# Sherlok - Version 1.4

Java Application Monitor

*13. September 2005*

# 1 Introduction

Sherlok is a Java application monitor, that allows to get detailed information about the behavior of java programs. It implements the following aspects of monitoring:

- Monitor memory allocation and memory leaks
- Monitor performance
- Monitor program execution (application trace)

## 1.1 Monitoring Memory

Memory is a resource, which is shared by all components of a Java application without any quotas. The overall application will fail, if only one component does not cooperate (e.g. has a memory leak) or if several components with huge demand on memory will compete for this resource when they run at the same time (in threads). Once the application reaches the system limits, the infamous 'out-of-memory' occurs and it is hard to tell which components are at fault.

Memory consumption also impacts performance. Each chunk of memory, which is used by a Java component, will have to go through the garbage collector to make it available again for other components. The allocation of memory, the initializing of objects and finally the garbage collector need time for execution.

In order to help diagnosing memory problems, Sherlok offers these features:

- Show memory usage by component with allocated, deallocated, retained memory and a history for these values,
- A *Memory Leak Detector* mode that watches for growing classes, and a mode that can trap 'out-of-memory' situations.
- A quick heap dump to analyze the component's memory usage.

## 1.2 Monitoring Performance

Performance analysis, i.e. monitoring execution times, is also hard for a complex client-server application. The request response time might depend more on LDAP, SQL or file system services, than on the executed Java code. What you need to improve your software is reliable data to compare the current state with prior ones and make the measurement independent from network and external services.

- Sherlok measures CPU and elapsed time for each Java method in scope and allows you to find the cause of any delay in a given response. Sherlok also allows you to 'drill down' and focus the performance measurements.
- Sherlok has a built-in HotSpot analyzer, which dumps the call stack for the most expensive method execution contexts. This allows to find the 'top slow methods'.

## 1.3 General Application Tracing

Sherlok offers tracing features as well. Tracing 'shows what the application is doing' dynamically. All trace functions dump information to the telnet console (and log file) and they are triggered by program execution events. These tracing trigger events include:

- Activation of the garbage collector: writes more detailed GC output
- Method *enter* and *exit* events: can be traced
- Unhanded exceptions: can write stack trace

# 1.4 Usage Modes

### Web UI and Console

Sherlok provides a web interface (IViews and servlet) for simpler use cases and a console / telnet interface for power users. A special use case, the iView TestBench has a separate very simple dedicated web interface. Not all Sherlok features are available through the web interfaces.

### Different Usage Types

Sherlok can be used by different types of users in different ways:

- *iView / Code Developers* can use it during development to identify resource usage and optimize the code. In general, the simple **iView TestBench** user interface will be sufficient to identify resource problems and pinpoint the causes in the code. For in-depth analysis the console mode is available.
- *Advanced Technical Support* can use Sherlok to do general analysis of overall system performance and to identify possible issues. This generally requires the console mode.
- A *content developer* combines iViews to pages and roles and can use the iView TestBench to measure the resource needs of certain iViews and pages / interactions.

### Usable for any Java program

Sherlok is described in the application context of the SAP Enterprise Portal, but it can be used on any Java program and JDK 1.3.1 and 1.4.

# 1.5 Changes from Version 1.3

- Sherlok now supports multiple configuration files, and to load them dynamically. The configuration files are located under the `sherlok` subdirectory by default. The default is `default.skp`.
- Some command names were changed to be more consistent and understandable.
- User interface changes were made to simplify usage.

# 2 Installation and Initial Configuration

In this chapter we will show how to install Sherlok on a Enterprise Portal node. To prepare for the installation, download the most recent version of the Sherlok package archive for your platform from the Sherlock homepage.

## 2.1 Requirements

- Operating system Linux, Windows, Solaris, HP-UX or AIX
- Java Runtime Environment (JRE) 1.3 or 1.4
- Telnet executable for command line interface
- SAP J2EEngine 6.20 or 6.30 for `sherlok.ear`
- Portal EP 6 for iView interfaces

## 2.2 Installation of Sherlok Package

The Sherlok Package consists of two parts: (1) The shared library and (2) the Sherlok support files, i.e. configuration files, UI components, documentation etc.

From now on we consider the case of SAP Enterprise Portal installed on SAP J2EE. Therefore the directory is one of the following depending on your portal configuration:

### 2.2.1 Install The Sherlok DLL

The DLL varies by platform:

| Platform | Library |
|---|---|
| MS Windows | `sherlok.dll,`<br>`(System: msvcr71d.dll, msvcp71d.dll)` |
| Solaris | `libsherlok.so` |
| Linux | `libsherlok.so` |
| HP-UX (32/64bit) | `libsherlok.sl` |
| AIX (64bit) | `libsherlok.so` |

Install the DLL at the root directory of the server instance, i.e. where the JVM is called. This directory can differ, e.g.

| Configuration | Installation directory inside of J2EE Engine to use |
|---|---|
| Standalone | …`/alone` |
| Cluster with one server node | …`/cluster/server` |
| Cluster with multiple server nodes | …`/cluster/server<n>` (where `<n>` is node number) |
| J2EE 6.40 | …`/cluster/os_libs` |

**Notes**

- To run Sherlok 1.4.0.7 on NT make sure, that **msvcp71d.dll** and **msvcr71d.dll** are available
- Standalone configuration is used for EP5.0, the cluster versions for EP6.
- The path leading to the DLL location must not contain spaces! (if you have problems here, see 'Central Installation Option' below)

### 2.2.2 Unpack the Sherlok Support File JAR

Unpack the Sherlok support files JAR (`sherlok.jar`) in the directory …`/cluster/server`. This will create a subdirectory `sherlok` that contains all the additional files such as the configuration files (`*.skp`) documentation etc. The file `default.skp` is the default configuration file, that you can also use as the template when creating your new configurations.

### 2.2.3 Adapting Sherlok configuration to your Namespace

Sherlok configuration files in the standard delivery are too generic for most cases. It is strongly recommended to adapt them, especially to restrict the count of classes monitored as much as possible.

For instance, when your portal components (you want to investigate) are implemented as the set of classes in the Java package **com.myfirm.myapplication.*** you should modify the **ProfilePackage** property in the configuration file `sherlok/default.skp` (and also in any other `*.skp` file you want to use) to the corresponding value like in the following sample:

```
#
# File: default.skp  (sample)

ProfileMemory       = on
ProfilePackages     = com.mycompany.test.
Timer               = on
…
```

For the exact meaning of the parameters and special usage of wildcards in the patterns have a look at the Parameter reference section.

### 2.2.4 Embed Sherlok into JVM

As next step, you have to embed Sherlok into the JVM by modifying the start script. Follow the instructions depending on platform and SAP J2EE version you use.

#### *2.2.4.1 For all Platforms*

You should also always run with the **-verbose:gc** option set that outputs GC information in the console log. This is always valuable information about the memory behavior of an application and needed to get the memory usage graphs described in the sherlok.jar.

If exists, put the shared library into the folder **os_libs**. In any other case put the shared library to the server directory and set the library path environment variable (on Linux LD_LIBRARY_PATH).

Create a directory …/cluster/server<n>/sherlok and put the configuration files into it. The two command line arguments are synonyms:

- -Xrunsherlok:ConfigPath=sherlok
- -Xrunsherlok:ConfigFile=sherlok/default.skp

#### 2.2.4.2 Windows with Portal running on J2EE 6.20

Open the start script `go.bat` (or `godebug.bat`*)* in a text editor. Locate the line starting with “set DEBUG_PARAMS” and add there following parameters (no line break)

```
set DEBUG_PARAMS=-Djava.compiler=NONE -Xdebug -Xnoagent
-Xrunsherlok:ConfigPath=sherlok
```

At the end, save all changes and restart the server node to make the changes effective.

#### 2.2.4.3 Windows with Portal running on J2EE 6.40 / 6.30

Start the **ConfigTool** from the SAP J2EE Engine and select there the correct server node instance.

Under the tab strip **General** you will find a **Java settings** / **Java parameter** text field, where you have to add following JVM parameters:

```
-Xdebug
-Xrunsherlok:ConfigPath=sherlok
```

Then switch to the tab-strip **Debug** and check there the value of the flag **Debug** / **Enable debug mode**. This flag has to be disabled to get the Sherlok running properly.

