocation Analysis Walkthrough ................................................................. 49
      4.1.1. Overview of the Allocation Analysis ................................................................. 49
      4.1.2. A Simple Example ................................................................. 50
      4.1.3. A Complex Example ................................................................. 59
   4.2. Combined Allocation and Method Parameter Analysis ................................................................. 71

5. Performance Hotspot Analysis ................................................................. 76
   5.1. Performance Hotspot Analysis Walkthrough ................................................................. 76
      5.1.1. Performance Hotspot Analysis Overview ................................................................. 76
      5.1.2. The Checksum Example ................................................................. 79
      5.1.3. Retrieving Profiling Data ................................................................. 82
      5.1.4. Evaluating Profiling Data ................................................................. 90
      5.1.5. Advanced Performance Hotspot Analysis ................................................................. 107
   5.2. Combined Performance Hotspot and Method Parameter Analysis ................................................................. 115
      5.2.1. Connect the SAP JVM Profiler to the SAP NetWeaver Application Server Java ............... 115
      5.2.2. Dynamically Enable and Disable the Analysis ................................................................. 118
      5.2.3. Analyze the Method Invocations ................................................................. 121
   5.3. Performance Hotspot Analysis - Technical Documentation ................................................................. 123
5.3.1. Methods of Data Collection .............................................................. 123
5.3.2. Performance Hotspot Analysis with the SAP JVM Profiler .................. 124
5.3.3. Technical Details ........................................................................ 124
5.3.4. VM Options .............................................................................. 126
5.3.5. Limitations .............................................................................. 127
5.3.6. Misconceptions ........................................................................ 127
5.3.7. Summary .................................................................................. 128

6. Synchronization Analysis ................................................................. 129
6.1. Synchronization Analysis Walkthrough ........................................... 129
6.1.1. Synchronization Analysis Overview ............................................ 131
6.1.2. The Pi Calculator Example ....................................................... 131
6.1.3. Profiling the Example .................................................................. 133
6.1.4. Evaluating Profiling Data .......................................................... 140

7. File I/O Analysis ............................................................................... 154
7.1. File I/O Analysis Walkthrough ....................................................... 154
7.1.1. File I/O Analysis Overview ....................................................... 154
7.1.2. The File Copy Example Program .............................................. 155
7.1.3. Profiling the Example .................................................................. 155
7.1.4. Evaluating Profiling Data .......................................................... 161

8. Advanced topics ............................................................................ 170
8.1. Profiling Configuration View .......................................................... 170
8.1.1. Profile Views Hierarchy ............................................................. 170
8.1.2. Profiling Configuration Toolbar ............................................... 172
8.1.3. Profiling Configuration Context Menu ....................................... 172
8.2. Profiling Lifecycle and Analysis Overview ...................................... 173
8.2.1. Profiling Overview ..................................................................... 173
8.2.2. Profiling Lifecycle ..................................................................... 174
8.2.3. Analysis Overview ..................................................................... 176
8.3. Comparison of Profiling Statistics .................................................. 177
8.3.1. Comparable Data ...................................................................... 178
8.3.2. A Simple Example ................................................................... 178
8.4. Profiling Properties Dialog ............................................................ 180
8.5. Advanced Profiling Techniques ...................................................... 181
8.5.1. SAP AS Java Entity- / Thread-Filter........................................... 181
8.5.2. Exceptions ................................................................................ 183
8.5.3. Finalizer ................................................................................... 186
8.5.4. Top of Stack Finder ................................................................... 187
8.5.5. Component Statistics ................................................................. 191

9. Tips & Tricks ................................................................................... 196
9.1. Smart Expanding of Trees ............................................................. 196
9.2. Displaying Source Code ............................................................... 196
9.3. Opening Profiling Files Using Drag and Drop ................................. 197
9.4. Exporting Statistics Using Drag and Drop ...................................... 198
9.5. VM Monitor View ......................................................................... 199

10. Reference ...................................................................................... 200
10.1. Internal Objects ........................................................................ 200
10.1.1. The {instance class} Object .................................................... 200
10.1.2. The {type array class} Object .................................................. 200
10.1.3. The {object array class} Object .............................................. 200
10.1.4. The {constant pool} Object .................................................... 200
10.1.5. The {constant pool cache} Object ........................................... 201
10.1.6. The {method} Object .............................................................. 202
10.1.7. The {const method} Object .................................................... 202
10.1.8. The {method data} Object ..................................................... 202
10.1.9. The {symbol} Object .............................................................. 202
10.1.10. The {array class} ................................................................. 202
10.1.11. The {other class} Object ...................................................... 203
10.1.12. The {compiledICHolder} Object ......................................... 203
Chapter 1. Overview

The SAP JVM Profiler helps you analyze the resource consumption of your Java application running on a SAP Java VM - be it a simple standalone Java application or a complex enterprise application.

The profiler consists of two parts: A back-end component integrated into the SAP Java VM itself and a front-end component. The back-end records the resource consumption information and sends it to the front-end for further processing via the SAP Java VM's debug connection. There are two types of front-ends: a stand-alone application for analyzing stand-alone and server applications, and an Eclipse plug-in, which in addition provides source code navigation and profiling of Java projects out of Eclipse.

As a first step you may check out our tutorials: Section 3.2, “Allocation Analysis Using SAP JVM Profiler” guides you through an example session analyzing the memory consumption of a Java application. Of course you can also measure in detail the runtime of a Java application. See Section 3.3, “Performance Hotspot Analysis Using SAP JVM Profiler” for an example session. The tutorial Section 3.4, “Method Parameter Analysis Using SAP JVM Profiler” explains how to analyze individual method calls in depth. In particular, you get information about method parameters, runtimes and the number of invocations.

Detailed information about the profiler's capabilities can be found in the provided walkthrough documents, which explain the wide range of options to analyze your Java applications. Chapter 4, Allocation Analysis guides you through the functionality and features of the memory analysis. If you are interested in performance measurement, Chapter 5, Performance Hotspot Analysis provides the details how to measure the performance of a Java application.

Chapter 8, Advanced topics gives you an in-depth insight into the structure of the profiler user interface and into advanced profiling topics. Finally, please take a look at Chapter 9, Tips & Tricks, which describes handy tools to simplify an analysis and share your findings with others.
Chapter 2. Installing SAP JVM Profiler (Eclipse Plug-in)

The SAP JVM Profiler consists of two parts:

- the profiling front-end, that is the Eclipse plug-in, which visualizes the profiling data
- the profiling back-end, that is the SAP Java VM to be profiled, which provides the profiling data

Within a profiling scenario the user directly connects to the SAP Java VM to be profiled. The profiling data can be requested interactively. Technically, this is implemented by establishing a socket connection between the profiling front-end and back-end. Thus, the profiling front-end and the profiling back-end can run - but do not necessarily need to - on different physical machines.

2.1. Installing the Profiling Front-End

In order to install the profiling front-end, the following prerequisites have to be met:

- Eclipse in version 3.3, 3.4, 3.5 or 3.6 installed (or alternatively an SAP NetWeaver Developer Studio based on Eclipse 3.3, 3.4, 3.5 or 3.6) running on top of a Java SE 5 or Java SE 6 compatible VM.

- If not already installed, you can download Eclipse from http://www.eclipse.org/downloads or alternatively the SAP NetWeaver Developer Studio from the Service Marketplace (SMP).

- In order to run Eclipse, a Java SE 5 or Java SE 6 is needed. You can either use a corresponding SAP Java VM or a Oracle HotSpot VM version.

In order to get an archived version of the SAP JVM Profiler, you have the following alternatives:

- Download the archive version from the SDN:  SAP JVM Profiler - SDN [http://www.sdn.sap.com/irj/scn/index?rid=/webcontent/uuid/104301f7-8c92-2b10-22a2-ecc5be0d5bb3]

- Go to the Service Marketplace [http://service.sap.com/support] and download the latest version of the SAP JVM Supportability Tools. You find the supportability tools through the following path: Software Downloads -> Support Packages & Patches -> Browse our Download Catalog -> SAP Technology Components -> SAP JVM Supportability Tools.

The downloaded archive have to be stored somewhere within your local file system.

Eclipse offers a built-in system which allows you to update existing plug-ins or install new ones automatically. The SAP JVM Profiler can be installed and updated using this system. The following installation steps applies to Eclipse 3.6. The installation procedure for older Eclipse versions and the SAP NetWeaver Developer Studio is analogous. In order to update the already installed SAP JVM Profiler the same steps have to be performed.

To use the Eclipse built-in installation and update system, go to the Help menu and open the Install New Software... dialog. The dialog is displayed in Figure 2.1, “Install New Software”.
Figure 2.1. Install New Software

Next you have to open the Add... repository dialog in order to specify the location from where you want to install the SAP JVM Profiler. Then press the Archive... button to enter the file location.

Figure 2.2. Add a Repository

In the file browser dialog specify the path to the downloaded SAP JVM Profiler.
Figure 2.3. Specify Path to SAP JVM Profiler Archive

After choosing the SAP JVM Profiler archive, you have to select the found SAP JVM Profiler version and navigate to the next dialog step.

Figure 2.4. Select SAP JVM Profiler

Then you have to agree to the installation details and license text. Then, you finish the installation dialog.
Installing SAP JVM Profiler (Eclipse Plug-in)

2.2. Notes on Security

Profiling essentially works like debugging, i.e., the communication between the profiling front-end and the back-end is using a debugging connection to the SAP Java VM. This implies that the profiling front-end has
the full power of a debugger. In addition, the various profiling traces provide very detailed information about sensitive information. For example, the performance hotspot trace samples complete stack trace information of the profiled system. Furthermore, the method parameter trace provides the functionality to look at individual method parameter values.

Using the debugging protocol, any software that manages to connect to the VM effectively gains complete control over the VM and can change its behavior as well as read sensitive information. For this reason, the debugging port range and the port used by jvmmond (if running) should be protected, e.g., by using a firewall.

Caution

When considering the access permissions for profiling, please keep in mind that profiling - just like debugging - means full control over the profiled VM and its information. It is strongly recommended to use the profiling functionality only in development, test or quality systems - and not in productive environments.
Chapter 3. Getting Started Tutorials

The following section contains some getting started tutorials for the core functionality of the SAP JVM Profiler. All tutorials are self-contained and do not build upon each other or other sections of the documentation. That means it is also not necessary to read them in a consecutive manner. Just jump to the tutorial you are most interested in.

• Section 3.1, “Remote Profiling” - Describes in detail how to connect to and profile a Java application running on a remote host.

• Section 3.2, “Allocation Analysis Using SAP JVM Profiler” - The allocation analysis provides an insight in Java object allocations being performed within an application. It shows the allocated objects as well as the methods being responsible for these allocations. The tutorial presents how to analyze the memory behavior of a simple Web application.

• Section 3.3, “Performance Hotspot Analysis Using SAP JVM Profiler” - Via the performance hotspot analysis it is possible to analyze the runtime behavior of a Java application. You can get an overview of the most expensive method calls. Based on a standalone Java application the basic feature set of the performance hotspot analysis is shown and explained in a detailed manner.

• Section 3.4, “Method Parameter Analysis Using SAP JVM Profiler” - Often it is very interesting to analyze individual method invocations in depth. So you want to see how many times a certain method is called, what are the parameter values and how much time we spent within a specific method. These capabilities are provided by the so-called method parameter analysis. You have to define upfront (i.e. before starting the analysis) what methods and parameters are of interest for you. This tutorial guides you through the process of defining method parameters, starting the method parameter analysis, and analyzing the results of real-world application server scenario.

• Section 3.5, “Synchronization Analysis Using SAP JVM Profiler” - Modern Java applications usually employ multi-threading to scale on current multi-core processors. Shared memory is generally used for communication between the threads of such an application. Synchronization, e.g. by means of locks or object monitors, is commonly utilized to prevent data inconsistencies due to parallel writes to shared memory. This may partially or completely serialize the execution of threads, effectively blocking some threads while others are in their critical sections. Contention caused by excessive synchronization may impact the application performance. The synchronization analysis provides insights in the contended Java synchronization performed within an application, i.e. it shows how often and how long parts of the application are blocked due to usage of synchronization. The tutorial presents the basic feature set of the synchronization analysis based on a standalone Java application.

• Section 3.6, “File I/O Analysis Using SAP JVM Profiler” - Besides CPU and memory consumption, another very common performance bottleneck is I/O. What if your program is running slow, even though there is plenty of free memory available and the CPUs are idle most of the time? Often, the problem is competing access to a slow disk. What part of the code is writing to that given file and why does it take so long? Or, what method opened this file and failed to close it properly? Is reading of a data file done byte-wise or in large chunks? The File I/O Analysis of the SAP JVM Profiler can help answering these questions.

3.1. Remote Profiling

Within a remote profiling scenario, where the Java application to be analyzed is running on a remote host, the user directly connects to the SAP Java VM to be profiled. It is then possible to interactively configure the profiling session and request analysis results. In addition to the actual profiling functionality, it also allows integrated debugging and profiling scenarios. Thus, it is possible to debug and profile simultaneously. This is especially useful for memory debugging scenarios.

The SAP Java VM to be profiled is switched into profiling mode by opening a debugging port, which is also used for profiling connections. This can be done using the Start Profiling Wizard and the SAP MC (SAP Management Console).
Note: remember the security implications of providing remote profiling access to an SAP Java VM (see Section 2.2, “Notes on Security”).

### 3.1.1. Connect to a VM Using the Start Profiling Wizard

The Start Profiling Wizard can be opened via the profile view or by selecting Run # Start Profiling from the menu bar.

![Start Profiling Wizard](image)

By selecting the jvmmond option and pressing the Next button, all SAP Java VMs running on the local machine are displayed in the Start Profiling Wizard.

![Start Profiling Wizard - Local SAP Java VMs](image)
Basic information about the running VMs is displayed within this view. Furthermore, the VM can be switched into profiling mode, that means it opens a debugging port which is used for profiling connections.

Of course the Start Profiling Wizard is not limited to displaying local VMs only. It is possible to display VMs running on remote hosts as well. As a prerequisite for monitoring VMs running on a remote host, the jvmmond tool, which is located in the sapjvm_5/bin directory, must be running on the remote host to be monitored. Once the jvmmond tool is started on the remote host, the remote host can be added to the Start Profiling Wizard by entering the host name in the host field as shown in the following figure.

![Start Profiling Wizard](image)

**Figure 3.3. Adding a New Cluster to the Start Profiling Wizard**

If no port was specified explicitly when starting jvmmond, the preconfigured default port will be used to connect to the jvmmond daemon running on the remote host. Otherwise the port has to be set explicitly by entering hostname:port into the host name field (e.g. wdfn00157455A:1099).

After the connection between the profiler front end and the jvmmond tool has been established, the VMs running on the remote host are displayed in the Start Profiling Wizard. In this example, a NetWeaver Application Server Java is running on the remote host.

To profile the NetWeaver Application Server Java, it has to be switched into profiling mode by selecting the NetWeaver Application Server Java VM and finishing the wizard.

Additionally, profiling analyses can be configured and enabled directly when attaching to the VM. Instead of finishing the wizard directly, the analyses can be enabled by choosing Next in the Start Profiling Wizard as seen below.
Another way of starting the application in profiling mode is to use the Profile action available on the Java perspective's toolbar menu. Similar to the Run and Debug toolbar actions, the Profile action will open the launch configuration dialog and from there you can select the type of application you want to profile. The debug port of the profiling back end needs to be open already in order to connect to the VM (this could have been done using the Start Profiling Wizard, the jvmmon tool or the SAP MC).

Figure 3.5. Profiling a Remote Application
After the profiling front end is attached to the VM, the VM can be profiled.

### 3.2. Allocation Analysis Using SAP JVM Profiler

In typical development scenarios, developers tend to focus more on the functional aspects of application execution, when testing, debugging, and code fixing. However, many problems do not easily surface until the application is running in production mode, 24 hours a day, 7 days a week, and is pushed to its limits during unexpected peak periods.

The kinds of performance problems encountered in production cannot be discovered during a debugging session. Before deployed and run in production mode, it is important to use a profiling tool to analyze application execution and identify memory related problems, such as high memory consumption.

To demonstrate an allocation analysis to identify and isolate memory related problems, we use a simple Web application which allocates some memory.

#### 3.2.1. Connecting the SAP JVM Profiler to a SAP NetWeaver Application Server Java

SAP JVM Profiler can be used to profile a Java application running on a local or remote host. Within a remote profiling scenario the user directly connects to the SAP Java VM. It is then possible to interactively configure the profiling session and request analysis results. For example, it is possible to request a class statistic and directly see the results in the profiling front end (Eclipse). In addition, the results of a certain request can be used as an input for additional requests.

In addition to the actual profiling functionality, SAP JVM Profiler also allows for integrated debugging and profiling scenarios. Thus, it is possible to debug and profile simultaneously, which is especially useful for memory debugging scenarios.

To switch the SAP Java VM into profiling mode, we have to open its debugging port which is also used for profiling connections. This can be done using the Start Profiling Wizard, and SAP MC (SAP Management Console).

In this example we use the Start Profiling Wizard to connect to an SAP Java VM running the SAP NetWeaver Application Server Java. The wizard can be opened via the profile view or by selecting `Run # Start Profiling` from the menu bar.
By selecting the `jvmmond` option and pressing the Next button, all SAP Java VMs running on the local machine are displayed in the Start Profiling Wizard (the tutorial Section 3.1, “Remote Profiling” describes in detail the necessary steps to display SAP Java VMs running on remote hosts).

The figure below shows the Start Profiling Wizard after the selection of the `jvmmond` option. The dialog can show basic information about the running VMs. To profile the SAP NetWeaver Application Server Java, it has to be switched into profiling mode by selecting its VM and then finishing the wizard, as seen in the figure below.
This example demonstrates an integrated debugging and profiling scenario in which we want to profile the memory consumption of our Web application. In order to profile our Web application, we later set breakpoints at the entry and exit points of our application. When these breakpoints are hit, we dynamically enable and disable the allocation analysis. The captured profiling analysis information then allows us to determine the memory consumption of our Web application.

Since we dynamically enable the allocation analysis when we hit the breakpoint at the entry point of our application, we don’t have to activate any profiling traces in the Start Profiling Wizard and can continue by choosing Finish.

Another way of starting the application in profiling mode is to use the Profile action available on the Java perspective’s toolbar menu in the SAP NetWeaver Developer Studio. Similar to the Run and Debug toolbar actions, the Profile action will open the launch configuration dialog and from there you can select the type of application you want to profile. The debug port needs to be opened already in order to connect to the VM.

3.2.2. Dynamically Enable and Disable the Allocation Analysis

We have connected the profiler to the SAP Java VM and now want to determine the memory consumption of our Web application. As a preparation we now have to set breakpoints at its entry and exit points. When these breakpoints are hit, we later dynamically enable and disable the allocation analysis.

The following figure shows the source code of the Web application including the breakpoints at the entry and exit points. As you can see, the sample Web application is a simple JSP which prints out 100 strings.

![Source Code](image)

**Figure 3.8. Source Code**

Once the breakpoints are set, we can execute our Web application by invoking the browser with the URL pointing to it. The debugger will then stop at the breakpoints. Now it is time to enable the allocation analysis to collect the profiling data necessary to determine the memory consumption of our application.
Figure 3.9. Start Analysis

This is done by choosing Start Analysis in the context menu of the configuration in the Profile View (as seen in figure above). The Analysis Start Dialog is shown in the figure below. Here we activate the allocation analysis.

Figure 3.10. Enable Allocation Trace
With the allocation analysis, it is possible to find the classes which are mostly allocated and to find the methods which allocate most. Allocating large amounts of objects can lead to two kinds of performance problems:

- Firstly, it takes time to allocate the objects and initialize the memory.
- Secondly, more time is needed during the garbage collection.

When the allocation analysis is enabled, then for each object allocation the stack trace, the size of the created object, and a thread identifier are saved. This information can be used in many ways:

- It allows for tracking down the components doing the most allocations.
- It is possible to identify components allocating very large arrays.
- It gives information about the distribution of array lengths used in allocations, so the user can spot points where the size of allocated arrays is normally quite small, but certain allocations create much larger arrays. It also provides the capability to track down the call paths which lead to such large arrays.
- The user can get a feel for the behavior of certain methods or components, since the amount of allocations done (especially in methods called indirectly) may be surprisingly high.

After enabling the allocation analysis, we resume the debugging. The information described above is collected for our Web application. After resuming the VM, we hit the second breakpoint. This breakpoint was set at the end of our application. Here we can stop the allocation analysis. This is done in the same place as enabling it, just select Stop Analysis from the context menu of the configuration in the Profile View.

### 3.2.3. Determine the Memory Consumption

Now we are interested in the memory consumption of our Web application. The allocation analysis provides information about the methods that performed memory allocations. It shows the number and bytes allocated directly in the method as well as the number and bytes allocated in the method(s) called by the method.

To open the Methods (Flat) View expand the configuration in the Profile View, select the Allocation Statistic snapshot and choose Show Methods (Flat) from the context menu. The Methods (Flat) View shown in the figure below displays the methods that performed allocations.

**Figure 3.11. Methods (Flat) View**

Our Web application creates 100 objects by itself (column Self Objects) and 400 objects are created by the method itself and the methods it invokes (column Total Objects).

Now we would like to know, what kind of objects were allocated by our simple Web application. As we are only interested in our own application, not in allocations of the server, we select only the method that handles
the JSP and choose Show Allocated Objects from the context menu of that method. You could select multiple methods of interest and show statistics for this selection as well.

The Allocated Objects Statistic shows the objects which have been created while executing the selected method. Once again, the statistic view differentiates between objects that have been created by the method itself (the Self Bytes | Self Objects columns) and the objects which have been created by the method(s) called from the selected method. Also, as we only selected a subset of all allocations (those that happened while executing the selected method), the bar diagrams show the percentage relative to this subset as well as relative to the complete snapshot.

You might wonder why java.lang.StringBuilder occurs in the Self Objects of our Web application as you cannot find a java.lang.StringBuilder when looking at the source code. This is due to the fact that the Java compiler optimizes the string concatenation in a way to use java.lang.StringBuilder instead of always creating new unchangeable java.lang.String objects.

Another interesting view is the Called Method Tree view. In this view, the number of bytes of the allocated objects is shown by the call stack during allocation in a tree. This answers questions such as: How many objects were allocated by a java.lang.StringBuilder when called from a specific method? To open the Called Method Tree view, select Show called methods from the context menu of the method.

In typical development scenarios, developers tend to focus more on the functional aspects of application execution when testing, debugging, and code fixing. However, many problems do not easily surface until the
application is running in production mode, 24 hours a day, 7 days a week, and gets pushed to limits during unexpected peak periods.

The kinds of performance problems encountered in production cannot be discovered during a debugging session. Before deployed and run in production mode, it is important to use a profiling tool to analyze application execution and identify performance problems, such as execution bottlenecks.

The basic concept behind the SAP JVM Profiler's performance hotspot analysis is the sampler instance built in the SAP Java VM, which is capable of collecting information about the execution of Java methods in the currently running VM. When the user decides to start a performance analysis, the sampler is activated and starts to send information about the execution of methods (more precisely the call stack states) in regular intervals. As the sampling process causes only a very small overhead, the running system is hardly touched and the measurement is valid. After the user has stopped or paused the analysis, the recorded information is composed to a snapshot in the SAP JVM Profiler which is the basis for the subsequent analysis. Generally the user is allowed to take snapshots as long as the VM is running. It's also possible to profile the run of a complete Java application. In this special case the sampler is attached over the complete runtime. This approach is more convenient for small applications as used for the getting started example below.

Based on the information about method call times and calling hierarchies, which is stored in the taken snapshot, the user is supported with different tabular views showing the running time for each executed method. It is easy now to detect the most time-consuming methods or control paths of the application.

To demonstrate a performance hotspot analysis that identifies and isolates performance problems, we use a small sample application parsing a set of product information stored in XML files and printing the content of these XML files to the console output.

The download and installation of this example is described in Section 3.3.9, “Running the Example” at the end of this document.

### 3.3.1. Starting the Application in Profiling Mode

After installing the sample application, the first step is to run the product catalog application in profiling mode. Before the application can be profiled, some necessary parameters initially have to be setup and stored in a new eclipse profile configuration. Choose Profile As (SAP JVM) # Open Profile Dialog from context menu on the Product class as shown in the figure below.
Figure 3.14. Profile the Product Catalog Application

The profile configuration dialog can be also opened directly by pressing the Profile action available on the Java perspective's toolbar menu in the SAP NetWeaver Developer Studio. Similar to the Run and Debug toolbar actions, the Profile action will open the launch configuration dialog in which you can select the type of application you want to profile.

3.3.2. Create and Manage the Launch Configuration

The next step is to create a launch configuration and set the profiling options appropriately to collect performance hotspot information. To create a launch configuration, choose Java Application # New. Now the new launch configuration for the product class must be setup correctly. To set the profiling options, choose the Profiling tab and set the Performance Hotspot Analysis indicator.
In order to profile the product class, the launch configuration needs to be configured to execute the Product class on an SAP Java VM. To configure the JRE, choose the JRE tab and choose a SAP Java VM. Using the SAP Java VM is essential since the SAP JVM Profiler needs an optimized way of interaction with the running VM.

### 3.3.3. Run the Application

Now run the product catalog application by choosing Profile on the Launch Configuration dialog. Choose Yes when asked to switch to the profiling perspective. You should see a result of the program execution similar to the figure below.

The performance hotspot snapshot view is the starting point for viewing several statistics. In the summary section on top of the view, the user is informed about the number of samples which were taken by the sampler during the sampling process. Low numbers will generally indicate questionable results. The collection period is also noted here. Below different kinds of statistics can be reached by pressing the corresponding button. In this short introduction only the Methods (Flat) View is of relevance. All statistics are described in detail in Section 5.1, “Performance Hotspot Analysis Walkthrough”.

![Figure 3.15. Performance Hotspot Analysis Option](image-url)
Figure 3.16. Catalog Application Executed

The SAP JVM Profiler allows you to interact with your profiled application. You can pause and resume the profiled application, set breakpoints or terminate the application.

3.3.4. Self and Total Times

As mentioned above, the sampler works by dumping a stack trace of all threads at regular intervals which are gathered in a snapshot. Since the complete stack traces are send, it is possible to distinguish between the self time and the total time of a method.

- The self time of a method is the percentage of total running time the VM spent directly in that method disregarding its called methods. The percentage is simply given by the number of stacktraces this method appears on top compared to the total amount of stacktraces.

- Similar the percentage given by the number of stacktraces where the method appears somewhere in the stack implies the method's total time. It is the time the VM was processing in that method or in one of its called methods.

Note that both values are only statistical. For methods that only have a very short self or total time the result might be not accurate. Note that this circumstance does not limit the usability of the profiler at all, because these methods are hardly a useful target for optimization.
3.3.5. Identify Performance Hotspots Using the Methods (Flat) View

The Methods (Flat) Statistic helps to identify performance hotspots, i.e. by providing a sorted list of each method's self and total time, it enables the user to find the methods that consume the most CPU resources of the application. Note that unlike the Methods (Hierarchical) View, the Methods (Flat) View does not show any information about calling hierarchies of methods (call graphs). So it only delivers the running time of a method from an application-wide view and therefore cannot give information about the CPU consumption of a special control path.

To open the Methods (Flat) View, follow the link named Methods (Flat) on the snapshot view. It's also possible to expand the configuration in the Profile view, select the Performance Hotspot Statistic Snapshot and choose Show Methods (Flat) from the context menu. The Methods (Flat) View shown in the figure below displays the methods executed, sorted by total time.

![Figure 3.17. Methods (Flat) View](image)

As mentioned before, the sample application parses a set of product information stored in XML files and prints the content of these XML files to the console output.

As presented in the figure above, the Methods (Flat) Statistic shows methods related to XML parsing (like `Product.parseContent`, `SAXParser.parse`) as well as the entry points (like `Product.main`, `Product.readCatalogFromFolder`) as the top methods with the highest total execution time. It is not surprising to see the entry points on top of the list, as they are the root for invoking lots of other methods. The actual work is done by methods related to XML parsing, consequently they belong to the top of the list.

What comes as a surprise for us is the fact, that the `createParser` method, which just creates a SAX parser instance used to parse the XML files, has such a high execution time. The Methods (Flat) Statistic has helped us to identify this method as a potential bottleneck (a hot spot) which highly influences the application's performance.

A hierarchical view of the information described above can be obtained from the Methods (Hierarchical) View. The following figure shows the Methods (Hierarchical) View starting from `Product.main` - our entry point. This statistic can be opened directly from the main snapshot view or alternatively by choosing Show Methods (Hierarchical) from the context menu of the `Product.main` method. It shows all methods called by `Product.main` and their contribution to the execution time. Once again, it shows that...
ProductCatalog.createParser (including its invocations to other methods) takes more than 30% of the total execution time.

Figure 3.18. Methods (Hierarchical) View

After having identified this method, let's drill down and see the createParser method's execution details in the view. As you can see, the method invokes 2 different methods. The newSAXParser and newInstance methods are responsible for the createParser method's overall execution time.

3.3.6. Define a Solution for the Identified Performance Problem

By analyzing this data, we have found out that the execution time of createParser directly depends on the execution time of the two SAXParserFactory methods. Since we have no control over these methods' implementation, we can only try to reduce to the number of createParser calls.

The solution is to create one parser instance and reuse it for parsing all XML files, instead of creating a new parser for every file. Let's open the source code and apply the fix.

Before making any such optimizations, make sure that they are supported by the code. For example, while the SAXParser cannot be simultaneously used by multiple threads, instances can be reused.

3.3.7. Open the Source Code and Apply the Performance Fix

To open the source code for the createParser method, select the method and use either the F3 shortcut or the context menu action Show Source in the Methods (Flat) View.
Figure 3.19. Source Code

The figure above shows the `createParser` source code. Notice that the method creates a new SAX parser instance on every call. Update the source code as shown in the following figure.

Figure 3.20. Source Code Fix

The performance fix defines a global `SAXParser` instance. The `createParser` method initializes the parser and returns this instance every time the method is called instead of creating a new instance on every call.

Let's go back now and validate the fix by running the Product catalog application in profiling mode once again.

### 3.3.8. Validate the Performance Fix

This time we can directly start the profiling by simply clicking on the `Profile` action in the toolbar or alternatively by choosing `Profile As (SAP JVM) # Performance Hotspot Analysis`. Of course it is also possible to open the
profile configuration dialog again as described above, adjust some parameters if necessary and then start the profiling directly from this dialog.

The following figure shows the execution times after the fix has been applied to the code:

![Performance Hotspot Snapshot View](image)

**Figure 3.21. Performance Hotspot Snapshot View**

As you can see in the figure above, the total execution time has decreased from 8 seconds in the first run to 3 seconds in the run with the improved version of the product catalog sample. When opening the Methods (Flat) View, you can see that the total time of method `createParser` has decreased from 39.02% to 4.6%.
Note that this improvement will prove to be even more valuable when the number of XML files to be parsed increases, i.e. the fix will reduce the application execution time the more product files are added to the catalog.

### 3.3.9. Running the Example

The file "ProductCatalog_example.zip" contains the complete source code and the XML files containing the product information for the example in this article. Extract the content of the ZIP file into a directory and import the Java project into Eclipse.

### 3.4. Method Parameter Analysis Using SAP JVM Profiler

In order to perform a detailed analysis of an application, it is often necessary to look at individual method calls. Assume you have written a cache for Java objects. The cache provides functionality for storing Java objects under a descriptive key. Afterwards this key can be used to request an object. In order to analyze the cache you normally want to get some basic metrics like the number of requests, the number of store operations and the duration of the individual operations. Moreover, a cache is in general associated with an eviction policy to remove objects when the cache gets too large. To analyze the behavior of these policies it is often very interesting to analyze the usage of the cache. For example, you want to see what objects are requested and how often.

Via the method parameter analysis it is possible to look at individual method invocations in detail. The values of method parameters, the number of invocations and the runtimes of method calls are provided. With respect to the cache example, the method parameter analysis allows you to check how many times a certain cache operation is performed, what cache objects are requested and how long it takes to get a certain object.

The following section describes the basics of the method parameter analysis. It shows you how to connect to a running SAP Java VM, to start the method parameter analysis and finally to perform an actual evaluation. As an example we want to analyze the JNDI (Java Naming and Directory Interface) lookup operations within a running SAP NetWeaver Application Server Java installation. JNDI is a core service of a J2EE application server. It is a Java API for a directory service that allows Java components to discover and look up data and objects via a name. In general, the application server itself or privileged applications are responsible for binding objects to a certain name.
3.4.1. Connecting the SAP JVM Profiler to a SAP NetWeaver Application Server Java

The SAP JVM Profiler can be used to profile a Java application running on a local or remote host. In both scenarios the user directly communicates with the SAP Java VM through a socket connection. It provides functionality to interactively configure the profiling session and request analysis results. For example, it is possible to start and stop traces or to create snapshots. In addition, the results of a certain request can be used as an input for additional requests.

To switch the SAP Java VM into profiling mode, we have to open its debugging port which is also used for profiling connections. This can be done using the Start Profiling Wizard or the SAP MC (SAP Management Console). In this example we use the SAP MC to open the debugging port and then the Start Profiling Wizard to connect this port. It is also possible to directly open the debugging port via the Start Profiling Wizard. But then either the SAP Java VM to get profiled must run on same the host as the profiler frontend or the jvmmond daemon must run on the remote host. The tutorial Section 3.1, “Remote Profiling” describes in detail the necessary steps to use the Start Profiling Wizard in combination with the jvmmond daemon.

To start debugging via the SAP MC we have to select the server node to get profiled (see AS Java Process Table entry in the SAP Systems overview) and choose the context menu entry Enable Debugging as shown in Figure 3.23, “SAP MC - Enable Debugging”. By pressing function key F5, the process table gets refreshed and you can see the opened debugging port in the Debug column of the process table.

![Figure 3.23. SAP MC - Enable Debugging](image)

Once we have opened the debugging port, we start the Start Profiling Wizard via the Profile View (see the Start Profiling button in the view toolbar) or by selecting Run # Start Profiling from the menu bar.
Figure 3.24. Start Profiling Wizard - Start Profiling

When selecting the open debugging port option and pressing the Next button, you get to a page where you have to enter the hostname and the debugging port of the SAP Java VM we want to profile. Once specified we finish the connection wizard. Now we get connected to the SAP Java VM.

Figure 3.25. Start Profiling Wizard - Connect to SAP Java VM via open debugging port

3.4.2. Dynamically Enable and Disable the Method Parameter Analysis

We have connected the profiler to the SAP Java VM and now we want to analyze the JNDI lookup calls. So we want to see the number of lookup calls, the names to the looked-up objects and the consumed CPU and elapsed
time. Therefore we have to enable the method parameter trace. On the configuration overview page which is created after successfully starting a profiling run, an Analysis Overview section is available. This section contains a button for enabling profiling traces as shown in the figure below.

![Figure 3.26. Configuration Overview - Enable Profiling Traces](image)

Traces can also be started via the Profile View. When we select the entry which corresponds to the connected SAP Java VM in the configuration tree and open the context menu, we find the option Start Analysis... for enabling profiling traces.

![Figure 3.27. Profile View - Enable Profiling Traces](image)

To enable the method parameter trace, we select the corresponding entry Method Parameter Analysis in the Analysis Start Dialog. Then we have to specify the method parameter we want to evaluate.
A method parameter is defined by its fully qualified method name and the index within the method parameter list. So for the method name we have to specify the package name, the class name, the actual method name, the parameter list and the return type. The complete definition of a method parameter is called a method parameter specification. The following picture shows you how to create a method parameter specification from within the Method Parameters Dialog.

After pressing the button for creating a new method parameter specification, a panel appears on the right side of the dialog where we have to enter the actual data of the specification. First we have to provide a name for the specification and an optional description. For our example we enter `JNDI Lookup(String)` as a name and `The string key requested in a JNDI InitialContext look-up call.` as a description. Then we have to specify the fully qualified method name. To help users avoid typos an input assistance is available which provides proposals for completion. These proposals come from method definitions found in the associated JDK, in the workspace Java projects and in the profiling data received from the SAP Java VM itself.
After defining the method name we have to specify the index of the parameter we want to analyze. 1 is the first parameter with respect to the method definition, 2 is the second parameter, etc. 0 denotes the *this* pointer (i.e., the actual object this method is called on). In our case we want to trace the first parameter of the `lookup(...)` method, the key to the looked-up object. The result of the complete method parameter definition is shown in Figure 3.31, “Method Parameters Dialog - JNDI Lookup(String) Method Parameter Specification”.

Besides the `InitialContext.lookup(String)` method an overloaded method exists which takes a `javax.naming.Name` object as an argument. Due to the fact that we are interested in all looked-up objects we create analogously a second method parameter specification related to the `Name` parameter.

If the parameter to be traced has a primitive type (`char`, `short`, `int`, `long`, `float` or `double`) or is of type `java.lang.String`, `java.lang.StringBuffer` or `java.lang.StringBuilder`, the profiler makes the assumption that we are interested in the actual value. In case of `StringBuffer` and `StringBuilder` that means the created `String` value. For arbitrary other objects the user has to explicitly specify what value should be provided. Therefore, a so-called *modifier* has to be specified and applied to the parameter value. A modifier defines the UI representation of a value which is presented to the user as the parameter value of a certain method call.

A modifier can be one of the following entities:

- an instance method without parameters which is part of the parameter's class definition
- a static method which take the parameter value as an argument
- class - returning the class of the parameter value
- length - returning the length of the parameter value (can only be applied to arrays)
- `[i]` - returning the array element at position `i` of the parameter value (can only be applied to arrays)
• com.sap.MyClass fieldName - returning the value of a field which is part of the parameter’s class definition
• (com.sap.MyClass) - casts the parameter value to the specified type

For one parameter value a chain of modifiers can be specified. When we have modifier \( A \), \( B \) and \( C \), then the first modifier \( A \) gets applied to the traced parameter value. Next the modifier \( B \) gets applied to the return value of modifier \( A \). At last the return value of modifier \( B \) is taken as input for modifier \( C \). The return value of modifier \( C \) is reported by the profiler as the parameter value.

![Figure 3.31. Method Parameters Dialog - JNDI Lookup(String) Method Parameter Specification](image)

Note the return value of the last modifier must be a primitive type or of type String, StringBuffer or StringBuilder. Typical modifiers are for example String toString() returning the String representation of the parameter value or int hashCode() returning the hash value of the parameter value. They are also available as predefined modifiers as shown in Figure 3.32, “Method Parameters Dialog - JNDI Lookup(Name) Method Parameter Specification”.