At the end, save all changes and restart the server node to make the changes effective.

#### 2.2.4.4 Solaris / HP-UX / AIX / Linux with Portal running J2EE 6.20

Open the starting script `go.bat` (or `godebug.bat`*)* in a text editor. Locate the line starting with “set DEBUG_PARAMS” and add there following parameters. (no line break!)

You have to also add the environment variable `LD_LIBRARY_PATH=.` to the script.

```
set DEBUG_PARAMS=”-Xdebug -Xrunsherlok:ConfigPath=sherlok”
set LD_LIBRARY_PATH=.
```

At the end, save all changes and restart the server node to make the changes effective.

#### 2.2.4.5 Solaris / HP-UX / AIX / Linux with Portal running J2EE 6.40 / 6.30

Start the **ConfigTool** from the SAP J2EE Engine and select there the correct server node instance.

Under the tab strip **General** you will find a **Java settings** / **Java parameter** text field, where you have to add following JVM parameters:

```
-Xdebug
-Xrunsherlok:ConfigPath=sherlok
```

in the **Startup Framework Settings** dialog for the **Java parameters** section. Make sure, that the debug option on this node is disabled.

Copy the shared library into .../`cluster/os-libs`

At the end, save all changes and restart the server node to make the changes effective.

### 2.2.5 Changing the Telnet Port

With the settings to the JVM invocation done in previous section, the Sherlok offers its console interface for the Telnet host **localhost** at port 2424.

In special cases (i.e. when this port is already occupied for other services), you may have to change this settings by adding one or more parameters to the JVM invocation as the comma-separated list of key/value pairs like in following sample.

```
-Xrunsherlok:ConfigPath=sherlok,TelnetPort=2222
```

### 2.2.6 Testing the Console

Having previous steps completed, you get elementary output (property dump) from Sherlok on the corresponding server console whenever the Portal server is started.

Sherlok can be then used via a console interface (via telnet) and various web UIs. For the console mode you access the interface via port 2424 on **localhost** (or corresponding values when this was changed in the JVM invocation).

```
telnet localhost 2424
```

You can then use the console interface via telnet, even without waiting until the portal is fully started. You can just open the console and type 'exit' to close it. See the 'Console Reference' section for more details.

## 2.3 Install Sherlok User Interfaces (optional)

Sherlock provides not only a command based interface but also more comfortable interfaces that can be installed separately. You wil find the necessary deployable files inside of the `sherlok` subdirectory.

### 2.3.1 Upload Portal iView to the Enterprise Portal

There is a special portal component archive called `com.sap.portals.sherlok.par` that can be easily deployed to the portal and instantiated in the portal pages in a common way.

Being a portal administrator, you can upload this component via Administration console *(System Administration* -> *Support* -> *Support Desk* -> *Area: Portal Runtime* -> *“Portal Anywhere” Admin Tools : Administration Console).*

You can get the Archive Uploader also by typing the following URL in your browser (adapted to the true server name

```
<portal_server>/irj/servlet/prt/portal/prtroot/PortalAnywhere.ArchiveUploader
```

### 2.3.2 Servlet

For non portal application Sherlok exposes the interfaces in

- `sherlok.ear` for J2EEngine
- `sherlok.war` for arbitrary servlet container, contained in the ear file
- com.sap.portals.sherlok.par: portal iView monitor
- com.mycompany.test.par: portal iView test component

Please refer to the documentation of the servlet container how to deploy and modify startup parameters.

## 2.4 Sherlok documentation

The Sherlok documentation (this file) is also located in the `sherlok` subdirectory.

## 2.5 Memory Considerations

Note that Sherlok itself needs memory to run. To monitor all classes and objects in a Java system you can expect that Sherlok needs about 10-15% of the Java memory size. This amount of memory requirement will be added to the JVM, i.e. the java process!

Please make sure the java process has enough physical memory to avoid paging. Also, on Windows, the address space is limited to 1.3 GB. In case you had the Java heap set to 1.3 GB to fully utilize all memory, you must reduce the Java heap size by about 200 MB to make room for the Sherlok memory needs.

# 3 QuickStart: The iView TestBench

The simplest way to use Sherlok is to use the iView TestBench in the portal. The purpose is to be able to simply measure how much CPU time and memory a certain interaction (iView, page) needs and offer quick feedback on possible resource problems.

The following description does not explain all details to the last level but tries to give you enough to get started quickly. For full details see the detailed descriptions after this section.

**Note:** As memory profiling can degrade portal performance by up to ten times (depending on the profile mode) it is strongly recommend to use a dedicated system, i.e. a development/test system.

In order to get valid results it is also necessary to allocate the development system for exclusive use. It is further assumed that there is no concurrent activity taking place in the system as the profiling is in progress.

The basic idea of the TestBench is to actively use an iView in one windows and to observe its behavior in a second window: the windows of the iView TestBench. It will show memory consumption and timing of the monitored classes as the application is used.

In order to limit the output to the significant classes of the application, it is necessary to configure Sherlok appropriately. Otherwise the results will be scattered among several classes that are out of scope. Sherlok uses the default preset file passed in the parameter **ConfigFile** when the TestBench is used. See 11.1 for explanation.

## 3.1 Starting the TestBench

As the first step in testing your iView, the iView TestBench has to be started. This application can be found in the **Sherlok** role.

The TestBench user interface is fairly straightforward, offering only two different commands:

- **Next Step** will take a snapshot of the current state of execution and display it in the table below.
- **Reset** will clear the history and also recognize changes made to the configuration in the mean time.

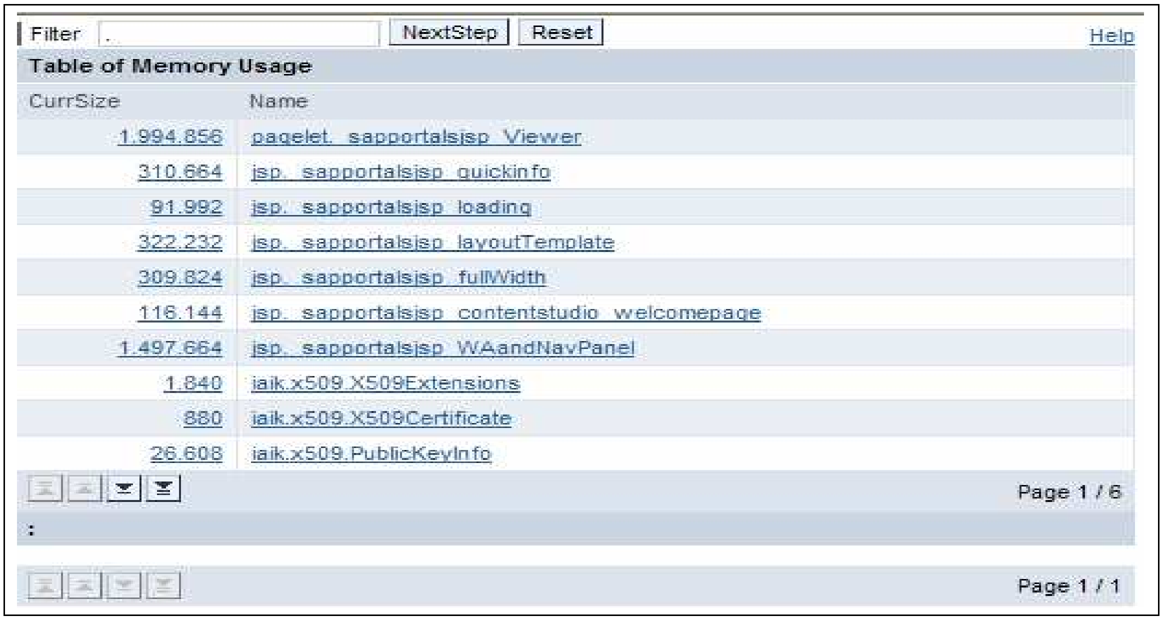


Table of Memory Usage

| CurrSize | Name |
|---|---|
| 1.994.856 | pagelet._sapportalsjsp_Viewer |
| 310.664 | jsp._sapportalsjsp_quickinfo |
| 91.992 | jsp._sapportalsjsp_loading |
| 322.232 | jsp._sapportalsjsp_layoutTemplate |
| 309.824 | jsp._sapportalsjsp_fullWidth |
| 116.144 | jsp._sapportalsjsp_contentstudio_welcomepage |
| 1.497.664 | jsp._sapportalsjsp_WAandNavPanel |
| 1.840 | iaik.x509.X509Extensions |
| 880 | iaik.x509.X509Certificate |
| 26.608 | iaik.x509.PublicKeyInfo |

**Picture 1: iView TestBench**

This table below the buttons contains a list of classes, ordered by their memory consumption. In the first column the table shows the total memory requirements of an application part (i.e. a Java class) so far, the second column contains the full qualified name of the Java class responsible. For every entry, detail information can be shown by clicking on the corresponding table cell (see 3.2.3 and 3.3.3 for a detailed explanation).