In our example we simply want to see the String representation of the Name object. So we choose the corresponding predefined toString() modifier.
Now we have defined the method parameter we want to trace. Thus, we select both parameter definitions and start the trace by confirming the Method Parameters Dialog and the Analysis Start Dialog.
Figure 3.33. Method Parameters Dialog - JNDI Lookup Method Parameter Definitions

We can now run our test scenario on the SAP NetWeaver Application Server Java or if you want to analyze the JNDI lookup calls within a live system, then you should wait for a certain amount of time until you have collected a significant amount of profiling data. You can see the total number of method invocations collected so far in the Profile View. The invocations refer to invocations of the methods defined in the parameter specifications.

Figure 3.34. Profile View - Number of Method Invocations

When we have collected the profiling data, we disable the trace and disconnect from the SAP NetWeaver Application Server Java. There are several ways to disable the trace:
• Press the Disconnect button within the toolbar of the Profile View. The trace gets disabled and the connection to the profiled VM gets closed.

• Press the Stop Analysis button within the toolbar of the Analysis Overview section. Note that only the trace gets disabled and we stay connected.

• Open the context menu of the profiled VM entry within the Profile View and choose Stop Analysis. Note that only the trace gets disabled and we stay connected.

In our case we want to disable the trace and disconnect from the VM. So we press the Disconnect button within Profile View toolbar.

![Profile View - Disable the Method Parameter Trace and Disconnect from the SAP NetWeaver Application Server Java](image)

Figure 3.35. Profile View - Disable the Method Parameter Trace and Disconnect from the SAP NetWeaver Application Server Java

On disconnect a so-called snapshot is created. In general a snapshot refers to a certain timeframe where profiling data was collected. In our case the snapshot spans the whole lifecycle of the method parameter trace. All analysis functionality within the profiler (cf. Section 3.4.3, “Analyze the Method Invocations”) are related to snapshots. An entry for the created snapshot Method Parameter Statistic_1 is added to the configuration tree within the Profile View as shown in the following figure.

![Profile View - Snapshots](image)

Figure 3.36. Profile View - Snapshots

By double-clicking on the snapshot entry or by opening the context menu of the snapshot entry (see figure above) you can get an overview of all the available entry points for analyzing the profiling snapshot data.
We start the analysis of method parameter values with the so-called Method Parameter Statistic. This statistic can be opened by choosing the Method Parameters entry on the snapshot overview page or by choosing the Show Method Parameters entry in the context menu of the snapshot as shown in the previous section.

For each individual defined method parameter specification, which refers to one method parameter value, the statistics presents the total number of method invocations, the consumed CPU time and the total elapsed time accumulated for all method invocations. As shown in Figure 3.38, “Method Parameter Statistic - JNDI Lookup Calls”, you can see that the JNDI lookup method InitialContext.lookup(String) was called 1276 times. The CPU time the VM consumed for executing the lookup functionality was 3.54 seconds in total for the 1276 invocations. The elapsed time (i.e., the wall clock time) was 6.49 seconds in total. The elapsed time for one method invocation is defined as the difference between the timestamp when the method exits and the timestamp when the method gets entered. So the elapsed time includes not only the actual method execution time but also potentially times for executing other processes or threads due to process or thread switches performed on behalf of the operating system.
When started the method parameter trace, we also defined a parameter specification for the `InitialContext.lookup(Name)` method. But as you can see within the Method Parameter Statistic, there is no corresponding entry in the table available. That means this overloaded method was not called during our test scenario.

The next step in our analysis is to check the actual parameter values of the various invocations. Therefore, we select the entry within the Method Parameter Statistic and open the context menu. Here we find the option for showing the so-called Parameter Values Statistic.

The Parameter Values Statistic shows for each applied parameter value the corresponding number of invocations, the total CPU and elapsed time. The displayed parameter value refers to the modifier definitions in the method parameter specifications. Thus, if you had defined for example a `class` modifier, then you would have seen the class definitions of the parameters as a parameter value. In our case, we have not defined a modifier for the `InitialContext.lookup(String)` specification. So per default, we get the actual value of the `String` parameter. As shown in Figure 3.40, “Parameter Values Statistic - JNDI Lookup Calls”, you see for instance that the object to the key `/broker/services/com.sap.portal.themes.lafservice.laf` is requested 180 times. For all method invocations, which relates to this parameter value, 31.2 ms CPU time and 120 ms elapsed time was consumed.
Figure 3.40. Parameter Values Statistic - JNDI Lookup Calls

As discussed the Parameter Values Statistic shows only accumulated values over all invocations for a certain parameter value. Especially when implementing a cache infrastructure, it is often very interesting to see the variance of the individual method invocations. For example, if an object is requested the first time, it may happen that operations have to be performed which are only done on first request. A typical example is that the object data has to be loaded from the database. So the first request to the object is very expensive and further requests may be very cheap. In order to analyze these situations the so-called Method Invocation Statistic is available. For a chosen parameter value you can get overview of the individual method invocations. To open this statistic you have to select a certain parameter value within the Parameter Values Statistic and open the context menu as shown in the figure below.

Figure 3.41. Parameter Values Statistic - Context Menu
In this example, we select the `/broker/services/com.sap.portal.themes.lafservice.laf` parameter value and open the Method Invocation Statistic. When looking at the results you can see some variance both in the CPU and the elapsed time.

![Method Invocation Statistic - JNDI Lookup Calls](image)

**Figure 3.42. Method Invocation Statistic - JNDI Lookup Calls**

This tutorial has shown you the basic steps to perform an analysis of method parameter values. You have learnt how to switch an SAP NetWeaver Application Server Java instance into profiling mode and how traces can be started. We also performed a basic evaluation of the received profiling data.

### 3.5. Synchronization Analysis Using SAP JVM Profiler

In typical development scenarios, developers tend to focus more on the functional aspects of application execution when testing, debugging, and fixing code. However, many problems do not easily surface until the application is running in production mode, 24 hours a day, 7 days a week, and gets pushed to limits during unexpected peak periods.

The kinds of performance problems encountered in production cannot be discovered during a debugging session. Before deployed and run in production mode, it is important to use a profiling tool to analyze application execution and identify synchronization related problems, such as a high amount of contention with respect to synchronization primitives, e.g. locks. Contention may cause threads to be blocked for extensive periods of time even though the processing resources would allow a higher load. This can become a performance bottleneck when scaling the system.

The basis for SAP JVM Profiler's synchronization analysis is SAP Java VM's built-in capability to collect information about contended synchronization operations in the currently running VM. When the user decides to start a synchronization analysis, this functionality is activated and starts to send information about contended synchronization operations including call stacks of the involved threads. After the user has stopped or paused the analysis, the recorded information is composed to a snapshot in the SAP JVM Profiler which is the basis for the subsequent analysis. Generally the user is allowed to take snapshots as long as the VM is running.
also possible to profile the run of a complete Java application. In this special case the contention information is emitted over the complete runtime. This approach is more convenient for small applications as used for the getting started example below.

Based on the information about the timestamps of the synchronization events and the corresponding call stacks, which are stored in the taken snapshot, the user is supported with different tabular views showing the blocked time and block count for each executed method which was temporarily blocked due to synchronization. This eases the task of detecting the method that were blocked most of the time and also the reasons for this blockage.

To demonstrate a synchronization analysis that identifies and isolates synchronization related performance problems, we use a small sample application that does some number crunching creating a list of calculated values.

The download and installation of this example is described in the subsection Section 3.5.6, “Running the Example” at the end of this section.

### 3.5.1. Starting the Application in Profiling Mode

After installing the sample application, the first step is to run the number cruncher application in profiling mode. Before the application can be profiled, some necessary parameters initially have to be setup and stored in a new eclipse profile configuration. Choose Profile As (SAP JVM) # Profile Configurations from the context menu of the NumberCruncher class as shown in the figure below.

![Figure 3.43. Profile the Number Cruncher Application](image)

The profile configuration dialog can be also opened directly by pressing the Profile action available on the Java perspective's toolbar menu in the SAP NetWeaver Developer Studio. Similar to the Run and Debug toolbar
actions, the *Profile* action will open the launch configuration dialog in which you can select the type of application you want to profile.

### 3.5.2. Create and Manage the Launch Configuration

The next step is to create a launch configuration and set the profiling options appropriately to collect synchronization information. To create a launch configuration, choose *Java Application # New*. Now, the new launch configuration for the number cruncher class should be setup correctly. To set the profiling options, choose the *Profiling* tab and set the *Synchronization Analysis* indicator.

![Profile Configurations](image)

**Figure 3.44. Synchronization Analysis Option**

In order to profile the number cruncher class, the launch configuration needs to be configured to execute the *NumberCruncher* class on an SAP Java VM. To configure the JRE, choose the JRE tab and choose a SAP Java VM. Using the SAP Java VM is essential since the SAP JVM Profiler needs an optimized way of interaction with the running VM.

### 3.5.3. Run the Application

Now run the number cruncher application by choosing *Profile* on the *Launch Configuration* dialog. Choose *Yes* when asked to switch to the profiling perspective. You should see a result of the program execution similar to the figure below.

![Synchronization Snapshot View](image)

The synchronization snapshot view is the starting point for viewing several statistics. In the summary section on top of the view, the user is informed about the overall time any thread spent blocked due to synchronization and the overall number of such occurrences indicated by the *Block Count*. The collection period is also noted here. Below different kinds of statistics can be reached by pressing the corresponding button. In this short introduction only the Methods (Flat) View is of relevance. All statistics are described in detail in Section 6.1, “Synchronization Analysis Walkthrough”.
The SAP JVM Profiler allows you to interact with your profiled application. You can pause and resume the profiled application, set breakpoints or terminate the application.

### 3.5.4. Self and Total Blocked Times

As mentioned above, the synchronization analysis is based on SAP Java VM’s capability to dump information concerning contended synchronization operations including timestamps and stack traces of the affected threads, which is gathered in a snapshot. Since the complete stack traces are send for all contention data, it is possible to distinguish between the **self blocked time** and the **total blocked time** of a method. Analogously, it is also feasible to differentiate between the **self block count** and the **total block count** of a method.

- **Self blocked time** of a method is the time the VM spent blocked directly in that method disregarding its called methods. It is the sum of the durations of all blockages that occurred directly in that method. The **self block count** of a method is the number of blockages that happened directly in that method.

- In a similar fashion, the appearance of the method somewhere in the stack trace of a blockage due to synchronization implies a contribution to the method’s **total blocked time**. It is the time the VM was blocked in that method or in one of its called methods. The **total block count** is defined analogously. It is the number of times the VM was blocked in that method or in one of its called methods.

### 3.5.5. Identify Synchronization Issues Using the Methods (Flat) View

The Methods (Flat) Statistic helps to identify synchronization issues. By providing a sorted list of each method’s self/total blocked time and self/total block count, it enables the user to find the methods of the application that
were blocked the longest and/or most often. Note that unlike the Methods (Hierarchical) View, the Methods (Flat) View does not show any information about calling hierarchies of methods (call graphs). So it only delivers the blocked time and block count of a method from an application-wide view and therefore cannot give information about the blocked time and block count of a special control path.

To open the Methods (Flat) View, follow the link named *Methods (Flat)* on the snapshot view. It's also possible to expand the configuration in the Profile view, select the Synchronization Statistic Snapshot and choose *Show Blocked Methods (Flat)* from the context menu. The Methods (Flat) View shown in the figure below displays the blocked methods, sorted by self blocked time.

![Methods (Flat) View](image)

Figure 3.46. Methods (Flat) View

As mentioned before, the sample application calculates a set of scientific values using several threads and returns the results as a list of values.

As presented in the figure above, the Methods (Flat) Statistic shows some methods related to the number cruncher application, e.g. `NumberCruncher$Worker.run` and `NumberCruncher.main`, as well as several methods from the JDK library, e.g. `Vector.add` and `Object.wait`. It is not surprising to see high total values for the entry points, e.g. `Thread.run`, as they are the root for invoking the other methods. Since the actual work, i.e. the number crunching, does not require synchronization in itself, no corresponding methods are displayed in the list.

What comes as a surprise for us is the fact, that the `Vector.add` method, which adds a value to the result list, has such a high blocked time. A multi-threaded number crunching application should use synchronization only for merging the calculated results. In comparison to the overall runtime, the example application threads seem to spend a lot of time blocked. The Methods (Flat) Statistic has helped us to identify `Vector.add` as a potential synchronization bottleneck which highly impacts the application's performance.

To drill down into the details of the found synchronization issue it is useful to check were the method `Vector.add` was called in the example application. This information can be obtained using a Calling Methods (Hierarchical) View. The following figure shows the Calling Methods (Hierarchical) View starting from `Vector.add` - our contention hotspot. This statistic can be opened from the Methods (Flat) Statistic discussed above by choosing *Show Calling Methods (Hierarchical)* from the context menu of the Vector.add method. It shows all methods that were calling `Vector.add` and their contribution to the blocked time and block count.

![Calling Methods (Hierarchical) View](image)

Figure 3.47. Calling Methods (Hierarchical) View
After having identified this method and its call sites, we can see the problem arose because each worker thread adds its calculated values to the shared result list in every iteration. Hence, the worker threads contend for the result list to add their newly calculated values to it. In this case, the obvious optimization is to temporarily store the calculated values for some time in thread local data structures and do the updates of the shared result list less often. It may even be possible to delay the update of the result list until all calculations have finished.

The implementation of optimizations identified using the synchronization analysis and their validation is left as an exercise for the reader.

### 3.5.6. Running the Example

The file "NumberCruncher.zip" contains the complete source code of the example application used in this article. Extract the content of the ZIP file into a directory and import the Java project into Eclipse.

### 3.6. File I/O Analysis Using SAP JVM Profiler

Besides CPU and memory consumption, another very common performance bottleneck is I/O. What if your program is running slow, even though there is plenty of free memory available and the CPUs are idle most of the time? Often, the problem is competing access to a slow disk.

What part of the code is writing to that given file and why does it take so long? Or, what method opened this file and failed to close it properly? Is reading of a data file done byte-wise or in large chunks? The File I/O Analysis of the SAP JVM Profiler can help answering these questions.

The SAP Java VM can collect information about file operations like opening a file, reading from or writing to the file and closing it. When the user decides to start a File I/O Analysis, this functionality is activated and starts to send information about the file, the call stack, the amount of transferred data and the required time for each file operation until the user stops or pauses the analysis. The recorded information is composed to a snapshot in the SAP JVM Profiler which is the basis for the subsequent analysis. Generally the user is allowed to take snapshots as long as the VM is running. It's also possible to profile the run of a complete Java application. In this special case, the file I/O information is emitted over the complete runtime. This approach is more convenient for small applications like the getting started example below.

Based on the collected information in the taken snapshot, the SAP JVM Profiler offers various analyses that focus on different aspects of the recorded I/O operations. It is possible to get an overview considering the files that were operated on, the methods that caused or executed the operations and even a detailed view on operations on single selected files.

To demonstrate some features and views of the File I/O Analysis, we use a small sample application that performs some file operations. It will copy a text file to a new file and create another copy with all characters converted to lower case. Well, not the most useful program in the world, but it will do to demonstrate the most important features of the File I/O Analysis.

### 3.6.1. Starting the Application in Profiling Mode

The file "FileIOExample.zip" contains the complete source code of the example application used in this article. Extract the content of the ZIP file into a directory and import the Java project into Eclipse.

After installing the sample application, the first step is to run the FileCopyExample application in profiling mode. Before the application can be profiled, some necessary parameters initially have to be setup and stored in a new eclipse profile configuration. Choose Profile As (SAP JVM) # Profile Configurations from the context menu of the FileCopyExample class as shown in the figure below.
The profile configuration dialog can be also opened directly by pressing the Profile action available on the Java perspective’s toolbar menu in the SAP NetWeaver Developer Studio. Similar to the Run and Debug toolbar actions, the Profile action will open the launch configuration dialog in which you can select the type of application you want to profile.

3.6.2. Create and Manage the Launch Configuration

The next step is to create a launch configuration and set the profiling options appropriately to collect file I/O information. To create a launch configuration, choose Java Application # New. Now, the new launch configuration for the FileCopyExample class should be setup correctly. To set the profiling options, choose the Profiling tab and set the File I/O Analysis indicator.

Figure 3.48. Profile the FileCopyExample Application
In order to profile the sample program, the launch configuration needs to be configured to execute the `FileCopyExample` class on an SAP Java VM. To configure the JRE, choose the JRE tab and choose a SAP Java VM. Using the SAP Java VM is essential since the SAP JVM Profiler needs an optimized way of interaction with the running VM.

### 3.6.3. Run the Application

Now run the example application by choosing `Profile` on the `Launch Configuration` dialog. Choose `Yes` when asked to switch to the profiling perspective. You should see a result of the program execution similar to the figure below.

The File I/O Snapshot view is the starting point for viewing several statistics. In the summary section on top of the view, the user is informed about the overall time and volume of collected read and write operations. The collection period is also noted here, but as our program finished in less than a second, all we see here are zeros. Below, different kinds of statistics can be reached by pressing the corresponding buttons. In this short introduction only the Files View is of relevance. All statistics are described in detail in Section 7.1, “File I/O Analysis Walkthrough”.

**Figure 3.49. File I/O Analysis Option**

In order to profile the sample program, the launch configuration needs to be configured to execute the `FileCopyExample` class on an SAP Java VM. To configure the JRE, choose the JRE tab and choose a SAP Java VM. Using the SAP Java VM is essential since the SAP JVM Profiler needs an optimized way of interaction with the running VM.

### 3.6.3. Run the Application

Now run the example application by choosing `Profile` on the `Launch Configuration` dialog. Choose `Yes` when asked to switch to the profiling perspective. You should see a result of the program execution similar to the figure below.

The File I/O Snapshot view is the starting point for viewing several statistics. In the summary section on top of the view, the user is informed about the overall time and volume of collected read and write operations. The collection period is also noted here, but as our program finished in less than a second, all we see here are zeros. Below, different kinds of statistics can be reached by pressing the corresponding buttons. In this short introduction only the Files View is of relevance. All statistics are described in detail in Section 7.1, “File I/O Analysis Walkthrough”.

**Figure 3.49. File I/O Analysis Option**
Figure 3.50. File I/O Example Application Executed

3.6.4. Overview of File I/O Operations Using the Files View

The Files Statistic provides an analysis of the file I/O operations of the profiled application that is presented per file name. For each file that was read or written, this view offers accumulated data about the number of transferred bytes, the durations and how often that file was opened or closed. This way, one can easily spot the files where the most I/O time is spent.

To open the Files View, follow the link named Files on the snapshot view. It's also possible to expand the configuration in the Profile view, select the File I/O Statistic Snapshot and choose Show Files from the context menu. The Files View shown in the figure below displays the names of files with I/O operations, sorted by the number of times a file was opened.
We can see that besides the files that are used in our example application, there are more files listed because of class loading (in our case just the rt.jar and the simple application class). Additionally, we find the standard input / output streams `<stdin>`, `<stdout>` and `<stderr>`, as the example did not redirect them. Actually, we can attest that some bytes were sent to the standard output by the code writing the "Finished." message.

We are more interested in the files that are actually read and written from our program. The only written files are the two result files of our program, `copy.txt` and `lowerCase.txt`. While the number of written bytes is the same, it is obvious, that writing the `lowerCase.txt` file took much longer. Looking at the example code, the reason for this becomes clear: the copy is written in large blocks from a buffer, while the `lowerCase.txt` file is written in text lines that are converted to all lower case characters beforehand.

All of the (non class loading related) reading was done on the `LoremIpsum.txt` file, which was opened twice. To get a better idea about what was done with that file, we can have a look at the Detailed Files View for that file. Select *Show Files (Details)* form the context menu of that entry to open this view.

---

**Figure 3.51. Files View**

We can see that besides the files that are used in our example application, there are more files listed because of class loading (in our case just the rt.jar and the simple application class). Additionally, we find the standard input / output streams `<stdin>`, `<stdout>` and `<stderr>`, as the example did not redirect them. Actually, we can attest that some bytes were sent to the standard output by the code writing the "Finished." message.

We are more interested in the files that are actually read and written from our program. The only written files are the two result files of our program, `copy.txt` and `lowerCase.txt`. While the number of written bytes is the same, it is obvious, that writing the `lowerCase.txt` file took much longer. Looking at the example code, the reason for this becomes clear: the copy is written in large blocks from a buffer, while the `lowerCase.txt` file is written in text lines that are converted to all lower case characters beforehand.

All of the (non class loading related) reading was done on the `LoremIpsum.txt` file, which was opened twice. To get a better idea about what was done with that file, we can have a look at the Detailed Files View for that file. Select *Show Files (Details)* form the context menu of that entry to open this view.

---

**Figure 3.52. Show Detailed Files Statistic**
The Detailed Files View that opens now is very similar to the Files View, but now we get information about I/O operations performed on the file we selected and for each time this file was opened. We can see some additional information like the thread that opened or closed the file (with timestamps, if these columns are enabled).

**Figure 3.53. Detailed Files Statistic**

In our example, the first time that file was opened (indicated with ID 0 in the table), there was no corresponding close operation. This is a strong indication of a program error! By selecting the row and navigating further, we can find the methods responsible for opening and reading this file for that particular instance.

An easier way to find files that were never closed, together with more information about other views available in the File I/O Analysis is described in Section 7.1, “File I/O Analysis Walkthrough”.
Chapter 4. Allocation Analysis

4.1. Allocation Analysis Walkthrough

This is a walkthrough for the Allocation Analysis. It explains the basic concepts of this analysis and shows it in action using two examples. The first example is a short, self-contained program. It is used to show the basic features of the analysis. The second example is just one transaction of a Web Dynpro application. Here you will see more complex analysis methods. Additionally you will learn to profile code you do not understand, because it is either complex or you did not write it or both.

You can proceed from here in one of the following ways:

• Start the walkthrough with Section 4.1.1, “Overview of the Allocation Analysis”. Here we will explain what exactly this analysis does.

• Go directly to Section 4.1.2, “A Simple Example” for an overview of the basic analysis methods.

• Go to Section 4.1.3, “A Complex Example” for an overview of the more complex analysis methods.

4.1.1. Overview of the Allocation Analysis

Here we describe the basic workings of the Allocation Analysis, which is used to track the allocations of objects in the profiled VM.

When the Allocation Analysis is enabled, the VM we profile reports all allocations of Java objects to the profiler. We say 'Java objects' because the VM allocates some internal objects as well, which are not reported (see Section 10.1, “Internal Objects”).

Here is what the VM reports on each allocation:

• The class of the allocated object. Examples would be `java.lang.String` or `char[][]`.

• If the allocated object is an array, the size of the array is reported too. Note that even for multi-dimensional arrays only one size is reported, since even such an array is only allocated with one size (e.g. `new int[10][]` would report the allocation of `int[][]` with an array size of 10). The size of the array is used to calculate the number of bytes it consumes.

• The thread in which the allocation occurred.

• If the VM is running the NetWeaver AS Java and the user responsible for the allocation is known, the user is reported too. Note that this feature depends on the fact that the AS Java knows which threads are currently doing work for a specific user and tells this to the VM. All recent versions of the AS Java have this feature, but an older version might not. In this case you will not see any user information.

• Like the user, the session, application and request ID are reported if they are known.

• The stack trace of the thread in which the allocation was performed. This is actually one of the most important bits of information, since it allows us to determine the methods which are responsible for the allocation.

As you might imagine, this can lead to a huge amount of data sent to the profiler, which itself can lead to a large amount of memory used by the profiler itself. Additionally the overhead in the VM can be large too (which is mostly runtime overhead, since the additional memory overhead is quite small).

Depending on the average depth of the stacktraces, the profiled application runs slower by a factor of 3 to 10 typically. But factors of 100 have been observed too in special benchmarks, although it is unlikely you will ever see such drastic slowdowns. The profiler itself will use ten to fifty megabytes when profiling only for a
few seconds (e.g. a single click in a web application) up to several 100 megabytes when you profile a complex scenario for minutes or even hours. Because of this there are several methods to reduce the overhead of the Allocation Analysis both in the profiled VM and in the profiler:

1. You can specify to record only allocations performed in certain threads or by certain users or sessions.

2. You can let the VM omit the line numbers in stack traces. This leads to less different stack traces, which reduces the amount of memory used in the profiler. Additionally it makes the creation of the stack traces in the VM faster. Usually the effect of this optimization is only moderate, but you do not lose much information either.

3. You can enable the adaptive Allocation Analysis. When this is enabled, the VM records an allocation only after a specific amount of other objects have been allocated. The amount of skipped objects (in fact it's not the number of objects, but the size they consume) is zero when you start profiling and is then increased to a maximum value the longer the profiling run lasts. The error you will get by this statistical method is lower than \(0.01\) percent of the overall number of allocated bytes, so it is a good alternative as long as you do not have to know the exact number of allocated objects. Especially for long lasting profiling runs it is recommended to enable the adaptive Allocation Analysis.

So now that you know the basics of what the Allocation Analysis does, let's go to the first example in Section 4.1.2, “A Simple Example”.

### 4.1.2. A Simple Example

Here we describe the basics of the Allocation Analysis, using an example program.

#### 4.1.2.1. The Program

To keep it simple, we profile a small program that is only a few lines long. We want the program to be small enough for you to remember, so you don’t always have to look at the program again as we describe the various analysis methods. This is the program we will use:

```java
package com.sap.jvm.profiling.examples.alloctrace;
import java.util.*;
public final class Example {

    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.load(Example.class.getResourceAsStream("/com/sun/corba/se/impl/logging/LogStrings.properties"));
        System.out.println(getWithPrefix(props, "OMG").length);
        System.out.println(getWithPrefix(props, "IOR").length);
        System.out.println(getWithPrefix(props, "POA").length);
        System.out.println(getWithPrefix(props, "OMG").length);
    }

    public static String[] getWithPrefix(Properties props, String prefix) {
        ArrayList<String> result = new ArrayList<String>();
        Enumeration<?> propEnum = props.propertyNames();
        while (propEnum.hasMoreElements()) {
            String key = (String) propEnum.nextElement();
            if (key.startsWith(prefix)) {
                result.add(key + "=" + props.getProperty(key));
            }
        }
        return result.toArray(new String[result.size()]);
    }
}
```

It reads a property file (we use one of the properties that comes with the SAP JVM). A property file is basically just a bunch of key/value pairs. This particular property file is used to store the log messages used by the
CORBA implementation. After the property file is read into a Properties object, we call a method which returns the key/value pairs for which the key starts with a given prefix. We just print out the number of found keys matching the given prefix. If you run the program you'll see the following output:

131  15  71  131

So nothing spectacular here. We now want to know which and how many objects are allocated during the execution of the program.

### 4.1.2.2. Profile the Program

So let's start with the profiling. We assume that the program is part of an Eclipse project and you have installed the SAP JVM Profiler plug-in. The easiest way to profile the program is running it directly with the Allocation Analysis enabled. Just select the Example class and choose Profile As (SAP JVM) # Allocation Analysis from the context menu:

If you haven't set the SAP JVM as your default VM in Eclipse, you will get the following message box:

You are asked to change the configuration, but what configuration is meant? As you probably know, whenever you run or debug a program in Eclipse, an entry in the Run and Debug menus is added, so you can rerun that program easily. This entry is called a configuration and we add a configuration for the program we profile too. In the configuration you can set all kinds of properties for the program to run (e.g. the arguments to pass to the program). And one of the properties is the VM used to run the program. So just choose Yes button and the configuration will be opened, where you can change the VM:
As you can see, the currently selected VM is a SUN 1.6 VM (OK, you cannot really see it, but trust us). We will have to select an alternative JRE by selecting Alternate JRE. If this is the first time you profile, you will see that the selection you get is quite limited:

So we will first have to make the SAP JVM known to Eclipse. Just click on the Installed JREs... button, click Add in the following dialog and click the Browse... button. Select the directory of the SAP JVM. Then click OK and OK again. Now you can select the SAP JVM as an alternative JRE:

4.1.2.3. The Profiling Perspective

Now you can choose OK and the program is run with Allocation Analysis enabled. You should now be in the SAP JVM Profiling perspective. Let us first look at the Profile View at the upper left corner of the perspective (we have expanded the tree fully here):

Let us shortly discuss what we are seeing here. The root node is called 'Example' and represents the profiling run. Below you see the name of the class we profiled. Since the program is small, it is already terminated. You can see that the profiler was connected at port 1406 of the VM, but that is not very interesting for us now. But what is interesting are the snapshots, of which we have exactly one called 'Allocation Statistic_1'. But what is a snapshot? A snapshot is the result of the Allocation Analysis for a specific amount of time. You can have more than one snapshot, each representing the allocations of different time spans. This means that you can request the generation of a snapshot at specific times. But since the program was so fast, we couldn't do it here and the profiler generated a snapshot for us that covers the whole run of the application. Now double-click the snapshot and you will see the following view on the right side of the perspective:
This view gives you a good overview of the Allocation Analysis. At the top of the view you will find the number of allocated objects and their total size. As you can see about eight thousand objects were allocated, which consumed about 600 kB. Additionally you see when the snapshot was started and the time it represents (in our case this is 1 second). Below you see the entry points for some of the most important analysis views and a short description of them.

4.1.2.4. The Allocated Objects

The first analysis we will use is the Allocated Objects Statistic. Just click on the corresponding icon and you will get a new view which looks like this:
You can change the visibility of individual columns by selecting the pull-down and checking the columns you want to see.

In the column titled *Bytes* you see the number of bytes used up by the allocation of different classes (note that these are the classes of the allocated objects and not the class of the allocating method). We see that most of the allocated bytes are used for *char* arrays, something you will see most of the time. If you want to know the exact number of bytes and the percentage, you can get this from the tool tip:

The column titled *Objects* shows the number of objects. Here we find the first hint of something strange going on in our example program. We see that about 1,800 *String* objects were allocated, but more than 3,000 *Hashtable$Entry* objects. If we assume that the keys of the hash table are strings, we should have more strings than entries. But for now we don't know who used the tables at all, so we cannot know if this is a real problem or not, and by the way, haven't we always been told that we should use *HashMap* instead of *Hashtable*? As we will find out later, we can see the methods allocating the specific objects from this view, but for now we will first show the Methods (Flat) Statistic.

### 4.1.2.5. The Method Statistic

Now go back to the snapshot overview view and select 'Methods (Flat)' under the headline *Allocations per Method*.

This will open a new view that shows object allocations counted per method where the allocation took place. Basically, you can find out, which methods have allocated how much. So let's take a look at it:
Let us look in more detail at what we are seeing here. First note that now we have two kinds of bytes, termed 'self' and 'total'. What does this mean? The Self bytes are the number of bytes directly allocated in the method, while the Total Bytes are the number of bytes allocated directly in the method plus the bytes allocated in methods called from that method. You can clearly see the distinction between the two values in the Example.main() method. This method has allocated only 304 bytes directly (the few String objects and the Properties object), but most of the overall bytes indirectly (525 kByte). This does not surprise us very much, so let's look at the next method, which is the Example.getWithPrefix() method.

First take a look at how the full method name is displayed. As you can see the class and the method name are rendered somewhat more visible, so you can spot the most important information more easily. Additionally if the space is not large enough for the whole method including package and signature, we first make sure the method name is visible, after that the class, package name and the signature if there is enough room. You can see this for the Example.getWithPrefix() above, where only a part of the package name and no signature is shown.

Now back to the Example.getWithPrefix() method. We see that it allocated 307 kByte in total, but we are unsure if this is too high or not. So we select the method and can now get more information in the context menu:

After selecting Show Allocated Objects you will get a new view which looks like this:

This looks a lot like the Allocated Objects Statistic from where we started, but there are differences. The first is that this view shows the objects allocated in the Example.getWithPrefix() method. And this explains...
the second difference too. Here we have both 'self' and 'total' values. Again 'self' means the objects allocated in the method itself and 'total' includes the objects allocated in called methods too.

Here we have the first example of a view that depends on the context where it was created. This view doesn't show all allocated objects, but just the ones happening in Example.getWithPrefix(). To make this visible, the profiler annotates the view with this information. Let us look back at the Profile View from which we started. Right now it looks like this:

You see the two views we opened first. These are direct children of the 'Allocation Statistic' snapshot. The new Allocated Objects View on the other hand is a child of the Methods (Flat) View, because it was created by that view. And you can even see the method for which the Allocated Objects View was created (in the small font behind the view name). Apart from displaying the open views of the statistic, the Profile View can be used to navigate to specific views just by double-clicking their nodes in the view.

But let’s get back to the Allocated Objects View. As you can see, the percentage bar has now two colors instead of one. What does this mean? If you look at the tool tip, the meaning of the additional bar becomes clear:

The smaller and darker bar shows the number of bytes in relation to the total bytes allocated in the snapshot and the larger and lighter bar shows the number of bytes in relation to the bytes allocated by the Example.getWithPrefix() method. The former is used to see if a number of bytes is significant and the latter is used to get the relative quantities in a view. Note that the bar for the 'self bytes' is relative to the self bytes of the method and the bar for the 'total bytes' is relative to the total bytes of the methods.

So what can be learned from this view? We see that the Example.getWithPrefix() allocates mostly char arrays, which isn't really surprising considering the StringBuilder we use. But can we verify this? In fact we can. One option is to mark the line with the char[] and select Show Methods (Flat) from the context menu:

This opens a view that shows which methods are allocating char arrays:
You see the methods called by `Example.getWithPrefix()` which lead to allocations of character arrays. This can be seen in the Profile View:

We now see that our guess was correct, that the character arrays were allocated during the string processing. Additionally we see that class loading causes the allocation of character arrays too, but the amount is negligible.

Now that we know where the `char[]` objects were allocated, we want to know where the `Hashtable` $Entry$ and their arrays were allocated. Just select the two classes and choose `Show Methods (Hierarchical)` from the context menu. This will show methods that allocate objects of these classes in a tree. You can expand the tree to find out, who is calling these methods and what call-stacks result in the majority of allocations:

We see that getting the iterator of the `Properties` object seems to trigger the copying of the key/value pairs into a hash table, which is quite surprising. This is one of the things you should look at: Seemingly harmless methods which cause a lot of unexpected allocations. The JavaDoc of `Properties.propertyNames()` doesn't give a hint that this might be an expensive operation, but as we saw it actually is.

Now that we have inspected the two main memory consumers in the `getWithPrefix()` method, we want to know which of the four calls in the `main()` method actually leads to the allocation of the most memory. To do this we go back to the Methods (Flat) View, select the `main()` method and choose the `Show Methods (Hierarchical)` entry from the context menu:
This shows us the methods called from the `main()` main method in a tree as we have seen before:

Here we see a new feature of the Methods (Hierarchical) View. As you can see, four methods that are called by `main()` are not explicitly shown, since they allocate only an insignificant amount of memory. But how do we define insignificant? We just sum the bytes allocated in all the children of the `main()` method and show as many methods as needed to get at least 95 percent of the allocated memory, starting from the first child. This means in fact, that depending on which column you sort, different methods might be considered insignificant (and if you sort in ascending order, we never hide children). If we sort the view by the self bytes, we get the following result:

If you are interested in the combined methods, you can click on the triangle right to the method name to expand the combined methods:

But now let's get back to our original question: Which call to `getWithPrefix()` is the most expensive? As you can see, you cannot decide now, since you cannot differentiate between the four calls. To do this click on the triangle right of the `main()` method to display the line number information:
Instead of the called methods of main() you now see all the line numbers of main() which triggered allocations. Comparing the result with the source code of the example, we see that the most memory was allocated during the reading of the properties file and the second most in the first call to getWithPrefix() with the "OMG" string (line 10). The second call to getWithPrefix() with the "OMG" string (line 13) consumes 1 kByte less memory. Now the question is why there is a difference between these two calls. To get the answer we simply expand both nodes:

As you can see, the difference is the call to ClassLoader.loadClassInternal() which is only present at line 10. In the source code we never called this method explicitly. Instead it is called by the VM the first time a class is seen during the execution of a class's method. In our case the System class is first seen at line 10. The VM calls loadClassInternal to get the actual class and then stores it in the Example class, so we never try to load it again. This is one of the one-time effects you have to be aware of. Another one-time effect you will often see is the triggering of the statistic initializers by the first user of a class.

We are now at the end of our example, where you have seen the most basic operations of the profiler. If you like, you can now look at Section 4.1.3, “A Complex Example” for some of the more advanced topics.

### 4.1.3. A Complex Example

After the simple example of the last section, we will now look at a more complex example. 'Complex' means that we profile a program we don't fully understand, which does a lot of things and has large call stacks. We will explain some basic strategies to pinpoint the most important methods regarding allocations.

#### 4.1.3.1. Getting the Data

The scenario we will use in the example is a search in the Web Dynpro Content Administrator. To avoid seeing one-time effects we have done the search before.

There are several methods we could use to profile the scenario, but we will use the one that will work in all settings: We enable debugging on the node we want to profiler, attach the profiler, enable the Allocation
Analysis perform the login and stop the profiling. This works because the profiler uses the debug port of a VM for profiling (this means that we can even debug while profiling). Let's start by enabling debugging on the node. We assume that you're using the SAP MMC on Windows in this example:

If you refresh the view you should see that debugging is enabled on a specific port (in our example it’s 54026):

The next step is to create a profiling configuration for remote profiling on the given port. Select Profile Configuration... from the Profile menu:

In the following dialog we create a new 'Remote Profiling' configuration:

Now you have a new configuration. We give it a better name and insert the connection properties (the debug port 54026 and the host the VM is running on, which is the local host in our case):
After this you can start the profiler. It will connect to the debug port of the VM we want to profile. Since we haven't enabled any kind of analysis in the configuration, the profiler will only show some basic information about memory and CPU usage. To actually collect profiling data, we will now start the allocation analysis.

In the following dialog select the Allocation Statistic indicator and make sure that Include Line Numbers is selected and the optimization for long running analysis is deselected.
From now on the VM will report every allocation happening to the profiler, which means we can now start our scenario by performing the search:

You have probably noticed that this time the search took significantly longer than without the profiler. This is the overhead of reporting all object allocations to the profiler. If you expand the entry called 'Allocation Analysis' you’ll see the number of allocated objects the VM has reported so far. You should see this number rising fast when you start the search:

After the search is finished, we want to stop the allocation analysis and look at the results. The simplest way to do this is to disconnect the profiler from the VM. This stops the profiling in the VM and returns the result of the allocation analysis up to that point. Just select the node containing the host and port and click the button. Now we have created a snapshot, which we will examine in detail below.

4.1.3.2. Analyzing the Snapshot

Now that we have the data, let us see how we can determine problematic areas. In principle there are many ways by which you can determine these problematic areas and not all work in all situations. Because of this, we will show several ways you can start the analysis.

4.1.3.2.1. Starting with the Allocated Objects

One way to start is by inspecting the allocated objects. In our example you get the following result:
In most cases you will see char arrays at the top, which are created during string processing, followed by the String objects itself. As you can see, while we see char[] at the top, it's directly followed by int[] and Object[].