In order to reduce the number of display items, the list can be filtered using the text field at the top of the application. Here either a prefix, infix or suffix of the classes of interest can be entered. A “.” dot is used at the beginning and/or the end of the filter string to define an arbitrary string. An infix string (substring) is enclosed in dots, a prefix ends with a dot and a suffix starts with a dot.

**Examples:**

- “.sun.” matches **sun.java.util.***, **com.sun.java.net.*, com.sun.***, …
- “.Map” matches any class ending with “Map” like **java.util.HashMap**
- “com.sap.” matches **com.sap.util.***, **com.sap.portals.***, **com.sapportals.***, …

## 3.2 Memory Profiling

Excessive memory consumption is a major problem in Java programs. Neither the portal nor the Java base technology restrict the amount of memory an application can allocate. As a result, a single iView can bring down the entire portal installation by eating up all the memory available to the Virtual Machine.

It is strongly recommended to measure the memory consumption of all portal content before it is deployed on a production system. The TestBench will give a deeper insight into the memory requirements of the iViews. Within a few steps, those parts that claim major amounts of memory can be detected. iViews that show non-cooperative behavior show up soon, because of their eye-catching numbers.

The iView TestBench is collecting data about allocated and deallocated memory by intercepting both the creation of new objects (allocations) performed by a class and the reclaiming of memory (deallocations) by the garbage collector. Both allocation- and deallocation-values are accumulated for the time between two runs of the garbage collector (called GC cycles).

The following sections will outline a step-by-step approach to testing iViews for memory consumption.

### 3.2.1 Step 1: Start TestBench

First, the TestBench has to be opened. If it has been used recently it should be initialized by pressing the **Reset** button. See 3.1 on how to start TestBench.

### 3.2.2 Step 2: Use the iViews of interest

To drive memory-consumption, the iView that is to be inspected must be used actively. This is best accomplished by starting it from the **Content Inspection** area using **Preview** and then going through several iterations of the measurement procedure that is outlined here:

- Perform some activity in the custom iView.
- Press **Next Step** in the TestBench

After every step the table is updated with the latest results and Sherlok attempts to induce a garbage collection in the JVM. This will create some detail information on the memory consumption.

### 3.2.3 Step 3: Inspect the numbers

From time to time the results should be examined closer. When there are multiple classes on the display, concentrate on those classes that consume large amounts of memory or have a steadily increasing total (**CurrSize** column). The classes can be easily sorted by clicking on the column-labels.

To see detailed information on the memory requirements of a class, click on the size entry that is displayed left to the class name. Now a table will be show up below containing the garbage collection history of the class.

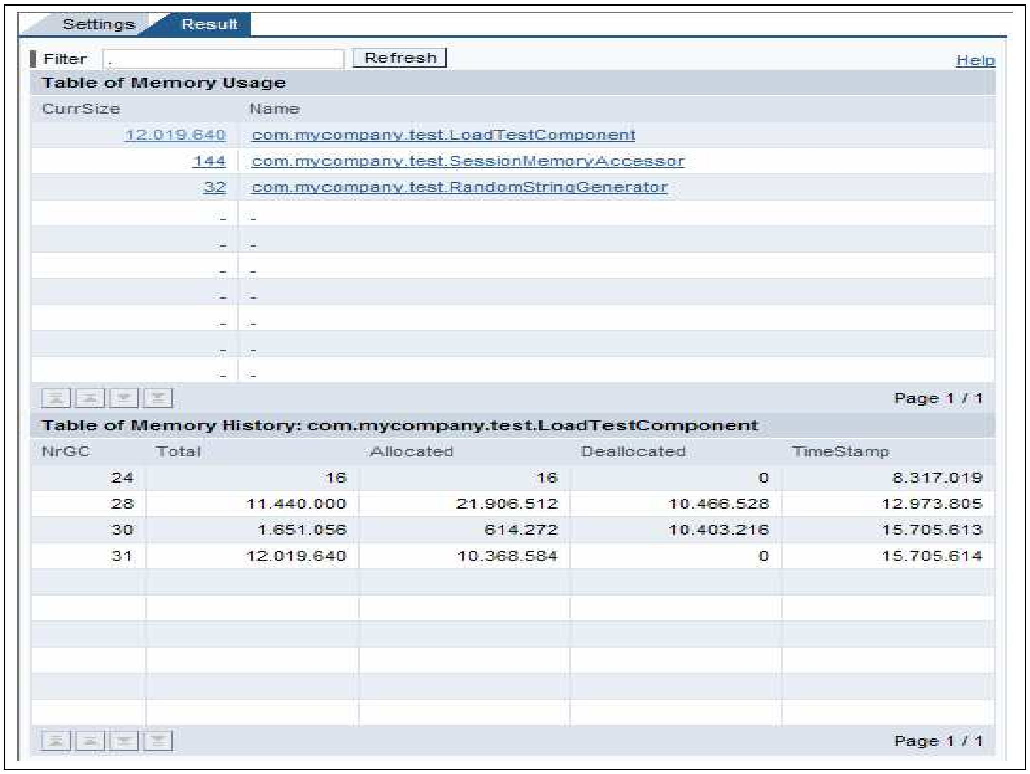


Table of Memory Usage

| CurrSize | Name |
|---|---|
| 12.019.640 | com.mycompany.test.LoadTestComponent |
| 144 | com.mycompany.test.SessionMemoryAccessor |
| 32 | com.mycompany.test.RandomStringGenerator |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |

Table of Memory History: com.mycompany.test.LoadTestComponent

| NrGC | Total | Allocated | Deallocated | TimeStamp |
|---|---|---|---|---|
| 24 | 16 | 16 | 0 | 8.317.019 |
| 28 | 11.440.000 | 21.906.512 | 10.466.528 | 12.973.805 |
| 30 | 1.651.056 | 614.272 | 10.403.216 | 15.705.613 |
| 31 | 12.019.640 | 10.368.584 | 0 | 15.705.614 |

**Picture 2: Memory History**

The history contains memory allocations and deallocations for the current and privious GC cycles. The size of the table depends on the configuration of Sherlok and can be adjusted with the property **MemoryLimitHistory** (see 11.1.4.3).

| Column | Description |
|---|---|
| GCNr | Sequence number of the garbage collection.<br>**Note:** This number is assigned by the JVM. Sometimes multiple garbage collections are combined, carrying only the number of the last GC. Therefore there may be gaps in the numbering. |
| Total | The number of bytes that is actually consumed by the class of the application (in bytes). This is the total amount of memory that has been allocated, minus the total of deallocated memory. |
| Allocated | The amount of memory being newly allocated during the GC cycle (in bytes).<br>This number includes all memory that has been requested, no matter if it is still referenced or available for reclaiming. |
| Deallocated | The amount of memory being deallocated during the GC cycle (in bytes).<br>This number includes all memory that has been reclaimed by the garbage collector since the previous garbage collection, no matter if it was allocated in this cycle or in one of the previous ones. |
| TimeStamp | Time stamp value, depending on operating system, with zero meaning current time.<br>The time stamp is a relative value that be used to compare intervals to each other. |

For a single GC cycle, the amount of deallocated memory may well exceed the amount of allocated memory. This is often confusing to the novice, but comes from the fact the allocated and deallocated values do not necessarily refer to the same memory *space* or objects.

### 3.2.4 Step 4: Classify the results

Looking at the memory history of a class, it can be assigned one of these cases:

| | |
|---|---|
| Well behaving | The memory requirements are constantly low (< 2 MBytes). |
| High Demand | The memory requirements are constantly high. |
| Peak Demand | The allocation reaches critical levels in peak situations, but remains otherwise normal. |
| Critical Demand | The memory requirements are very high and may cause outages under load. |
| Leak | The memory requirements are constantly growing, a memory leak seems likely. |

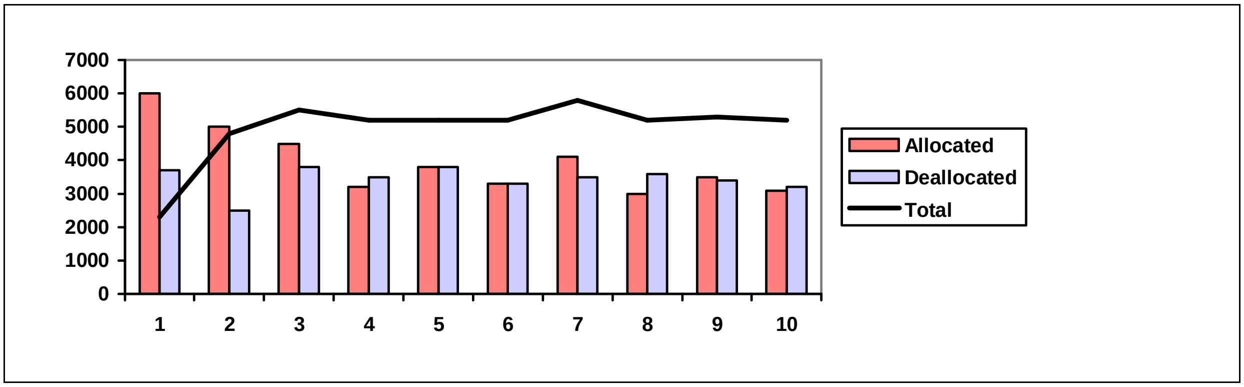
To do this assignment it is very helpful to display the entries of the memory history table as a graph. You can export the history table for iView TestBench to Microsoft Excel using copy-and-paste from the table. You may use the xls sample in the subdirectory `sherlok/gcgraph`.

Every kind of issue has its very own fingerprint or pattern that shows up in the history graph.

#### High / Critical Demand

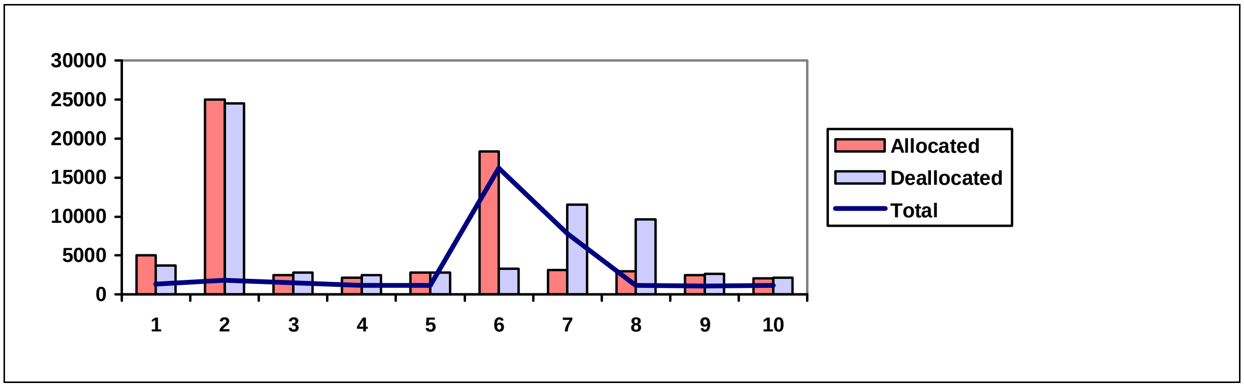
The amount of memory is relatively stable for both the *allocated* and the *total* value. The iViews following this patterns usually have no memory leak and their memory demand is predictable. Still, high values still point to possible improvements in the implementation. When the total value reaches critical levels (like >10

MB), memory requirement have to be reduced in order to keep the portal installation operational. Application logic that requires that high levels of memory should be moved to a separate application server.



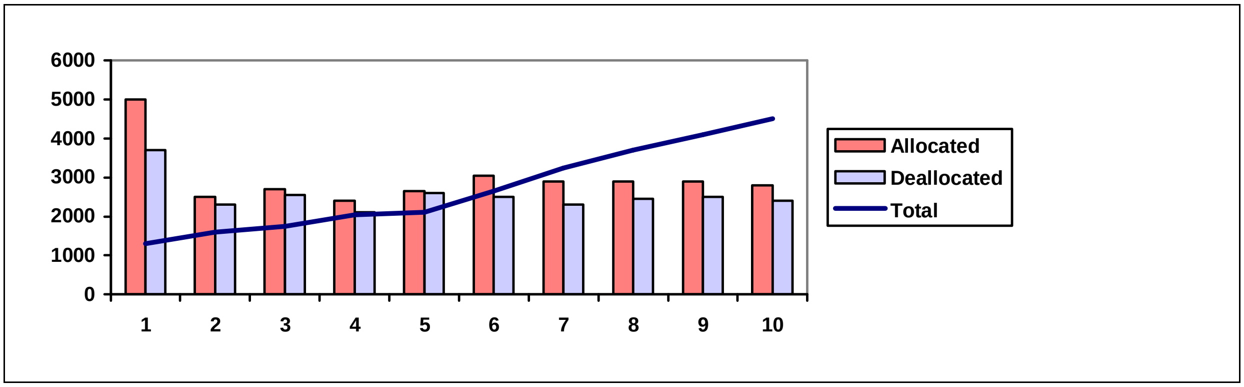
## Peak Demand

There are high peak values for the *allocated* and *deallocated* values, with the total staying at constant levels or growing only for a very short period of time (1-2 GC cycles). High peak values point to efficiency problems in the implementation leading to high amounts of garbage objects. Though the memory is not claimed for a longer period, this is still a critical issue in a multiuser environment. When several peaks of different applications and/or users concur, the combined total may exceed the available memory and the system may be injured by expensive full GCs or even fail with an out-of-memory error.



## Leakage

The memory footprint of the iView is constantly growing, so independently from the **allocated** and **deallocated** value, the **total** is growing over time. The reason is the class is collecting data *in memory*. Because business applications normally collect their data in a database or some other external storage facility, this points to a bug in the software. The application is keeping objects *by accident*. Sooner or later the leaking application may have claimed so much memory that the system will fail. This way, even a small leak may bring down the entire system.



### 3.2.5 Step 5: Set Up Action Items

Depending on the memory consumption pattern, the following measures can be planned.

#### High Demand

iView whose memory requirements are staying constantly above acceptable levels should be inspected for excessive caching. This may happen either in the iView itself or depending on the usage of external subsystems like JCo or a third-party library.

The iView should be inspected in detail using the iView Monitor (see 5.2.1). Once the problem can be assigned to a specific subsystem, the iView has to be reviewed by the developer to check the usage of the subsystem's components and their APIs.

#### Peak Demand

Peaks in memory consumption point to inefficient usage of the Java programming language, leading to a multitude of temporary objects. As with the previous case, you should track-down to the source of the problem to identify the responsible group of developers or the vendor of the subsystem.

#### Leakage

A memory leakage is caused by the iView itself in most cases. Therefore a deeper inspection of the subsystems should be left to the developer of the iView, as it is impossible to assign a leak correctly without looking at the implementation of the iView. Still, it might be helpful to apply the **Monitor Subsystems** preset anyway. If there is no leakage in the inspected subsystems, this source can be safely ruled out.

## 3.3 Performance Monitoring

Performance is a key factor in the usability of portal applications. Slow applications lead to dissatisfaction on behalf of the user and eat up processing time that could better be used otherwise.

Portal content that shows lengthy response times should be inspected to trace the performance leak back to the source. The test bench will give a deeper insight into the execution times of the various methods an application is composed of.