The next step is to determine which object types contribute significantly to the overall allocations. You can get this information by selecting first the top entry and extending the selection by objects that contribute less. If you look at the status bar, you will see the total size and percentage the selected objects contribute to the overall allocations. Here is the result when only the first four classes are selected:

As you can see, the first four object types contribute 89 percent of the allocated bytes, but only about 75 percent of the number of allocated objects. Since the number of allocated bytes is typically more important than the number of allocated objects, we can conclude that it’s enough to inspect the methods which allocate objects of the first four classes. We hope that only a few methods are responsible for most of the allocations of these objects. While this is often the case, there are situations where the allocations are scattered across a large number of methods, without any significant methods. In this case starting with the allocated objects is not the best solution.

How do we get the information, which methods are allocating objects of a specific class? The first step is to select the class of the objects we are interested in (you could select more than one class, but we will ignore this here). Then you can choose from the context menu:

There are two methods to choose from. The first shows a flat statistic of the methods that directly or indirectly allocate one of these methods. We will start with this view first, but as you will see it’s usually not the best method to choose, especially in complex scenarios. But let us nevertheless take a look. Select the char[] entry and select Show Methods (Flat) from the context menu:
We have sorted the view by the self bytes (the number of bytes allocated directly in the method). Note that in this view we don't see the bytes allocated in the method, but only bytes allocated for character arrays. This is an important concept to remember. Whenever we open another view from this view, only allocations of character arrays will be considered. Let's get back to the method statistic above. As expected, the constructor of `String` and method `AbstractStringBuilder.expandCapacity` are high at the top of the statistic, but the top method `Normalizer.next()` is quite unusual. There's a great chance that these 'unusual' methods are called only from a few or even one method itself, so we can pinpoint the 'offending' method. The next step is to check, which methods are calling the `Normalizer.next()` method. Select it and choose Show Calling Methods (Hierarchical) from the context menu:

What you see is a call tree of the methods calling `Normalizer.next()` which leads to allocations of character arrays. This means that all calls that don't lead to these allocations in `Normalizer.next()` are not shown. The next step is to expand the tree by some levels:

Let us examine in some detail what we are seeing here. First of all we see that the `Normalizer.next()` method is entered by a chain of methods, which is triggered by the merge sort method of the Arrays class. The `Normalizer.next()` is used to compare two strings in a locale-sensitive way. The next step would be to see who triggers the sorting, but let us first determine the meaning of the 'self' and 'total' values of the children of `Normalizer.next()`. In contrast to the method statistic we have seen before, these are not the number of bytes allocated in the method itself, but the number of bytes allocated in the root method (the method at the top of the tree, which in our case is `Normalizer.next()`), when called by the chain of methods up to the method itself. For example 2.27 MB are allocated in `Normalizer.next()`, when called by the chain `Arrays.sort()`
The next thing we would like to find out are the callers of the sort method, but this is not as easy as it might sound. If you look at the screenshot above, you see nine different nodes for the sort() method. That is because the Normalizer.next() method was called from nine different paths. What should we do now? We could just use any of the nodes to dig deeper, since we just want to see who called sort(). But now we use the topmost sort() method because calls to sort() using only a small array will use this path too. Doing this we see that the sort() method was called to sort the entries of our search, an operation you can hardly avoid if you want to have sorted entries.

Now we can go back to the Allocated Objects View we have seen before and select char arrays again. Last time we chose to look at the flat method statistic, this time we select Show Methods (Hierarchical) from the context menu. This shows the methods allocating char arrays and the path by which they were called:

This Methods (Hierarchical) View looks a lot like the flat view we saw before, but you can now dig into the methods without having to make a detour and creating a new view. Expanding the Normalizer.next() method, we get the now familiar merge sort calling chain again.
Very often, you can find very long calling chains with a lot of methods that do nothing more than forwarding:

Since it is cumbersome to expand node after node to get the point where the forwarding chain ends, the profiler has a feature called Smart Expanding. Whenever you press the Shift key while expanding a node, the profiler checks, if one of the children is responsible for almost all of the allocations and if this is the case this child is expanded, too. This goes on until the simple forwarding ends. If you smart expand the first node of the above example, you will get a view like the following:

You see that 78 entries were ’skipped’. What does this mean? It means that we skipped 78 methods in the call hierarchy which are more or less only forwarding to the next method. The method above the skipped methods has allocated 12.2 KB and the method below the skipped method has allocated 12.2 MB, too. So nothing
interesting happened in the skipped methods. But if you really want to see them, just click on the green triangle to the right of the '78 skipped methods’ line.

As you can see, when the call trees get large, smart expansion is a useful tool not only for avoiding many clicks, but also to keep a better eye on the overview. You can even combine items yourself. Just select the items to combine and choose Combine Selected Items from the context menu. Note that you must have selected at least two items and the items must either be children of the same parent or have a parent-child relationship. Here is an example:

![Example of combining items](image)

Note that all items in the parent-child hierarchy are selected. After the items have been expanded, we see the following, less cluttered view:

![Expanded view example](image)

### 4.1.3.2.2. Starting With the Method Statistic

Instead of starting our analysis with the allocated objects, we can start the Methods (Flat) View instead. Just click the Methods (Flat) icon in the overview page of the Allocation Statistic snapshot:

<table>
<thead>
<tr>
<th>Method Statistic</th>
<th>Allocations per Method...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat</td>
<td>Methods (Flat)</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>Methods (Hierarchical)</td>
</tr>
</tbody>
</table>

- **Methods (Flat)**: Allocated bytes and objects per method. The methods are displayed in a flat manner.
- **Methods (Hierarchical)**: Allocated bytes and objects per method. The methods are displayed hierarchically according to their call relationship.
The opening view will show you, which method allocates how many bytes, either directly (the 'self' value) or directly and indirectly (the 'total' value):

Here the methods are sorted by the 'total' number of bytes. As you can see, it is not very interesting in our case, since the top methods are just the ones which are simply called to start the request. But sorting by 'self' bytes is more interesting in our case:

The Normalizer.next() method that we have seen before when inspecting allocations of char arrays is at the top of the allocating methods also generally.

The following entries in the top list are methods of JDK classes, like AbstractStringBuilder or String. Normally, you are not really interested in these methods themselves, you want to know who is calling these methods instead. So the Methods (Flat) Statistic is usually not directly interesting, since it doesn't allows you to directly see these methods. But there is a feature which helps you to make more sense of the method statistic, which 'inline' specific methods. What do we mean by inlining specific methods? It means that allocations done in these methods are treated as if they had been done by their caller (and if the caller is one of the specific methods itself, the allocation is attributed to that method's caller and so on). This means that:

1. None of the specific methods which are inlined will have any 'self' bytes and objects.
2. The 'self' values of the inlined methods are added to that of their callers.
Let us now try this inlining. We want to inline all Methods of the JDK. To do this just choose the button on the tool bar. In the next dialog you can select which methods you want to inline. You do this via a class or method filter. As it turns out the profiler comes with a predefined set of filters which will match all JDK classes, so you just have to check it:

Now you get a new version of the method statistic, where all the methods of the JDK are inlined. As you can see, it looks significantly different from the method statistic with no inlining:

Now the methods with a high ‘self’ value are more likely the ones we are interested in. We can now see that nearly 75 percent of all allocated bytes are allocated directly and indirectly (in inlined methods) in the `compare` method of class `SearchView$SearchResultTextComparator`. For educational reasons, let us look at the same method in the non-inlined method statistic. To do this just select the line and copy the name by using Ctrl-C. Then open the old method statistic view and paste the method in the search field:
As you can see you wouldn’t have found the method when sorting by ‘self’ bytes. And if you sort by ‘total’ bytes, the method is the 68th method, so it doesn’t catch ones eye either, since it is hidden behind the uninteresting methods which do just simple forwarding. So ‘inlining’ is quite a useful concept (at least if you have an idea what to inline). Our next step would be to see, what methods the compare method calls. But can we do this, considering we have inlined the called methods? Yes you can. The profiler is smart enough to turn of the inlining when you want to see the called methods:

A feature, which is sometimes useful, is the ability to group the method statistic by classes, class loaders or packages. This means that all methods of a specific class (or methods of classes loaded by a specific loader or in a specific package) are grouped together. This makes it easy to see for example how much memory was allocated in String methods. To group a method statistic by classes, just click on the button and a new view will be created, which shows only the classes (these are not the classes of the allocated objects, but each class represents all methods defined in the class):

In our example this grouping reveals nothing spectacular, but it is less cluttered than the method statistic itself. And at position 23 (out of 1165) we see Throwable, a class we would not expect to see here, because as you’ve probably been told many times, exceptions are for exceptional situations. Now let us examine which methods of Throwable are called. Selecting the entry and showing the calling methods we get the following result:
To our comfort, we find that the allocations don't happen in the constructor of Throwable, but mostly in tracing. Still interested, we could now check, who is causing exceptions to be thrown. An easy method to achieve this is to open the Allocated Objects View and click on the 🏛 to show only specific types of allocated objects. In the following dialog, check the predefined 'Exceptions' class filter, which matches all Exception objects:

This leads to the following filtered view (we have sorted by the 'Objects' column to see the number of exceptions, since the size of the exceptions is not particularly interesting):

If we found an unexpected number of exceptions this way, we could once again check, who allocates these exceptions by simply selecting them and choosing Show Methods (Hierarchical) from the context menu. This way, we would quickly find the paths that generate most of the exceptions, without having to look at every single stack trace.

### 4.2. Combined Allocation and Method Parameter Analysis

In Section 4.1, “Allocation Analysis Walkthrough” we have learned, how to get a lot of useful information about object allocations and who is responsible for them. The Allocation Analysis makes it easy to find methods that are responsible for the creation of large or numerous objects. But still, there are interesting questions that sometimes can't be answered this way.

Imagine, you know a specific method that has allocated a huge amount of objects. Unfortunately, this method is quite complex and may do completely different things, depending on its parameter values. With the Allocation Analysis alone, it will be very difficult to find out more about the circumstances that lead to the allocations. Do all calls to this method result in allocations or are there some certain 'expensive' parameters, while other
invocations are harmless? Do the allocations depend on the parameters at all? Perhaps they all happen at the first call to this method?

If you already read Section 3.4, “Method Parameter Analysis Using SAP JVM Profiler”, you know that you can answer some of these questions using the Method Parameter Analysis. By defining a method parameter specification for the method in question, you can find out exactly, how many times the method was called, including a distribution of the parameter values together with running times for each value. But now we are interested in information about object allocations and their relation to method parameters!

For questions like these, Allocation Analysis and Method Parameter Analysis can be combined. As you perhaps already noticed, there is a field for defining method parameters in the dialog where the Allocation Analysis is enabled. This field is marked with a red frame in Figure 4.1, “Enable allocation analysis dialog”.

**Figure 4.1. Enable allocation analysis dialog**

Let’s have a look at the short program of Section 4.1.2, “A Simple Example” again. There is a method

```java
getWithPrefix(Properties props, String prefix)
```

that is called for different values of the `prefix` parameter. We found that this method causes quite a lot of allocations and would like to know, if the number or size of allocated objects somehow depends on the invocation and parameters of the method. So we enable the Allocation Analysis and at the same time enter a method parameter specification for this method. To do this, press the button to select method parameters.

**Figure 4.2. Select method parameters**

The Method Parameters Dialog is shown, where defined method parameter specifications can be selected for the analysis. As we have not defined a specification for our method before, we create a new one now.
Figure 4.3. Create new method parameter

We fill in the new method parameter specification to include the `getWithPrefix` method and the second parameter, which is the prefix string. (For more information about this dialog, please refer to Section 3.4, “Method Parameter Analysis Using SAP JVM Profiler”).

Figure 4.4. Specification of the example method parameter

After making sure that the new method parameter specification is selected, confirm the Method Parameters Dialog by pressing the OK button. The new specification is now listed in the profiling configuration and the analysis can be started.

Figure 4.5. Method parameter specification in Allocation Analysis
After the short example program has finished and the profiling run has terminated, we get the well known Allocation Statistic snapshot with all the possibilities of a normal Allocation Analysis. In addition, there is a new entry labeled ‘Method Parameters’ in the snapshot’s overview page.

**Figure 4.6. Method Parameters entry in Allocation Snapshot overview**

After clicking on this entry, a new Method Parameter View is created. For each selected method parameter specification (in our case this is only the `Example.getWithPrefix` method), the number of invocations is shown. What’s new is the information about allocations for each parameter definition. We find that the 4 invocations of the `getWithPrefix` method did not allocate a lot of objects directly, but 307kB were allocated in total because of these invocations.

**Figure 4.7. Method Parameter View for the example**

As we wanted to know whether the object allocations depend on the actual parameter values, we select Show Parameter Values from the context menu of the method parameter definition. This will show the number of invocations and the allocation information for each parameter value that the method was called with.

**Figure 4.8. Method Parameter View with parameter values**

We see that the method was called with the parameter value ‘OMG’ twice, as expected when looking at the source code. It is interesting to note that the parameter values indeed differ in the caused allocated bytes. To further drill down to a potential top-memory-consumer, we could even look at individual invocations. We had two invocations of the method for the parameter value ‘OMG’. Select ‘Show Invocations’ from the context menu of that value to get the object allocations per invocation.
Figure 4.9. Method Parameter View with individual invocations

Note that all these views are perfectly integrated into the normal Allocation Analysis, so for every individual invocation or for every method parameter value, you can get statistics about the allocated objects, the methods that were called for these invocations, and the calling methods as well.

Figure 4.10. Show new statistics for method parameters
Chapter 5. Performance Hotspot Analysis

5.1. Performance Hotspot Analysis Walk-through

This is a detailed walkthrough for the SAP JVM Profiler’s Performance Hotspot Analysis. Target audience are all those, who want to get profound knowledge about the opportunities this profiler offers to analyze the running behavior of Java applications systematically.

The SAP JVM Profiler is an analysis tool that measures the performance of a Java program being executed, particularly the duration of method calls. When we talk about "Performance Hotspot Analysis", we not just mean starting a profiler and interpreting the results afterwards. Performance Hotspot Analysis is rather an iterative process of gathering information about the method calls in the VM (sampling), preparing a statistical summary of the collected data in order to point out potential performance bottlenecks (so called hotspots), trying to minimize the number of hotspots, and then redo the steps to make sure, that the performance really has been improved.

The SAP JVM Profiler's Performance Hotspot Analysis supports users to perform the first step (collecting the data from the system) and the second step (processing and analyzing the data). Therefore the profiler tool is essential for understanding program execution behavior, especially in complex scenarios. Clearly, the profiler can only point to critical code sections, that probably have a high potential for performance improvement. In the end, it’s up to the user to find a better solution in the coding. To sum up, the Performance Hotspot Analysis helps to identify performance problems and ensures, that improvements, done by the user, really have a positive effect.

This walkthrough treats two different scenarios where all necessary steps to perform a complete analysis are explained in detail. The first example, a simple self-contained Java standalone application, shows the basic concepts and approaches to interact with the profiler, whereas the second example concentrates on the more sophisticated analysis methods that are needed to profile parts of complex productive systems.

You can proceed from here in one of the following ways:

• Start the walkthrough with Section 5.1.1, “Performance Hotspot Analysis Overview”. Here you will get a short overview about the objectives of the Performance Hotspot Analysis and how the process of removing performance hotspots is supported by the SAP JVM Profiler step by step.

• Get to know the sample Java application ChecksumServer in Section 5.1.2, “The Checksum Example”

• Go directly to Section 5.1.3, “Retrieving Profiling Data” to learn how to collect profiling data of a Java VM in action, which is the basis for statistical evaluations afterwards.

• Section 5.1.4, “Evaluating Profiling Data” summarizes the various types of statistics, views and charts which help you to identify the performance hotspots very quickly.

• In Section 5.1.5, “Advanced Performance Hotspot Analysis” more advanced concepts and features of the profiler are presented. We show how the profiler can be applied to complex systems without a system restart.

5.1.1. Performance Hotspot Analysis Overview

5.1.1.1. Objectives of Performance Hotspot Analysis

One of the basic facts about writing software is, that developers tend to estimate the most expensive code sections completely wrong (with regard to performance), which leads into a dilemma. One the one hand, developers
are expected to write functional correct, robust, maintainable and of course efficient code. One the other hand, it is very hard to forecast the runtime behavior of an integrated system a priori, even if the single components are understood well. Consequently, it makes no sense to put a major focus on performance issues during the coding phase of the development, but rather in the testing phase when there’s a runnable system with test applications.

The well-known 80-20-rule also underlines this statement: 80% of the CPU time is spent in only 20% of the coding. Identifying these 20% of the coding a priori is nearly impossible. In fact, sophisticated algorithms often show to be ineffective in the context of a productive system in the end.

The suggested approach is to write the code with focus on correctness, robustness and maintainability, without losing sight of general performance issues on a high level (i.e. choose correct collection types and algorithms with respect to O notation). After having a running system (for example a prototype), developers may tune the performance of the system, if the runtime behavior is not acceptable. This is done during the performance hotspot analysis process as described in the next section.

5.1.1.2. Process of Performance Hotspot Analysis

Briefly, the major aim of a Performance Hotspot Analysis tool like the SAP JVM Profiler is to support the user to identify the 20% critical sections in the coding, that lead to 80% CPU consumption. This is usually done by means of an iterative process:

1. Trace the runtime behavior of the running system, i.e. collect profiling data that reflects the actual execution time of method calls in the SAP Java VM. The SAP JVM Profiler makes usage of a sampler backend integrated in the VM to perform this step (see Section 5.1.3, “Retrieving Profiling Data”).

2. After the collection phase, the recorded data is processed offline in the SAP JVM Profiler's frontend (described in Section 5.1.4, “Evaluating Profiling Data”), where various kinds of statistical evaluations can be used to identify the most time-consuming methods (or even code sections).

3. Based on the performance bottlenecks found in the previous step, the developer now may adjust the coding in order to improve the overall performance. It is recommended to start top down with the most expensive calls (see Section 5.3.6, “Misconceptions”).

4. Do step 1 again to verify, that the last changes have a positive effect on the execution time. Check if there are still hotspots worthy to be removed. If yes, repeat the steps until the runtime behavior is acceptable.

5.1.1.3. Evaluating Profiling Data

What kind of data do we need to be able to characterize performance hotspots? Of course, we need time-related data that reflects the execution time of methods which were processed during the profiling run. We’ll learn, that this information alone is not sufficient to find the actual program control paths which cause the performance bottlenecks.

5.1.1.3.1. Self and Total Times

Typically, as a result of the profiling run, profilers show self and total times of methods. They quantify the execution costs of all methods, the VM spent time for. Given that overview, the user may decide which methods should be adapted to achieve a better performance.

The definition of the self and total time of a method is given as follows:
Performance Hotspot Analysis

- The *self time* of a method is the portion of execution time the VM spends with the execution of bytecode instructions within that specific method *excluding* its called methods.

- Whereas the *total time* of a method is the accumulated execution time the VM spends with executing instructions of a method *including* its called methods.

Obviously, the sum of self times exactly makes up the total CPU time consumed in the VM (only in Java code and possibly shared among several CPU cores). Similarly, the total times of the application’s `main`- and `Thread.run`-methods make up the overall CPU time (in Java code).

Reducing the self time of a method by doing some improvements in code will tune the application globally. Clearly, the overall execution time can’t be improved more than the method’s self time. That’s why it’s recommended to first try to improve the methods with the highest self time. Note that the analysis of self times does not take program control paths into consideration (we see an example later).

Reducing the number of calls of a method will lead to a lower execution time depending on the total time of that method. Regarding total times does not take control paths into consideration either, since that method could have been called in multiple control paths.

Let’s demonstrate self and total times of methods in a simple example:

```java
public final class SelfTotalExample {

    private void A() {
        // do some work with 1 unit of cost
        int n = 1;
        // n is set to 9 accidentally, nearly no time consumption
        ...  
        for(int i=0; i<n; ++i) {
            C();
        }
    }

    private void B() {
        // do some work with 9 units of cost
        ...  
        C();
    }

    private void C() {
        // do some work with 1 unit of cost
        ...  
    }

    public static void main(String[] args) {
        A();
        B();
    }
}
```

`SelfTotalExample`’s `main`-method calls `A` and `B`. `A` itself calls `C` `n` times. Assume now that `n` is unnecessarily much higher than expected, let’s say 9 (instead of 3). `B` makes some calculations which last 9 units of time and then calls `C`. Executing `C` costs exactly 1 unit of time. Then the corresponding self and total times are as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Self time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

As expected, the self times sum up to 20, which is `main`’s total time. As `n` is too high accidentally, we get a much higher running time than expected. Do the listed self times respectively the total times give a strong hint to the problem?
Let's have a closer look to the values now. When looking at the self times, only B and C seem to have potential for optimization. But we know from source code, that both methods can’t be tuned. What about the total times? All total times of called methods are equal, so we can’t identify a bottleneck here either. In practice, the profiled applications are much more complex with thousands of different method calls, which makes the analysis even harder on this global-orientated level.

Most profilers manage this problem by providing the user with additional information which take call stack information into account. As we can see in the next section, this helps to drill down the performance hotspots hierarchically.

### 5.1.1.3.2. Method Call Hierarchy

As we have seen, self and total times help to estimate the CPU time which can be saved, when a method is optimized or the number of its calls is reduced. The point of view is globally in this context, which forms a major obstacle in searching for hotspots. Therefore most profilers offer calling hierarchy dependent times, i.e. the self and total time don’t refer to all calls of a method anymore, but to the calls of a method with the same program control path to it (a unique sequence of method calls on the stack with main as the root).

The definition of the self and total time of a method sequence is given as follows:

- The **self time** of a method sequence is the overall execution time, the VM spent within the last method in the sequence (and not in its called methods), where the sequence reflects the current call stack.

- Similarly the **total time** of a method sequence is the accumulated execution time, the VM spent with executing the last method of the sequence or one of its called methods. Again, the method sequence must correspond to the current call hierarchy.

Providing the user with hierarchy-dependent self and total times, it is possible now, to split up the self and total time of a method call into the hierarchical self and total times of all call stacks with that method call on top. The table below shows the values for the different possible call stacks:

<table>
<thead>
<tr>
<th>call stack (bottom to top)</th>
<th>Self time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>main, A</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>main, A, C</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>main, B</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>main, B, C</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It's easy now to isolate the code section which causes the performance problem. Beginning from main, we see that the total time is split into equal parts via calls to A and B. As the total time of "main,B,C" is only 1, it is clear, that "main,B" has a self time of 9. Since in this example we can't improve self times, we consider "main,A" with 10 units of total time. In contrast to "main,A", "main,B" has nearly no self time. 90% of the running time on this path are spent in "main,A,C", which leads to the conclusion, that C is called from A more often than expected.

### 5.1.2. The Checksum Example

In Section 5.1.1, “Performance Hotspot Analysis Overview” we introduced the basic ideas behind the process of Performance Hotspot Analysis. In particular, we pointed out, how users can make usage of profiler tools in order to:

1. measure the runtime behavior of running systems
2. interpret the delivered self and total times to identify performance problems (hotspots)

3. verify, that improvements in code has removed hotspots

In Section 5.1.3, “Retrieving Profiling Data” and Section 5.1.4, “Evaluating Profiling Data” we show the Performance Hotspot Analysis with the SAP JVM Profiler in practice on the basis of the simple Checksum example introduced in this section.

5.1.2.1. Installing the Checksum Example

The file "checksum_example.zip" contains the complete source code and the sample XML files on which the checksum application operates. Extract the content of the ZIP file into a directory and import the Java project into Eclipse:

1. choose Import... in context menu of the package explorer
2. select Existing projects into Workspace and click Next >
3. press Browse... and select the unzipped example folder in the file dialog
4. end the wizard by pressing Finish

The resulting package view should look like this now:

![Package Explorer image]

The example project consists of

1. the ChecksumTest class located in the default package
2. the ChecksumServer class in package com.sample.checksum
3. the products folder containing a set of sample XML files

The source codes of the example can be opened and viewed by double-clicking the corresponding Java files in the package explorer.

5.1.2.2. Running the Checksum Example

We demonstrate the Performance Hotspot Analysis by profiling a simple Java-based server, which calculates a binary checksum over a set of files in the local file system on demand. The Checksum example consists of three parts: the ChecksumServer class implements the checksum service, the test class ChecksumTest is
the test driver (with the main method) and the test data files, the server operates on, are located in the products folder.

Have a closer look now to the ChecksumServer class. After having been created and started via start(), the ChecksumServer provides a service to calculate a checksum over a set of files. A client may create a ChecksumRequest object with a list of absolute file names stored in fileNames. Such a request object may be passed to the server object by calling calcChecksum. The client thread is blocked until the server has stored the result into ChecksumRequest.checksum together with a unique request id in ChecksumRequest.requestID. Several client threads may pose requests asynchronously. The server internally queues up requests to be completed until it is shutdown by calling its method stopService.

ChecksumTest in the default package contains a test driver for the server posing thousands of (partial parallel) checksum requests. In the main method, the server object is created and the service started. Then, for a period of time passed as a program argument (by default 30s), the test code generates new requests by choosing a random subset (every second file) of the XML files in the product folder. Each parallel request, represented by an active thread in a thread pool, waits for the server to return the checksum of the specified files. After the request period is over, some statistical information is printed out, in particular the number of requests the server could finish during the active time (its throughput).

We now start a test run. In the package explorer open the context menu for ChecksumTest and choose Run As...->Java Application. The currently finished requests should be printed out to the console output as long as the test is running:

```
Finished request 32 (result: 6574)
Finished request 33 (result: 33069)
Finished request 34 (result: 19779)
Finished request 35 (result: 46521)
Finished request 36 (result: 68104)
Finished request 37 (result: 64104)
Finished request 38 (result: 26471)
Finished request 39 (result: 26666)
Finished request 40 (result: 56295)
Finished request 41 (result: 55889)
Finished request 42 (result: 12971)
Finished request 43 (result: 38285)
Finished request 44 (result: 60773)
Finished request 45 (result: 65104)
Finished request 46 (result: 55991)
Finished request 47 (result: 65700)
Finished request 48 (result: 47921)
```

The summary at the end shows that the server processed only 48 requests within 55.18s, making a throughput of about 0.87 request/s. There seems to be a high potential for performance improvements, but it’s hard to say, which part of the server is responsible for that poor running behavior. The low number of finished jobs could be caused by the huge number of checksum calculations, intensive I/O-operations or even by a bad synchronization in the server when processing the request queue. As outlined in Section 5.1.1, “Performance Hotspot Analysis Overview”, this is where the SAP JVM Profiler comes into play, helping point out the performance bottleneck(s) explicitly.

By the way, Eclipse automatically created a new run configuration with name ChecksumTest when we started the application. This configuration can be viewed in the Run Configuration dialog under Java Application (click menu Run->Run Configurations...). A run configuration is a collection of all the parameters that are needed to run the Java application. Amongst others, it particularly specifies the main class, the arguments for the program as well as for the VM, the classpath and the path to the JRE. In order to adapt the request period in the checksum example (which is 30s by default), just type another value into the input field named Program arguments in tab Arguments.
After a short theoretical introduction into different basic types of Java profilers, Section 5.1.3, “Retrieving Profiling Data” describes how the SAP JVM Profiler can be applied to gather performance-related information about the running server.

5.1.3. Retrieving Profiling Data

Before we get started with the Performance Hotspot Analysis of the checksum example, we take the liberty to have a short look into the very basic technical backgrounds of Java profilers. We can't discuss that in great depth, but it's essential to understand the different approaches of profilers, as the chosen technique to measure the VM not only has a great impact on the quality of the results, but also on the provided analysis features.

5.1.3.1. Classification of Java Profilers

Typically, during being active in profiling period, a profiler backend observes the running VM and generates a stream of profiling data reflecting the execution times of methods. This stream is recorded by a frontend instance, which processes the gathered data into statistic views afterwards (some profilers like the SAP JVM Profiler also allow to open statistics while currently collecting data).

Depending on the way profilers gather profiling data about the VM, Java profilers can be divided into two main categories:

1. **Instrumentation-based** profilers need to adapt the code of the executed application in order to perform the measurement. The code added by the profiler may trigger actions like recording the elapsed time between
entry and exit points of Java methods, build up a call graph and count the method invocations. Most Java profilers perform the instrumentation via the JVMTI interface of the VM (see http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/). As most VMs don't offer online attaching of JVMTI agents, it is generally necessary to restart the VM before the profiling session may begin.

The major advantage of this approach is, that the execution can be traced in detail, leading to accurate results (if some additional conditions are met). The downside of this approach is, that the instrumented methods are expected to execute very slowly as (nearly) all methods have to perform additional instructions and less optimizations can be performed by the JIT compiler.

2. In contrast, statistical profilers take samples of the running system at regular intervals without having to touch the code to be executed. While being active, the sampler shortly interrupts the current Java threads, examines their stacks and assembles the information into a single sample. Based on a sufficient number of samples, the execution times of method calls can be statistically estimated.

Very little overhead is created, because the points of time when the system is interrupted for taking the sample have a distance of several milliseconds (e.g. 10ms). This way the Performance Hotspot Analysis can be performed without influencing the running system too much which prevents from distorted results. Moreover the sampler can be attached to a running system, which does not need to be restarted for the purpose of instrumentation. The main risk is the quality of the results, which depends on an adequate setup of the profiling parameters such as sampling interval, sampling duration etc.

The SAP JVM Profiler is based on statistical sampling and therefore creates low overhead. For each current Java thread in the VM, the samples sent to the frontend in the Eclipse application contain the following information:

- the unique id of the stacktrace when the thread was stopped. This corresponds to the ordered list of method calls in the call graph (the stacktrace information itself is communicated to the frontend only once separately)
- a flag if this specific Java thread was currently running or not (also when executing in native code).

The number of samples with a specific stacktrace implies the self and total time of a method call in the call graph for a thread. Simple aggregations can be obtained when relaxing from the call graph and the current thread to get a global view of the method calls. Similarly, sleeping threads can be filtered out in the analysis when only running times are of interest (see the profiling parameter Ignore sleeping threads).

Given a running VM, the user may decide about the profiling periods during which the sampler is active. How to make the sampler taking samples periodically, is explained in Section 5.1.3.2, “Starting a Performance Hotspot Analysis”. The collected profiling data (the samples), which is gathered during such a sampling period, is packed into a so-called profiling snapshot. The SAP JVM Profiler offers comprehensive methods for evaluating and generating statistics based on snapshots (see Section 5.1.4, “Evaluating Profiling Data”). For getting a deeper understanding about the used techniques in the profiler backend and the resulting advantages and limitations, please have a closer look into Section 5.3, “Performance Hotspot Analysis - Technical Documentation”.

5.1.3.2. Starting a Performance Hotspot Analysis

To sample the checksum server with the SAP JVM Profiler, we have to start the ChecksumTest application in profile mode. Technically, this means that

- the sampler backend in the VM is setup for profiling during the program execution
- a communication connection is established between the sampler backend and the profiler frontend located in Eclipse
- the profiler frontend collects the sample packets periodically sent by the profiler backend
- the sampling process is controlled by the frontend (start, stop, take snapshots etc.)
Similar to run configurations used to start Java applications from Eclipse, so-called profile configuration are necessary to start applications in profile mode. Open the profile configuration dialog by selecting **Run->Profile...** (Alternatively choose Profile Configurations... from the Launch application button). Since we have already started the checksum example in normal mode, a run configuration has already been created. This configuration can be completed with profiling specific parameters that can be edited in the **Profiling** tab (the profile configuration is therefore simply an add-on to the existing run configuration and not a completely new configuration).

Before we start the application in profile mode, we set the correct parameters in the **Profiling** tab. On the left side (**Profiling Analysis**) the different kinds of analysis the profiler can perform are listed (i.e. **Allocation Analysis**, **Performance Hotspot Analysis** and **Method Parameter Analysis**). We choose the Performance Hotspot Analysis by simply activating its check box. The **Analysis Options** on the right can be left untouched as the default values are sufficient for the moment. We'll see some use cases for these parameters in Section 5.1.5, "Advanced Performance Hotspot Analysis".

Having a profile configuration, the application now can be profiled by clicking the **Profile** button. A message box appears:

Since the SAP JVM Profiler's frontend tightly works together with a special profiler backend that is implemented in the SAP Java VM (either version 5 or 6), it is not possible to profile the checksum server running on a different VM.

After having confirmed the suggestion, we are directly taken to the **JRE** tab where we can choose an installed SAP Java VM in the combo box under **Alternate JRE**. If no SAP Java VM appears in the combo box already, you'll have to add an entry for it by clicking **Installed JREs**...
Now the profiling run can be started by clicking OK. Depending on the current perspective in Eclipse, another message box may appear suggesting to switch to the Profiling Perspective:

Now the checksum application is running and the profiler is active. As long as the VM is working, the profiler backend (the sampler) will send the taken samples to the frontend where an overview of the profiling process is presented to the user.

### 5.1.3.3. Profiling Configuration View

In the Profile view the profiler gives feedback about the profiling scenario, the current status of the profiler and the chosen analysis. The same information is provided for scenarios, where the profiler is attached online to an already running VM (see Section 5.1.5, “Advanced Performance Hotspot Analysis”) as well as for imported scenarios that were stored to a snapshot file before.

While the checksum application is in progress of being sampled, the Profile view shows information about the profiling session which is going on. After the application has completed, the tree in the view changes slightly. The different icons have the following meaning:
Performance Hotspot Analysis

<table>
<thead>
<tr>
<th>Icon</th>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Icon" /></td>
<td>ChecksumTest</td>
<td>This is the root element embracing all information about a profiling session.</td>
</tr>
<tr>
<td><img src="image2" alt="Icon" /></td>
<td>ChecksumTest at localhost:52034</td>
<td>Indicates that the profiler is currently connected to the localhost at debug port 52034.</td>
</tr>
<tr>
<td><img src="image3" alt="Icon" /></td>
<td>Performance Hotspot Analysis</td>
<td>Shows that a performance analysis is running, i.e. that the frontend is receiving profiling samples from the backend.</td>
</tr>
<tr>
<td><img src="image4" alt="Icon" /></td>
<td>&lt;terminated&gt; ChecksumTest at localhost:52034</td>
<td>Indicates that the profiler is disconnected now.</td>
</tr>
<tr>
<td><img src="image5" alt="Icon" /></td>
<td>Duration 0:00:16</td>
<td>Shows the connection time.</td>
</tr>
<tr>
<td><img src="image6" alt="Icon" /></td>
<td>Performance Hotspot Statistic_1</td>
<td>Represents the (taken) performance hotspot snapshot. This is the starting point for opening further statistics.</td>
</tr>
</tbody>
</table>

See Section 8.1, “Profiling Configuration View” for further reading.

5.1.3.4. Application Profiling Overview

Besides the periodic samples, the profiler backend sends various other information to the frontend while the application is being profiled (so-called heartbeat information, detailed description see ??). On top of the Application profiling tab, which can be opened by clicking the corresponding profiling session icon in the Profile view, the Profiling Lifecycle is presented graphically. With a resolution of one second, the CPU as well as the memory consumption are illustrated in a graph with different colors. The points of time a GC is triggered in the VM are marked with small blue bullets.

![Graph](image7)

The graph gives a first overview about the used resources. Obviously, the checksum server does not exhaust the CPU power, only about a half of it is consumed. As we ran the application on a dual core machine, one core seems to be idle in the average. We’ll examine this behavior in Section 5.1.4, “Evaluating Profiling Data” where we try to identify the performance bottlenecks of the checksum server.

The timeline in the Analysis overview shows at each point of time, which kind of analysis trace is currently active. In the example we have only started a Performance Hotspot Analysis which is indicated by the red bar.

![Timeline](image8)

After the application has finished, the Performance Hotspot Analysis automatically ends and a snapshot is created. In the Analysis overview appears now a new row for that snapshot named Performance Hotspot Statistic_1.
Additional information about the taken snapshot are presented in several columns. The number of samples the sampler sent to the frontend is 5,415 covering a period of time of 57s. Thus the sampling rate in this snapshot is about 10ms (as expected). Since we profiled the complete application run, the snapshot consequently covers the complete execution time. We'll see in the next section, that it's also possible to create snapshots that cover an arbitrary time span within the execution time span.

The newly created snapshot named *Performance Hotspot Statistic_1* is added as a new tree node in the *Profile* view under the profiling session's root element. The connection icon has changed to indicate, that the profiler is no longer connected to the VM:

In the snapshot's context menu there are two commands to manage the snapshot:

- **Delete**: removes the snapshot (and all statistics based on it)
- **Rename**: shows a dialog to enter a meaningful name for the snapshot

Since we have a valid snapshot now, we can take it as the basis for further analysis in various statistics. A double-click on the row *Performance Hotspot Statistic_1* in the *Analysis overview* directly takes us to the statistic overview. Alternatively, the context menu for the snapshot entry in the *Profile* view offers direct links to the possible statistics. We'll illustrate the numerous possibilities in Section 5.1.4, “Evaluating Profiling Data”.

### 5.1.3.5. Working with Snapshots

Before proceeding with opening the different kinds of statistics for our taken snapshot, as described in the next section, we give a short overview about the various ways to create snapshots of even different analysis traces.

In general, if the profiler is connected to the running SAP Java VM, the user is allowed to take direct control about how the system is profiled and how profiling data is organized into snapshots. This is done in the *Analysis Overview's* toolbar within the profiling view.

When the profiler is active (i.e. currently collecting data from the backend), the user is allowed to take snapshots whenever he wants to (by pressing the button). A snapshot finishes the current data collection and
starts a new collection right afterwards. The trace itself is not stopped. Thus a snapshot represents a profile view within a specific period of execution time. Snapshots help to analyze dedicated periods of execution time independently.

The currently running trace (either for Performance Hotspot Analysis, Allocation Analysis or Method Parameter Analysis trace) can be stopped by clicking this button. A snapshot is taken and the frontend remains connected to the SAP Java VM.

When this button is active (i.e. currently no analysis trace is running), the user may start a new Performance Hotspot Analysis, Allocation Analysis or Method Parameter Analysis trace without having to establish a new connection to the VM. The Analysis overview will then show another bar for the newly started trace.

We illustrate the possibilities of controlling the sampler with the checksum example. First we make the checksum application work for a longer time by adjusting the request time. As described in Section 5.1.2.2, “Running the Checksum Example”, change the checksum program argument to 1000 in the Profile Configuration dialog and start a new profiling session.

After a certain time of sampling, click the snapshot button in the Analysis Overview's toolbar.

A dialog appears with an input field containing the proposed name for the snapshot to be created. You can type in a different name or just click OK.

All the samples collected so far are grouped into this snapshot which appears as a new row now:

Capturing a second snapshot leads to the following picture:
Now we try out how the stop and start trace buttons interact. As the performance trace is currently active, we are allowed to stop it by pressing the corresponding button in the Analysis overview. As an effect, the red trace bar in the first row of the table stops growing with time, which indicates, that the trace is turned off now.