Essentially, the iView TestBench takes snapshots of the timing of an application. For every method in every monitored class the total execution time and the number of calls is recorded. The results are added to a table.

We will explain here the steps to take in order to find bad performing classes.

### 3.3.1 Step 1: Start iView TestBench in a separate window

See section 3.1 to learn how TestBench can be started.

A description of the performance data offered by TestBench can be found in section 3.3.3.

### 3.3.2 Step 2: Use the iViews of interest

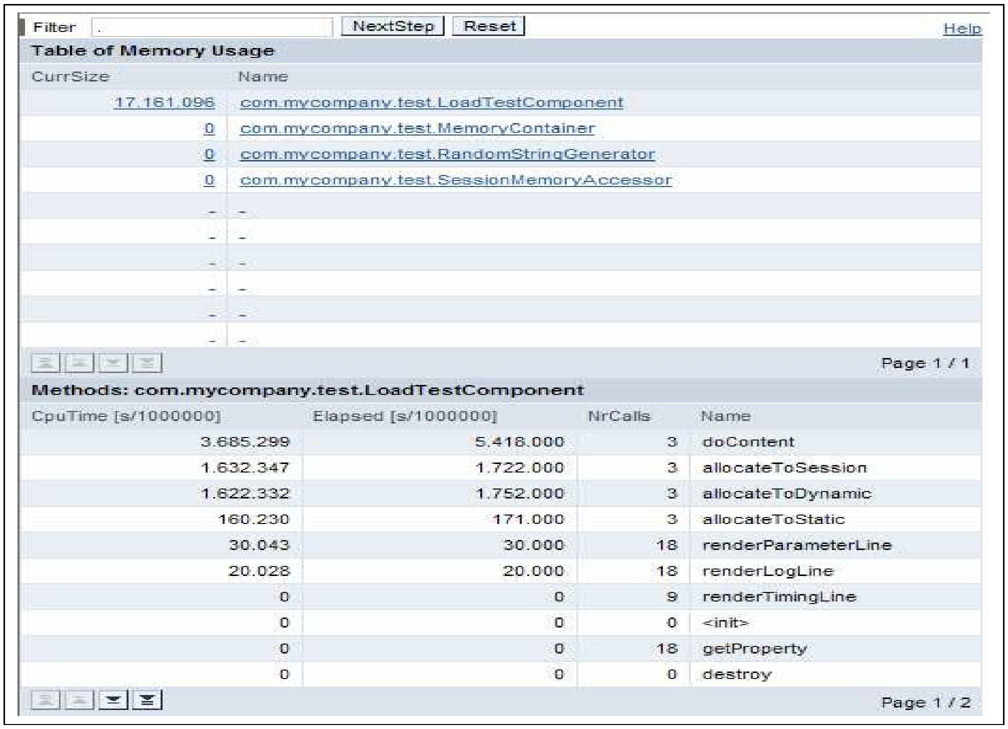
To drive CPU-consumption, the iView that is to be inspected has to be actively used. The iView can be started from any place to obtain valid performance data. Either from the page it is normally embedded or directly from the **Content Inspection** role.

After the iView is loaded the usage should focus on the functionality that shows long response times. When the initial displaying of the iView is the key-issue, the iView should be started over and over using the **Content Inspection** role.

### 3.3.3 Step 3: Inspect the numbers.

After some activity the **Next Step** button of the TestBench should be pressed in order to get results. The results will be updated each time this button is pressed.

In order to get the total execution time of the iView the class actually implementing the iView has to be located. A click on the class name will bring up the detail view on performance data. Note, that there is no history created for performance. So the **Next Step** button has only the effect of updating the table.



Filter: .

Table of Memory Usage

| CurrSize | Name |
|---|---|
| 17.161.096 | com.mycompany.test.LoadTestComponent |
| 0 | com.mycompany.test.MemoryContainer |
| 0 | com.mycompany.test.RandomStringGenerator |
| 0 | com.mycompany.test.SessionMemoryAccessor |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |

Page 1 / 1

Methods: com.mycompany.test.LoadTestComponent

| CpuTime [s/1000000] | Elapsed [s/1000000] | NrCalls | Name |
|---|---|---|---|
| 3.685.299 | 5.418.000 | 3 | doContent |
| 1.632.347 | 1.722.000 | 3 | allocateToSession |
| 1.622.332 | 1.752.000 | 3 | allocateToDynamic |
| 160.230 | 171.000 | 3 | allocateToStatic |
| 30.043 | 30.000 | 18 | renderParameterLine |
| 20.028 | 20.000 | 18 | renderLogLine |
| 0 | 0 | 9 | renderTimingLine |
| 0 | 0 | 0 | <init> |
| 0 | 0 | 18 | getProperty |
| 0 | 0 | 0 | destroy |

Page 1 / 2

**Picture 3: Method Timing Details**

The table that shows up at the bottom contains the execution time and the total number of calls for each method in the selected class. The numbers here grow with every step, there is no history like in the memory case.

| Column | Description |
|---|---|
| CPU-Time | Cumulated CPU time usage<br>This parameter measures the plain *processing* time. This is the time the thread was in *running* state. This is the time it took the method to complete minus the time other threads or the operating system owned the CPU. See 4.2 for details. |
| Elapsed | Cumulated time measured between enter and exit of a method. In short: This is the accumulated execution time of the method.<br>The difference between CPU to elapsed time is the time a thread was in *ready* or *waiting* state. A thread is in waiting state, when it is waiting for a signal of the operating system, like incoming data of an I/O or an object monitor. It is in ready state, when the scheduler has elected other threads to own the CPU. |
| NrCalls | Number of calls to the method |
| Name | Name of the method |

### 3.3.4 Step 4: Classifiy the results

By looking at the execution time of a method, the problem can be categorized:

#### Complex computations

The method shows a high amount of CPU time and an almost equal elapsed time. This shows that the method is really busy doing its work.

#### High latency

The method shows a high amount of elapsed time and significantly lower value for the actual CPU time. Therefore it is waiting for another task to be completed most of the time. This may either be a complex database operation, or another thread that is holding a monitor (contention) or an arbitrary I/O operation.

### 3.3.5 Step 5: Set up Action Items

After the general problem has been named, it is necessary to look for the reasons.

#### *3.3.5.1 High latency*

Most of the problems come from high latencies the application is suffering. These can be spotted by comparing the elapsed time with the CPU time. When the elapsed time is significantly higher than the CPU time, the reason for the loss of performance does not lie in the computations performed inside the application’s thread. Instead the application is waiting for another party to complete.

Common reasons are contention, broad queries and a bad system configuration.

#### Contention

Sometimes a method requires exclusive access to some object or resource that it does not own. Depending on the competition, it may spend most of the time in waiting for another thread to release it. This kind of bottleneck is a performance bug that prevents an application from being scalable. When the object is shared with the infrastructure or other applications the entire portal installation may suffer. In both cases the iView sourcecode has to be inspected in detail in order to find and remove the bottleneck.

#### Broad Queries

When an external database or R/3 repository is queried for information, a lot depends on efficiency of the query. When the query (like an SQL statement) is too general, the time needed to perform it is very long. As a result the calling thread is spending most of the time waiting for the results to come in from the external system. The method takes too long to execute and the connected system may be overloaded with queries. The iView sourcecode has to be inspected for complex queries using JDBC, JCo or other middleware technology.

#### Bad System Configuration

High latencies during the execution of a method may come from slow connections to an external system or from an overloading of the system itself. The iView *and* the environment have to be inspected in order to find the responsible subsystem. Every stage of the portal request has to be examined for performance leaks.

#### 3.3.5.2 Complex computations

Applications that perform complex computations can be found by comparing the elapsed time with the CPU time. When the CPU time is close to the elapsed time, the method is spending most of the time being busy. While this may be normal for scientific applications, expensive computations are a rare exception in portal applications. So More likely is a defect in the iView.

Examples of such defects are:

- **Busy waiting:** A status is queried over and over to observe a change. Correct implementations would register for some event or signal in order to be notified and sleep in the meantime. In broken implementations the sleeping phase is missing or broken, leading to continuous requeries (polling).
- **Inefficient algorithms:** An application that searches for a record using linear lookup algorithms is wasting CPU time. Intelligent usage of well-known algorithms for data processing are a prerequisite for responsive applications. The iView should be checked in a performance codereview.