To start the trace again (i.e. to make the sampler in the backend send profiling packets), click on the start button. Now we are asked which specific trace we want to start.

It's possible now to setup a complete new set of parameters for the new trace. We decide to continue with the performance trace using the default parameters and confirm the dialog with OK. We now get a new table view like this:

Obviously, the profiling session has a break of about 40s where no trace was running. It's legal to switch the kind of current trace within a profiling session (i.e. within the phase of being connected to the VM) as often as necessary, however traces never can overlap.

Suppose now we are only interested in the performance behavior of the server after a certain warm-up phase. More precisely, let's say we want to examine a 30s - time period after 1min warm-up execution time of the
server. We could restart the application and take a first snapshot after 1 min and a second 30s later. The problem with this approach is, that it's hardly possible to do that exactly by creating the snapshot on the fly. Moreover the user is forced to watch the process very carefully and take the necessary actions online.

To overcome these obstacles, the SAP JVM Profiler offers the possibility to define new snapshots that cover a certain period of time in the Profiling lifecycle view. The profiler may be either connected or disconnected to perform this action. Just click into the life-cycle graph at point of 1 min and drag the mouse to cover 30s of execution time.

Then push on the Create Snapshot button the create a new snapshot that has exactly this range.

The new snapshot now is listed in the Analysis Overview and can be used for further analysis just like the snapshots directly taken during profiling. It's a favored approach to trace the system over a longer time of period including all time spans interesting for analysis. After the profiler is disconnected, the snapshots are “cut out” from the overall snapshot and can be analyzed in various statistics.

5.1.4. Evaluating Profiling Data

Based on the information the profiler backend sent to the frontend in form of a sample stream, the frontend provides comprehensive analysis methods to find performance bottlenecks. As we saw in the last Section 5.1.3.5, “Working with Snapshots”, snapshots form the basic unit of sample collections on which the hotspot analysis is performed on. So the starting point for an analysis is always a snapshot.

5.1.4.1. Snapshot Overview

After having profiled the checksum example (let's say for 30s), we got a default snapshot named Performance Hotspot Statistic_1, which appears firstly in the Profile view and secondly as an entry in the Analysis Overview. The Performance Hotspot Snapshot view can be opened by double-clicking one of these locations.

The Performance Hotspot Summary within the Performance Hotspot Snapshot view shows basic information about the taken snapshot:
• The number of samples that make up the snapshot (5367 in the example)

• The collection period is the time between the last and the first sample (56s here). Note: short collection periods should be avoided as they imply a low number of samples. Having gathered only a few samples may be a hint, that the quality of the measurement is not acceptable due to its statistical nature.

• There's also a label informing us, if sleeping threads were ignored or not. As we have turned on this flag in the former profiling run, the statistics we open for this snapshot won't show any sleeping times (i.e. they all will be zero).

The Statistics overview offers links to the various statistics that can be opened for the snapshot. This section describes the different types of performance hotspot statistics that can be opened for a snapshot. Each statistic will be opened in an own view. The following basic types of statistics are available:

**Methods (Flat)**

The Methods (Flat) statistic provides the self and total times of method calls in a flat table, i.e. all threads are accumulated and no calling hierarchy is regarded. This corresponds to a global point of view (see Section 5.1.4.2, “Called Methods (Flat) Statistic”).

**Methods (Hierarchical)**

The Methods (Hierarchical) statistic presents self and total times of method calls in a tree view beginning with the threads' root methods. In this statistic the user can recursively inspect the runtime behavior of the application's control paths (see Section 5.1.4.3, “Called Methods (Hierarchical) Statistic”).

**Threads by ID**

**Threads by Name**

The Threads statistic summarizes the running times the VM spent in the different Java threads (Section 5.1.4.5, “Threads Statistic”).

Beside the very basic statistics presented so far, there are also more advanced statistics available at the bottom of the overview. These are mainly helpful when sampling an application on the SAP NetWeaver Application Server Java. Basically, they help to drill down the performance issues related to a specific user, session, request or application. We'll demonstrate the usage of these statistics in Section 5.1.5, “Advanced Performance Hotspot Analysis”.

**Users**

The Users statistic shows the running times separated by the different users.
The *Session* statistic shows the running times separated by the different sessions.

The *Requests* statistic shows the running times separated by the requests (e.g. different HTTP requests to a web application).

The *Applications* statistic shows the running times separated by the applications.

The *Components* statistic shows the running times of methods grouped into the components.

Note that the different statistic views (even some more) are also available via the context menu in the *Profile* view:

### 5.1.4.2. Called Methods (Flat) Statistic

Now please open the *Called Methods (Flat)* statistic either via the hyperlink in the *Statistics* view or the context menu of the snapshot in the *Profile* view. Initially, the called methods are listed in a flat table together with their self and total times.
5.1.4.2.1. Statistic View Navigation

On top of the Methods (Flat) View we see a navigation bar that helps us keep the overview on the currently active statistic. We see in the example, that the Methods (Flat) Statistic directly is based on the snapshot we took, i.e. it contains system wide methods calls within the collection period. We’ll demonstrate some examples, where derived statistics are created out of other statistics. Those derived statistics generally represent only a special subset of the snapshots method calls, e.g. the method calls a specific thread invoked (see Section 5.1.4.4, “Called and Calling Method Statistics”). The previous views in the navigation chain get activated when their hyperlink is pressed.

5.1.4.2.2. Methods (Flat) Statistic Toolbar

The first button group of the toolbar, the input field together with the previous and next button, enables the user to search for an occurrence of a special method in the list.

Although the method list is quite short in our example, we use this feature to search for all `run` methods (which are the entry methods for Java threads) in the table. Just start to type "run" into the search field.

All occurrences having the typed search string as prefix will be highlighted yellow. With the arrow buttons you are able to iterate all matches until the desired method is found. To remove the selection, just clear the search field.

The remaining buttons on the right side of the toolbar have the following meaning:

<table>
<thead>
<tr>
<th>Icon(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Controls the layout of the table. Several pre-defined columns can be chosen as described in Section 5.1.4.2.3, “Methods (Flat) Statistic Table”.</td>
</tr>
</tbody>
</table>
5.1.4.2.3. Methods (Flat) Statistic Table

For each method call, or to be more precisely for each method call for which there is at least one sample, the Methods (Flat) Statistic shows self and total times by default. As we profiled with ignoring sleeping threads, these values are the same as Self Running Time and Total Running Time. Click the Total Time column header to sort the methods by decreasing times (if the methods are sorted increasing, just click on the header again).

We get the following result:

Generally the table may be sorted by any column, either increasing or decreasing. Another nice feature is to get accumulated values for the selected set of method calls. Selecting an arbitrary set of rows (Ctrl-a will select all) will trigger the output of accumulated self times in the status bar of the main view:

Note that the sum of all self times is limited by the collection time multiplied by the number of CPU cores. When we select all 160 methods, we get an accumulated self time of 57.4s which is only a bit more than the elapsed execution time of 55.18s (here the execution time is the collection time). This implies, that the application runs nearly single-threaded in the average.

We now want to figure out, in which method (plus its called methods) the VM was active most of the time. On top of the list, there are four run methods: Thread.run(), ThreadPoolExecutor$Worker.run(), ThreadPoolExecutor$Worker.runTask(), ChecksumServer$1.run() all taking about 57.1s, which is the overall collection time.

Why are there four run methods with that high amount of total time? To answer this question we dig into the most general one, Thread.run() by opening the Called Method Statistic of this single method. Select the row of Thread.run() in the table, open the context menu and click Show Called Methods.
The statistic we have just opened is of hierarchical type. It shows all total and self times of the called methods in a tree view. As we generated this statistic view for a selected method (i.e. `Thread.run()`), only its called methods are listed. We easily see that the 57.1s are consumed on a single control path starting from `Thread.run()` to `ChecksumServer$1.run()`, which means that most of the CPU time is spent in `ChecksumServer$1.run()`. A closer look reveals, that it is mainly `FileInputStream.read()` the VM has to process. Here the hierarchical view helped us to figure out the concrete performance bottleneck, whereas the flat view of our initial method statistic nearly had no informative value in this context. Close the temporary hierarchic view by clicking its close button. We’ll have a closer look on the Methods (Hierarchical) Statistic in Section 5.1.4.3, “Called Methods (Hierarchical) Statistic”.

Alternatively, we open a method statistic with JDK methods inlined into application’s methods. In such a statistic we should see only times of methods from our project Checksum with accumulated JDK methods. We do this by clicking the Inline Methods by Filter... button in the toolbar and activate filter group JDK classes.

It's clear immediately, that the 57s running time is consumed exclusively in `ChecksumServer$1.run()` (just like the previous analysis has shown), only a very small fraction can be found in the client's main method.

We know now, that most time is spent in `ChecksumServer$1.run()`, but it's not yet clear, why this method has this poor runtime behavior. `ChecksumServer$1.run()`'s self time of 306ms gives us the hint, that the code within this method hardly can have such negative effects, so one of its called method must be the hotspot. To answer the question, we sort the methods in the Methods (Flat) Statistic by descending self times:

Obviously, the view with sorted self times points to `FileInputStream.read()` as the major CPU consumer and is therefore the top candidate for being improved.

Now we want to examine the source code, where `FileInputStream.read()` is called.

```java
public void run() {
    task.checksum = 0;
    for(String fileName : task.fileNames) {
        try {
            FileInputStream fis = new FileInputStream( new File(fileName) );
            int nextChar = -1;
            while( (nextChar = fis.read()) != -1 ) {
                task.checksum = updateSum(task.checksum, nextChar);
            }
            fis.close();
        } catch (FileNotFoundException e) {
        } catch (IOException e) {
        }
    }
    synchronized(task) {
        task.notify();
    }
}
```
One does not need to be a Java expert to detect the problem: the developer of the method simply forgot to perform a buffered IO. FileInputStream.read() reads every single byte from disc in a single OS operation, which is extremely expensive.

The performance problem is fixed easily. We just wrap a BufferedReader around the FileInputStream. Now a larger chunk of bytes is retrieved from the file in each IO operation:

```java
public void run() {
    task.checksum = 0;
    for(String fileName : task.fileNames) {
        try {
            FileInputStream fis = new FileInputStream( new File(fileName) );
            BufferedReader br = new BufferedReader( new InputStreamReader(fis) );
            int nextChar = -1;
            while( (nextChar = br.read()) != -1 ) {
                task.checksum = updateSum(task.checksum, nextChar);
            }
            br.close();
            fis.close();
        } catch (FileNotFoundException e) {} catch (IOException e) {} 
    }
    synchronized(task) {
        task.notify();
    }
}
```

As expected, when we run the new server code, the throughput has increased from about 0.8 requests/s to 8.65 request/s (see the Console output), which is an enormous improvement.

### 5.1.4.3. Called Methods (Hierarchical) Statistic

In last Section 5.1.4.2, “Called Methods (Flat) Statistic” we introduced the (flat) Methods (Flat) Statistic which offers a global point of view over the methods’ performance. In this section, we get the know the Methods (Hierarchical) Statistic which mainly adds the opportunity, to distinguish the performance of method calls with a different call graph location (calling context).

Just go back to the Performance Hotspot Statistic overview and open the Methods (Hierarchical) View. Alternatively press the button Show Called Methods... in the context menu of the snapshot.

Unlike of the table-based Methods (Flat) View, the Methods (Hierarchical) View shows a method tree that reflects the call graph of the application. Starting from the root methods, you may follow the call stacks by clicking the small arrow left of the method name.

Note that a specific method may now appear in different rows simultaneously, which is clear since it could have been called with a different call graph (i.e. the call stacks differ). As an illustration we completely expand the tree by pressing the Expand All Items toolbar button in the view and search for close.
In the subtree of `com.sample.checksum.ChecksumServer$1.run()`, we find two different usages of `java.io.FileInputStream.close()`. The first call is a child node of `java.io.BufferedReader.close()` and has a total time of 2.25s, whereas the second is a direct successor of the root node with a total time of 53.4ms.

In order to find the corresponding locations in the source code, we fade in the line numbers where the methods are called in `ChecksumServer$1.run()`. Therefore we click the small arrow on the right side of the methods name, and the line numbers appear now in its subtree:

```
We can read that line 98 and line 99 are the source code locations corresponding to the different subtrees that call `FileInputStream.close()`. Obviously the first call does the complete job, so line 99 could be removed (but that's not our main focus when profiling the application).

Now let's see where the most CPU time is consumed. First, collapse the tree in the Methods (Hierarchical) View by clicking the corresponding toolbar button. When looking at the total times of the root elements, which directly correspond to the threads' entry points, it is eye-catching, that most of the time falls into `Thread.run()` (we saw that already in the Methods (Flat Statistic)). So we follow the call graph starting from this root element, since all other nodes surely have only very small potential for performance improvements.
A handy feature is to expand recursively a specific subtree in a way that all expensive paths are visible. We make use of this by clicking the expand arrow left of the node for `Thread.run()` with the ctrl-key pressed:

![Screen capture showing the performance hotspot analysis](image)

This immediately leads us to the view we need to find the most expensive calls within `ChecksumServer$1.run()`. Move the mouse cursor over the total time cell of this method to get a tooltip, which shows the exact total time in nanoseconds and its overall percentage (98.11% here).

![Tooltip showing the total time of `ChecksumServer$1.run()`](image)

The successors of `ChecksumServer$1.run()` in decreasing order in total time are `FileInputStream.<init>()` (62.18%), `BufferedReader.read()` (22.41%), `BufferedReader.close()` (7.03%) and the remaining calls (3.91%). Clearly, `FileInputStream.<init>()` is the top consumer and not `BufferedReader.read()` as before.

Just open `FileInputStream.<init>()`'s branch to get an idea, why this method is expensive.

![Screen capture showing the `FileInputStream.<init>()` call tree](image)

The call of `java.io.FileInputStream.open(String)` nearly makes up the total time of its parent. Being a native call, we can't expand the node further, neither can we optimize the `open` method as it is part of the JDK. Additionally, the method’s self time equals its total time, consequently we can stop our search here anyway.

How could we obtain better results? Each request comes with a (large) set of files for which the server must calculate the checksum. Keeping a mapping between a filename and the file's checksum is not an option since the checksum operation is not associative. Also caching the file handles is not possible, as we don’t want to keep lots of files open.

We try what happens, if we replace the IO logic with an approach based on `java.nio`. The `run` method now look like that:

```java
public void run() {
    task.checksum = 0;
    for(String fileName : task.fileNames) {
        try {
            FileInputStream fis = new FileInputStream( new File(fileName) );
            FileChannel fc = fis.getChannel();
            MappedByteBuffer bb = fc.map(FileChannel.MapMode.READ_ONLY, 0, (int)fc.size());
            while( bb.hasRemaining() ) {
                task.checksum = updateSum(task.checksum, bb.get());
            }
            fc.close();
        }
    }
}
```
Having adapted the method, we do another profiler run and examine the results. In fact, we get a slightly better throughput of about 9.52 requests/s. We analyze the profiling run in the Methods (Hierarchical) View.

Just as expected, `java.io.FileInputStream.<init>(File)` is as expensive as before. Also closing the channel (i.e. the file) doesn't seem to be more efficient compared to the previous version. The little improvement obviously comes from the way, the bytes are read. Internally making use of caching on OS level, `FileChannelImpl.map` is faster than `BufferedReader.read` in this context.

### 5.1.4.4. Called and Calling Method Statistics

Beyond doubt, the Methods (Flat) Statistic and Methods (Hierarchical) Statistic provide the user with well-arranged global information about method execution times. However, to perform a Performance Hotspot Analysis with a local focus on parts of the method call graph (e.g. consider only a special subtree in the call graph), the Called Method Statistic or the Calling Method Statistic of a method statistic can be applied. Both types of statistics are secondary, i.e. they are derived from a method statistic which represents a complete snapshot.

#### 5.1.4.4.1. Called Method Statistics

A popular use case of the Called Method Statistics is to investigate all the callees of a specific method in the Methods (Flat) Statistic. As we learned before, the Methods (Flat) Statistic doesn't give us any hint how a method call's total time is contributed to its callees, as there is no call graph information (flat view).

In our checksum example, have a look into the Methods (Flat) Statistic again. The values for `FileInputStream.<init>(File)` show a high total time of 21.8s, whereas its self time is only about 34ms. We can conclude, that this constructor itself makes expensive calls. In order to investigate these calls, just choose Show Called Method Statistic in the context menu of the selected constructor.
A new Methods (Hierarchical) Statistic is opened with `FileInputStream.<init>(File)` as the single root node. The tree, after having been expanded, shows all self and total times of the callees. It's clear now, that `FileInputStream.open` is the cause of the high CPU consumption.

Note that the derived statistic is a complete Methods (Hierarchical) Statistic that offers all the operations introduced in Section 5.1.4.3, “Called Methods (Hierarchical) Statistic”.

Given a method node (or even a set of) in a hierarchical method statistic, just as the called method statistic created before, the user is enabled to generate a flat method statistic of that node. This means, that all called methods in the subtree of this node are accumulated and presented in a table. We illustrate this by choosing Show Called Methods (Flat) from the context menu of `FileInputStream.<init>(File)` in the called method statistic:

The flat view provides a fast overview of all called methods somewhere in the subtree of the root method. Of course, deriving new statistics based on others, always operate on the currently selected rows.

5.1.4.4.2. Calling Method Statistics

In general, the self time column in the Methods (Flat) View shows the methods, where most CPU time is consumed within their own code. But it can't tell the user, in which context or better say control paths the expensive calls are performed. Just imagine a method of the `String` class with a very high self time and hundreds of different control paths in the program, that call this method. As this `String` method can't be improved (it's part of the JDK), we could only try to reduce the number of calls. The problem is, that this is very hard to do, because the statistic doesn't tell us anything about the context. The Methods (Hierarchical) Statistic helps to overcome this problem.

We try to examine, how the self and total time of `String.charAt(int)` in our example is distributed over the different possible control paths that call this `String` method. In the basic Methods (Flat) View, select `String.charAt(int)`'s row and click Show Calling Methods in the context menu.

We get a hierarchical tree, which is orientated down the stack as the green arrows indicate. This Calling Method Tree can be interpreted as follows:

- `String.charAt(int)` has total time of 67.9ms
• There is only a single caller of the method `charAt` in the profiled code: `Win32FileSystem.normalize(String)`. So the overall total time of `charAt` is caused on the control path `normalize` calling `charAt`.

• `normalize` itself has two parent callers: `File.<init>` and `Win32FileSystem.getUserPath`. 34ms of `normalize`'s total time is spent on `File.<init>` calling `normalize` and the other 34ms of `normalize`'s total time are spent on `Win32FileSystem.getUserPath` calling `normalize`.

• The control path `ChecksumTest$1.run` calling `File.getAbsolutePath` calling `Win32FileSystem.resolve` calling `Win32FileSystem.getUserPath` calling `Win32FileSystem.normalize` calling `String.charAt` causes 34ms of `charAt`'s total time.

• The control path `ChecksumServer$1.run` calling `File.<init>` calling `Win32FileSystem.normalize` calling `String.charAt` causes the other 34ms of `charAt`'s total time.

5.1.4.5. Threads Statistic

So far the method calls were not separated by the thread in which they were invoked. In some situations it is really helpful to get an overview about the activities of threads. The `Threads` statistic offers a way for the user to examine the execution of each thread individually.

To open the `Threads` statistic, simply click on the corresponding icon in the snapshot overview or by choosing either `Show Thread Statistic By Id` or `Show Thread Statistic By Name` in the snapshots context menu. The latter statistic accumulates thread objects which have the same name. In the checksum example, all thread names are unique and therefore both statistics show the same table.

![Threads Statistic Table](image)

We get 18 different thread objects listed in the statistic together with its execution times:
• ReferenceHandler, Finalizer, and main thread are the minimum set of threads each SAP Java VM needs

• pool-1-thread-1 is clearly the pool thread started in ChecksumServer.handleTask which has the most CPU consumption of 31s

• pool-2-thread-* are the pool threads in the test driver that poses the requests

• But what kind of thread is Thread-0?

Normally, the Threads statistic is only the starting point for examination. To figure out, which methods Thread-0 calls, open the called method statistic for that thread by choosing Show Called Methods from the context menu of the selected row.

Looking at the called methods tree it is clear that Thread-0 is the server thread that takes the request from the queue and takes pool-1-thread-1 to handle the requests.

The context menu for one or multiple selected threads offers the same statistics as the Methods (Flat) Statistic or the Methods (Hierarchical) Statistic.

It’s crucial to understand, that these statistics are based on the set of method calls that were executed in this thread(s).

The Profiling Lifecycle gave us a first hint that the server application does not really fully utilizes the test machine (which is a dual core here). The curve for the overall CPU consumption is about 50% in the average. On the other hand the Methods (Hierarchical) Statistic clearly shows us, that only the single pool thread named pool-1-thread-1 does the complete work sequentially. The idea now is to handle requests queued in the server concurrently by several worker threads. As the server code already parameterizes the number of maximum worker threads in the server it is easy to adapt the ChecksumServer class’ field initialization accordingly

```
private final static int MAX_PARALLEL_TASKS = Runtime.getRuntime().availableProcessors();
ExecutorService taskWorker = Executors.newFixedThreadPool(MAX_PARALLEL_TASKS);
```

A new profiling run with the same configuration settings now has the throughput of 15.64 requests/s compared to 9.52 requests/s before. The Profiling Lifecycle shows an utilization above 90% in the average:
It's important to mention, that the profiler's sampling causes some overhead. Without sampling, the 2 CPU cores would be fully utilized by the executing Java code.

A short look into the Threads statistic ensures that there are now two pool threads performing the checksum processing in the server:

Note that each of the worker threads is loaded equally (20.6s).

### 5.1.4.6. Sleeping Times

So far method statistics we opened all had two columns in common: the Self Time and Total Time column. A single sample sent by the profiler backend contributes to the total times of the called methods on the corresponding stack and to the self time of the method on top of the stack. This contribution does not consider the state of the profiled Java thread or method being executed. E.g. it is not possible to separate the time a thread is sleeping in a method from its running time.

Therefore each sample holds some additional information about this states in the specific moment the sample is taken. That helps the profiler frontend to offer fine-grained classifications:

- Thread state: running or sleeping
- Method state: interpreted, compiled, native, inlined (see Section 5.1.4.7, “Visualizing Method States”)

Based on these states, the profiler is able to show much more detailed information about method calls in separate columns. The set of visible columns can be configured by the user:

We now want to check our example for sleeping times. We ought to make sure, that no CPU time is wasted due to synchronization problems (i.e. waiting threads). Before we add the columns with sleeping times, we have to profile the checksum server again with parameter Ignore Sleeping Threads turned off. By default, this flag
is checked, since the profiler creates less overhead in this case. In particular, this is important when profiling applications with lots (maybe hundreds) of threads.

We resample our project with the correct parameters and open the Threads statistic view. Now we add the Time (Running) and Time (Sleeping) by choosing the entries in the column configuration toolbar button named Add and remove columns.

For each thread we now have exact values about its running and sleeping times. As expected, pool-1-thread-1 and pool-1-thread-2 have only a small fraction of sleeping time, which is not surprising cause they are handling the tasks. All other threads are blocked most of the execution time, only the Reference Handler and the main thread posing the requests were active for some seconds.

We now try to find out, why both worker threads still have a sleeping time of about 1.5s. Best qualified statistic for such a question is the Methods (Hierarchical) Statistic of this thread, as we can recursively go down the call graph searching for the methods that were found in sleeping state.

Open the called method statistic for the thread pool-1-thread-1 by choosing Show Called Methods from its context menu in the view. Then change the column settings to make the columns for running and sleeping time visible. Beginning from the root node Thread.run(), follow the path with the attached sleeping times.

We find out, that the complete sleeping time of 1.52s is taken within FileChannelImpl.map(), whereas 1.42s fall into the called method Thread.sleep(). As we can't directly influence the implementation of FileChannelImpl, we know that the server code in our sample project is not responsible for the time of being inactive.
5.1.4.7. Visualizing Method States

Another factor that influences the performance of Java applications is the percentage of compiled code. Java methods are interpreted by the VM before they get compiled by the JIT compiler. Compiled Java code is executed much faster than interpreted code. Moreover short methods might be inlined into the calling method to save a method call and to allow better optimization. Last but not least, methods might execute native code.

As mentioned before, the method statistics also offer self- and total columns for each type of method execution. To illustrate this concept, switch back to Performance Hotspot Snapshot overview (e.g. via the view navigator on the top) and open the Methods (Flat) View. We want to examine how all the methods are executed (i.e. compiled vs. interpreted etc.) and therefore don’t need total times.

Now activate the following columns:

- Running Time (sleeping time is not relevant in this context)
- (Native)Interpreted Time, (Native) Compiled Time, (Native) Inlined Time
- Self Time (we are only interested in the methods)

and sort the table by descending Self Time column.

We get a table like this:

For example FileInputStream.open(String) has an overall time of 40.7s, where 39.9s are executed in compiled native code (i.e. the native stub was already compiled) and only about 0.8s fall into interpreted native. ChecksumTest.main obviously was only running interpreted (for 317ms), which is clear since it was called only once.

Additionally, the contribution of the different method states can be made visible in special columns, representing the values in a single bar diagram. Just remove the currently selected columns and choose
Performance Hotspot Analysis

- Running Time (just as before)

- Detailed Distribution

- Self Time (just as before)

What we get is the very same information as before, but in a graphical manner:

<table>
<thead>
<tr>
<th>Method</th>
<th>Self Time (Running)</th>
<th>Detailed Self Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>40.7 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>6.08 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>5.93 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>5.74 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>1.95 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>1.61 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>1.34 s</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>393 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>355 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>317 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>233 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>122 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>114 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>101 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>88.8 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>76.1 ms</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>76.1 ms</td>
<td></td>
</tr>
</tbody>
</table>

When hovering the mouse over the diagram cells, a tooltip with the displayed values appears.

5.1.4.8. Export Statistics

All presented types of statistics can be exported to the following file formats:

1. HTML

2. CSV

You may either export a complete statistic or just the subset of currently selected rows. For example to perform a HTML export of the selected rows in a statistic, call Export Selection To -> HTML from the context menu:
In the *Export* dialog type in a title, setup the export parameters (only for HTML export) and choose the location for the new file:

If export parameter *With tool tips* is activated, the generated HTML page offers tool tips. Similarly *Show selection* will regard the current selection in the statistic by highlighting the corresponding rows in the HTML output.

The result may looks like this:

**Method Statistic**

<table>
<thead>
<tr>
<th>Method</th>
<th>Self Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChecksumTest51.run()</td>
<td>16m 55.935s</td>
<td>1h 20m 49.833s</td>
</tr>
<tr>
<td>com.example.checksum.ChecksumServer.concat(String)</td>
<td>6m</td>
<td>15m 01.109s</td>
</tr>
<tr>
<td>sun.misc.Unsafe.Println(100, long)</td>
<td>0 ns</td>
<td>15m 01.077s</td>
</tr>
<tr>
<td>ChecksumTest.main(String)</td>
<td>1m 24.581s</td>
<td>1m 24.581s</td>
</tr>
<tr>
<td>java.lang.Thread.sleep(100)</td>
<td>0 ns</td>
<td>10.7 ms</td>
</tr>
<tr>
<td>java.lang.Thread.sleep(200)</td>
<td>0 ns</td>
<td>10.7 ms</td>
</tr>
<tr>
<td>java.lang.Thread.sleep(300)</td>
<td>0 ns</td>
<td>10.7 ms</td>
</tr>
<tr>
<td>java.util.concurrent.LinkedBlockingQueue.take()</td>
<td>385 ms</td>
<td>57.4 s</td>
</tr>
<tr>
<td>JavaUtil.concurrent.LinkedBlockingQueue.take()</td>
<td>0 ns</td>
<td>57.4 s</td>
</tr>
<tr>
<td>67 methods</td>
<td>1m 27.203s</td>
<td>4m 53.426s</td>
</tr>
</tbody>
</table>

Similarly, a CSV export can be triggered. If the complete statistic should be exported, just choose *Export All To* from the statistic's context menu.

Statistic exports can be embedded into other applications and views or can be used to perform additional analysis (e.g. import to Microsoft Excel).

### 5.1.5. Advanced Performance Hotspot Analysis

We now want to demonstrate how the SAP JVM Profiler can be applied to a complex scenario: we measure the runtime behavior of a user request in a specific application on the SAP NetWeaver Application Server Java. More detailed, we try to find out, where most CPU time is spent in the *System Information* application (*sap.com/tc-lm-itsam-ui-systeminfo-wd*) when selecting components in the the *Component Info* tab.

From an external point of view, we may have multiple users, sessions, applications and requests on the server instance simultaneously. Internally, many different Java components interact to perform parallel requests. The
execution is spread over lots of threads with very large call stacks. It's clear that we need more sophisticated analysis procedures to get along with that complex setting and to extract the information we need.

5.1.5.1. Online Profiling

First of all we prepare the application on the server for the measurement. Therefore we switch to the main access page of the SAP NetWeaver Application Server Java (URL here: http://localhost:50000/index.html. Of course port number may vary) and start the System Information application by simply clicking the corresponding button:

By default, the System Info tab is active, showing the database instance, the message server and the application server:

Before we activate the Components Info tab, we have to attach the profiler in order to record the system behavior when performing this action.

As the Java server instance is already running, we have to attach the profiler online. This means we have to specify a Remote Profiling configuration in the Profile Configurations dialog, where some parameters like the Java server instance’s host and (debug-)port have to be setup. We possibly need to open the debug port on the Java server instance before we are able to attach the profiler (it’s recommended to use SAP Management Console for that purpose). How to perform these steps can be adopted directly from Section 4.1.3.1, “Getting the Data”.

Instead of the Allocation Analysis in the last chapter, we now choose the Performance Hotspot Analysis in the Start Remote Profiling dialog.
Since we are logged in the application server as Administrator we are only interested in performance data which is caused by requests by this user. Therefore we filter the users with 'Administrator' on the analysis' parameter page. In the Main tab we additionally activate Ignore Sleeping Threads as we put the focus on CPU consumption. Click on Finish to start the analysis.

The Profiling Lifecycle (see Section 8.2, “Profiling Lifecycle and Analysis Overview”) gives a first overview about the system's state and behavior:

As there are no requests to be processed by the Java server instance, the memory usage is quite stable (425 MB) and no GCs happen. CPU usage varies between 1% and 7% which comes from the overhead the profiler backend creates in the VM during sampling.
Now we switch to the System Info application and open the Components Info tab and click on several components.

Back to the Lifecycle view, we recognize increasing CPU activity for some seconds (related to the period of time the clicks on the Web page were performed). We select this period of time and create a snapshot named ComponentInfo:

![Profiling Lifecycle](image)

After we have taken the relevant snapshot, we may stop the analysis by pressing the Stop Analysis button in the Analysis Overview. We will evaluate the profiling data in next Section 5.1.5.2, “Advanced Performance Hotspot Analysis”.

5.1.5.2. Advanced Performance Hotspot Analysis

To inspect the snapshot ComponentInfo, we double-click it in the Analysis Overview section. The snapshot’s overview page opens being the starting point for further analysis.

5.1.5.2.1. User-, Session-, Request- and Application-Statistics

The Performance Hotspot Summary view tells us that 1500 samples were taken within the collection period of 18s, which seems to be sufficient for getting reliable results.

First of all, we try to get an overview about involved users, sessions, requests and applications. As we performed the measurement within a single session and users were filtered by 'Administrator', we expect only a single session in the Session Statistic and a single user in the User Statistic:
Note that there is a small fraction of execution time (169ms) falling into *<no user>* respectively *<no session>*. This user-independent effort is due to basic infrastructure services in the Java server instance, which are provided by system threads.

All other time-consuming method calls clearly relate to the user *Administrator* within the single session number -472652067.

What about the applications? Are there several applications contributing CPU time? Let's have a look into the *Application Statistic*:

Two applications seem to be involved: *sap.com/tc~lm~itsam~ui~mainframe~wd* and *sap.com/tc~wd~dispwda* (similarly *<no application>* sums up basic services). Obviously most time is consumed in the first application. Open the *Request Statistic* for this application by invoking *Show Request Statistic* in the context menu. This provides us an overview about requests and their contribution of CPU time to the overall application time.
Each click on a row in the component info table is mapped to a single request which then is processed by a HTTP Worker thread. This can be seen by opening the thread statistic for the application via the context menu’s entry Show Threads by Name. We see 5 different worker threads. Looking into the thread statistic of a single request (opened similarly) reveals that each request is handled by single thread from the HTTP worker pool.

The statistics described in this section help the user concentrate on those profiling information he is interested in, e.g. the method calls caused by a single request or the overall CPU consumption of an application. Opening statistics out of other statistics reduces the call graph to context specific method invocations, i.e. user-, session-, application-, request- or thread-dependent.

Although nested statistics act as a powerful feature to reduce the profiling data, the method calls that belong to a single request may be still unmanageable due to huge call stacks. Section 5.1.5.2.2, “Advanced Method Statistics” shows variations of method statistics that help to identify performance hotspots on a very coarse-grained level of analysis.

5.1.5.2.2. Advanced Method Statistics

We continue the performance analysis for the single request with id 561 (id may vary). The objective is to find the most time-consuming Java calls in WebDynpro. In the Request Statistic, we select request 561 and choose Show Components (Flat)... The Filters dialog is opened, where we have to select method or class filters that should be applied in the Component Statistic.

For convenience, there are some typical filters predefined: Cache, Configuration Management, OpenSQL, etc. Of course, users are allowed to specify arbitrary sets of methods and classes that should belong to a filter. To learn more about filters, see Section 10.3, “Class Filters” and Section 10.5, “Method Filters”.

Applying a filter to a given statistic means to omit all methods, that do not match the specified filter (or their class respectively). In our example, we want to restrict to methods which relate to OpenSQL, consequently we select the predefined filter named ‘OpenSQL’ and open the corresponding Filter Statistic by confirming with OK.
Performance Hotspot Analysis

To inspect the called methods, choose Show Methods (Hierarchical) For Self from the context menu of the OpenSQL entry to get the hierarchical Method Statistic:

All methods shown in the statistic are directly called by OpenSQL framework. In order to get all method called by this methods to find out the most time consuming, select all and choose Show Methods (Flat) from the context menu of the selection.

All methods shown in the statistic are either directly or indirectly called by methods located in the OpenSQL framework. Sorting the methods by decreasing self times identifies the most time-consuming method: `java.net.SocketInputStream.socketRead0(java.io.FileDescriptor,byte[],int,int,int)` taking about 26.8s.

The first observation is, that this method belongs to the standard JDK and therefore can’t be optimized. When thinking about it, we are rather interested in the execution time of OpenSQL methods where the total time for called JDK methods is included. Adding the execution times of a method’s callees which match a specific filter to the method’s own execution time is named ‘inlining’. We can make usage of this feature to assign execution times of JDK methods to the corresponding callers out of WebDynpro.

Now click on the button to switch to the inlined version of the current statistic. In the Filters dialog we have to specify the class/method filters that should be applied for the inlining operation:
We can directly choose the JDK filter being already predefined. Just confirm with OK to open the inlined statistic:

Clearly, `com.sap.dbtech.rte.comm.BasicSocketComm.receiveData()` is the top consumer of CPU power. To find the reason for that it's recommended to have a look into the Called Method Statistic of that single method. Click on Show Methods (Hierarchical) from the method's context menu to get the following statistic:
5.2. Combined Performance Hotspot and Method Parameter Analysis

This section provides you an overview of how to perform a performance hotspot analysis in combination with the functionality offered by the method parameter analysis. As described in Section 3.4, “Method Parameter Analysis Using SAP JVM Profiler”, the method parameter analysis gives the user detailed information about individual method invocations. In particular, it provides the actual values of method parameter, the number of invocations and the runtimes of individual method calls. The methods and the corresponding parameters you want to analyze in detail have to be specified upfront - that means when enabling the actual analysis. However note that the analysis is completely dedicated to the analysis of individual method calls and their corresponding parameters. That means, for instance, it is not possible to look in the methods being called by the analyzed methods. So when you see that a certain method consumes a huge amount of CPU time, you cannot determine whether the reason behind the high resource consumption lies in the method implementation itself or in the implementations of the called functionality. The performance hotspot analysis gives you aggregated runtime information over all invocations of a certain method, but it allows you to look at the context in which the particular method is called. You can analyze both the methods calling a certain method and the methods being called by that method. So in some scenarios it would be very desirable to combine the functionality provided by the performance hotspot and method parameter analysis.

As an underlying use case, we want to analyze the lookup operations of the JNDI (Java Naming and Directory Interface) service within an SAP NetWeaver Application Server Java installation. JNDI is one of the core services of a J2EE application server. It is a Java API for a directory service that allows Java components to discover and look up data and objects via a name. In general, the application server itself or privileged applications are responsible for binding objects to a certain name.

5.2.1. Connect the SAP JVM Profiler to the SAP NetWeaver Application Server Java

The first step in our analysis is to connect to the SAP NetWeaver Application Server Java system. Therefore, we choose the corresponding entry Profile SAP AS Java on the Profiling Intro page as shown in the picture below.
To perform the configuration of the profiling run in a convenient way, we should start the \texttt{jvmmond} daemon process on the machine the SAP NetWeaver Application Server Java is running on. The \texttt{jvmmond} executable is part of the SAP Java VM delivery and can be found in the \texttt{bin} directory of the SAP Java VM installation. It provides remote access to all SAP Java VMs running on a remote host. To start \texttt{jvmmond}, you simply have to execute the corresponding executable. On start-up, the \texttt{jvmmond} process opens a server socket waiting for client requests. The default port number is \texttt{1099}. If for instance this port is already in use, you can also configure \texttt{jvmmond} to use a different port via the \texttt{-port} option. For details how to start and configure the \texttt{jvmmond} process please refer to Section 3.1, “Remote Profiling”. So, in the opened Start Profiling Wizard we choose the \texttt{jvmmond} option.
Figure 5.2. Start Profiling Wizard - Connect to an SAP Java VM via jvmmond

On the next wizard page, we enter the name of the host the SAP NetWeaver Application Server Java is running on and choose the process which corresponds to our server instance we would like to profile. Then we finish the Start Profiling Wizard dialog and get connected to the SAP NetWeaver Application Server Java system.

Figure 5.3. Start Profiling Wizard - Enter Host Name and Choose Server Process to Profile
5.2.2. Dynamically Enable and Disable the Analysis

When getting connected, a new entry within the Profile View is created which displays the profiled SAP Java VM. On the right hand side of the profiling perspective, you get an overview page for the whole profiling run. This page is named according to your created configuration. Within this page there is a section called Analysis Overview which contains an overview of the started profiling traces and created snapshots. The toolbar of this section also contains buttons for starting and stopping traces. So, in order to start the combined performance hotspot and method parameter analysis, we press the corresponding button as shown in the picture below.

![Figure 5.4. Analysis Overview - Start an Analysis](image)

Within the opened Analysis Start Dialog we select the Performance Hotspot Analysis. Next we have to specify the methods and the corresponding parameters for which we want to get detailed information like the method invocation numbers and the actual method parameter values. This can be done through the Method Parameters option page on the right side of the configuration dialog. By pressing the button for selecting method parameters we get to the Method Parameters Dialog.
Within the Method Parameters Dialog we are able to specify the method parameters we are interested in. For each method parameter a corresponding definition has to be created. The definition contains a description name for the parameter specification, the fully qualified method name and the index of the parameter. The index of the parameter refers to the position within the method parameter list. \( i \) denotes the first parameter, \( 2 \) the second one, etc. \( 0 \) refers to the \textit{this} pointer of the object calling the specified method.

If the type of the parameter corresponds to a primitive type (\texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{float} or \texttt{double}) or is of type \texttt{java.lang.String}, \texttt{java.lang.StringBuffer} or \texttt{java.lang.StringBuilder}, the actual value is reported as a parameter value per default. In case of \texttt{StringBuffer} and \texttt{StringBuilder} that means the created \texttt{String} value. However, in order to provide a user-readable parameter value for arbitrary object types a so-called modifier has to be specified. A modifier defines the UI representation of a value which is presented to the user as the parameter value of a certain method call.

A modifier can be one of the following entities:

- an instance method without parameters which is part of the parameter's class definition
- a static method which take the parameter value as an argument
- class - returning the class of the parameter value
- length - returning the length of the parameter value (can only be applied to arrays)
- \([i]\) - returning the array element at position \( i \) of the parameter value (can only be applied to arrays)
- \texttt{com.sap.MyClass fieldName} - returning the value of a field which is part of the parameter's class definition
- \texttt{(com.sap.MyClass)} - casts the parameter value to the specified type

With respect to our use case we want to analyze the lookup operations of the JNDI functionality. There are two basic methods for looking up objects: \texttt{javax.naming.InitialContext.lookup(String)} and

**Figure 5.5. Analysis Start Dialog - Enable the Performance Hotspot Trace**

Within the Method Parameters Dialog we are able to specify the method parameters we are interested in. For each method parameter a corresponding definition has to be created. The definition contains a description name for the parameter specification, the fully qualified method name and the index of the parameter. The index of the parameter refers to the position within the method parameter list. \( i \) denotes the first parameter, \( 2 \) the second one, etc. \( 0 \) refers to the \textit{this} pointer of the object calling the specified method.

If the type of the parameter corresponds to a primitive type (\texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{float} or \texttt{double}) or is of type \texttt{java.lang.String}, \texttt{java.lang.StringBuffer} or \texttt{java.lang.StringBuilder}, the actual value is reported as a parameter value per default. In case of \texttt{StringBuffer} and \texttt{StringBuilder} that means the created \texttt{String} value. However, in order to provide a user-readable parameter value for arbitrary object types a so-called modifier has to be specified. A modifier defines the UI representation of a value which is presented to the user as the parameter value of a certain method call.

A modifier can be one of the following entities:

- an instance method without parameters which is part of the parameter's class definition
- a static method which take the parameter value as an argument
- class - returning the class of the parameter value
- length - returning the length of the parameter value (can only be applied to arrays)
- \([i]\) - returning the array element at position \( i \) of the parameter value (can only be applied to arrays)
- \texttt{com.sap.MyClass fieldName} - returning the value of a field which is part of the parameter's class definition
- \texttt{(com.sap.MyClass)} - casts the parameter value to the specified type

With respect to our use case we want to analyze the lookup operations of the JNDI functionality. There are two basic methods for looking up objects: \texttt{javax.naming.InitialContext.lookup(String)} and
javax.naming.InitialContext.lookup(Name). So, we create two method parameter specifications accordingly. Aforementioned, for the String parameter it is not necessary to specify a modifier. However, the Name parameter needs a modifier definition to get enabled. We are interested into the keys of the looked up objects. So we simply want to see the String representation of the parameter value and specify the toString()java.lang.String as a modifier. The result of the definition can be seen in the following figure.

Figure 5.6. Method Parameters Dialog - Specifications for the JNDI Lookup Methods

Now we select the two parameter definitions and confirm the Method Parameters Dialog and the Analysis Start Dialog. Then the analysis gets enabled and we can run our application scenario we want to profile. When we are finished with the application scenario, we can disable the trace via the Stop Analysis button in the Analysis Overview section.

Figure 5.7. Analysis Start Dialog - Disable the Performance Hotspot Trace

After disabling the trace, a snapshot is created. A snapshot refers to a time frame where profiling data was collected. In our use case, the snapshot spans the whole lifecycle of our started profiling trace. You can see the added snapshot in the Profile View and in the Analysis Overview section of our configuration overview page.
Figure 5.8. Analysis Start Dialog - Newly Created Snapshot

A snapshot overview page is also created which contains the entry points for starting a detailed analysis of the profiling data.

Figure 5.9. Analysis Start Dialog - Snapshot Overview

5.2.3. Analyze the Method Invocations

In addition to the general statistic views of the performance hotspot analysis (see Section 5.1, “Performance Hotspot Analysis Walkthrough”), a new entry point for starting your detailed analysis is available called Method Parameter Statistic. You can see the corresponding entry in the Figure 5.9, “Analysis Start Dialog - Snapshot Overview”. The Method Parameter Statistic provides detailed runtime information for each individually defined method parameter specification. That means in our use case, you get runtime information about the two defined JNDI lookup calls. By selecting the Method Parameters entry in the snapshot overview, you open the corresponding statistic view.
Figure 5.10. Method Parameter Statistic - JNDI Lookup Calls

In the statistic only the entry referring to the `InitialContext.lookup(String)` is available. That means in our application scenario the other overloaded lookup method `InitialContext.lookup(Name)` has not been called. However, the `InitialContext.lookup(String)` method is called 1246 times. The time we spent directly within the `lookup(String)` method is 12.6ms for all invocations, whereas the total running time is 2.68 s.

The next step in our analysis is to look at the individual parameter values. Therefore, we select the corresponding entry in the Method Parameter Statistic and open the context menu. There, we find an entry for showing the parameter values. We select this option and get the following Parameter Values Statistic.

Figure 5.11. Parameter Values Statistic - Keys to the Looked-up JNDI Objects

The Parameter Values Statistic shows for each applied parameter value the corresponding number of invocations and the aggregated runtime information for all invocations of a specific value. The displayed parameter value refers to the modifier definitions in the method parameter specifications. Thus, if you had defined for example a `class` modifier, then you would have seen the class definitions of the parameters as a parameter value. In our case, we have not defined a modifier for the `InitialContext.lookup(String)` specification. So per default, we get the actual value of the `String` parameter. As shown in Figure 5.11, “Parameter Values Statistic - Keys to the Looked-up JNDI Objects”, you see for instance that the object to the key `com.sap.tc.gl.srv` is requested 142 times. For all method invocations, which relates to this parameter value, 37.7 ms total running time was consumed.

As discussed the Parameter Values Statistic shows only accumulated values over all invocations for a certain parameter value. For a chosen parameter value you can get an overview of the individual method invocations.
To open this statistic you have to select a specific parameter value within the Parameter Values Statistic and open the context menu. There you find an entry for opening the so-called Method Invocation Statistic. In this example, we select the `com.sap.tc.pdc.gl.srv` parameter value and open the Method Invocation Statistic. When looking at the results you can see some variance in the total running time.

![Figure 5.12. Method Invocation Statistic - JNDI Lookup Calls](Image)

### 5.3. Performance Hotspot Analysis - Technical Documentation

Performance Hotspot Analysis is, as opposed to static code analysis, the investigation of a program's behavior with respect to execution time using information gathered while the program runs. The objective is to determine parts of a program with a huge optimization potential for execution speed. It shows where the program spent most time and which control paths are extraordinary expensive. This information can show which pieces of the program are slower than expected and might be candidates for rewriting to make the program run faster.

Performance bottlenecks are code segments that contribute significantly to the overall program execution time, and are responsible for slowing down or even halting a program. Identifying and improving these code segments has tremendous impact on a program's performance.

According to the 80/20 rule, 80% of the time is spent in executing 20% of the code. Optimizing 80% of the code will have little impact on overall performance. The performance bottlenecks - the 20% of the code where 80% of the execution time is spent - must be pinpointed and improved.

### 5.3.1. Methods of Data Collection

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, system hooks and performance counters. We can categorize profilers by the chosen technique of data collection.

- **Event based profilers**

  Event based profilers use specific events such as method call events to trace the exact execution path as the program runs. For Java, the JVM Tool Interface (JVMTI) generates events like method calls, class loading or class unloading.

  To exactly measure the time passed or record events happening during the execution of code regions (e.g. a method), additional measurement code needs to be inserted before and after the given region. This code reads the time or a global event count and calculates differences. Thus, the original code has to be adapted before execution. This is called instrumentation. Instrumentation generally can be done by the programmer itself, the compiler or by the runtime system.
With Java SE 5, the VM's runtime system provides services that allow the instrumentation of programs running on the VM. Some profilers instrument the target program with additional instructions to collect the required information. Instrumentation of the program will cause changes in the performance of the program, leading to inaccurate results. Instrumentation can potentially be very specific but also slows down the target program the more specific information is collected.

• Statistical profilers

Some profilers use sampling. A sampling profiler probes the target program's execution state (such as program counters or callstacks) at regular intervals. Sampling profiles are typically less accurate and specific, but allow the target program to run at nearly full speed.

If the sampling rate is too high, the interruptions to sample the execution state will overwhelm the executed program. On the other hand the profiler must collect enough samples to gather a distribution of samples representing the execution time as accurately as possible.

5.3.2. Performance Hotspot Analysis with the SAP JVM Profiler

The Performance Hotspot Analysis of the SAP JVM Profiler uses a statistical analysis. Imagine a running program being halted at a specific point of time. The threads' callstacks then represent the program's current state in execution and also the control path to reach that point. The method on top of the callstack is the currently executed method and the methods down the stack are its predecessors.

As the program execution is interrupted at regular intervals, each sample represents a fixed portion of execution time, namely the sample distance in time. Thus the total number of samples with a method on top of the callstack directly corresponds to the execution time within this method (self time). Similarly, the total number of samples with this method contained in the callstack refers to the time the VM spends in that method or in a called method (total time).

Regularly sampling can at most approximate what actually was executed. But in general, this statistical approximation is good enough to reflect the real execution times and even more reveal the performance bottlenecks. Provided an arbitrarily long sampling period, the quality of the offered approximation depends mainly on two factors:

1. The sampling rate should be as constant as possible, i.e. the distance in time between two adjacent samples may not vary a lot in the average. This requirement comes directly from the fact that each sample is assumed to contribute the same proportion of the execution time. Briefly, all Java profilers have a problem here, because it is not possible to stop a VM at arbitrary points of time to obtain valid callstacks (see Section 5.3.3.2, “Safepoints”). The SAP Java VM tries to overcome this problem by the introduction of special Safepoints.

2. There must not be any correlation between the sampler (i.e. the points of time to take the so-called samples) and the Java code execution. Unfortunately, this requirement can't be met completely either. The SAP Java VM was optimized to minimize these correlations.

5.3.3. Technical Details

5.3.3.1. Runtime Behavior

Taking samples should not remarkably interfere with the running program. Consequently, the profiling code must be as cheap as possible and therefore cannot do the analysis of the profiling data (e.g. assemble the callstacks, propagate times, building up call graphs, etc.) during program execution. So, the SAP JVM Profiler
backend (the part of the profiler which resides in the VM) sends the profiling data during program execution directly to the SAP JVM Profiler frontend (the Eclipse plugin) where it is stored in snapshots. The stored samples then can be processed for an offline analysis afterwards.

### 5.3.3.2. Safepoints

The Performance Hotspot Analysis samples the callstacks in a regular manner. Sampling the callstacks requires the Java threads not changing their state during this taking the sample. The SAP Java VM is a native threaded VM; therefore the callstack sampling needs to be synchronized with the threads running. The SAP Java VM utilizes the VMs Safepoint mechanism to sample the callstacks, as it's guaranteed that the callstacks of a thread is unchanged on a Safepoint and can be examined by another thread.

A Safepoint is a point during program execution at which all GC roots (i.e. root memory pointers in the VM) are known, all heap object content is consistent and the callstacks of the threads can be examined. From a global point of view, all threads must block at a Safepoint before a Safepoint operation can run. From a local point of view, a Safepoint is a distinguished point in a block of code where the executing thread may block for a special operation. There are strong invariants which hold true at every Safepoint, which may be disregarded at non-Safepoints.

In simple terms, when the VM is at a Safepoint, all threads inside the VM have been blocked and any threads executing in native code are prevented from returning to the VM while the Safepoint is in progress. This means that an operation can be executed knowing that no thread can be in the middle of modifying the Java heap, and all threads are in a state such that their callstacks are fixed and thus can be examined.

The most familiar operation which is executed on a Safepoint is the garbage collection, or more specifically the "stop-the-world" phase of garbage collection. Many other Safepoint-based operations exist, for example biased locking revocation, thread stack dumps, thread suspension or stopping and numerous inspection or modification operations requested through JVMTI.

Safepoints are initiated using a cooperative, polling-based mechanism. In simple terms, every thread asks from time to time: "should I block for a Safepoint?".

Asking this question efficiently is not so simple. However, it's guaranteed that the question is asked in a regular manner, for instance when returning from a method or on a backward branch in a loop. As a method consists of a limited number of byte codes, it's guaranteed that the end of the method is reached after a certain amount of time when no backward branch has been taken. As it's checked for Safepoints on backward branches as well, a Safepoint check is guaranteed to be executed in a regular manner.

The listing below shows an annotated source snippet where Safepoints can occur. Safepoints are only issued at certain points, like method exits, backward branches, monitor enters, monitor exits or native calls. Whenever a callstack is triggered, it will be triggered on one of the points where a Safepoint can occur. Other places will never show up in the callstack. Technically the SAP Java VM's JIT compiler performs various optimization in between Safepoints as these optimizations are not observable. However, on Safepoints, the JIT compiler must be able to reconstruct the actual state.