#### 3.3.5.3 Missing caches

Some methods are inherently complex. They require a lot of time to be executed and there is no way to accelerate them substantially. In this case caching is a helpful strategy to avoid repeated execution of these expensive methods. When an expensive operation is called several times with the same parameters yielding the same result, a missing cache is a defect that may seriously degrade the overall performance of the system.

# 4 Fundamental Sherlok Concepts

## 4.1 Memory Profiling Concepts

### 4.1.1 Basic Concepts

Sherlok allows to monitor memory consumption at different levels. By recording the point of memory allocations and relating it to garbage collection events, memory consumption can be traced back to the source, i.e. a class. In order to limit the amount of data, Sherlok allows to limit profiling to a defined set of classes (the *scope*) and aggregates all allocations that happen outside.

This chapter will outline a procedure to isolate the classes responsible for out-of-memory situations. The following values will be used to monitor memory consumption:

| | |
|---|---|
| Allocated memory | The amount of memory newly requested during the time-span between two GC cycles |
| Deallocated memory | The amount of memory released during a GC cycle. This memory is called garbage. |
| Total memory | The amount of memory that survived the GC cycles. In general this should be the contents of caches and pools and the application state itself. In case of a memory leak, this amount also contains the memory that actually defines the leak, i.e. unused objects. |

In Sherlok all memory allocation is associated with *classes*. To link an allocation with a class, all memory allocation statements (*new* …) in a program are intercepted and the calling method is determined. Now the class containing the method is charged for the allocated amount of memory. For each subsequent invocation the values are added up until the next garbage collection cycle occurs.

## Example

An application for a library implements a simple model-view-controller architecture, consisting of the classes **LibraryView**, **LibraryModel** and **LibraryController** for the MVC part and a class **BookManager** that is used by the model to store data in a database. For some methods the implementation has been omitted for simplicity reasons. These methods are highlighted in *`italics`*. Codelines that have no effect on the results are greyed-out. Note that the source code has only been provided to show the interconnection between Java statements and the values measured by Sherlok. Sherlok can be used though without deeper Java knowledge. It is sufficient to know basic OOP concepts like packages, classes and operations to inspect software with Sherlok.

The memory requirements used in the examples do not reflect the real values that the example program would actually produce. We have chosen simple numbers to make calculations easier.

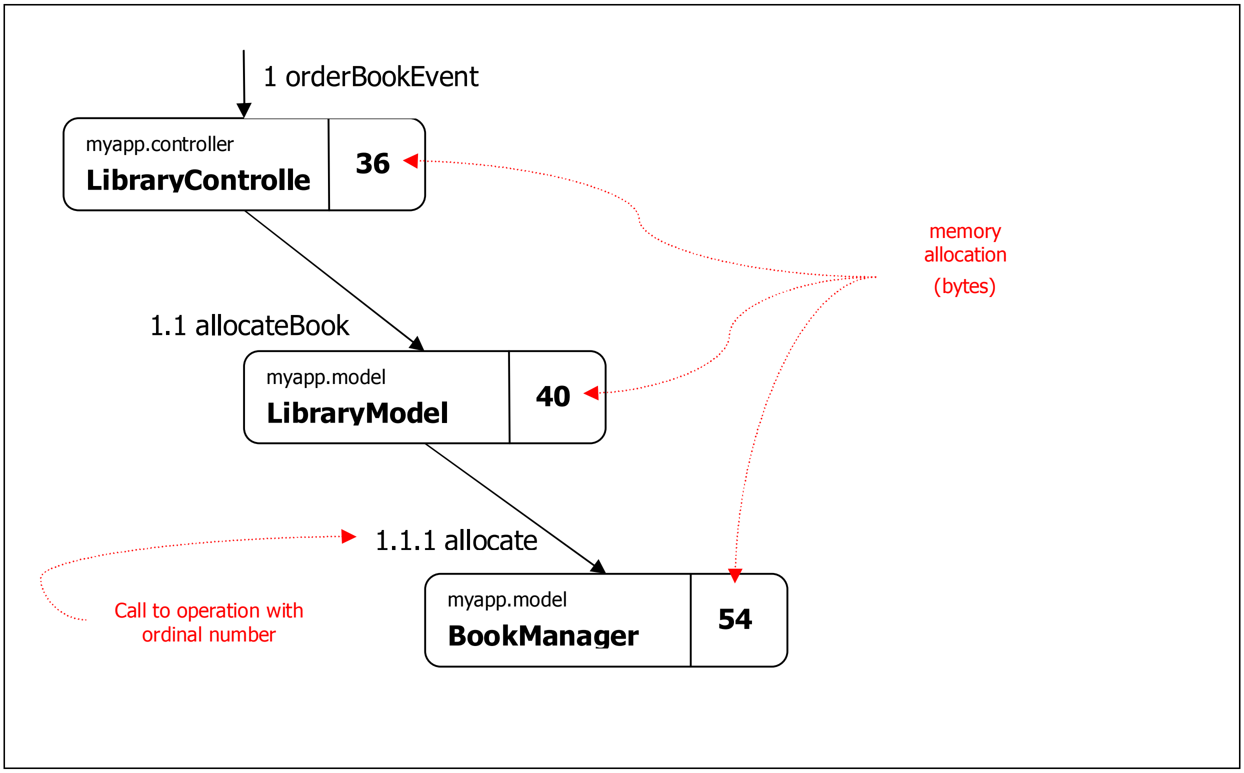
**package myapp.view:**

```
public class LibraryView implements LibraryModelObserver {
  private String _userId;
  private String _status;
  private String _orderId;

  public void bookOut(String isbn) {
    _status = _userId+" allocates "+isbn; // 50 for the string (non-garbage)
    updateTable();
    refresh();
  }

  public void displayOrderId(String orderId) {
    _orderId = orderId;
    refresh();
  }
}
```

**package myapp.model:**

```
public interface LibraryModelObserver {
  void bookOut(String isbn);
}

public class LibraryModel {
  private BookManager _bookman;
  private LinkedList _observers;

  public void allocateBook(String  isbn) {
    _bookman.allocate(isbn); // result is thrown away (becoming garbage)
    Iterator i = _observers.iterator(); // 40 bytes for the Iterator object (garbage)
    while(i.hasNext()) {
      LibraryModelObserver o = (LibraryModelObserver) i.next();
      o.bookOut(isbn);
    }
  }
}

class BookManager {
  private static String ALLOC_STMT = “INSERT INTO ALLOCS (isbn, code) VALUES (?, ?)”;

  private Connection _dbConn;

  String allocate(String isbn) {
    String code = isbn+’#’+System.currentTimeMillis();  // 54 bytes for the string
    storeAllocation(isbn, code);
    return code;
  }

  private storeAllocation(String isbn, String code) {
    PreparedStatement stmt = _dbConn.prepareStatement(ALLOC_STMT);
    stmt.setString(1, isbn);
    stmt.setString(2, code);
    stmt.executeUpdate();
  }
}
```