```java
public static void main(String []args) {
    byte count = 0;
    List<Object> l = new ArrayList<Object>();
    Object o = new Object();
    while (true) {
        if (count == 100) {
            l.removeAll(l); /* Safepoint on method exit */
            count = 0;
        } else {
            l.add(o); /* Safepoint on method exit */
            ++count;
        }
        /* Safepoint due to backward branch in loop */
    }
```

Usually, when trying to reach a Safepoint on a method exit, the method's frame is popped first and only now the Java-thread blocks as a reaction to the Safepoint request. Thus, methods with only Safepoint at method exit will never show up in the callstack.

The following method statistic shows this behavior. The method `java.util.AbstractCollection.removeAll` seems to be responsible for most of the runtime in this example (self time of 19.604s), but in reality its callees (especially `AbstractList$Itr.remove`) consumed most CPU time.

<table>
<thead>
<tr>
<th>Method</th>
<th>Self Time (Running)</th>
<th>Total Time (Running)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.util.AbstractCollection.remove</code></td>
<td>1.007s</td>
<td>22.002s</td>
</tr>
<tr>
<td><code>java.util.ArrayList.remove</code></td>
<td>0.0015s</td>
<td>0.076s</td>
</tr>
<tr>
<td><code>java.util.AbstractList$S$trimremove</code></td>
<td>0.0012s</td>
<td>0.003s</td>
</tr>
</tbody>
</table>

By default, the SAP JVM Profiler issues more Safepoints when profiling is activated. This results in more accurate callstacks and helps to track down problems.

The next method statistic is a result of profiling with more Safepoints. Here, most of the time is spent in `java.util.ArrayList.remove` which is called by `java.util.AbstractCollection.removeAll`.

The more accurate callstacks are not for free and add a runtime overhead of about 5%. The runtime overhead is only paid when profiling is active (when profiling is switched off, the VM should run in normal speed). To achieve this the compiled code is decompiled when the profiler is attached to the VM. And the newly generated code contains more Safepoint checks.

5.3.4. VM Options

There are some VM-flags which control the behavior of the Performance Hotspot Analysis. As they are all startup options, they must be passed to the VM as VM common VM arguments. This can be done in the launch configuration dialog: ...

- `-XX:TimeBasedSamplingInterval=n`

  The interval in milliseconds how often a callstack should be sampled. The default value is 10 milliseconds. The minimum value is 5 milliseconds. Note that high sampling rates will have a higher impact on the overhead, a low sampling rate will lead to inaccurate results (unless the sampling time is extended).

- `-XX:+TimeBasedSamplingEnableSafepointSupport`

  Adds more Safepoint checks in the compiled code leading to more accurate callstacks. Furthermore a runtime overhead of around 5 percent is added. By default, the option is activated.

- `-XX:+TimeBasedSamplingEnableSmoothSafepointTransition`

  When `TimeBasedSamplingEnableSafepointSupport` is active, attaching the profiler to a running system requires all methods to be recompiled for inserting the additional Safepoints. This would abruptly
slow down the system for some time until the recompilation process has finished. If `TimeBasedSamplingEnableSmoothSafePointTransition` is set to `true` (and it is by default), the SAP Java VM performs this transition as fluently as possible, i.e. methods get recompiled only on demand. After detaching the profiler, the newly compiled methods can be exchanged with copies made before the recompilation. Therefore detaching won't touch the system again.

- **-XX:+TimeBasedSamplingSpinSafePoint**
  
  This flag makes the sampling thread taking its samples uncorrelated to the Java threads. The sampler randomly delays the point of time when a sample has to be taken (but 0.5ms at maximum). It's also enabled by default.

- **-XX:+TimeBasedSamplingFairSafePoints**
  
  As described above, Java threads halt on SafePoint request mechanism. If a thread is currently processing in native or VM code, it can't be stopped directly. Instead, such a thread is not allowed to transition its state back to Java when a SafePoint is currently requested. Therefore the thread normally blocks on the next state transition. This asymmetric way of stopping threads leads to overestimated threads in native or vm and underestimated threads in Java. When `TimeBasedSamplingFairSafePoints` is active, threads are only stopped on transitions in the phase of signaling the Safepoint, when the polling page is seen protected (now the Java threads have the same chance to stop).

### 5.3.5. Limitations

Due to the implementation of the Performance Hotspot Analysis, there are several (partly architectural) restrictions:

- Since the profiler uses information collected during the actual execution of the program, it can be used on programs that are too large or too complex to be analyzed by reading the source. However, the actual program execution will affect the information that shows up in the profiling data. If some feature of the program is not used while being profiled, no profile information will be generated for that feature.

- Callstacks can only be sampled on SafePoints. Therefore only certain places (like method exits, backward branches, monitor enters, monitor exits or native calls) will show up in the callstacks. As a consequence the execution time between the last and the current Safepoint is completely accumulated to the current Safepoint.

- As the compiled code is modified in order to contain more Safepoint checks, the compiled code is deoptimized when the profiler attaches to the VM. At this point in time, the VM runs interpreted and new code is created as the VM runs. When only a short scenario is measured, the warm up, i.e. the compiling of the methods, could influence the measurement. Therefore it's recommended to attach the profiler, run the scenario in order that the hot methods are compiled, start the Performance Hotspot Analysis and rerun the scenario - now with the hot methods compiled.

- The overhead of sampling and storing the data will dilate the runtime of the program. For the most part, we estimate that dilation to be less than 15%.

### 5.3.6. Misconceptions

At this time, there are certain misconceptions in the Performance Hotspot Analysis.

- **Timing is important**

  Knowing how much time is spent in methods is doubtless good for reporting improvements, but it provides only vague help in finding problems. The information that matters is the fraction of time that individual statements reside on the callstack.

- **Statistical precision matters**
Typical performance problems sit on the callstack between 5 and 95% of the time. The larger they are, the fewer samples are needed to find them. As in sport fishing, the object is to catch them first, and measure them later, if ever.

As an example, the iteration of performance problem removal tends to go something like this: Performance problem X1 could be taking 50% of the time and X2 could be taking 25% of the time. If X1 is removed, execution time is cut in half, at which point X2 takes 50% of the time. If on the first pass X2 is removed instead, the time is only reduced by 1/4, but then X1 is seen as taking 67% of the time, so it is even more obvious, and can be removed. Either way, removing both X1 and X2 reduces execution time by 75%, so the remaining performance problems are four times larger. This "magnification effect" allows the process to continue through X3, X4 and so on until all the easily-removed performance problems have been fixed.

• Statistical Sampling Error

The results of the SAP JVM Profiler are based on a sampling process, so they are subject to statistical inaccuracy. If a method runs only a small amount of time so that on the average the sampling process ought to catch that method in the act only once, there is a pretty good chance it will actually find that method zero times, or twice.

The rule of thumb is that the result is accurate if the sample time is considerably bigger than the sampling period.

The actual amount of error can be predicted. For n samples, the expected error is the square-root of n. For example, if the sampling period is 0.01 seconds and foo's runtime is 1 second, n is 100 samples (1 second/0.01 seconds), sqrt(n) is 10 samples, so the expected error in foo's runtime is 0.1 seconds (10*0.01 seconds), or ten percent of the observed value. Again, if the sampling period is 0.01 seconds and bar's runtime is 100 seconds, n is 10000 samples, sqrt(n) is 100 samples, so the expected error in bar's runtime is 1 second, or one percent of the observed value. It is likely to vary this much on the average from one profiling run to the next.

This does not mean that a small runtime figure is devoid of information. If the program's total runtime is large, a small runtime for one method does tell that that method used an insignificant fraction of the whole program's time. Usually this means it is not worth optimizing.

One way to get more accuracy is to give the program more (but similar) input data in order to make the overall execution time longer.

5.3.7. Summary

Optimizing every part of the code is not worth the effort. The efforts should focus on code segments where returns are higher, or rather highest. The SAP JVM Profiler helps identify these code segments (performance bottlenecks).

Once these performance bottlenecks have been identified, they can be removed. Another set of profiling information should then be collected. Comparing both sets help to quantify the improvements. When satisfied with the result, the next bottleneck can be taken into account.

The SAP JVM Profiler is best used in an iterative approach: profiling the program, eliminating one bottleneck, and then finding some other part of the program that begins to dominate execution time.
Chapter 6. Synchronization Analysis

6.1. Synchronization Analysis Walkthrough

This is a detailed walkthrough for the SAP JVM Profiler's Synchronization Analysis. It explains the basic concepts of this analysis and shows it in action using an example application. The example is a short, self-contained Java program. It is used to show basic and advanced features of the Synchronization Analysis.

You can proceed from here in one of the following ways:

- Start the walkthrough with Section 6.1.1, “Synchronization Analysis Overview”. Here you will get a short overview about the objectives and the main concepts of the Synchronization Analysis. Furthermore, it is discussed how the process of analyzing synchronization issues is supported by the SAP JVM Profiler step by step.

- Get to know the sample Java application Pi Calculator in Section 6.1.2, “The Pi Calculator Example”

- Go directly to Section 6.1.3, “Profiling the Example” to learn how to collect profiling data of a Java VM in action, which is the basis for statistical evaluations afterwards.

- Section 6.1.4, “Evaluating Profiling Data” summarizes the various types of statistics and views which help you to identify the synchronization issues very quickly.

6.1.1. Synchronization Analysis Overview

6.1.1.1. Objectives of Synchronization Analysis

One of the main problems of writing scalable multi-threaded software is the correct synchronization of the shared memory. Some developers take the dangerous route of premature optimization or ‘works for me’ synchronization. The main issue of this approach is that workstations used for development are usually not highly parallel systems so that synchronization problems may arise only when the application is run in production. Since synchronization issues are often hard to find and nearly impossible to debug, this may cause a very high correction effort. These kinds of problems affect the functional correctness of an application and need to be checked during acceptance tests. The SAP JVM Profiler's Synchronization Analysis cannot help finding these application specific issues.

Developers who have found the method described above to be seriously flawed usually tend to overuse synchronization, i.e. they use large critical section with only few commonly utilized synchronization primitives, e.g. locks. While this approach is normally quite successful in preserving functional correctness of an application, it may impact the scalability of a system. The SAP JVM Profiler's Synchronization Analysis can help identify and analyze these scalability issues by providing means to go from a comprehensible overview of the contention caused by synchronization down to the gory details of each and every contended synchronization usage.

The main objective of the Synchronization Analysis is to show where the application was blocked most of the time because of contention due to synchronization. Furthermore, it enables the user to detect the dependencies between the threads that were involved in contended synchronization. With the knowledge gained through the Synchronization Analysis developers may be able to improve the synchronization usage and reduce the contention thereby improving the scalability of their applications.

6.1.1.2. Blocked and Blocking in the Synchronization Analysis

The Synchronization Analysis focuses on the temporary blockages caused by contended synchronization. It shows all involved parties, i.e. it presents the ones blocked due to synchronization as well as the reasons of these blockages namely the ones blocking the former ones. Throughout the Synchronization Analysis the terms blocked and blocking are used to indicate in which role the entity at hand is currently displayed.
It is important to note that all statistics shown in the Synchronization Analysis exist in both flavors, i.e. for any statistic there is a blocked variant as well as a blocking one. The difference of these two flavors is the point of view they take. On the one hand, the blocked statistics show the situation from the perspective of a blocked entity, e.g. a blocked thread statistic shows all threads that were blocked during acquisition of a lock. On the other hand, the corresponding blocking statistic answers the question which entities, e.g. which threads, caused the former ones to be blocked. The type of the statistic is displayed in the statistic tab name as shown in the figure below.

Figure 6.1. Blocked and Blocking Statistic Examples

Even though the columns of both blocked and blocking statistics seem to be identical the semantic differs due to the opposing view. The blocked time of a blocked statistic depicts the actual time a shown entity was blocked due to contention with respect to synchronization. Similarly, the block count indicates how often a displayed entity was blocked for the same reason.

In contrast to that, the blocked time presented in a blocking statistic displays how long a shown entity caused another entity to be blocked, i.e. how long it was blocking any other entity. Correspondingly, the block count points out how often a displayed entity was blocking another entity because of synchronization usage.

Due to the serializing nature of most synchronization primitives, it is possible in highly contended applications that one thread may block several threads at a time. This can happen when several threads try to acquire the same synchronization primitive while another one already owns it. Having said that, it is obviously also possible that a thread may be blocked by several threads during one blockage. This case can occur when several threads try to enter a critical section at about the same time. Then only one of them can acquire the synchronization primitive at a time while all others have to wait. Therefore, one particular thread may have to stand in line while other threads take their turns.

6.1.1.3. Self and Total Blocked Times

The basis for SAP JVM Profiler's Synchronization Analysis is SAP Java VM's built-in capability to collect information about contended synchronization operations in the currently running SAP Java VM. The recorded information include timestamps, identifiers of the synchronization primitives being utilized and the involved threads. Even though this information alone may provide some value for the Synchronization Analysis, the call stacks supplied for each contended synchronization operation add the necessary context so that the user can identify which code path causes the contention.

Since the complete stack traces of all contended synchronization operations are available, it is possible to distinguish between the self blocked time and the total blocked time of a method. Analogously, it is also feasible to differentiate between the self block count and the total block count of a method.

• The self blocked time of a method is the time the VM spent blocked directly in that method disregarding its called methods. It is the sum of the durations of all blockages that occurred directly in that method. The self block count of a method is the number of blockages that happened directly in that method.

• In a similar fashion, the appearance of the method somewhere in the stack trace of a blockage due to synchronization implies a contribution to the method's total blocked time. It is the time the VM was blocked in that method or in one of its called methods. The total block count is defined analogously. It is the number of times the VM was blocked in that method or in one of its called methods.

Using the definitions of self blocked time and total blocked time from above, it is easy to see that the sum of all self blocked times make up the overall time the application was blocked due to contended synchronization. Similarly, the total blocked times of the application's main method and all Thread.run methods constitute the overall blocked time.

The self and total values provided by most statistics of the SAP JVM Profiler's Synchronization Analysis allow the user to view for example the methods involved in contended synchronization in a flat manner without losing
the overview. The distinction between the self and total values eases the task of identifying which entities actively contribute to the contention by employing means of synchronization. In addition to that, it furthermore clearly highlights the innocent bystanders, i.e. the entities that do not use synchronization themselves, but call methods that do so. These entities can be spotted without difficulty because they have zero as self values but non-zero total values.

6.1.2. The Pi Calculator Example

In this section we describe the example application that is used in the following sections to discuss the features of the SAP JVM Profiler's Synchronization Analysis in detail. Since a sensible example showing synchronization usage requires to be multi-threaded, its results may not be deterministic and will probably differ from run to run. Nevertheless, this does not limit the value of the example because the basic concepts can be demonstrated very well.

The example application used for this walkthrough of the Synchronization Analysis tries to find an approximation of the mathematical constant Pi. For the multi-threaded approximation of Pi Buffon's needle is utilized which is described in detail in Section 6.1.2.1, “Calculating Pi using Buffon’s Needle”. In case you are already familiar with the algorithm of Buffon's needle you may also skip directly to Section 6.1.2.2, “Example Source Code”.

The file "PiCalculator.zip" contains the complete source code of the example application used in this section. Extract the content of the ZIP file into a directory and import the Java project into Eclipse to execute it.

6.1.2.1. Calculating Pi using Buffon's Needle

The foundation of Buffon's needle problem is the question asked by mathematician Georges-Louis Leclerc, Comte de Buffon in the 18th century:

Assuming the floor is made of parallel wooden strips with constant width and a needle is dropped onto the floor. What is the probability that the needle will lie across a line between two strips?

A schematic representation of the experimental setup is shown in the figure below. In the illustration below t denotes the height of one band while l stands for the length of the needle. a and b indicate two possible needle positions. The former one does not cross a line while the latter does.

Figure 6.2. Schematic representation of Buffon's needle
Integrating the uniform probability density function of the needle fall it is possible to derive the following formula for the probability $P$ shown below.

$$P = \frac{2l}{t\pi}$$

**Figure 6.3. The Probability of Buffon's needle crossing a boundary**

The example application attempts to approximate $\pi$. Using the formula above we can rearrange it to have $\pi$ alone on one side as shown in the following formula. From this we can conclude that it is possible to estimate $\pi$ if we can estimate the probability $P$ by conducting an experiment, i.e. throwing a lot of needles.

$$\pi = \frac{2l}{tP}$$

**Figure 6.4. Calculating $\pi$ using Buffon's needle and the Probability**

Assuming we drop $n$ needles and $h$ of these needles cross a boundary, we can approximate the probability $P$ by the fraction $h/n$. Given this fraction for $P$ the formula above can be estimated by the one below.

$$\pi \approx \frac{2l \cdot n}{t \cdot h}$$

**Figure 6.5. Estimating $\pi$ using Buffon's needle**

The example application shown in detail in Section 6.1.2.2, “Example Source Code” uses this approximation to calculate $\pi$. It does so by randomly 'dropping' a lot of needles on a floor made of equal width strips of wood. Each random throw of a needle can obviously be done in parallel to others as they are completely independent from each other. Ideally, the experiment and hence the estimation of $\pi$ should become better the more needles are dropped.

At the end of the parallel throwing of the needles $\pi$ is approximated by the formula above and the results are printed to the console.

### 6.1.2.2. Example Source Code

```java
double pi = 0; // Store calculated pi
    private static void main(String[] args) {
        long start = System.currentTimeMillis();
        double pi = 0;
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new Worker(), "WorkerThread-" + i);
            threads[i].start();
        }
    }
```
Synchronization Analysis

for (int i = 0; i < threads.length; i++) {
    try {
        threads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
System.out.println("Attempts : " + attempts);
System.out.println("Hits : " + hits);
System.out.println("Estimated pi: " + estimatePI());
System.out.println("Real pi : " + Math.PI);
double runtime = (System.currentTimeMillis() - start) / 1000.0;
System.out.println("Runtime : " + runtime + "s");
}

public static synchronized void throwNeedle() {
    attempts++;
    if (NeedlePosition.generate().crossesLine()) {
        hits++;
    }
}

private static double estimatePI() {
    return 2.0 * NEEDLE_LENGTH * attempts / (ROW_SIZE * hits);
}

private static class NeedlePosition {
    private final double x1;
    private final double y1;
    private final double x2;
    private final double y2;

    private NeedlePosition(double x1, double y1, double x2, double y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    public static NeedlePosition generate() {
        return new NeedlePosition(x1, y1, x2, y2);
    }

    public boolean crossesLine() {
        for (int line = 0; line <= TABLE_LENGTH; line += ROW_SIZE) {
            if ((x1 <= line && x2 >= line) || (x1 >= line && x2 <= line)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public String toString() {
        return "[" + x1 + "/" + y1 + "] [" + x2 + "/" + y2 + "]"
    }
}

private static class Worker implements Runnable {
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            throwNeedle();
        }
    }
}

6.1.3. Profiling the Example

In this section we describe the steps that are necessary to run and profile the example application introduced in Section 6.1.2, “The Pi Calculator Example”. The main tasks that are required for this include:
• Import the example application into Eclipse. This is explained in detail in Section 6.1.3.1, “Importing the Example”.

• Create and adjust a launch configuration for the example application. Section 6.1.3.2, “Creating the Launch Configuration” focuses on this topic.

• Profile the example application while running on a SAP Java VM. This is the subject of Section 6.1.3.3, “Executing the Profiling Run”.

6.1.3.1. Importing the Example

The source files of the example application are included in the file "PiCalculator.zip". After extracting the files into a directory of your choice it is possible to import the example project into Eclipse using the import wizard. The wizard can be opened by choosing Import from the File menu as shown in the figure below. An alternative way to open the import wizard is to use the context menu of the package explorer and choose Import.

![Figure 6.6. File menu of Eclipse](image)

The example application is available as a complete Eclipse project so that it can be directly imported through the import wizard. On the first page of the wizard choose General # Existing Projects into Workspace and select Next. The directory, into which the example application was extracted, can be entered into the marked text field on the next page of the wizard as indicated in the screenshot below. It is also possible to use the Browse button to select the directory using the file system tree.
Figure 6.7. Importing the example into Eclipse

Selecting Finish will finalize the import of the example application project into Eclipse. Now, the package explorer should include the imported project with the content as displayed in the figure below.

Figure 6.8. The imported Pi Calculator project
6.1.3.2. Creating the Launch Configuration

After importing the example application into the Eclipse workspace, you may want to run it to see how it works. The example application can be executed as any simple Java program by creating a run configuration in the Java perspective and execute it. Depending on the system the example application is run on, it may take several seconds until the program completes. It prints its results at the end of a run directly to the console.

The steps required to profile the example application are similar to an ordinary execution of the program. First a launch configuration has to be created and configured. This is analogous to the run configuration but has a few profiling related derivations. To create a launch configuration, choose Profile As (SAP JVM) # Profile Configurations from the context menu of the PiCalculator class as shown in the figure below.

![Figure 6.9. Profile the Example Application](image)

An alternative way to open the emerging profile configuration dialog is to use the Profile action available on the toolbar menu. Similar to the Run and Debug toolbar actions, the Profile action will open the launch configuration dialog in which you can select the type of application you want to profile.

In the profile configuration dialog, a new launch configuration has to be created and adjusted to collect synchronization information. The launch configuration can be constructed by choosing Java Application # New. This should automatically fill the most important fields like Name, Project and Main class when started from the PiCalculator class.

Even though most attributes of the launch configuration automatically receive sensible default values, some properties have to be adapted before starting the profiling run. The most important ones are the profiling options that are found in the Profiling tab as shown in the figure below. For this walkthrough the Synchronization Analysis indicator should be set.
The **Profiling** tab also allows to set filters to reduce the amount of data that is created during a profiling run. These filters are usually more relevant when profiling an application server and enable the user to analyze only selected entities, e.g. applications or users. The small example application used for this walkthrough does not require filtering so that all corresponding options can be left empty.

Please note that it is also possible to start a profiling run without selecting any analysis type. If no analysis is chosen at the beginning, the profiling run will only include lifecycle information (see Section 8.2, “Profiling Lifecycle and Analysis Overview” for details). Any profiling analysis can be started or stopped at any point in time during a profiling run.

The last step necessary before actually profiling the example application is to check that the Java runtime environment used for the profiling run is a SAP Java VM. Using a SAP Java VM is essential since the SAP JVM Profiler needs an optimized way of interaction with the running VM. The Java runtime environment associated to the launch configuration can be configured in the **JRE** tab as shown in the screenshot below.
After selecting a correct runtime environment all changes made to the launch configuration can be saved using the **Apply** button. The launch configuration can be reused for profiling run at a later point in time.

### 6.1.3.3. Executing the Profiling Run

Now, that the launch configuration is finished a new profiling run can be started from the profile configuration dialog by pressing the **Profile** button. In case the launch configuration is already saved, a profiling run can also be launched directly using the **Profile** action from the toolbar menu. Analogous to the **Run** and **Debug** action, this action shows the last few launch configurations and allows to launch a profiling run with one of the listed launch configurations directly.

After starting the profiling run and switching to the profiling perspective the profiling configuration view should look similar to the figure below. During the runtime of the example application it shows the connection status, the analysis currently active and a summary of the profiling events of the enabled analysis. The summary information is updated while the application is being profiled.

During a profiling run the lifecycle view displays some general information about the profiled VM. Among the information shown in the lifecycle view are the CPU and memory load and the enabled analysis type. After
the profiling run finished, it also includes summary information similar to the one displayed during a profiling run in the profiling configuration view. A figure of the lifecycle view after profiling the example application is shown in the picture below.

Figure 6.13. Lifecycle View of Synchronization Analysis

The Blocked Time visible for example in the lifecycle view indicates how much time was spent in the current profiling run being blocked due to synchronization usage. This duration is accumulated over all threads of the profiled application.

The other summary value used in the profiling configuration view during the profiling run as well as in the lifecycle view after the profiling run has finished is the Block Count. It denotes how often a blockage occurred because of parallel synchronization utilization.

Figure 6.14. Configuration View after Profiling Run

When a profiling run has finished, a new snapshot is assembled from the profiling data which was collected during the profiling run. The snapshot created from the example application profiling run is shown in the profiling configuration view as Synchronization Statistic 1 in the figure above. Usually, the snapshot is also directly opened when created even though this behavior can be changed in the profiling preferences. The overview page for the new snapshot is shown in the screenshot below. Besides some summary information concerning the
profiling run the overview page includes several entry point of the Synchronization Analysis that are discussed in detail in Section 6.1.4, “Evaluating Profiling Data”.

![Synchronization Summary](image)

**Methods (Flat)**
Shows the total times and frequencies methods are blocked on entering a synchronized section. The methods are displayed in a flat manner.

**Methods (Hierarchical)**
Shows the total times and frequencies methods are blocked on entering a synchronized section. The methods are displayed hierarchically according to their call relationship.

**Blocked Time per Thread...**

**Threads by ID**
Shows the total times and frequencies threads were blocked on entering a synchronized section. The threads are listed by their id.

**Threads by Name**
Shows the total times and frequencies threads are blocked on entering a synchronized section. The threads are listed by their name.

**Blocked Time per SAP AS Java Entity...**

**Users**
Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per user.

**Sessions**
Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per session.

**Requests**
Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per request.

**Applications**
Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per application.

**Blocked Time per Component...**

**Components (Flat)**
Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per component. The components are displayed in a flat manner.

**Components (Hierarchical)**
Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per component. The components are displayed hierarchically according to their call relationship.

**Figure 6.15. Snapshot Overview of Synchronization Analysis**

### 6.1.4. Evaluating Profiling Data

This section focuses on the evaluation of the profiling data generated in the previous section. The Synchronization Analysis walkthrough presented here uses the Pi Calculator example application introduced in Section 6.1.2, “The Pi Calculator Example”. Even though the example application is only rather small, it helps to familiarize with the statistics that are available in the Synchronization Analysis of the SAP JVM Profiler.
The analysis will start with a Methods (Flat) Statistic in Section 6.1.4.2, “The Method Statistics” to get an overview of the profiled application. In the subsequent sections the other statistics of the Synchronization Analysis are presented in detail. As described in Section 6.1.1.2, “Blocked and Blocking in the Synchronization Analysis” the statistics of the Synchronization Analysis come in two flavors: Blocked and Blocking. To keep this walkthrough concise, some of the statistics are introduced only in one of the two flavors. The semantics of the statistics that have been omitted can nevertheless be easily derived from the description of the related statistic in conjunction with the corresponding perspective as mentioned in Section 6.1.1.2, “Blocked and Blocking in the Synchronization Analysis”.

6.1.4.1. The Synchronization Snapshot Overview

Before diving directly into the Methods (Flat) Statistic in Section 6.1.4.2, “The Method Statistics” we take a step back and first have a look at the snapshot overview page as shown in Figure 6.15, “Snapshot Overview of Synchronization Analysis”. The most important information about the synchronization snapshot is displayed in the synchronization summary (see figure below).

![Figure 6.16. Synchronization Summary of a Snapshot]

The Collection Period denotes the duration of the snapshot or, as in this case, the whole profiling run. Next to it, the Blocked Time is shown which is the overall duration accumulated over all application threads that have been spent blocked due to synchronization usage. This value may surpass the Collection Period because it is the sum of the Blocked Times of all threads. The Blocked Times of each thread is obviously limited by its runtime and hence the Collection Period. Therefore, the overall Blocked Times is limited by the number of threads times the Collection Period.

The last summary information is the Block Count. This performance counter indicates how many blockages occurred due to synchronization usage, i.e. how often any application thread was blocked because it tried to acquire a synchronization primitive that was already held by some other thread.

Looking at the actual summary information from the example application, it seems a bit peculiar that the amount of Blocked Time by far exceeds the Collection Period. As mentioned above, this is possible but it is generally a sign of synchronization issues. In addition to that, the Block Count is also rather high as we would have expected the application threads to work independently without interfering with each other. These are the first strong indicators that there are some issues in the example application with respect to synchronization usage.

6.1.4.2. The Method Statistics

After having found initial evidence that there might be some synchronization problems ahead of us we use the Methods (Flat) Statistic for a first insight into the location of the issue. The Methods (Flat) Statistic can be opened directly from the synchronization snapshot overview page by selecting the corresponding entry. An alternative entry point for the Methods (Flat) Statistic is the context menu of the corresponding synchronization snapshot in the profiling configuration view. This is displayed in the figure below.
Synchronization Analysis

Figure 6.17. Context Menu of Synchronization Snapshot

Please note that we start with a *Blocked* variant of the statistic because we want to know where the example application was blocked due to utilization of synchronization. Using the context menu of the synchronization snapshot, it is also possible to navigate directly to the *Blocking* variant of the Methods (Flat) Statistic. The snapshot overview page only provides entry points to *Blocked* statistics.

After opening the Methods (Flat) Statistic in any of the two ways described above we see something similar to the picture below. The table shows all methods that were involved in contended synchronization as well as their calling method, i.e. the methods that have directly or indirectly called a method that was blocked due to contended synchronization. For each of these methods the *Blocked Time* and *Block Count* are shown as *Self* and *Total* values (described in detail in Section 6.1.1.3, “Self and Total Blocked Times”).

Figure 6.18. Blocked Methods (Flat) Statistic

The Pi Calculator example application is small enough that all entries fit into the viewable area of the statistic table. For more complex applications one of the numerous filtering capabilities of the SAP JVM Profiler may be used to reduce the amount of data shown in the table. For any statistic, the standard filtering options are shown in the toolbar right above the table. The toolbar of a Methods (Flat) Statistic is displayed in the figure below.

Figure 6.19. Methods (Flat) Statistic Toolbar

You may search directly for table entries using the text input field on the left of the toolbar. Next to the search options you can change the visibility of individual columns by selecting the pull-down menu and checking...
the columns you want to see. After that in the toolbar the compare action can be found which is useful to compare consecutive profiling runs.

The next two toolbar actions can be used to filter and/or inline methods according to some user-defined criteria. Methods filters can be specified using arbitrary patterns. The last four actions of the toolbar enable you to look at the Methods (Flat) Statistic from different perspectives. For example, it is possible to view the statistic accumulated by class, i.e. all entries are summarized per class. A screenshot of this is shown in the figure below. It is also possible to look at the Methods (Flat) Statistic from a class loader, package or package tree perspective. The semantic of these views are analogous to the view by class.

![Figure 6.20. Blocked Classes Statistic](image)

Getting back to the example application, we see clearly that there seems to be a synchronization problem in the method `PiCalculator.throwNeedle`. Its **Self Blocked Time** accounts for nearly the whole blocked time we saw on the snapshot overview page. The same goes for the **Self Block Count**.

From the Total Blocked Times we can assume that `PiCalculator.throwNeedle` seems to be called from `PiCalculator$Worker.run`. We can prove this assumption using any of two approaches without looking into the source code:

1. The Calling Methods (Hierarchical) Statistic
2. The Methods (Hierarchical) Statistic

To see the differences between both approaches we will use the Calling Methods (Hierarchical) Statistic as well as the plain Methods (Hierarchical) Statistic in this order.

The Calling Methods (Hierarchical) Statistic can be utilized to view the methods that have been calling the selected entry. To open this statistic just select the corresponding entry, in this case the method `PiCalculator.throwNeedle`, and choose Calling Methods (Hierarchical) Statistic from its context menu as shown in the figure below.

![Figure 6.21. Opening the Calling Methods (Hierarchical) Statistic](image)

The resulting view, as presented in the picture below, shows the methods and their calling methods in a hierarchical manner. If we expand the tree like in the figure we see that our assumption was right. The statistic
Synchronization Analysis

shows the complete call graph until it reaches the roots of the call stack which is either Thread.run or the corresponding main method.

Figure 6.22. Blocked Calling Methods (Hierarchical) Statistic

Tackling the problem from the other direction, it is possible to use the Methods (Hierarchical) Statistic for this purpose. This statistic shows the same information as the Methods (Flat) Statistic, but does so in a hierarchical manner, i.e. it displays the contended synchronization information along the call graphs. This statistic can be opened analogous to the Methods (Flat) Statistic: The snapshot overview page is the main entry point for the Methods (Hierarchical) Statistic, but the context menu of the synchronization snapshot also includes entries for this statistic. In addition to this, you may start from the Methods (Flat) Statistic as well and choose any item, in this case the method PiCalculator.throwNeedle, and select Methods (Hierarchical) Statistic from its context menu. While the other two start points show all contended synchronization information the last one filters the information with respect to the selection.

Figure 6.23. Blocked Methods (Hierarchical) Statistic

The figure above shows the result when starting from the snapshot overview page or the synchronization snapshot. Our assumptions hold here as well. This time we see the call graph in the opposite direction starting from the method roots. Since we did not use any filtering we see all contended synchronization information and not only the data related to PiCalculator.throwNeedle.

As we have seen in the two examples above, the Calling Methods (Hierarchical) Statistic and the Methods (Hierarchical) Statistic show similar information, but do so in different ways. Due to the similarities it is possible to confuse the one statistic type with the other. Nevertheless, there are several things that help you distinguish between both statistics types. Obviously, the name as displayed in the statistic tab shown above the table is a first indicator of the statistic type. The profiling configuration view also shows the name of the statistics in a hierarchical manner. In addition to that, please note the arrows to the left of the method names in the Calling Methods (Hierarchical) Statistic and the Methods (Hierarchical) Statistic. They indicate the direction of the method call hierarchy, i.e. they point from caller to callee. Therefore, they also show what kind of statistic is currently shown.

In contrast to the Methods (Flat) Statistic the Methods (Hierarchical) Statistic and the Calling Methods (Hierarchical) Statistic both include information about the actual line numbers of the method calls, i.e. the hierarchical views show exactly were in the method execution a blockage took place. The line numbers are displayed after clicking on the arrow icon at the end of the method names.

Now that we know where a potential synchronization problem might be and we already identified the call graph, it may be interesting to see which method actually blocked the execution of others. This can be the same method or any other method that uses the same synchronization primitive, i.e. in this case object monitor. The Synchro-
Synchronization Analysis

Synchronization Analysis provides the user with several choices how to achieve this task. What we are looking for is a blocking method statistic. Since the call graph of the blocking method may be of interest, too, a hierarchical view might be the way to go. We could once again start from the synchronization snapshot and use the context menu to get a Blocking Methods (Hierarchical) Statistic for the whole profiling run of the example application. Nevertheless, in this walkthrough we will focus on another way and open a Blocking Calling Methods (Hierarchical) Statistic from the Methods (Flat) Statistic. After selecting the method PiCalculator.throwNeedle we choose the Blocking Calling Methods (Hierarchical) Statistic. The resulting statistic is shown in the figure below.

![Figure 6.24. Blocking Calling Methods (Hierarchical) Statistic](image)

As we see in the screenshot above, the method PiCalculator.throwNeedle is only blocked by itself. This eases the analysis because we do not have to keep the interactions between different methods in mind and can focus on this method for optimizations. More complex synchronization issues usually will show more blocking methods than this example. We postpone the discussion of the peculiar high Block Counts until we discuss the Synchronization Event Statistics in Section 6.1.4.4, “The Synchronization Event Statistics”.

6.1.4.3. The Object Monitor Statistics

In the last section we found that a potential synchronization problem may be in the method PiCalculator.throwNeedle. Before looking into the source code to inspect the method we can use the Object Monitor Statistic to have a look at the kind of object monitor that was so heavily contended. This can also be used as an alternative entry point into the Synchronization Analysis if the application is rather small or the used object monitors of the application are well known.

To open the Object Monitor Statistic for a whole synchronization snapshot you can utilize the corresponding context menu entry of the snapshot in the profiling configuration view. In case you are only interested in the object monitors of a specific method or thread it is also possible to navigate from most statistics of the Synchronization Analysis to the Object Monitor Statistic. This is achieved by selecting the corresponding entries and choosing Object Monitor Statistic from the context menu. This is shown for the example application in the figure below.
The screenshot below displays the Object Monitor Statistic when opened from the Methods (Flat) Statistic for the method `PiCalculator.throwNeedle`. The statistic presents the object monitors that were used for contended synchronization, i.e. the object monitors shown in the statistic were the causes for blockages due to synchronization usage. As several instances of any class can be used as object monitors in Java the statistic contains an identifier for each object besides its class.

The Object Monitor Statistic of the example application only shows one entry of type `java.lang.Class`. This means that only one object monitor was contended in the method `PiCalculator.throwNeedle`. From the type of the object monitor we can assume that a class object was used for the contended synchronization of the example application. Normally, this indicates a synchronized class method or a code block synchronized using the class object. Let us now have a look at the source code of the method to verify the hypothesis.
public static synchronized void throwNeedle() {
    attempts++;
    if (NeedlePosition.generate().crossesLine()) {
        hits++;
    }
}

As expected we see that PiCalculator.throwNeedle is a synchronized class method. When we look at its call site as identified using the (Calling) Methods (Hierarchical) Statistic and displayed below, it is quite obvious where the high contention comes from. The synchronized class method is called by all worker threads in a tight loop. Only one of the worker threads may execute the method itself due to synchronization. Hence, the threads will compete heavily to get the corresponding object monitor.

... public void run() {
    for (int i = 0; i < 1000000; i++)
        throwNeedle();
} ...

At this point the origin of the synchronization problem of the example application is clear. There are several possible ways how this obstacle can be solved, e.g. the worker threads could store the results in a thread local data structure and merge them only after the all iterations have finished. However, the actual fix and the subsequent comparative measurements are left as an exercise for reader. Depending on the kind of fix, the comparative measurements may show no contention at all.

6.1.4.4. The Synchronization Event Statistics

Even though we found the synchronization problem of the example application, this and the following sections go into more detail of the Synchronization Analysis showing other aspects that have not been covered, yet. In this particular section we concentrate on the Synchronization Event Statistics. These statistics show contended synchronization at the lowest level. In contrast to the other statistics that accumulate the information, each and every blockage is listed separately in the Synchronization Event Statistics. This allows you to check all blockages individually to distinguish between cases where one long blockage dominates all others or where most of the blockages have a similar duration.

Like for most of the statistics of the Synchronization Analysis there are two kinds of Synchronization Event Statistics that will be described in the order listed below:

1. The Block Event Statistic

2. The Blocking Event Statistic

First, we start with the Block Event Statistic which shows every blockage separately. To open the statistic we once again use the Methods (Flat) Statistic described in Section 6.1.4.2, “The Method Statistics”. After selecting the method PiCalculator.throwNeedle we choose the statistic from the context menu as indicated in the figure below.
Figure 6.27. Opening the Block Event Statistic

The new view should look similar to the screenshot below.

Figure 6.28. Block Event Statistic

Every blockage has an index which shows the order from a time perspective, i.e. the blockage with index 1 happened before the blockage with index 2. For each entry the duration of the blockage is shown. Obviously, the Block Count of each blockage is 1 in the Block Event Statistic. To verify that we really have all blockages we can select all entry using the Edit menu or the keyboard short cut `CTRL + A`. Since the statistic contains a lot of entries it may take a while to finish. The result is shown in the status bar as depicted in the figure below. The status bar always displays this kind of summary if more than one entry is selected.
Figure 6.29. Block Event Statistic Status Bar

If we compare the status bar summary to the values of the blockages shown in the statistic we see that no blockage completely dominates the others because even the ones on top, i.e. the ones with the longest duration, each make up less than 0.2 percent of the overall Blocked Time. The exact percentage information is shown as a tool tip for each entry.

Another tool of the Synchronization Analysis that might be handy now is the Blocking Event Statistic. With the Block Event Statistic we have seen each individual blockage. However, we may want to know who was the reason for the blockage, i.e. we require information about the entity that held the object monitor we tried to acquire it. Since several threads may try to obtain the same synchronization primitive at a similar point in time and only one of them may succeed, it is possible for more than one thread to cause the blockage of another thread. Afterwards the threads take their turns using the object monitor before the original thread acquires it. The original thread was therefore blocked by several threads. The Blocking Event Statistic would show several Blocking Events for the blockage of the original thread each denoting one thread that held the lock while the original thread waited.

Figure 6.30. Opening the Blocking Event Statistic

We demonstrate the Blocking Event Statistic using the example application. Therefore, select one entry of the Block Event Statistic and choose the Blocking Event Statistic from the context menu as indicated in the figure above. The resulting Blocking Event Statistic is shown in the screenshot below.
From the Blocking Event Statistic we see that there are far more *Blocking Events* than the example application has worker thread. This might be surprising at first, but it is due to the fact that the SAP Java VM, like most Java VMs, has an optimized synchronization subsystem. The upside of this is that it is in general quite fast. The downside, however, is that fairness is seldom achieved with respect to synchronization. This means that it is not guaranteed by the VM that the order in which threads arrive at a critical section corresponds to the acquisition order of the corresponding synchronization primitive. For the Blocking Event Statistic this suggests that more than one entry correspond to the same thread, but as mentioned above this statistic is not accumulated so that each entry is listed separately. The thread perspective is discussed in Section 6.1.4.5, “The Thread Statistics”.

The example statistic shows that most of the *Blocking Events* were quite short while only a few of them took more than a millisecond. Again, this points to the highly optimized synchronization subsystem indicating that most of the *Blocking Events* took the fast path of locking. From each table entry it is possible to navigate to most other statistics of the Synchronization Analysis. Note however that most navigation links from the Blocking Event Statistic lead to other *Blocking* statistics unless indicated otherwise. This is analogous to the *Blocked* statistics.

### 6.1.4.5. The Thread Statistics

The Thread Statistics form the last major statistic group left in this walkthrough of the Synchronization Analysis. These statistics present *Blocked Time* and *Blocked Count* on a per thread basis. This is especially useful if the threads of an application have names that imply their field of work. The Thread Statistics may prove to be less valuable in the face of thread pools.

It is possible to use the Thread Statistics as an entry point into the Synchronization Analysis. The *Blocked* variants of the statistics are available on the synchronization snapshot overview page as depicted in Figure 6.15, “Snapshot Overview of Synchronization Analysis”. The context menu of the synchronization snapshot include both the *Blocked* and *Blocking* variants as shown in Figure 6.17, “Context Menu of Synchronization Snapshot”. An example of a Blocked Thread By ID Statistic for the PiCalculator application is displayed in the figure below.

<table>
<thead>
<tr>
<th>Index</th>
<th>Blocked Time</th>
<th>Block Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>37.9 ms</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>24.6 ms</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>18.6 ms</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>12.4 ms</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5.59 ms</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5.30 ms</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2.07 ms</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>201 us</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>32.6 ms</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>17.6 us</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>13.6 us</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>10.8 us</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>11.6 us</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>11.4 us</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>10.6 us</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>10.4 us</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>10.2 us</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>10.2 us</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>20.0 us</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>9.73 us</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>9.90 us</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>9.06 us</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>8.69 us</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>8.48 us</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>8.87 us</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>7.97 us</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>7.67 us</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>2.07 us</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>8.48 us</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.31. Blocking Event Statistic
Synchronization Analysis

Figure 6.32. Blocked Threads by ID Statistic

Since we already found the synchronization issue which causes the execution to be nearly serialized it is not surprising to see all worker threads listed in the Blocked Thread Statistic. The Synchronization Analysis allows the usual navigation links for the entries of the statistic. However, there are special navigation links to the Blocking versions of the Thread Statistics. Using this link you can get a statistic as shown in the figure below.

Figure 6.33. Blocking Threads by ID Statistic

Besides the entry statistics, the Thread Statistics are also available from most other statistics of the Synchronization Analysis, i.e. it is possible to create a Thread Statistic for any given entity, e.g. a method, that shows the thread(s) executing this entity. This can be useful for various reasons. In the last section for example (see Section 6.1.4.4, “The Synchronization Event Statistics” for details) we wondered which thread was the cause of a blockage. With the Thread Statistic it is easy to answer this question. The screenshot below shows the same information as Figure 6.31, “Blocking Event Statistic”, but aggregates the data on a per thread basis.

Figure 6.34. Blocking Threads by ID Statistic for a Blockage
6.1.4.6. Advanced Statistics for Complex Scenarios

The example application used for this walkthrough of the Synchronization Analysis is intentionally rather simple so that the shown statistics are very concise. However, real world applications tend to be much more complex in their nature leading to a huge amount of data to dig through in the Synchronization Analysis. Even though the SAP JVM Profiler's Synchronization Analysis offers a wide range of filtering capabilities as introduced in the previous sections, the discussed facilities may not suffice to handle the mass of data successfully. For complex scenarios, the Synchronization Analysis of the SAP JVM Profiler provides the user with advanced statistics which assist in narrowing down the available data to be comprehensible again.

Application server scenarios are among the complex applications mentioned above. For applications running in an SAP NetWeaver Application Server Java environment the SAP JVM Profiler's Synchronization Analysis provides special means for profiling. This means that it is possible to use SAP AS Java entities for filtering the information about contended synchronization. A general introduction into these kind statistics can be found in Section 8.5.1, “SAP AS Java Entity- / Thread-Filter”.

The SAP AS Java entity and thread filter enable the user for example to view Blocked Time and Block Count from a user or session perspective. Using this approach of the Synchronization Analysis it is possible to identify the higher level entities like users or sessions that cause other higher level entities to be blocked due to contention with respect to synchronization. The Synchronization Analysis furthermore permits to dig from these higher level entities to the lower level statistics, for example the Methods (Flat) Statistic. Of course, the other way round is available as well.

Besides being the navigation targets the SAP AS Java entity and thread filter can also be used as entry statistics either from the synchronization snapshot overview page or the synchronization snapshot context menu. The entry points from the overview page are shown in the figure below.

![Figure 6.35. Opening the SAP AS Java Entity and Thread Filter Statistics](image)

**Figure 6.35. Opening the SAP AS Java Entity and Thread Filter Statistics**

Another advanced statistic available in SAP JVM Profiler's Synchronization Analysis are the Component Statistics. This is based on the fact that most large applications are composed from several more or less well defined components. One goal of the Synchronization Analysis of a large application may be to analyze the dependencies of the components concerning contended synchronization usage. The Components Statistics provide means to view Blocked Time and Block Count accumulated per component. Furthermore, they can help in identifying the dependencies with respect to synchronization. A general introduction into these kind of statistics can be found in Section 8.5.5, “Component Statistics”.

Just like the SAP AS Java entity and thread filter it is possible to use the Component Statistics as entry statistics as well as navigation targets during the Synchronization Analysis. As a navigation target the Component Statistics can be opened using the corresponding context menu entries. The resulting statistic displays the components of the formerly selected entities.

The Component Statistics can also be utilized as entry points either using the synchronization snapshot overview page or the synchronization snapshot context menu. The resulting statistics show the contended synchronization information on a per component level for the whole application. The corresponding entry points from the overview page are displayed in the figure below.
Figure 6.36. Opening the Component Statistics

**Components (Flat)**

Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per component. The components are displayed in a flat manner.

**Components (Hierarchical)**

Shows the total times and frequencies threads are blocked on entering a synchronized section aggregated per component. The components are displayed hierarchically according to their call relationship.
Chapter 7. File I/O Analysis

7.1. File I/O Analysis Walkthrough

This is a detailed walkthrough for the SAP JVM Profiler's File I/O Analysis. It explains the basic concepts of this analysis and shows it in action using an example application. The example is a short, self-contained Java program. It is used to show basic and advanced features of the File I/O Analysis.

You can proceed from here in one of the following ways:

• Start the walkthrough with Section 7.1.1, “File I/O Analysis Overview”. Here you will get a short overview about the objectives and the main concepts of the File I/O Analysis. Furthermore, it is discussed how the process of analyzing file I/O issues is supported by the SAP JVM Profiler step by step.

• Get to know the sample Java application FileCopyExample used to demonstrate features of the File I/O Analysis in Section 7.1.2, “The File Copy Example Program”

• Go directly to Section 7.1.3, “Profiling the Example” to learn how to collect profiling data of a Java VM in action, which is the basis for statistical evaluations afterwards.

• Section 7.1.4, “Evaluating Profiling Data” summarizes the various types of statistics and views which help you to identify the synchronization issues very quickly.

7.1.1. File I/O Analysis Overview

7.1.1.1. Objectives of File I/O Analysis

The main objective of the File I/O Analysis is to show where the application was using most of the time because of file I/O operations. Furthermore, it enables the user to detect errors like files that are opened, i.e., for reading but never closed again. With the knowledge gained through the File I/O Analysis developers may be able to improve the file I/O usage for example using caching or other techniques to avoid or reduce I/O operations.

7.1.1.2. Self and Total Times

The basis for SAP JVM Profiler's File I/O Analysis is SAP Java VM's built-in capability to collect information about file I/O operations in the currently running SAP Java VM. The recorded information include timestamps, identifiers of the files in use, the involved threads and call stacks and the number of transferred bytes.

Since the complete stack traces of all file I/O operations are available, it is possible to differentiate reading or writing operation times by the self time and the total time of a method. Analogously, it is also feasible to distinguish between the self bytes and the total bytes transferred by I/O operations caused by a method.

• The self time of a method is the time the VM spent in I/O operations directly because of that method disregarding its called methods. The self bytes of a method are the number of bytes read or written directly in that method.

• In a similar fashion, the appearance of the method somewhere in the stack trace of an I/O operation implies a contribution to the method's total time. It is the time the VM spent in that method or in one of its called methods doing I/O operations. The total bytes are defined analogously as the number of bytes read or written in that method or in one of its called methods.

Using the definitions of self time and total time above, it is easy to see that the sum of all self times make up the overall time the application was spending doing file I/O. Similarly, the total bytes of the application's main method and all Thread.run methods constitute the overall transferred bytes.
The self and total values provided by most statistics of the SAP JVM Profiler's File I/O Analysis allow the user to view for example the methods involved in file I/O operations in a flat manner without losing the overview. The distinction between the self and total values eases the task of identifying which entities actively contribute to file I/O.

7.1.2. The File Copy Example Program

In this section we describe the example application that is used in the following sections to discuss the features of the SAP JVM Profiler's File I/O Analysis in detail.

This example program is kept very basic on purpose. It doesn't solve any interesting problem and serves the sole purpose of performing some file operations. It will copy a supplied text file "LoremIpsum.txt" to a new file "copy.txt". Furthermore, it creates another copy "lowerCase.txt" with all characters of the original document converted to lower case. Well, not the most useful program in the world, but it will do to demonstrate the most important features of the File I/O Analysis.

To make it a little bit more interesting, a coding error was "hidden" in the example and we will see, how the File I/O Analysis will help to spot the problem.

7.1.3. Profiling the Example

In this section we describe the steps that are necessary to run and profile the example application introduced in Section 7.1.2, “The File Copy Example Program”. The main tasks that are required for this include:

• Import the example application into Eclipse. This is explained in detail in Section 7.1.3.1, “Importing the Example”.

• Create and adjust a launch configuration for the example application. Section 7.1.3.2, “Creating the Launch Configuration” focuses on this topic.

• Profile the example application while running on a SAP Java VM. This is the subject of Section 7.1.3.3, “Executing the Profiling Run”.

7.1.3.1. Importing the Example

The source files of the example application are included in the file "ProfilerIOExample.zip". After extracting the files into a directory of your choice it is possible to import the example project into Eclipse using the import wizard. The wizard can be opened by choosing Import from the File menu as shown in the figure below. An alternative way to open the import wizard is to use the context menu of the package explorer and choose Import.

![Figure 7.1. Context menu of the package explorer](image)
The example application is available as a complete Eclipse project so that it can be directly imported through the import wizard. On the first page of the wizard choose General # Existing Projects into Workspace and select Next. The directory, into which the example application was extracted, can be entered into the marked text field on the next page of the wizard as indicated in the screenshot below. It is also possible to use the Browse button to select the directory using the file system tree.

![Figure 7.2. Importing the example into Eclipse](image)

Selecting Finish will finalize the import of the example application project into Eclipse. Now, the package explorer should include the imported project with the content as displayed in the figure below.

![Figure 7.3. The imported example project](image)

### 7.1.3.2. Creating the Launch Configuration

After importing the example application into the Eclipse workspace, you may want to run it to see how it works. The example application can be executed as any simple Java program by creating a run configuration in the
Java perspective and execute it. Depending on the system the example application is run on, it will take less than a second until the program completes. It prints a "Finished" message to the console and creates two new files in its working directory, which should be visible in the Eclipse package explorer.

The steps required to profile the example application are similar to an ordinary execution of the program. First a launch configuration has to be created and configured. This is analogous to the run configuration but has a few profiling related derivations. To create a launch configuration, choose `Profile As (SAP JVM) # Profile Configurations` from the context menu of the `FileCopyExample` class as shown in the figure below.

![Figure 7.4. Profile the Example Application](image)

An alternative way to open the emerging profile configuration dialog is to use the `Profile` action available on the toolbar menu. Similar to the `Run` and `Debug` toolbar actions, the `Profile` action will open the launch configuration dialog in which you can select the type of application you want to profile.

In the profile configuration dialog, a new launch configuration has to be created and adjusted to collect file I/O information. The launch configuration can be constructed by choosing `Java Application # New`. This should automatically fill the most important fields like `Name`, `Project` and `Main class` when started from the `FileCopyExample` class.

Even though most attributes of the launch configuration automatically receive sensible default values, some properties have to be adapted before starting the profiling run. The most important ones are the profiling options that are found in the `Profiling` tab as shown in the figure below. For this walkthrough the `File I/O Analysis` indicator should be set.
Figure 7.5. File I/O Analysis Option

The Profiling tab also allows to set filters to reduce the amount of data that is created during a profiling run. These filters are usually more relevant when profiling an application server and enable the user to analyze only selected entities, e.g. applications or users. The small example application used for this walkthrough does not require filtering so that all corresponding options can be left empty.

Please note that it is also possible to start a profiling run without selecting any analysis type. If no analysis is chosen at the beginning, the profiling run will only include lifecycle information (see Section 8.2, “Profiling Lifecycle and Analysis Overview” for details). Any profiling analysis can be started or stopped at any point in time during a profiling run.

The last step necessary before actually profiling the example application is to check that the Java runtime environment used for the profiling run is a SAP Java VM. Using a SAP Java VM is essential since the SAP JVM Profiler needs an optimized way of interaction with the running VM. The Java runtime environment associated to the launch configuration can be configured in the JRE tab as shown in the screenshot below.

Figure 7.6. Java Runtime Environment Options
File I/O Analysis

After selecting a correct runtime environment all changes made to the launch configuration can be saved using the *Apply* button. The launch configuration can be reused for profiling run at a later point in time.

### 7.1.3.3. Executing the Profiling Run

Now, that the launch configuration is finished a new profiling run can be started from the profile configuration dialog by pressing the *Profile* button. In case the launch configuration is already saved, a profiling run can also be launched directly using the *Profile* action from the toolbar menu. Analogous to the *Run* and *Debug* action, this action shows the last few launch configurations and allows to launch a profiling run with one of the listed launch configurations directly.

After starting the profiling run and switching to the profiling perspective the profiling configuration view should look similar to the figure below. During the runtime of the example application it shows the connection status, the analysis currently active and a summary of the profiling events of the enabled analysis. The summary information is updated while the application is being profiled.

![Configuration View during Profiling Run](image)

**Figure 7.7. Configuration View during Profiling Run**

During a profiling run the lifecycle view displays some general information about the profiled VM. Among the information shown in the lifecycle view are the CPU and memory load and the enabled analysis type. After the profiling run finished, it also includes summary information similar to the one displayed during a profiling run in the profiling configuration view. A figure of the lifecycle view after profiling the example application is shown in the picture below.
Figure 7.8. Lifecycle View of File I/O Analysis

The Transferred Bytes visible for example in the lifecycle view indicates the total volume of data transferred in file I/O operations while profiling the application.

Figure 7.9. Configuration View after Profiling Run

When a profiling run has finished, a new snapshot is assembled from the profiling data which was collected during the profiling run. The snapshot created from the example application profiling run is shown in the profiling configuration view as File I/O Statistic 1 in the figure above. Usually, the snapshot is also directly opened when created even though this behavior can be changed in the profiling preferences. The overview page for the new snapshot is shown in the screenshot below. Besides some summary information concerning the profiling run, the overview page includes several entry point of the File I/O Analysis that are discussed in detail in Section 7.1.4, “Evaluating Profiling Data”.
Figure 7.10. Snapshot Overview of File I/O Analysis

7.1.4. Evaluating Profiling Data

This section focuses on the evaluation of the profiling data generated in the previous section. The File I/O Analysis walkthrough presented here uses the File Copy example application introduced in Section 7.1.2, “The
File Copy Example Program”. Even though the example application is only rather small, it helps to familiarize with the statistics that are available in the File I/O Analysis of the SAP JVM Profiler.

The analysis will start with a Files Statistic in Section 7.1.4.2, “The Files Statistic” to get an overview of the profiled application. In the subsequent sections the other statistics of the File I/O Analysis are presented in detail.

7.1.4.1. The File I/O Snapshot Overview

Before diving directly into the Files Statistic in Section 7.1.4.2, “The Files Statistic” we take a step back and first have a look at the snapshot overview page as shown in Figure 7.10, “Snapshot Overview of File I/O Analysis”. Some high-level information about the file I/O snapshot is displayed in the file I/O summary (see figure below).

Figure 7.11. File I/O Summary of a Snapshot

The Collection Period denotes the duration of the snapshot or, as in this case, the whole profiling run. Next to it, the number of read bytes is shown which is the overall volume accumulated over all file reading operations. The total time spent in these operations is given in the next field. The second row of the summary shows the transferred volume and total time spent in file-writing operations.

This summary helps to spot, whether the application is doing an unusual amount of file I/O at all and if it's worth having a further look into potential optimization.

7.1.4.2. The Files Statistic

The Files Statistic is a good starting point to get an overview of the file I/O operations that happened in the profiled program. It can be opened directly from the file I/O snapshot overview page by selecting the corresponding entry. An alternative entry point for the Files Statistic is the context menu of the corresponding file I/O snapshot in the profiling configuration view. This is displayed in the figure below.

Figure 7.12. Context Menu of File I/O Snapshot
The Files Statistic provides an analysis of the file I/O operations of the profiled application that is presented per file name. For each file that was read or written, this view offers accumulated data about the number of transferred bytes, the durations and how often that file was opened or closed. This way, one can easily spot the files where the most I/O time is spent. The Files View shown in the figure below displays the names of files with I/O operations, sorted by the number of times a file was opened.

Figure 7.13. Files View

The File Copy example application is small enough that all used files fit into the viewable area of the statistic table. For more complex applications one of the numerous filtering capabilities of the SAP JVM Profiler may be used to reduce the amount of data shown in the table. For any statistic, the standard filtering options are shown in the toolbar right above the table. You may search directly for table entries (file names in this case) using the text input field on the left of the toolbar.

Not all available columns are displayed by default, you can change the visibility of individual columns by selecting the pull-down menu and checking the columns you want to see.

We can see that besides the files that are used in our example application, there are more files listed because of class loading (in our case just the rt.jar and the simple application class). Additionally, we find the standard input / output streams <stdin>, <stdout> and <stderr>, as the example did not redirect them. Actually, we can attest that some bytes were sent to the standard output by the code writing the "Finished." message.

We are more interested in the files that are actually read and written from our program. The only written files are the two result files of our program, copy.txt and lowerCase.txt. While the number of written bytes is the same, it is obvious, that writing the lowerCase.txt file took much longer. Looking at the example code, the reason for this becomes clear: the copy is written in large blocks from a buffer, while the lowerCase.txt file is written in text lines that are converted to all lower case characters beforehand.

All of the (non class loading related) reading was done on the LoremIpsum.txt file, which was opened twice. In Section 7.1.4.3, “The Detailed Files Statistic” we will learn how to get more detailed information about operations on selected files of interest.

7.1.4.3. The Detailed Files Statistic

To get a better idea about what was done with one particular file, we can have a look at the Detailed Files Statistic for it by selecting the file entry in the Files Statistic and choosing Show Files (Details) form the context menu of that entry to open the Detailed Files View.
The Detailed Files View that opens now is very similar to the Files View, but now we get information about I/O operations performed on the file we selected and for each time this file was opened. We can see some additional information like the thread that opened or closed the file (with timestamps, if these columns are enabled).

In our example, the file was opened twice, so we get two rows. The first time that file was opened (indicated with ID 0 in the table) there was no corresponding close operation. This is a strong indication of a program error! By selecting the row and navigating further to a Calling Methods Statistic, we can find the methods responsible for opening and reading this file for that particular instance. In this case, it is the copy method that opens the file but "forgets" to close it.

An easier way to find files that were never closed is described in Section 7.1.4.4, "The Open Files Statistic".

### 7.1.4.4. The Open Files Statistic

A common problem with file I/O is code that opens a file and performs some reading or writing operations, just to forget to properly close that file in the end. Especially in cases where the file is passed around to many objects it is sometimes hard to follow all possible execution paths and a simple close is missing in a dark corner.

The Open Files Statistic makes it easy to find files that were opened by the profiled code but never closed. You can access it directly from the snapshot overview page of the File I/O Analysis by clicking the Open Files link. It is also available in the context menu of the file I/O snapshot in the profiling configuration view.
The Open Files View that opens up will look exactly like a normal Files Statistic, except that it is restricted to files that were never closed. The figure below demonstrates the result for our example program.

Figure 7.16. Open Files Statistic

It is no surprise that the example program did not close the standard input / output streams `<stdin>`, `<stdout>`, and `<stderr>`. Also, it is not responsible for closing the `rt.jar`. But we can see that the `LoremIpsum.txt` was opened one time without a matching `close`.

By selecting the file and navigating further to a Calling Methods Statistic, we can find all the methods that perform I/O operations on this file and for this particular instance. That means that we can find out where the file is opened and read. In this case, it is the `copy` method that opens the file but "forgets" to close it.

7.1.4.5. The Method Statistics

Besides finding evidence that there are file I/O related problems in a profiled application, we are naturally interested to get an insight into the location of the issue. The Methods (Flat) Statistic helps to find methods that perform file I/O operations or call other methods that cause file I/O.

The Methods (Flat) Statistic can be opened directly from the file I/O snapshot overview page by selecting the corresponding entry. It is also possible to use the context menu of the snapshot. After opening the Methods (Flat) Statistic in any of these ways we see something similar to the picture below. The table shows all methods that performed file I/O operations, as well as their calling method, i.e. those methods that have directly or indirectly caused a file I/O operation. For each of these methods the volume of transferred bytes and the time spent for reading or writing operations are shown as `Self` and `Total` values (described in detail in Section 7.1.1.2, "Self and Total Times").

Figure 7.17. Methods (Flat) Statistic

Any of the numerous filtering capabilities of the SAP JVM Profiler may be used to reduce the amount of data shown in the table. For any statistic, the standard filtering options are shown in the toolbar right above the table. The toolbar of a Methods (Flat) Statistic is displayed in the figure below.
Figure 7.18. Methods (Flat) Statistic Toolbar

You may search directly for table entries using the text input field on the left of the toolbar. Next to the search options you can change the visibility of individual columns by selecting the pull-down menu and checking the columns you want to see. After that in the toolbar the compare action can be found which is useful to compare consecutive profiling runs (see Section 8.3, “Comparison of Profiling Statistics” for details on comparing).

The next two toolbar actions can be used to filter and/or inline methods according to some user-defined criteria. Methods filters can be specified using arbitrary patterns.

Maybe you noticed that among the methods displayed in the table, there are no methods that one might expect to see in this context, such as `java.io.File.open()` and other I/O related methods of the JDK. These methods are inlined by default and the profiling data for the JDK methods is assigned to the calling method. As it is typically interesting to know the cause of the I/O operation and not, what exact operation was performed, this inlining filter is applied for convenience. If the actually called JDK I/O methods are of interest, the inlining can be switched off using the toolbar action.

The last four actions of the toolbar enable you to look at the Methods (Flat) Statistic from different perspectives. For example, it is possible to view the statistic accumulated by class, i.e. all entries are summarized per class. A screenshot of this is shown in the figure below. It is also possible to look at the Methods (Flat) Statistic from a class loader, package or package tree perspective. The semantic of these views are analogous to the view by class.

Figure 7.19. Classes Statistic

Besides the flat presentation above, there is also a hierarchical variant of the Methods Statistic. This shows the same information presented as a tree. This statistic can be opened analogous to the Methods (Flat) Statistic using the snapshot overview page or the context menu of the snapshot as the entry point. The resulting Methods (Hierarchical) Statistic for our example will look like the figure below.

Figure 7.20. Methods (Hierarchical) Statistic

In this view, it is much easier to recognize the main causes of I/O operations: class loading and the `FileCopyExample` class of the example program. By expanding the tree nodes, one can further explore the call
paths and how they contributed to the total file I/O. The two prominent methods of our example program are the `copy` and `toLowerCase` methods. By expanding further, the actually called I/O methods become visible.

Up to now, we always opened the Method Statistics for the complete snapshot, showing all of the collected information. Often, it is helpful to show the information only for a selected subset of all file I/O operations. In Section 7.1.4.4, “The Open Files Statistic” we learned how to find files that were opened but never closed, with the "LoremIpsum.txt" file being one example. It is very easy to show only the method calls that performed I/O operations on this particular file and on exactly that instance that was never closed. All we need to do is select this file in the Open Files Statistic and open the Method Statistic for the selection from the context menu. The Methods (Hierarchical) Statistic for this example will look like this:

![Figure 7.21. Methods (Hierarchical) Statistic for the Open File](image)

All calls that caused I/O operations on this file started in the `FileCopyExample.main()` method, which called the `copy()` method in turn. There, the file was opened and read, so the chances are high that it should have been closed there as well.

In this basic example, the execution path was very short and we only had to expand a top level node to find the offending method. In larger real-world examples we would benefit from a view where we could directly see the methods calling I/O operations on the selected file. For these cases the SAP JVM Profiler offers the Calling Methods Statistic. We can open a hierarchical view of the calling methods by going back to the Open Files Statistic and choosing `Show Calling Methods (Hierarchical)` from the context menu of the selected "LoremIpsum.txt" file. The result will look like the figure below:

![Figure 7.22. Calling Methods (Hierarchical) for the Open File](image)

Using this presentation, it is obvious at the first glance, that the `FileCopyExample.copy()` method is the only method operating on the open file. We can expand the tree node to find out, who is calling this method in turn. The presented information is very similar to the Methods (Hierarchical) Statistic, only that the Calling Methods (Hierarchical) shows the calling hierarchy in the opposite direction, as indicated by the arrow-icons in front of the tree nodes.

7.1.4.6. The Thread Statistics

The Thread Statistics form the last major statistic group left in this walkthrough of the File I/O Analysis. These statistics present I/O operations on a per thread basis. This is especially useful if the threads of an application
have names that imply their field of work. The Thread Statistics may prove to be less valuable in the face of thread pools.

It is possible to use the Thread Statistic as an entry point into the File I/O Analysis. A link is available on the file I/O snapshot overview page as depicted in Figure 7.10, “Snapshot Overview of File I/O Analysis” and in the context menu of the file I/O snapshot as shown in Figure 7.12, “Context Menu of File I/O Snapshot”. An example of a Thread By ID Statistic for the File Copy example application is displayed in the figure below.

![Figure 7.23. Threads by ID Statistic](image)

It is no surprise that in our small example program all work is done in the main thread, so the thread statistic won’t help a lot in this simple case.

Besides the entry statistics, the Thread Statistics are also available from most other statistics of the File I/O Analysis, i.e. it is possible to create a Thread Statistic for any given entity, e.g. a method, which shows the thread(s) executing this entity. This can be useful to find threads responsible for most calls to a given method or most I/O operations on a particular file.

### 7.1.4.7. Advanced Statistics for Complex Scenarios

The example application used for this walkthrough of the File I/O Analysis is intentionally rather simple so that the shown statistics are very concise. However, real world applications tend to be much more complex in their nature leading to a huge amount of data to dig through in the File I/O Analysis. Even though the SAP JVM Profiler’s File I/O Analysis offers a wide range of filtering capabilities as introduced in the previous sections, the discussed facilities may not suffice to handle the mass of data successfully. For complex scenarios, the File I/O Analysis of the SAP JVM Profiler provides the user with advanced statistics which assist in narrowing down the available data to be comprehensible again.

Application server scenarios are among the complex applications mentioned above. For applications running in an SAP NetWeaver Application Server Java environment the SAP JVM Profiler’s File I/O Analysis provides special means for profiling. This means that it is possible to use SAP AS Java entities for filtering the information about file I/O operations. A general introduction into these kind statistics can be found in Section 8.5.1, “SAP AS Java Entity- / Thread-Filter”.

The SAP AS Java entity and thread filter enable the user for example to view a Files Statistic from a user or session perspective. Using this approach of the File I/O Analysis it is possible to identify the higher level entities like users or sessions that cause problematic amounts of file I/O. The File I/O Analysis furthermore permits to dig from these higher level entities to the lower level statistics, for example the Methods (Flat) Statistic. Of course, the other way round is available as well.

Besides being the navigation targets the SAP AS Java entity and thread filter can also be used as entry statistics either from the file I/O snapshot overview page or the file I/O snapshot context menu. The entry points from the overview page are shown in the figure below.
Another advanced statistic available in SAP JVM Profiler’s File I/O Analysis are the Component Statistics. This is based on the fact that most large applications are composed from several more or less well defined components. The Component Statistics provide means to view file I/O operations accumulated per component. A general introduction into these kinds of statistics can be found in Section 8.5.5, “Component Statistics”.

Just like the SAP AS Java entity and thread filter it is possible to use the Component Statistics as entry statistics as well as navigation targets during the File I/O Analysis. As a navigation target the Component Statistics can be opened using the corresponding context menu entries. The resulting statistic displays the components of the formerly selected entities.

The Component Statistics can also be utilized as entry points either using the file I/O snapshot overview page or the file I/O snapshot context menu. The resulting statistics show the file I/O operations on a per component level for the whole application. The corresponding entry points from the overview page are displayed in the figure below.

Figure 7.25. Opening the Component Statistics
Chapter 8. Advanced topics

8.1. Profiling Configuration View

The Profiling Configuration View, shown by default on the left upper side in the default perspective, shows the structured representation of running SAP JVM Profiling configurations. It provides you with the running state information; entry points to the collected profiling data (snapshots) and also the hierarchy of opened statistic views of the profiling analysis.

Each profiling configuration is structured in a tree, whereas the root node denotes the profiling configuration type and its name. The first child of the root node determines the profiling resource. In the online scenarios this is the connection to a profiling VM, in the offline scenarios the profiling file resource like a snapshot or .prf file. In the offline scenarios the profiling resource node provides no information, but in the online scenarios it shows the analysis state information of the running profiling. The further nodes of the configuration in the tree provides the collected profiling data. The profiling data is bundled into snapshots. A snapshot contains profiling data for a specific period of time and for associated analysis type, like allocation or performance analysis. The snapshots are used as entry points into the profiling analysis.

- Section 8.1.1, “Profiling Views Hierarchy”
- Section 8.1.2, “Profiling Configuration Toolbar”
- Section 8.1.3, “Profiling Configuration Context Menu”

8.1.1. Profiling Views Hierarchy
Within short time during the profiling analysis a lot of profiling statistic views may be created, which can confuse the user on the views desk. The profiling views hierarchy helps the user to keep an overview about the profiling statistic views. Due to the hierarchical presentation the user is able to track the creation sequence, called view path, of profiling views which are created by applying the selection or filter criteria to its parent views. Thus, during the profiling analysis several branches (view paths) are created in the view tree as shown above. Besides the hierarchical presentation, the Profiling Views Hierarchy provides also some navigation and views management possibilities like showing, closing and restoring.

8.1.1.1. Layout of a View Node

A profiling view is built up of three parts:

- The view icon, showing the statistic type or the closed state.
- The view name, describing the statistic type.
- And the view creation context, describing the creation origin of this view.

8.1.1.2. View Node Actions

The actions listed below are associated to the view nodes in the views hierarchy:

<table>
<thead>
<tr>
<th>Command</th>
<th>Name</th>
<th>Description</th>
<th>Available for</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Close View" /></td>
<td>Close View</td>
<td>Closes the profiling view. If not a leaf, the creation context is remains in order to provide the restoring possibility.</td>
<td>View Node</td>
</tr>
<tr>
<td><img src="image" alt="Close With All Subsequent Views" /></td>
<td>Close With All Subsequent Views</td>
<td>Closes the profiling view with all its child views.</td>
<td>Non-Leaf View Node</td>
</tr>
<tr>
<td><img src="image" alt="Restore View" /></td>
<td>Restore View</td>
<td>Restores the closed profiling view.</td>
<td>Closed View Nodes</td>
</tr>
<tr>
<td><img src="image" alt="Rename..." /></td>
<td>Rename...</td>
<td>Renames profiling view</td>
<td>View Nodes</td>
</tr>
</tbody>
</table>

Table 8.1. View Actions
8.1.2. Profiling Configuration Toolbar

The Profiling Configuration Toolbar provides actions helping the user to manage its running profiling configurations. Some actions are selection-specific and some are even configuration-type-specific. For instance, the action *Disconnect*, is only available in the online configurations. One of the most interesting actions is the *Start Profiling* action, which enables the user to find a running SAP JVM in order to attach the profiler and start the profiling analysis.

<table>
<thead>
<tr>
<th>Command</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Expand / Collapse</strong></td>
<td>Expands and collapses the entire content tree.</td>
</tr>
<tr>
<td><code>[0x00000002]</code></td>
<td><strong>Disconnect</strong></td>
<td>Disconnects the profiler from the selected online configuration when profiling remotely.</td>
</tr>
<tr>
<td><code>[0x00000003]</code></td>
<td><strong>Remove All Profiling Configuration</strong></td>
<td>Removes all profiling configurations (Note that only the offline and terminated online configurations can be removed).</td>
</tr>
<tr>
<td><code>[0x00000004]</code></td>
<td><strong>Remove Selected Profiling Configuration</strong></td>
<td>Removes the selected profiling configuration (Note that only the offline and terminated online configurations can be removed).</td>
</tr>
<tr>
<td><code>[0x00000005]</code></td>
<td><strong>Start Remote Profiling</strong></td>
<td>Opens the VM connection dialog helping the user to find to be profiled remote applications.</td>
</tr>
</tbody>
</table>

Table 8.2. Toolbar Actions

8.1.3. Profiling Configuration Context Menu

The *Profiling Configuration Context* menu has three types of contextual actions. The profiling navigation action, the profiling analysis action and the profiling views navigation actions. The profiling analysis actions are...
described in the corresponding analysis section. The profiling views navigation actions are described in the Profiling Views Hierarchy section. Only the profiling navigation actions are described in this section.

<table>
<thead>
<tr>
<th>Command</th>
<th>Name</th>
<th>Description</th>
<th>Available for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System Info...</td>
<td>Provides the system information of the profiled VM.</td>
<td>Configuration Nodes.</td>
</tr>
<tr>
<td></td>
<td>Save...</td>
<td>Saves the entire profiling configuration into a file.</td>
<td>Configuration Nodes.</td>
</tr>
<tr>
<td></td>
<td>Start Analysis...</td>
<td>Opens the Profiling Runtime Configuration Dialog in order to start a new profiling analysis for the selected configuration.</td>
<td>Resource Nodes of connected online configurations.</td>
</tr>
<tr>
<td></td>
<td>Stop Analysis...</td>
<td>Stops the running profiling analysis.</td>
<td>Resource Nodes of connected online configurations.</td>
</tr>
<tr>
<td></td>
<td>Create Snapshot</td>
<td>Creates a new snapshot of the selected profiling analysis. This snapshot contains collected profiling data for the time period from the moment of the last created snapshot for the same profiling analysis until the moment the new snapshot was requested.</td>
<td>Analysis Nodes.</td>
</tr>
<tr>
<td></td>
<td>Disable</td>
<td>Disables selected profiling analysis.</td>
<td>Analysis Nodes.</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>Deletes the selected snapshot. The data of the closed snapshot is going to be lost.</td>
<td>Snapshot Nodes.</td>
</tr>
<tr>
<td></td>
<td>Rename...</td>
<td>Renames the selected snapshot.</td>
<td>Snapshot Nodes.</td>
</tr>
</tbody>
</table>

Table 8.3. Context Actions

8.2. Profiling Lifecycle and Analysis Overview

This section is a detailed introduction into the Profiling Lifecycle view (see Section 8.2.2, “Profiling Lifecycle”) as well as the Analysis Overview view (see Section 8.2.3, “Analysis Overview”). The Profiling Lifecycle provides the user with a graphical representation of the resource consumption of the VM (CPU and memory), the Analysis Overview shows which kind of analysis is currently active and which snapshots were already taken.

Chapter 3, Getting Started Tutorials is a short introduction into the basic features of the SAP JVM Profiler, and Chapter 8, Advanced topics provides detailed descriptions and methods for profiling Java applications. We assume in this section, that the reader has read these chapters and is familiar with starting a profiling session, the three different kinds of analysis (Allocation Analysis, Performance Hotspot Analysis and Method Parameter Analysis) as well as with the concept of snapshots.

8.2.1. Profiling Overview

A profiling run (or profiling session) is defined by the profiling configuration that was used (i.e. the profiling parameters) and the period of time, the profiler was attached to the running VM. Each profiling run is represented as a root node in the Profiling view. Double-clicking on this node will open the Profiling overview in the Profiling Perspective, which shows general information about the current profiling run. This overview is labeled with the name of the used profiling configuration used (here: New_configuration).
The first label tells the user to which kind of VM the profiler is attached to (here: SAP Java VM) and to which machine and port (here: localhost:8000).

The second label shows the current connection state (here: connected).

When the profiler becomes inactive, either when the profiled application finishes or the user stops the profiling explicitly, the icon for disconnected appears.

During the profiling session, the user is enabled to perform some additional actions by clicking buttons in the toolbar:

- As long as the profiler is attached to the VM, the user may save the current state of the configuration in a snapshot file.
- Opens a dialog providing information about system- and VM-attributes, VM-properties and -arguments, loaded libraries etc.
- Opens the help dialog providing context sensitive help.

The System Info dialog appears by pressing the System Info... button in the toolbar:

8.2.2. Profiling Lifecycle

When connected to a running VM, the SAP JVM Profiler collects not only sampling information that is sent by the profiler backend, but also the resource consumption values. The Profiling Lifecycle is a graph showing these values:
For each second the following information is provided:

- green line: CPU utilization (in percent)
- red line: memory usage of Java heap (in percent)
- blue dots: GCs

Note that the time bar's origin denotes the start of the profiling session, i.e. the point of time when the profiler was attached to the VM.

The example above shows a SAP NetWeaver Application Server Java instance processing a HTTP request in the time period between 5:04 and 5:41. The peak in CPU utilization is about 50%. As the server runs on a dual core machine, it is likely, that the user request is handled by a single Java thread on one core. At 5:07 a GC happened which freed about 10% of unused memory. Of course, after the GC the memory consumption is increasing monotonically.

In order to get exact values at a special point of time in the diagram, the vertical slider may be moved horizontally. The time-related values (i.e. CPU consumption, memory usage and point of time) can be retrieved from the attached labels to the slider.

As profiling sessions may last for a very long period of time (maybe hours) and the lifecycle graph displays only a time window of few seconds, it's important to offer the user a convenient way to scroll the view in the time dimension. This is done in the small overview window at the bottom of the lifecycle graph, where the complete course is projected into the available horizontal space. A small rectangle within the window indicates the current section of time drawn in the lifecycle graph. To adapt the displayed section of time, this rectangle may be moved horizontally updating the lifecycle graph accordingly.

Moreover the lifecycle view's toolbar provides tools to directly influence the scale of the time bar:

- Zooms in the lifecycle window
- Zooms out the lifecycle window
- Switches back to the last view

To sum up, the lifecycle window helps the user to concentrate on relevant period of time where the VM shows conspicuous behavior and to cut off sections where a further investigation isn't promising (in our example 4:45 up to 5:00).

As snapshots form the basic units of information on which analysis may be performed on, it's crucial to allow the user to choose arbitrary periods of time within the profiling session and create a new snapshot for them. This can be achieved by dragging the time period with the mouse in the lifecycle view:
When selecting this way, the snapshot button becomes active. A new snapshot covering the current selection will be created by pressing this button. It is added to the set of taken snapshots in the Analysis Overview (see Section 8.2.3, “Analysis Overview”).

8.2.3. Analysis Overview

The Analysis Overview below the Profiling Lifecycle view tells the user

1. which kind of analysis trace is currently active (Allocation Analysis, Performance Hotspot Analysis or Method Parameter Analysis)

2. which snapshots have been already taken

The table on the left side of the Analysis Overview is divided in up to three sections according to the different kinds of analysis traces: Allocation Analysis (blue), Performance Hotspot Analysis (red) and Method Parameter Analysis (yellow). Each section title provides further information about the currently active trace. A row below corresponds to a taken snapshot. Note that the number of snapshots is not limited.

In the example shown above, we see an allocation analysis snapshot named Allocation Statistic_1 and a performance analysis snapshot named Performance Hotspot Statistic_1. The first covers 10,641 object creations with a memory consumption of 523kB, the latter includes 1,807 samples within a sampling duration of 19s.

On the right side of the Analysis Overview colored timeline bars are drawn that reflect the course of analysis traces. As long as the profiling session is active (i.e. as long as the profiler is attached to a running VM), there is either none or exactly one analysis trace active at one time. The currently running analysis trace collects the profiling data sent by the profiler backend into the current snapshot. A snapshot is closed,

• when the profiler becomes disconnected (either explicitly by the user or on VM shutdown)

• when the user decides to stop the current analysis

• when the user explicitly requests a snapshot but continues the current analysis
Advanced topics

- when the user decides to start a new analysis trace. A new snapshot is then started (in general profiling starts with an active analysis trace).

Consequently, when an analysis trace is active, exactly one section title's bar is growing in time. During this time, the collected data is put into a new (invisible) snapshot. In the example above, currently the Allocation Analysis is in progress, whereas two snapshots were already taken.

Several actions can be triggered by pressing one of Analysis Overview's toolbar buttons:

<table>
<thead>
<tr>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lets the user choose the type of a new analysis trace to be started</td>
</tr>
<tr>
<td>Stops the current analysis trace and closes the current snapshot. This snapshot will be displayed in the overview table.</td>
</tr>
<tr>
<td>Closes the current snapshot (it will be displayed in the snapshot table). The analysis trace continues with a new snapshot.</td>
</tr>
<tr>
<td>Adjusts the table columns' widths in a way that all labels are visible</td>
</tr>
<tr>
<td>Minimizes the overall width of the table in order to optimize the display of the analysis trace bars</td>
</tr>
</tbody>
</table>

After having stopped the Allocation Analysis trace in the example, the Analysis Overview looks like follows:

Currently no analysis trace is turned on and three snapshots have been taken and are ready for further analysis. A double-click on the snapshot bar will open its corresponding statistic, where several kinds of statistics are available for further analysis.

Also note that the collection of taken snapshots directly refer to the tree in the Profiling view:

8.3. Comparison of Profiling Statistics

This section describes in detail how profiling statistics can be compared with each other. This feature can be especially useful in software performance tuning or analyzing of impacts on changed software parts.

- Section 8.3.1, “Comparable Data”
• Section 8.3.2, “A Simple Example”

### 8.3.1. Comparable Data

Before any profiling statistics can be compared the comparable data has to be determined. There is some restriction to the profiling statistics:

• Only flat statistics can be compared. That means that only tables can be compared; trees are excluded.

• The tables must have the same column types. Essentially, that means that only statistics of the same type can be compared, while it is not possible to compare, for example, an allocated objects statistic with a method statistic (see below).

<table>
<thead>
<tr>
<th>Method</th>
<th>Smallest Bytes</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[]</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>int[]</td>
<td>80</td>
<td>120</td>
</tr>
<tr>
<td>char[]</td>
<td>7.37 kB</td>
<td>320</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Bytes</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>7.37 kB</td>
<td>223</td>
</tr>
<tr>
<td>String</td>
<td>2.18 kB</td>
<td>177</td>
</tr>
<tr>
<td>Class</td>
<td>3.66 kB</td>
<td>39</td>
</tr>
<tr>
<td>byte[]</td>
<td>1.55 kB</td>
<td>4</td>
</tr>
<tr>
<td>int[]</td>
<td>1.02 kB</td>
<td>1</td>
</tr>
</tbody>
</table>

• Section 8.3.2, “A Simple Example”

### 8.3.2. A Simple Example

Let’s have a look at the allocation analysis example. We have just a simple main method allocating some bytes. Now we profile this method with the allocation analysis and requests the class and method statistic (see below).