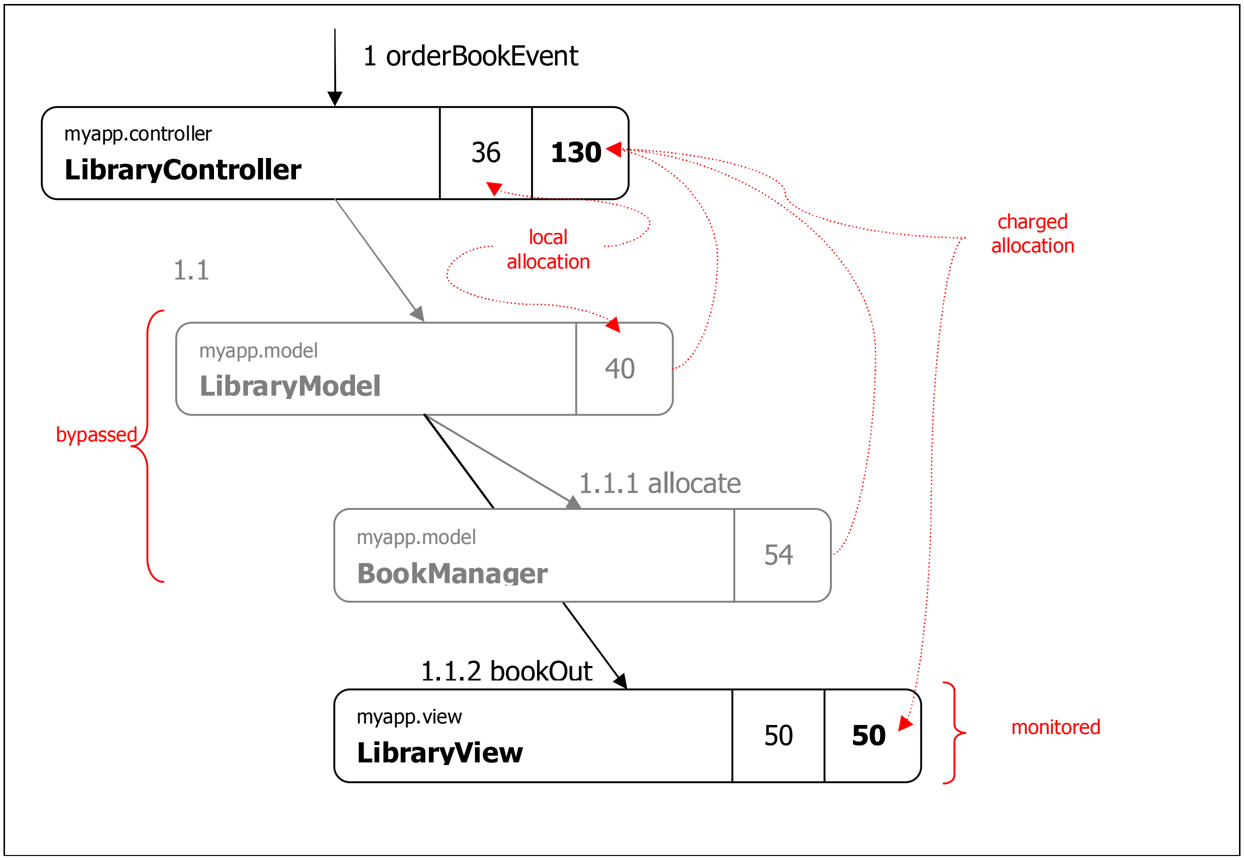
The corresponding call-graph shows, how Sherlok intercepts the memory allocations happening during the call to the method **LibraryController.orderBookEvent**:



### 4.1.2 Scope and Aggregation

From the graph we can easily read the number of bytes each class consumes. It is good to have detailed results, but with applications consisting of hundreds or thousands of classes there is simply too much data to inspect. A better approach is to pick out the most important classes and summarize all allocations that are out of scope. For this matter, Sherlok offers the capability of filtering. A wise filter defines which classes are to be *monitored* for memory consumption. Classes that are not in scope of the filter (i.e. are *bypassed*) do not undergo individual measurement. Their memory allocations are instead propagated up the call-stack until a method of a monitored class is reached. Therefore any monitored class inherits the memory allocation taking place during calls to methods of a *bypassed* class.

For our example we could restrict monitoring to the classes **myapp.controller.*** and **myapp.view.***, bypassing the backend (**myapp.model.***) and platform classes (**java.***, **javax.***, **com.sun.***, …*)*. The result looks like this.

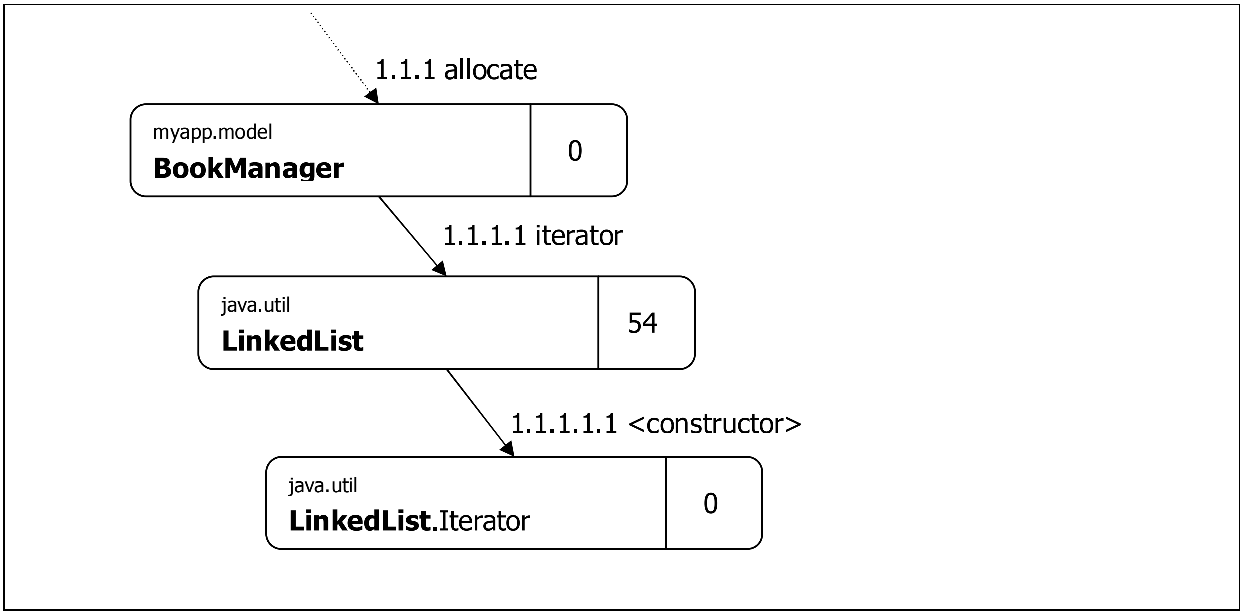


For the picture above the following allocation would be charged to each class:

| Class | Allocated memory (bytes) |
|---|---|
| LibraryController | 130 = 36 + 40 + 54 |
| LibraryView | 50 = 50 |

As can be seen in the picture, all memory that has been allocated by bypassed classes in the package **myapp.model** is assigned to the monitored class **LibraryController**. This is just because those allocations all occurred during the call to **orderBookEvent** method of said class. In other words: the monitored class **LibraryController** has *inherited* the memory allocation from their unmonitored delegates **LibraryModel** and **BookManager**.

When we take a closer at the sourcecode we also discover that the same kind of inheritance applied to class **LibraryModel**. The 40 bytes we noted for the creation of an **java.lang.Iterator** could also have been illustrated in detail:



Sherlok accumulates the allocated size for each monitored class until the next garbage collection cycle occurs. As the allocated memory is linked to its originating class, also the *deallocations* during the GC can be clearly assigned. Now the *total* memory consumption can be calculated by subtracting the deallocated size from the allocated size for each GC cycle and adding the results up.

For a single GC cycle, the amount of *deallocated* memory may well exceed the amount of allocated memory. This is often confusing to the novice, but comes from the fact the allocated and deallocated amounts do generally not refer to the same memory *space* or objects.

The following table shows the allocation and deallocation for the monitored classes in our example. We show the values for two GC cycles and one call to **orderBookEvent** in each cycle. In our example all objects except for the status string are temporary and therefore reclaimed at the end of the cycle.

| GC # | Class | Allocated | Deallocated | Total |
|---|---|---|---|---|
| 1 | LibraryController | 130 | 130 | 0 |
| | LibraryView | 50 | 0[1] | 50 |
| 2 | LibraryController | 130 | 130 | 0 |
| | LibraryView | 50 | 50[2] | 50 |

## Example

When we change line 3 and 7 to collect the status messages we would have created a memory leak and the table would look quit different.

```
03 private StringBuffer _status = new StringBuffer();
07   _status.append(_userId+" allocates "+isbn);
```

---

[1] A reference to the status message is kept in *_status*.

[2] The old status value is reclaimed, due to the field *_status* being overwritten with the new status message.