```java
public static void main(String[] args) {
    // allocate some bytes
    byte[] byteArray = new byte[1024*2];
    int[] intArray = new int[256*2];
}
```

Afterward we alter some allocated values in our main method and perform the same profiling run second time (see below).

```java
public static void main(String[] args) {
    // allocate some bytes
    byte[] byteArray = new byte[1024*4];
    int[] intArray = new int[256];
}
```
Now we have two profiling runs with supposedly different statistic results. So we can use the profiling comparison features to find the differences in object and method statistic of both profiling runs. Before we do this we have to understand the meaning of the comparison results. The comparison results are dependent on the statistic role. That means if both statistics are comparable they are divided into the old and a new statistic. If a value to be compared is greater in a new statistic then in the old, a growth is notified in the comparison statistic and vice versa. For equal values, 0 is provided.

The definition of new and old is not necessarily related to the execution time, it’s just the decision of the user. The selected statistic which the user intends to compare with any other statistic is marked as “new”, the other one is marked as “old”. Typically, the user would profile an application, look at some statistics and spot a problem. Then, after fixing the problem in the code, he or she will start another profiling run to see, if the problem is indeed fixed by the modification. For that, the user will select the statistic of the second run (which will then be marked as “new”) and compare that with the statistic of the first run (which will be marked as “old”).

Note, that it might be a good idea to rename the statistics to something meaningful like “before optimization” and “after optimization” before comparing. This way, it will be much easier to understand what was compared later on.

In our example we want to see the comparison between the both method statistics and we choose the statistic of the second profiling run as new. So we select this statistic and perform the “compare with…” action at the context menu (see below).

A new dialog shows all possible proposals to be compared with the selected statistic. In our case it’s only the method statistic of the first profiling run. Selecting it and pressing “OK” button performs the comparison and opens a new statistic with the comparison results.

If you can select the new comparison statistic in the configuration tree, the two compared statistics are both highlighted, showing also their roles (see below).

Now let’s see the comparison statistic. There is a growth of allocations in our main method, as expected.
But as you can see, there are a lot of statistical values where there was no difference at all, displayed as zero values. Actually we are not interested to see them, so we can use a convenience action in the toolbar, hiding them all and just showing the differences (see below).

Going back to our example we can inspect what exactly differs in our main method. Just comparing the allocated objects statistics of the both profiling runs, we can see that the second run allocates byte-arrays of greater size, but smaller integer arrays.

We can also find, that the total number of allocated arrays stays the same.

### 8.4. Profiling Properties Dialog

The Profiling Properties Dialog displays the properties of the profiling statistic items. This could be the class, method or class loader. The information about the statistic items is structured in a similar manner and provides only essential info of an item (see below).
The content of the properties dialog is structured dynamically. There are links providing users the navigation to other items. So it’s up to the user to navigate through the content, for instance the properties of a method provides links to the properties content of its parameter types, return types and parent type. The types are only shown in their simple form (without package definition) but the tooltips contains the full qualified type specification.

In order to simplify the navigation, the properties dialog provides the navigation history (see below).

8.5. Advanced Profiling Techniques

This section shows some advanced profiling techniques to track down memory and performance problems.

8.5.1. SAP AS Java Entity- / Thread-Filter

When running in an SAP NetWeaver Application Server Java environment, the SAP JVM Profiler gets additional information about the following entities from the application server.
• The user name.

• An identifier of the user session.

• An identifier of the user request.

• The name of the application being executed.

This information is available for each individual Java thread. So, the SAP JVM Profiler knows at any point in time which user, session, request and application is currently running or being executed within a specific thread. We can use this kind of information for filtering purpose. A user, session, request or application filter can be supplied when enabling an analysis.

Applying a filter for these SAP AS Java entities has two advantages:

• Information needs only be collected for the entity in question. Thus, the performance impact of the profiling session is less than if information for all users, session, requests and applications needs to be collected.

• The information collected only relates to the entity in question. For instance, when profiling a user request, the data collected only relates to this particular user request, not to any other requests and not to any background noise in the SAP NetWeaver Application Server Java.

Figure 8.1. Enabling an Analysis - User Filter

Note that it is also possible to apply filters for several entities. For instance, you can specify an application filter in addition to a user filter. Then, you collect only profiling data for the specified user and application. All the other applications and users running in the corresponding SAP NetWeaver Application Server Java system are not affected.

Applying a filter when enabling a trace has the advantage that only the necessary information is collected and other users, session, requests and applications are not affected. So, the impact of the SAP JVM Profiler with regards to performance and memory overhead is limited. In addition to this upfront filtering (i.e., when the trace gets started), the front end itself is also able to filter as well. This allows for analyzing profiling data only for
a particular user, session, request or application. For instance, if you have performed an allocation analysis, you are able to account the allocated objects to users, sessions, requests and applications. The same holds for the performance hotspot analysis. Here you can determine which application server entity is responsible for executing certain methods.

Basically, you can navigate from every statistic to the corresponding user, session, request and application statistics. Furthermore, you are able to start your snapshot analysis from one of these statistics and then dig into the details. For example, in case of an allocation analysis, you get the information how many objects were allocated in the application statistic. Then by selecting certain applications, you can check what kind of objects are allocated, which methods are responsible for these allocations, etc.

Figure 8.2. Application Filter on the Front End

Similar to the entity statistics, the thread statistics allow for filtering particular threads. Note this functionality is only available within the front end. It is not possible to filter threads upfront when a trace gets started. There are basically two kinds of thread statistics: thread name and thread id statistics. Java threads have a name and an identifier as main attributes. The identifier is a unique value over all created threads. In contrast, the name of a Java thread is not necessarily stable over time and can change during the lifetime of a thread. It is also possible that several threads share the same name. Therefore, two thread statistics are provided. The thread id statistic lists all threads according to their unique identifiers, whereas the thread name statistic lists all thread names. So, one entry within the thread name statistic may correspond to several entries within the thread id statistic.

8.5.2. Exceptions

Java exceptions should indicate exceptional situations. Creating an exception is not a cheap operation and often subsequent tasks like logging are triggered by a thrown exception. However, profiling a sales order in an SAP Business by Design system showed that a lot of exceptions are thrown.

The Allocation Analysis can be used to check how many exceptions are thrown in a scenario. After the scenario has been profiled with the Allocation Analysis enabled, the Allocated Objects Statistic can be opened as seen in Figure 8.3, "Allocated Objects Statistic".
Advanced topics

Figure 8.3. Allocated Objects Statistic

The Allocated Objects Statistic shows all objects that have been created during the profiling run, grouped by their corresponding classes. This statistic can be filtered by clicking on the class filter as highlighted in Figure 8.3, “Allocated Objects Statistic”.

Figure 8.4. Class filter dialog

The class filter dialog allows for filtering the statistic. In our example, we are interested in exceptions. There is an already predefined filter for exceptions, which can be selected (if a filter is not predefined or does not suit your needs, you can create your own filters as well).

The result of applying a class filter to an Allocated Objects Statistic is a Filtered Allocated Objects Statistic as seen in Figure 8.5, “Filtered Allocated Objects Statistic”. This statistic only contains the allocated exceptions. You can see that during the sales order a lot of exceptions have been thrown (around 750 exceptions: 312
XMLConfigurationExceptions, 113 ClassCastExceptions ...). As mentioned earlier, throwing exceptions has its own costs and often subsequent tasks follow when an exception is thrown.

Figure 8.5. Filtered Allocated Objects Statistic

The next step is to find out where these exceptions have been thrown. This can be done by selecting Show Methods (Flat) from the context menu of an exception. This statistic shows all methods which have allocated, i.e., thrown, an exception. Furthermore, it can be seen how many exceptions have been allocated by a particular method. Figure 8.6, “Methods (Flat) View” shows the Methods (Flat) Statistic for the XMLConfigurationException. Sorting this statistic by Self objects shows the methods which have thrown the most XMLConfigurationException. In our example, the method ParserConfigurationSettings.checkFeature throws 312 exceptions. By double clicking the method, we can jump directly into the source code to check why so many exceptions are thrown.

Figure 8.6. Methods (Flat) View

Another interesting information is who called the method. This information can be obtained by choosing Show Calling Methods (Hierarchical) from the context menu of the method. The calling method tree consisting of the methods which called ParserConfigurationSettings.checkFeature, is shown in Figure 8.7, “Methods (Hierarchical) View”.

Figure 8.7. Methods (Hierarchical) View
With the Allocation Analysis, it is possible to find out how many exceptions have been thrown and who has thrown them. This information is valuable as throwing exceptions is not a cheap operation and throwing unnecessary exceptions should be avoided.

8.5.3. Finalizer

Java objects of classes that override the Object.finalize() method can put pressure on the garbage collector. If an unreachable object has a finalize() method, arrangements are made for the object's finalizer to be called. The following steps describe the lifetime of a finalizable object `obj`, which is an object whose class has a non-trivial finalizer:

- When `obj` is allocated, the VM internally records that `obj` is finalizable (this typically slows down the otherwise fast allocation path that modern VMs have).

- When the garbage collector determines that `obj` is unreachable, it notices that `obj` is finalizable (as it had been recorded upon allocation) and adds it to the VM's finalization queue. It also ensures that all objects reachable from `obj` are retained, even if they are otherwise unreachable, as they might be accessed by the finalizer.

- At some point later, the VM's finalizer thread will dequeue `obj`, call its finalize() method, and record that `obj`'s finalizer has been called. At this point, `obj` is considered to be finalized.

- When the garbage collector rediscovers that `obj` is unreachable, it will reclaim its space along with everything reachable from it (provided that the latter is otherwise unreachable).

Notice that the garbage collector needs a minimum of two cycles (maybe more) to reclaim `obj` and needs to retain all other objects reachable from `obj` during this process. If a programmer is not careful, this can create temporary, subtle, and unpredictable resource retention issues. Additionally, the VM does not guarantee that it will call the finalizers of all the finalizable objects that have been allocated; it might exit before the garbage collector discovers some of them to be unreachable.

The current view on finalization is that it should be avoided when possible. But, you may have an application (or use a library) that uses finalization. So, you may want to analyze your application and see which objects, if any, are queued for finalization.

The Allocation Analysis can be used to check how many objects with a non-trivial finalizer are created and where they are created in a scenario. After the scenario has been profiled with the Allocation Analysis enabled, the Allocated Objects Statistic can be opened as seen in Figure 8.8, “Allocated Objects Statistic”.

Figure 8.8. Allocated Objects Statistic

The Allocated Objects Statistic shows all objects which have been created during the profiling run, grouped by their corresponding classes. This statistic can be filtered by clicking on the class filter as highlighted in Figure 8.8, “Allocated Objects Statistic”.
Figure 8.9. Class Filter Dialog

The class filter dialog allows for filtering the statistic. In our example, we’re interested in objects with non-trivial finalizers. There is an already predefined filter for objects with non-trivial finalizers, which can be selected (if a filter is not predefined or does not suit your needs, own filters can be created as well).

The result of applying a class filter to an Allocated Objects Statistic is a Filtered Allocated Objects Statistic as seen in Figure 8.10, “Filtered Allocated Objects Statistic”. This statistic only contains the allocated objects with non-trivial finalizers.

Figure 8.10. Filtered Allocated Objects Statistic

Starting from this statistic, you can see who has allocated these objects and from where these methods have been called. To get this information, the same steps as mentioned in chapter Section 8.5.2, “Exceptions” can be applied.

With the Allocation Analysis, it’s possible to find out how many objects with non-trivial finalizers have been created and where they have been created. This information is valuable as creating objects with non-trivial finalizers means additional work for the VM and unnecessary finalization should be avoided.

8.5.4. Top of Stack Finder

Often a majority of the allocations are performed in JDK classes, like java.lang.StringBuilder, java.lang.String, etc. However, these classes are highly optimized and usually not the root cause for performance or memory
problems. Often the JDK classes are only used not optimally or even in a wrong manner. Figure 8.11, “Methods (Flat) Statistic” shows the Methods (Flat) Statistic of a framework layer Java EE Engine startup. As mentioned earlier, the top allocating methods are part of the JDK.

![Figure 8.11. Methods (Flat) Statistic](image)

The SAP JVM Profiler provides a "top of stack finder", with which the top of stack definition can be changed, e.g. the top of stack finder allows to not account the JDK classes for the allocations done. The allocations done by the JDK classes are accounted to the methods calling the JDK classes.

For instance, the code extract below creates 10000 strings in the method foo:

```java
public class TopOfStackFinder {
    private void foo() {
        for (int i = 0; i < 10000; i++) {
            new String("" + i);
        }
    }

    public static void main(String[] args) {
        new TopOfStackFinder().foo();
    }
}
```

This results in that java.lang.AbstractStringBuilder.<init> is the method with the highest memory as a character array is allocated there.

However, java.lang.AbstractStringBuilder.<init> is called by foo and foo is responsible that 468 kB of 470kB are allocated in java.lang.AbstractStringBuilder.<init>.

The top of stack finder allows for accounting foo for the 468 kB directly instead of accounting it to java.lang.AbstractStringBuilder.<init>. Technically, the top of stack finder specifies a filter (e.g. all JDK classes). Then the callstack is traversed from top to bottom and the filter is applied to every method.
The starting point is java.lang.AbstractStringBuilder.<init>. As this is a JDK class, no accounting is done. The next frame is java.lang.StringBuilder.<init>. This is a JDK class as well, so no accounting is performed. Then TopOfstackFinder.foo is found. This is the first class which is not a JDK class, so it’s not ignored and accounted for the allocations that it performed itself, plus the allocations of java.lang.StringBuilder.<init> and the allocations of java.lang.AbstractStringBuilder.<init>.

Having the top of stack finder applied leads to a new Methods (Flat) Statistic where no JDK classes show up and all allocations done by JDK classes are accounted to the corresponding calling methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Self Bytes</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopOfStackFinder.foo&lt;void&gt;</td>
<td>1.37 MB</td>
<td>1.37 MB</td>
</tr>
<tr>
<td>frowned</td>
<td>49.7 KB</td>
<td>49.7 KB</td>
</tr>
<tr>
<td>&lt;init&gt;</td>
<td>20.4 KB</td>
<td>20.4 KB</td>
</tr>
<tr>
<td>TopOfStackFinder-main&lt;java.lang.String&lt;void&gt;</td>
<td>8</td>
<td>1.37 MB</td>
</tr>
</tbody>
</table>

The starting point for the top of stack finder is the Methods (Flat) Statistic as shown in Figure 8.11, “Methods (Flat) Statistic”. The top of stack finder is applied by clicking the top of stack finder symbol as highlighted in Figure 8.11, “Methods (Flat) Statistic”. A filter dialog is opened. Here a filter can be entered. All frames which match the filter are not accounted when the callstack is traversed. Figure 8.12, “Top of Stack Finder Filter Dialog” shows the filter dialog with a filter which won’t account JDK classes.

Once this filter is applied, we get a Methods (Flat) Statistic as shown in Figure 8.13, “Methods (Flat) Statistic”. This Methods (Flat) Statistic does not include JDK classes any more. All the allocations done by JDK classes are accounted to the methods calling the JDK classes. What comes to a surprise is that the getWarName method is responsible for allocating 189 MB. 189 MB are 4.91% of all allocations of a framework layer Java EE Engine startup. Before applying the top of stack filter, getWarName just ranged in the middle field of the allocations and it was easy to overlook it there (although it’s a heavy consumer and responsible for 4.91% of allocations).
As this looks suspicious, we'd like to know what this method is doing. When selecting "Show called method tree" from the context menu of getWarName, the methods which are called by getWarName are shown (as seen in Figure 8.14, "Method Tree").

Figure 8.13. Methods (Flat) Statistic

Figure 8.14. Method Tree
The Methods (Flat) Statistic, shows the 160 MB are allocated in line 547, which is

```
return null;
```

When jumping into the source code of `Properties.propertyNames()` by double clicking on the method, you can see that every time when `propertyNames` is called, the hash table that holds the properties is copied and this newly created hashtable is returned to the caller, thus calling `propertyNames` involves a lot of memory allocations.

The top of stack finder allows for making the caller responsible for allocations made by certain classes (e.g. JDK classes) and find places where these classes are used in a not optimal manner.

### 8.5.5. Component Statistics

A component statistic is a variant of a method statistic or a method tree, but not showing the individual methods. Instead so called components are shown, which can be defined by the user. Consider the following stack trace (the actual method names are simplified):

```
jndi.split(...)
jndi.parseEntry(...)
String.substring(int)
String.substring(int, int)
jndi.lookup(...)
servlet.get_security_proivder()
servlet.get_permission()
webcontainer.call_servlet()
```

**Figure 8.15. Example stack trace**

We see the web container called a servlet, which does a kind of permission check. This check needs some information from a JNDI provider and the provider itself does some string handling. When we look at a method statistic we would see all the methods found on the stack, with the top of stack method getting the self value. The component statistic view try to give a more high level view, by grouping method or call sequences into a component. Below we show two possible componentizations of the callstack. As you can see, more than one way to do a sensible componentization is possible.
A component statistic can now be created by using the “stack” of defined components instead of the original method stack.

Let’s see, how a component will be defined in the profiler. As we have seen, a component is a sequence of method calls. In the profiler a component is defined by the entry method into the component. In both examples the `jndi.lookup()` was such an entry method. Whenever the `jndi.lookup()` method was called, the part of the stack starting at that method is considered as the JNDI component. The next question is, when do we exit from a component. The answer is simple: We exit a component whenever we enter another component. This of course means, that the size of a component is defined by the other component definitions too.

To create a component statistic select the menu entry 'Show Components (Flat)' from the context menu of a statistic or snapshot:

Figure 8.16. Two componentization possibilities
Next we enter a dialog where we define the entry methods of the components. Since the number of entry methods can be quiet large or hard to detect, a method filter is used to specify the entry methods. If we for example would like to have a component for String handling, a suitable filter would be \texttt{java.lang.String*}, which would include all the methods from \texttt{String}, \texttt{StringBuilder} and \texttt{StringBuffer}. Or if we would like to have a component for servlets, the filter for the entry method would be \texttt{overwrites javax.servlet.http.HttpServlet\_doGet(*) || overwrites javax.servlet.http.HttpServlet\_doPost(*).} Let's look at an example of a flat component statistic (using the result of allocation profiling). In addition to the two filters mentioned before, we have added an XML filter \( \texttt{(javax.xml.*)} \) and are inspecting a click in a web application. We choose the “Show Components (Flat)” entry from the context menu of the respective snapshot and a dialog opens, where we can selected the components we are interested in.

**Figure 8.17. Selecting the flat component statistic**
Figure 8.18. Selecting and creating components

In this dialog we create the three components by specifying the filter we described above. After selecting the three components, the flat component statistic is created.

<table>
<thead>
<tr>
<th>Component</th>
<th>Self Bytes</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servlet</td>
<td>60.3 MB</td>
<td>111 MB</td>
</tr>
<tr>
<td>String Handling</td>
<td>46.1 MB</td>
<td>46.1 MB</td>
</tr>
<tr>
<td>XML</td>
<td>13.0 MB</td>
<td>17.4 MB</td>
</tr>
<tr>
<td>&lt;none&gt;</td>
<td>14.4 MB</td>
<td>14.4 MB</td>
</tr>
</tbody>
</table>

Figure 8.19. The flat component statistic

We see the three defined components plus the “<none>” component. The latter is used for all allocation in which no entry method of the three defined components was on the stack when the allocation occurred. The meaning of the terms “self” and “total” are analog to a normal method statistic. Analog to a method tree, a hierarchical version of the component statistic is available. In the example it looks like this:
Figure 8.20. The hierarchical component statistic
Chapter 9. Tips & Tricks

This section shows some neat tips and tricks which make life easier.

9.1. Smart Expanding of Trees

Trees can be expanded in a smart manner when clicking the expand while holding the shift key pressed. The smart expanding is characterized by

- Methods that do nothing but delegate to other methods are collapsed and show up only as skipped frames.
- Methods that do not participate heavily in the overall consumption are condensed and show up only as the sum of the individual methods.

![Figure 9.1. Expand Tree in a Smart Way](image)

9.2. Displaying Source Code

The source code can be displayed by double clicking on methods in a statistic. If a method is double clicked, the beginning of the method is shown. When a line number is double clicked, the source code around the line number is shown directly.
9.3. Opening Profiling Files Using Drag and Drop

Profiling files can be opened by dropping them into the profiling view.
Figure 9.3. Drag and Drop Profiling Files

9.4. Exporting Statistics Using Drag and Drop

Statistics can be exported as HTML documents by selecting the interesting lines and dropping them to a folder or mail client.
9.5. VM Monitor View

The SAP JVM Profiler includes a small VM monitor which displays basic information about the VMs running. Furthermore, a VM can be switched into profiling mode, that means it opens a debugging port which can be used for profiling connections or a VM can be exited. By default, the SAP Java VMs running on the local machine are displayed within the VM Monitor View. The VM Monitor View is not limited to display only local VMs. It is possible to display VMs running on remote hosts as well. As a prerequisite for monitoring VMs running on a remote host, the jvmmond tool, which is located in the sapjvm_5/bin directory, needs to run on the remote host to be monitored. Once the jvmmond tool is started on the remote host, the remote host can be added to the VM Monitor View by entering the hostname into the host field. If no port was specified explicitly when starting jvmmond, the preconfigured default port should be used to connect to the jvmmond daemon running on the remote host. Otherwise the port has to be set explicitly by entering hostname:port into the hostname field (e.g. wdfn00172752a:1099).

The VM Monitor View can be opened by the shortcut Ctrl-Shift-M.

Figure 9.5. VM Monitor View
Chapter 10. Reference

10.1. Internal Objects

In the next few sections we will describe all the internal objects used by the SAP Java VM. Note that some of the explanations require some internal knowledge of the class file to fully understand them.

10.1.1. The \{instance class\} Object

The \{instance class\} object stores the basic data for each non-array class loaded by the VM. It contains the static fields and descriptions of the instance fields of the corresponding class, references to the \{method\} objects and so on. The \{instance class\} is the starting point for getting every possible information about a non-array class. So the \{instance class\} corresponds to the java.lang.Class object in the Java world and in fact a Class object contains an invisible reference to its corresponding \{instance class\}.

10.1.2. The \{type array class\} Object

The \{type array class\} object stores the basic data for each one-dimensional primitive array (multi-dimensional arrays of a primitive type are represented by \{object array class\} objects). Since arrays have no methods defined by themselves, the \{type array class\} objects do not contain any references to other objects in the \textit{perm generation}. The java.lang.Class object of a one-dimensional primitive array contains an invisible reference to its \{type array class\} object.

10.1.3. The \{object array class\} Object

The \{object array class\} object stores the basic data for each non-primitive array and multi-dimensional arrays of a primitive type. Since arrays have no methods defined by themselves, the \{object array class\} objects do not contain any references to other objects in the \textit{perm generation} besides a reference to their element classes. The java.lang.Class object of a non-primitive array or a multidimensional primitive array contains an invisible reference to its \{object array class\} object.

10.1.4. The \{constant pool\} Object

The \{constant pool\} is an object that represents the \textit{constant pool} of a class. The \textit{constant pool} of a class is where every constant like string literals, integer literals, method names and signatures, class names, field names and types are stored. The \textit{constant pool} is needed since the Java bytecode does not store these constants directly\footnote{There are few exceptions for small integers and some floating point constants.}. If for example a method should be invoked, the bytecode contains just the index in the \textit{constant pool} where the name and signature and class of the method are stored. This keeps the bytecode small, since if one constant is referenced more than once in a class it is stored only once in the \textit{constant pool}.

If you are interested in seeing the contents of the \textit{constant pool} of a specific class, you can use the \texttt{javap} command of the SAP Java VM. Figure 10.1, “Example of a constant pool” shows an excerpt (slightly edited to fit onto the page) of the \textit{constant pool} of java.lang.String:
Figure 10.1. Example of a constant pool

As you can see, the constant pool, beside holding direct constants has a lot of entries that combine other constant pool entries. A method entry for example consists of two entries, each designating another entry in the constant pool. The first entry is the index in the constant pool where the class of the method is stored. The second entry is the index where the name and signature of the method are stored. Both referenced entries themselves do not contain the data directly. The entry with the class just holds the index of the entry where the name is stored. The entry with the name and signature contains two indices where the name and signature of the method are stored. Each instance class has exactly one associated constant pool object, while the type array class and object array class objects have no constant pool associated with them, since they do not have their own methods or fields.

Note that the constant pool does not contain strings (class names, field names and so on) itself, but a reference to the symbol objects which contain the string. This means that the size of the constant pool is not representative of the amount of data associated with it. But since the symbol objects referenced by the constant pool are potentially shared between many constant pool objects, there is no definite way to attribute a meaningful size to the whole constant pool.


10.1.5. The constant pool cache Object

The constant pool cache objects are strongly associated with the constant pool objects. Recall from the last section, that the constant pool, among other entries, contains entries which refer to a field or method. These entries contain just the names of the fields and methods. When the VM first uses such an entry, the entry is said to be resolved. This means that it is replaced by a reference to the method or field itself. In the SAP Java VM we don't overwrite the original constant pool entries of these types during resolving. Instead for each field or method entry, we have an entry in the constant pool cache which holds the resolved result.
Since the {constant pool cache} is only needed when at least one object of the associated class is instantiated, {constant pool cache} objects are created lazily.

10.1.6. The {method} Object

The {method} objects store basic information about each method of a class, like access flags or the maximum number of local variables. They don’t contain the bytecode of the method, which is stored in {const method} object associated with each {method} object. This makes {method} objects relatively small.

10.1.7. The {const method} Object

The {const method} objects store the most important properties of a method. Most notably they contain the bytecode of the method, the exception table\(^2\), the line number table\(^3\), the local variable table\(^4\) and the stack maps\(^5\). This makes the {const method} objects together with the {constant pool} objects the largest non-array objects in the perm generation.

10.1.8. The {method data} Object

The {method data} object is associated with a {method} object and contains profiling information gathered during execution of the method. This profiling information is later used by the JIT compiler (more specifically the server compiler) to aid certain optimizations (e.g. the most probable class to encounter when invoking a virtual method or which cases clause in a switch statement is most likely to be taken). Not every {method} object has an associated {method data}, for example because the method was never executed or doesn’t contain constructs which should be profiled.

10.1.9. The {symbol} Object

The {symbol} object contains an UTF-8 string. {symbol} objects are mostly used by the constant pool objects to store strings. Note that {symbol} objects are unique, meaning that there are no two {symbol} objects which represent the same string. In this regard they are similar to an interned string, but instead of being a real java.lang.String object they are more like a C string. Since {symbol} objects are shared between other objects, it’s hard to attribute them to specific classes.

10.1.10. The {array class}

Apart from the arrays represented by {object array class} and {type array class} objects, the SAP Java VM uses a few other types of arrays internally. These are represented by {array class} objects.

---

\(^2\) The exception table describes which exception handler (read catch blocks) are active during which sections of the bytecode. You can get the exception table of a method via javap command.

\(^3\) The line number table has a mapping of sections of the bytecode to a line number of the Java method. It’s generated when you compile your Java classes with debug information (via the \(-g\) flag of javac). Once again you can inspect the line number table with the javap command.

\(^4\) The local variable table contains a mapping between the index of a local variable used in a section of the bytecode and its name and type. Like the line number table it is only generated when the Java class was compiled with debug information. And again you can inspect the local variable table with the javap command.

\(^5\) A stack map contains information about which local variables and slots of the execution stack of the method contain objects. This is needed by the GC to keep these objects alive. There has to be a stack map for each position in the methods bytecode at which a GC can happen. Note that the stack maps are not stored directly in the {const method} object, but in byte and short arrays referenced by the {const method} object.
10.1.11. The \{other class\} Object

In the SAP Java VM every object has a class. The class of Java objects are the \{instance class\}, \{object array class\} and \{type array class\} objects (from the perspective of VM that is; in terms of Java the class are of course the \texttt{java.lang.Class} objects). The internal objects have a class too (which are objects again), which are subsumed under the \{other class\} objects. Since the number of internal classes is fairly small, there are only a small number of \{other class\} objects.

10.1.12. The \{compiledICHolder\} Object

The \{compiledICHolder\} objects are used by the JIT compiler to switch from calling an inlined method to calling the interpreter to execute that method. These objects are very small and only used sparingly.

10.2. The Adaptive Allocation Analysis

10.2.1. The Problem

The Allocation Analysis by default reports every allocation performed by the VM. This has several consequences. The first is that the profiled VM will be slowed down drastically. This is mostly caused by getting the stack trace for each allocation. The time spent is proportional to the depth of the stack trace, which tends to be in the order of 100 to 200 in the NetWeaver Application Server Java. And since a Java Application can allocate literally millions of objects per second, even the few microseconds we need to get and send the stack traces lead to a severe slowdown.

The second problem is the amount of memory used in the profiler itself. Obviously it has to store each stack trace somewhere in memory. This means that the number of different stack traces dominates the memory used by the profiler. Of course the profiler uses some tricks to keep the needed memory low. For example, when storing a stack trace it looks if it has already stored a stack trace which matches the lower part of the new stack trace and then stores only the frames different to the old stack trace. But even with these tricks a stack trace typically consumes 50 to 100 bytes per stack trace, so if we have a few million different stack traces we would need a few 100 MB in the profiler.

10.2.2. The Solution

The solution to both of the problems outlined above is to avoid reporting every allocation. Imagine we would only report every 10th allocation. This would lead to a drastic reduction of the runtime overhead in the VM, since we would only have to perform 10 percent of the expensive operations. The impact on the memory consumption in the profiler is more subtle. Typically a lot of the objects are allocated by a few (relatively spoken) stack traces, so if we don't record every allocation we will see a lot less different stack traces.

But there is a problem with the scheme outlined above. Imagine we do some allocations of small objects and only one allocation of a very large object. The probability that the large allocation is ignored is very high, so the results would be useless. To avoid this we don't simply record only every n'th allocation, but instead we record an allocation if a specific amount of bytes has been allocated since the last reported allocation. So big allocations are never omitted.

Another important optimization to make is the amount of bytes after which we report the allocation dynamic. If the profiling is started, every allocation is reported. After a specific amount of overall bytes were allocated, we report only allocations after 16 bytes were allocated. After another amount of overall allocated bytes is allocated we report allocations only after 32 bytes were allocated and so on. This scheme has the advantage that the error we get is a constant relative to the overall number of allocated bytes. Currently the adaptive Allocation...
Analysis maintains an error of 0.01 percent of the overall allocated bytes. For example, if the profiling run has traced 1 GB of memory, the error is 100 kB. This means that the adaptive Allocation Analysis should be used if you don't have to know the exact number of allocated bytes and objects and you want to profile more than a few 10 seconds.

To enable the adaptive Allocation Analysis you just have to set the Adaptive indicator in the Allocation Analysis options page:

10.3. Class Filters

A class filter is used to group classes into two sets: The ones which match the filter and the ones which do not. Depending on the context in which the filters are used you might remove the non-matching classes from the current view or the filters are used to categorize classes. If you want to can directly jump to Section 10.3.7, “Examples”.

10.3.1. Basic Filter Syntax

In its simplest form a class filter just is the name of a class, so

java.lang.String

is a valid class filter. Note that you must specify the complete class name including the package (for example String would only match a String class in the default package.

In addition you can use * and ? in the class names to match more than one class. * matches zero or more arbitrary characters while ? matches exactly one arbitrary character. For example

java.lang.*

would match all classes in the java.lang package and its sub-packages.

10.3.2. Combining Filters

You can combine filters using && and ||. In the first case the combined filter only matches if both the filter in front and after the && match. On the other hand for || only one (or both) of the filter must match. Note that you can alternatively use & or | instead, which have the same meaning. Additionally you can prepend a filter with !, so the filter only matches if the filter prepended with the ! does not match.

You can use parenthesis ( and ) to group filters, so the &&, || and !, operators are applied to the grouped filters. For example

!(java.* || javax.*)

would match every class not in the java and javax packages (and their sub-packages).

10.3.3. Matching by Type

Instead of just matching by the name of the class, you can match classes by their type too. You can to this with the instanceof, implements or extends keywords. instanceof <filter> matches all classes which are, extend or implement a class which matches <filter>. For example
\texttt{instanceof (java.lang.RuntimeException || java.lang.Error)}

matches \texttt{java.lang.RuntimeException} and \texttt{java.lang.Error} and all classes which extend them (or in other words, it matches all unchecked exceptions). \texttt{instanceof} can be used to check implemented interfaces too.

\texttt{extends <filter>} matches all classes which extend a class which matches \texttt{<filter>}. For example:

\texttt{extends java.lang.RuntimeException}

would match all classes which have \texttt{RuntimeException} as a super class, but not \texttt{RuntimeException} itself.

Analog \texttt{implements <filter>} matches only classes that implement (directly or indirectly) an interface which matches \texttt{<filter>}. An example is:

\texttt{implements java.io.Serializable}

which matches only classes which implement the \texttt{Serializable} interface.

### 10.3.4. Matching by Package

As you already know, you can use \texttt{*} to match classes in a package or its sub-packages. But if you want to match only classes in a specific package, you cannot do that without including all sub-packages too. The solution to this problem is to use the \texttt{package <name>}, which matches only classes in the package with the given name (note that you cannot use \texttt{*} or \texttt{?} in the package name). Using \texttt{package} allows you to match only classes in the default package (the package without a name), by using \texttt{package <default>}.

### 10.3.5. Matching by Class Properties

Here we discuss the keywords which match a class on a special property of that class. Note that none of these keywords expects a parameter.

\texttt{is_array} matches every array class. The dimension of the array is not relevant. Note that you could archive the same effect with \texttt{*[]*}, but \texttt{is_array} is clearer and faster to evaluate.

The expression \texttt{has_finalizer} matches all classes which have a non-trivial \texttt{finalizers()} method. Since the \texttt{finalizers()} method is defined in \texttt{Object} itself, every class has a \texttt{finalizers()} method. But as long as you don't overwrite it or just call the super classes \texttt{finalizers()} method, this \texttt{finalizers()} method is considered trivial (as long as no direct or indirect super class has a non-trivial \texttt{finalizers()} method). Objects of classes with only a trivial \texttt{finalizers()} method are treated by the VM as having no \texttt{finalizers()} method at all and don't take part in the 'expensive' finalizer handling. So by using the \texttt{has_finalizer} keyword you can match all classes which take part in finalization.

The expressions \texttt{is_public}, \texttt{is_private}, \texttt{is_protected} and \texttt{is_package_private} match classes with the associated visibility. To match abstract classes use \texttt{is_abstract}. Final or static classes are matched by \texttt{is_final} and \texttt{is_static}.

If the class is an enum class it matches the \texttt{is_enum} filter. If the class is an annotation class it matches \texttt{is_annotation}. An interface class matches \texttt{is_interface}.

Two more specialized filters are \texttt{is_strict} and \texttt{is_synthetic}. The former matches all classes declared 'strict', which means all floating point calculations in methods defined in the class must adhere strictly to the Java standard. The latter matches synthetic classes. A class can be marked synthetic by the Java compiler e.g. if it was only created as an implementation detail.
The last keyword to discuss here is \texttt{is\_internal}. It matches only classes internal to the VM (e.g. used to represent the byte code of a method). All internal classes can be recognized by the curly braces in their names, but using \texttt{is\_internal} is cleaner. There are only a few places where you will ever encounter internal classes (e.g. in the statistic about the permanent generation). See Section 10.1, “Internal Objects” for more about these objects.

10.3.6. Matching by Class Loader

Every class is loaded by a specific class loader. More specifically, we call the defining class loader of a class simply its class loader (see the VM Specification [http://java.sun.com/docs/books/jvms/second_edition/html/ConstantPool.doc.html#72007] for an exact definition). Informally the defining loader is the one which has the class file in its class path. The VM generally uses three different class loaders by itself:

1. The bootstrap class loader (named \texttt{<bootstrap>})
2. The extension class loader (named Extension)
3. The system or application class loader (named System)

The bootstrap class loader loads all classes in the JDK (e.g. \texttt{String} or \texttt{Integer}). The extension class loader loads the classes that were found in the \texttt{ext/} directory of the JDK. And the system class loader loads all the classes found in the directories and \texttt{jar} files specified by the \texttt{-cp} or \texttt{-classpath} argument. Additionally a Java application can create their own class loaders, and this is used heavily in application servers or even in Eclipse to separate components, unload or update them, and many more things.

Since the application server uses different class loaders for different components (applications, services), the class loader of a class can be used to determine the component of a class. But since a class loader generally has no name, it is hard to take advantage of this. Fortunately the AS Java assigns meaningful names to its class loaders. And the VM itself names it class loaders too, so when profiling the AS Java most class loaders will have names.

One additional area where a lot of class loaders are generated are reflection and serialization. The VM optimizes the repeated invocation of method using reflection by creating a small class that directly invokes the method in question. Each of these generated classes is loaded by its own class loader (this is needed to get rid of the helper class when it is not used anymore). A similar thing is done for calling the default constructor during serialization. In the case of reflection the name of the class loader starts with “Reflection” followed by the name of the method invoked. For serialization the name starts with “Serialization” and the invoked default constructor.

So after all these remarks, how can you filter by the class loader of a class? First, if the class loaders to match have a name, the \texttt{loadername} keyword allows you to match a class by the name of its class loader. If the loaders have no name, you might use \texttt{loaderclass} to match a class by the class of it’s classloader.

Using \texttt{loadername <name>} you match all classes whose class loader has the specified name. Note that you can use * and ? in \texttt{name} to match a whole group of class loaders. For example:

\texttt{loadername <bootstrap>}

would match all the JDK classes loaded by the bootstrap class loader. Or for a more complicated example:

\texttt{loadername service:*}

matches all the classes of all services in the Web AS.

\texttt{loaderclass <filter>} matches all classes whose class loader’s class matches the given \texttt{<filter>}. Note that every class loader apart from the bootstrap class loader is a Java object and thus has a class (since the bootstrap class loader is always named, you can use \texttt{loadername}). For example

\texttt{loaderclass instanceof *.URLClassLoader}
would match all class loaders which are loaded by an URLClassLoader or a loader derived from that loader. As you can see `loaderclass <filter>` is a lot less useful than `loadername`.

### 10.3.7. Examples

Now follows a short set of examples for class filters. If you like, you can try to decipher their meaning before looking at the description of what they do.

#### 10.3.7.1. Simple examples

Here comes a bunch of simple, yet useful class filters:

1. `instanceof java.lang.CharSequence`
2. `*HashMap`
3. `byte[]`
4. `is_array`
5. `loadername <bootstrap>`
6. `loadername application:*`

And here is what the filters match:

1. Matches all `String` like classes. These includes `StringBuffer` and `StringBuilder`.
2. A good approximation for finding all the hash maps used.
3. Matches all one-dimensional byte arrays. As you can see, the name of the arrays is just the same as you would write them in Java source code.
4. Matches all arrays, regardless of their dimension.
5. Matches all classes loaded by the bootstrap class loader.
6. Matches all application classes in the AS Java.

#### 10.3.7.2. Complex Examples

Here are some more complex examples:

1. `*[][] && !*[][][]`
2. `java.lang.String*`
3. `*$*
4. `extends implements java.io.Serializable`
5. `instanceof java.io.Serializable && !extends implements java.io.Serializable`
6. `java.lang.* && !java.lang.*.*`
And here is what the filters match:

1. Matches all two-dimensional arrays.

2. Matches `String`, `StringBilder` and `StringBuffer and their arrays`. Matching the arrays might seem surprising, but their names start with `java.lang.String` too!

3. Matches all inner classes.

4. Matches all classes for that one of the super classes implements `Serializable`. Note that it is not enough that the class itself implements `Serializable`.

5. Matches all classes which are the first in their class hierarchy to implement `Serializable`.

6. Matches all classes in the `java.lang` package but not classes in its sub-package. Note that you can get the same result more clearly via `package java.lang`.

### 10.4. Class Filter Overview

Here is a short listing of the keywords and parameters used in class filters. For a more detailed description see Section 10.3, “Class Filters”.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Example</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pattern</code></td>
<td><code>java.lang.String</code></td>
<td>Matches all classes whose full name match the given pattern. The pattern can contain <code>*</code> and <code>?</code> wild-cards.</td>
</tr>
<tr>
<td><code>&lt;filter1&gt; &amp;&amp; &lt;filter2&gt;</code></td>
<td><code>java.* &amp;&amp; isarray</code></td>
<td>Matches only if both filters match.</td>
</tr>
<tr>
<td>`&lt;filter1&gt;</td>
<td></td>
<td>&lt;filter2&gt;`</td>
</tr>
<tr>
<td><code>! &lt;filter&gt;</code></td>
<td><code>! java.lang.*</code></td>
<td>Matches only if the given filter does not match.</td>
</tr>
<tr>
<td><code>instanceof &lt;filter&gt;</code></td>
<td><code>instanceof *.Serializable</code></td>
<td>Matches all classes for which one of their superclasses or (directly in and indirectly) implemented interfaces match the filter or the class itself matches the filter.</td>
</tr>
<tr>
<td><code>extends &lt;filter&gt;</code></td>
<td><code>extends java.lang.Error</code></td>
<td>Matches all classes for which the filter matches one of the super classes.</td>
</tr>
<tr>
<td><code>implements &lt;filter&gt;</code></td>
<td><code>implements *.Runnable</code></td>
<td>Matches all classes for which the filter matches one of the (directly in and indirectly) implemented interfaces.</td>
</tr>
<tr>
<td><code>package &lt;name&gt;</code></td>
<td><code>package java.lang</code></td>
<td>Matches all classes in the given package. <code>*</code> and <code>?</code> are not allowed in the package name.</td>
</tr>
<tr>
<td><code>subpackage &lt;name&gt;</code></td>
<td><code>subpackage java.lang</code></td>
<td>Matches all classes in the given package and its subpackages. <code>*</code> and <code>?</code> are not allowed in the package name.</td>
</tr>
<tr>
<td><code>loadernamel &lt;name&gt;</code></td>
<td><code>loadernamel &lt;bootstrap&gt;</code></td>
<td>Matches all classes for which their class loader matches the given name. <code>*</code> and <code>?</code> can be used as wildcards.</td>
</tr>
<tr>
<td><code>loadercalss &lt;filter&gt;</code></td>
<td><code>loadercalss *.URLClassLoader</code></td>
<td>Matches all classes for which the URL of their class loader matches the given filter.</td>
</tr>
<tr>
<td><code>is_array</code></td>
<td><code>is_array</code></td>
<td>Matches all array classes.</td>
</tr>
<tr>
<td>Construct</td>
<td>Example</td>
<td>Effect</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>is_internal</code></td>
<td><code>is_internal</code></td>
<td>Matches all internal classes (classes used only by the VM).</td>
</tr>
<tr>
<td><code>has_finalizer</code></td>
<td><code>has_finalizer</code></td>
<td>Matches all classes with finalizers.</td>
</tr>
<tr>
<td><code>is_public</code></td>
<td><code>is_public</code></td>
<td>Matches all classes which are declared to be public.</td>
</tr>
<tr>
<td><code>is_protected</code></td>
<td><code>is_protected</code></td>
<td>Matches all classes which are declared to be protected.</td>
</tr>
<tr>
<td><code>is_package_private</code></td>
<td><code>is_package_private</code></td>
<td>Matches all classes which are declared to be package private.</td>
</tr>
<tr>
<td><code>is_private</code></td>
<td><code>is_private</code></td>
<td>Matches all classes which are declared to be private.</td>
</tr>
<tr>
<td><code>is_abstract</code></td>
<td><code>is_abstract</code></td>
<td>Matches all abstract classes.</td>
</tr>
<tr>
<td><code>is_final</code></td>
<td><code>is_final</code></td>
<td>Matches all final classes.</td>
</tr>
<tr>
<td><code>is_static</code></td>
<td><code>is_static</code></td>
<td>Matches all static classes.</td>
</tr>
<tr>
<td><code>is_strict</code></td>
<td><code>is_strict</code></td>
<td>Matches all classes declared to be strict fp.</td>
</tr>
<tr>
<td><code>is_synthetic</code></td>
<td><code>is_synthetic</code></td>
<td>Matches all classes declared to be synthetic.</td>
</tr>
<tr>
<td><code>is_annotation</code></td>
<td><code>is_annotation</code></td>
<td>Matches all annotation classes.</td>
</tr>
<tr>
<td><code>is_enum</code></td>
<td><code>is_enum</code></td>
<td>Matches all enum classes.</td>
</tr>
<tr>
<td><code>is_interface</code></td>
<td><code>is_interface</code></td>
<td>Matches interfaces.</td>
</tr>
</tbody>
</table>

Table 10.1.

10.5. Method Filters

A method filter is used to group methods into two sets: The ones which match the filter and the ones which do not. In contrast to class filters the syntax of method filters is relatively simple.

10.5.1. Basic Filter Syntax

In its simplest form a method filter is just the name of a method. So let's clarify first what is meant to be the name of the method. Generally the full method name consists of four parts:

1. The package name of the class in which the method was defined (e.g. `java.lang`).
2. The name of the class in which the method was defined (e.g. `String`).
3. The name of the method (e.g. `toString`).
4. The signature of the method (e.g. `(int, int[], java.lang.String)`).

The most complex parts are usually the signature, followed by the method name. So let's start with the method name. Usually the method name is quite clear, but there are two exceptions: The name of the static initializer and the name of the constructors. As it turns out, the name of the static initializer in the class file is `<clinit>` and that is the name we use in the method filters too. More surprisingly, the name of the constructors is not the name of the class, but `<init>`. So every constructor in every class has the same name, making it easy to match constructors.

Now we come to signatures. In principle the signatures are just as you would declare them in Java code, with the exception that every type is fully qualified (so classes are always prepended with their package). Arrays have the usual amount of `[ ]` after the base type. Types itself are separated by `','` (note the space after the comma). Note that the signature does not include the return type.
Now that we know the format of full method names, let’s look at the most basic method filters, which simply consist of the full method name. E.g.

```java
java.lang.String.equals(java.lang.Object)
```

Note that the classes in the signature are fully qualified. As with class filters you can use the wildcards `*` and `?` to match by pattern. For example if you aren’t interested in the signature, use the following filter:

```java
java.lang.String.equals(*)
```

Or to get all `hashCode()` methods:

```java
*.hashCode()
```

That’s about anything you have to know about the basic method filters.

### 10.5.2. Combining Filters

You can combine filters using `&&` and `||`. In the first case the combined filter only matches if both the filter in front and after the `&&` match. On the other hand for `||` only one (or both) of the filter must match. Note that you can alternatively use `&` or `|` instead, which have the same meaning. Additionally you can prepend a filter with `!` so the filter only matches if the filter prepended with the `!` does not match.

You can use parenthesis `(` and `)` to group filters, so the `&&` `||` and `!` operators are applied to the grouped filters. For example

```java
*.equals(*) && !*.equals(java.lang.Object)
```

would match all `equals` methods which are not of the canonical form.

### 10.5.3. Checking for Overwritten Methods

An expression of the form `overwrites <filter>` matches all methods which overwrite a method of a super class which matches the filter. For example:

```java
overwrites java.lang.Object.toString()
```

would match all methods which implement an own `toString` method, but not `Object.toString()` itself. But since every class has `Object` as its super class, the same methods would be matched just by (apart from the fact that `Object.toString()` would be matched too)

```java
*.toString()
```

A more useful example would be:

```java
overwrites java.lang.Runnable.run()
```

This would match all the `run()` methods of runnables, but not run methods of other types.

### 10.5.4. Matching by Method Properties

The expressions `is_public`, `is_protected`, `is_package_private` and `is_private` match methods of the given visibility. `is_native` matches native methods and `is_static` matches static methods.

A method can be strict (meaning the floating point operations in the method are to be performed according to the Java standard) or final by being defined as strict or final or by being defined in a strict or final class. The `is_declared_strict` and `is_declared_final` expressions only match methods which are explicitly
defined to be strict or final. On the other hand the `is_strict` and `is_final` filters match methods which are declared strict or final or which are defined in classes which are declared to be strict or final.

An abstract method is matched by `is_abstract` and a synchronized method by `is_synchronized`. If a method is a synthetic method generated by the Java compiler it matches the `is_synthetic` filter.

### 10.6. Method Filter Overview

Here is a short listing of the keywords and parameters used in method filters. For a more detailed description see Section 10.5, “Method Filters”.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Example</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>java.lang.String.</td>
<td>Matches all methods which full name (including the signature) match the given pattern. The pattern can contain * and ? wildcards.</td>
</tr>
<tr>
<td></td>
<td>toString()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*toString()</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>equals(</em>)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*equals(java.lang.Object)</td>
<td></td>
</tr>
<tr>
<td>&lt;filter1&gt; &amp;&amp; &lt;filter2&gt;</td>
<td><em>.equals(</em>) &amp;&amp; java.lang.*</td>
<td>Matches only if both filters match.</td>
</tr>
<tr>
<td>&lt;filter1&gt;</td>
<td></td>
<td>&lt;filter2&gt;</td>
</tr>
<tr>
<td>! &lt;filter&gt;</td>
<td>! *.hashCode()</td>
<td>Matches only if the given filter does not matches.</td>
</tr>
<tr>
<td>overwrites &lt;filter&gt;</td>
<td>overwrites java.io.Closeable.close()</td>
<td>Matches all methods which overwrite a method matching the given filter.</td>
</tr>
<tr>
<td>is_public</td>
<td>is_public</td>
<td>Matches all public methods.</td>
</tr>
<tr>
<td>is_protected</td>
<td>is_protected</td>
<td>Matches all protected methods.</td>
</tr>
<tr>
<td>is_package_private</td>
<td>is_package_private</td>
<td>Matches all package private methods.</td>
</tr>
<tr>
<td>is_private</td>
<td>is_private</td>
<td>Matches all private methods.</td>
</tr>
<tr>
<td>is_static</td>
<td>is_static</td>
<td>Matches all static methods.</td>
</tr>
<tr>
<td>is_declared_final</td>
<td>is_declared_final</td>
<td>Matches all final methods which are explicitly declared final. This means they are not just final because they are defined in a final class, but the final keyword was explicitly added to the method.</td>
</tr>
<tr>
<td>is_final</td>
<td>is_final</td>
<td>Matches all final methods. This includes all methods defined in a final class and all methods which are declared final explicitly.</td>
</tr>
<tr>
<td>is_declared_strict</td>
<td>is_declared_strict</td>
<td>Matches all strict fp methods which are explicitly declared strict. This means they are not just strict because they are defined in a strict class, but the strict keyword was explicitly added to the method.</td>
</tr>
<tr>
<td>is_strict</td>
<td>is_strict</td>
<td>Matches all strict fp methods. This includes all methods defined in a strict class and all methods which are declared strict explicitly.</td>
</tr>
<tr>
<td>is_abstract</td>
<td>is_abstract</td>
<td>Matches all abstract methods.</td>
</tr>
<tr>
<td>is_synchronized</td>
<td>is_synchronized</td>
<td>Matches all methods declared synchronized. Method which just use synchronized blocks inside their body are not matched.</td>
</tr>
<tr>
<td>Construct</td>
<td>Example</td>
<td>Effect</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>is_synthetic</code></td>
<td><code>is_synthetic</code></td>
<td>Matches all methods which are marked as synthetic.</td>
</tr>
<tr>
<td><code>is_overwriting</code></td>
<td><code>is_overwriting</code></td>
<td>Matches all methods which are overwriting a method of a super class.</td>
</tr>
</tbody>
</table>

Table 10.2.
# Index

## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>{array class}, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{compiledICHolder}, 203, 203</td>
<td></td>
</tr>
<tr>
<td>{const method}, 202, 202, 202, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{constant pool cache}, 201, 201, 201, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{constant pool}, 200, 200, 200, 200, 200, 201, 203</td>
<td></td>
</tr>
<tr>
<td>{instance class}, 200, 200, 200, 200, 200, 201, 203</td>
<td></td>
</tr>
<tr>
<td>{method data}, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{method}, 200, 202, 202, 202, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{object array class}, 200, 200, 200, 200, 200, 201, 203</td>
<td></td>
</tr>
<tr>
<td>{other class}, 203, 203, 203</td>
<td></td>
</tr>
<tr>
<td>{symbol}, 201, 201, 202, 202, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{type array class}, 200, 200, 200, 200, 201, 202, 203</td>
<td></td>
</tr>
</tbody>
</table>

## A

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive Allocation Analysis</td>
<td>50, 50</td>
</tr>
<tr>
<td>Allocation Analysis</td>
<td>11, 13, 49, 49, 49, 49, 49, 50, 50, 51, 52, 52, 53, 59, 71, 71, 72, 72, 72, 74, 75, 84, 88, 88, 108, 173, 176, 176, 177, 177, 183, 183, 186, 186, 186, 187, 203, 203, 203, 204, 204, 204</td>
</tr>
<tr>
<td>Analysis</td>
<td></td>
</tr>
</tbody>
</table>

## C

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined Performance Hotspot and Method Parameter Analysis</td>
<td>115</td>
</tr>
<tr>
<td>constant pool, 200, 200, 200, 200, 200, 200, 200, 201, 201, 201, 201, 201, 201, 201, 201, 202, 202</td>
<td></td>
</tr>
</tbody>
</table>

## E

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception table, 202, 202, 202</td>
<td></td>
</tr>
</tbody>
</table>

## F

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
</table>

## G

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>generation</td>
<td></td>
</tr>
<tr>
<td>perm generation, 200, 200, 202</td>
<td></td>
</tr>
</tbody>
</table>

## I

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal objects</td>
<td></td>
</tr>
<tr>
<td>{array class}, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{compiledICHolder}, 203, 203</td>
<td></td>
</tr>
<tr>
<td>{const method}, 202, 202, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{constant pool cache}, 201, 201, 201, 201, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{constant pool}, 200, 200, 201, 201, 201, 201, 202, 203</td>
<td></td>
</tr>
<tr>
<td>{instance class}, 200, 200, 200, 200, 200, 201, 203</td>
<td></td>
</tr>
<tr>
<td>{method data}, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{method}, 200, 202, 202, 202, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{object array class}, 200, 200, 200, 200, 200, 201, 202, 203</td>
<td></td>
</tr>
<tr>
<td>{other class}, 203, 203, 203</td>
<td></td>
</tr>
<tr>
<td>{symbol}, 201, 201, 202, 202, 202, 202, 202, 202</td>
<td></td>
</tr>
<tr>
<td>{type array class}, 200, 200, 200, 200, 201, 202, 203</td>
<td></td>
</tr>
<tr>
<td>interned string, 202</td>
<td></td>
</tr>
</tbody>
</table>

## J

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac, 202</td>
<td></td>
</tr>
<tr>
<td>javap, 200, 202, 202</td>
<td></td>
</tr>
<tr>
<td>JIT compiler, 202, 203</td>
<td></td>
</tr>
</tbody>
</table>

## L

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>line number table, 202, 202, 202, 202</td>
<td></td>
</tr>
</tbody>
</table>

## M

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Parameter Analysis, 25, 27, 71, 72, 84, 88, 88, 173, 176</td>
<td></td>
</tr>
<tr>
<td>Method Parameters Dialog, 29, 29, 30, 31, 32, 32, 33, 72, 73, 118, 119, 120, 120</td>
<td></td>
</tr>
<tr>
<td>Analysis Start Dialog, 28, 29, 32, 118, 119, 120, 120, 121, 121</td>
<td></td>
</tr>
</tbody>
</table>

## P

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Hotspot Analysis, 16, 76, 76, 76, 76, 76, 76, 76, 76, 77, 77, 79, 80, 80, 82, 83, 83, 84, 84, 86,</td>
<td></td>
</tr>
</tbody>
</table>
Index

perm generation, 200, 200, 202
Profile View, 14, 15, 15, 26, 28, 28, 33, 33, 34, 34, 34, 34, 52, 56, 56, 57, 118, 120
profiling information, 202, 202
programs
  javac, 202
  javap, 200, 202, 202, 202

S
server compiler, 202
stack map, 202, 202, 202, 202
Synchronization Analysis, 38, 40, 129, 129, 129, 129, 129, 129, 129, 129, 129, 129, 130, 130, 130, 130, 131, 131, 136, 139, 140, 140, 140, 140, 141, 141, 144, 145, 145, 147, 147, 149, 150, 150, 151, 151, 152, 152, 152, 152, 152, 152, 152, 152, 152, 152

U
UTF-8, 202

V
Views
  Analysis Start Dialog, 28, 29, 32, 118, 119, 120, 120, 121, 121
  Profile View, 14, 15, 15, 26, 28, 28, 33, 33, 34, 34, 34, 34, 52, 56, 56, 57, 118, 120