The memory leak can be easily detected by looking at the total of allocated memory which is constantly rising.

| GC # | Class | Allocated | Deallocated | Total |
|---|---|---|---|---|
| 1 | LibraryController | 130 | 130 | 0 |
| | LibraryView | 100[3] | 50[4] | 50 |
| 2 | LibraryController | 130 | 130 | 0 |
| | LibraryView | 150 | 50 | 100 |
| … | … | … | … | … |
| n | LibraryController | 130 | 130 | 0 |
| | LibraryView | n*50 + 50 | 50 | n*50 |

### Summary

Sherlok Monitors memory allocations (in memory profile mode). Allocations are accumulated to classes and charged to the next monitored class in the stack trace up from the point of allocation. Released instances reduce the memory charged to a class.

## 4.2 Performance Profiling Concepts

Compared to the way Sherlok monitors memory consumption, the way performance is measured is slightly different. For each called method, the execution time is calculated by subtracting the time-of-entry from the time-of-exit for the method. Between the entry and the exit of a method, also the time a thread is in *running state* is recorded. This is called *CPU-time* and reflects the true processing time of a method.

**Thread States**

| | |
|---|---|
| **Running** | A thread is in running state when it is actually executing instructions on the CPU. |
| **Waiting** | A thread is in waiting state when it is waiting for a signal from the operating system. Signals are used to indicate various events, like incoming data (I/O), the availability of a semaphore, the releasing of a monitor. |
| **Ready** | A thread that is in ready state is waiting for the scheduler to gain CPU time. Once the scheduler assigns the CPU to a ready thread it is changing into running state. |

As a consequence each method inherits the execution time from all methods it calls, no matter whether those are monitored or not. In this case the filtering capability already known from the memory profiling section is only needed to determine which methods are actually recorded.

### Example

Let us assume our methods spend the following time in their body and not in some other method they call:

---

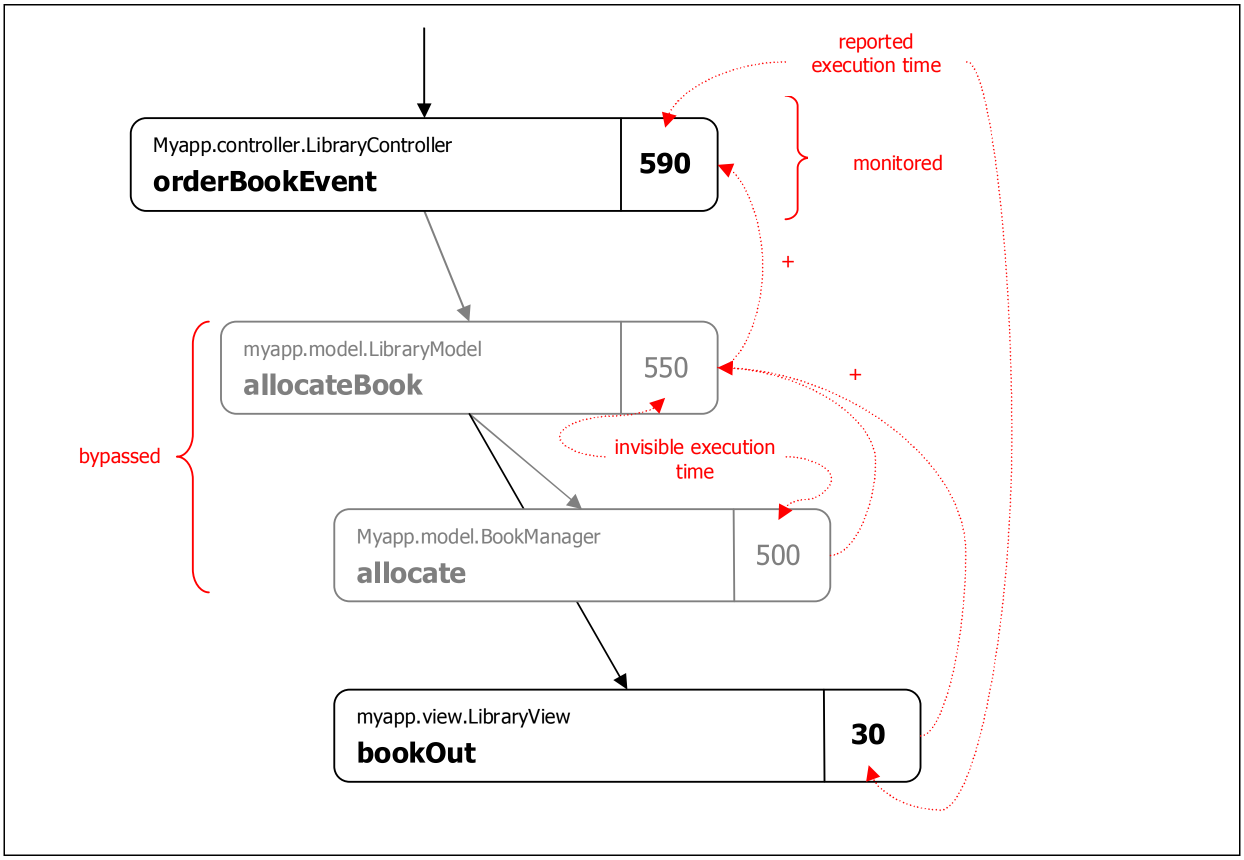
[3] = 50 + 50: 50 for the buffer inside of *StringBuffer*, 50 for the parameter to *append*.

[4] For the parameter to *StringBuffer.append*

| Method | Local execution time |
|---|---|
| LibraryController.orderBookEvent | 40 |
| LibraryModel.allocateBook | 20 |
| BookManager.allocate | 500 |
| LibraryView.bookOut | 30 |

The following results would be delivered by Sherlok:

| Method | Total |
|---|---|
| LibraryController.orderBookEvent | 590 = 40 + *550* |
| LibraryModel.allocateBook | 550 = 20 + *500* + *30* |
| BookManager.allocate | 500 |
| LibraryView.bookOut | 30 |



## 4.3 Tracing Concepts and Events

Beyond the profiling functionality of Sherlok which allows to take an overall picture of the software, tracing allows us to monitor the *execution* of a program itself, i.e. the dynamic behavior. This is done by intercepting certain events of the JVM and showing some output in those cases.

Triggers can be defined based on the following events:

- Garbage collections

- entering or exiting a method
- exceptions
- method execution times exceeding a given limit
- memory allocation exceeds a certain limit
- etc.

Tracing is a very advanced feature of Sherlok, much similar to debugging. For a description of the various events that Sherlok recognizes, visit section 10.

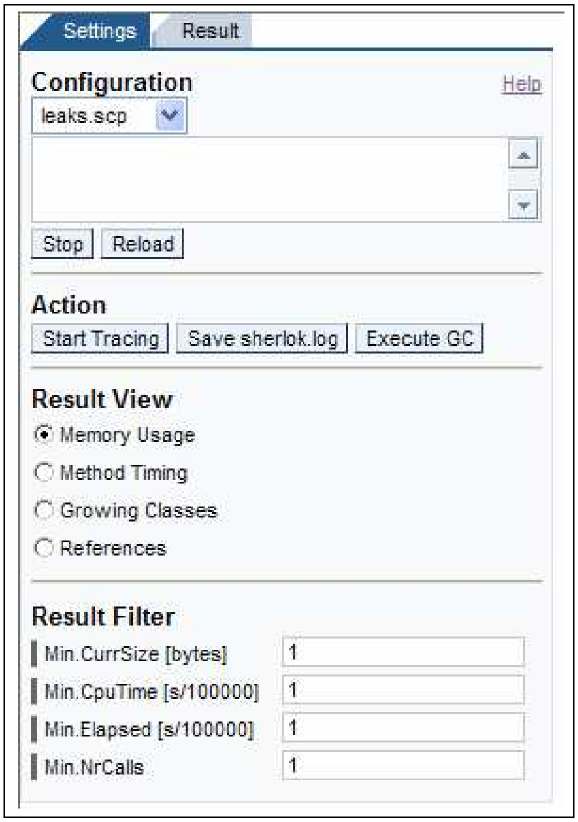
# 5 Monitoring iView

In addition to the bare-bones application **iView TestBench** there is another iView shipping with Sherlok that offers a convenient user interface to observe performance and memory requirements of portal content.

The display contains of two pages. The first page called **Settings** is subdivided in four sections to control different aspects of the Monitor. The second page called **Results** displays the results of the current operation. Its contents are controlled by the various configuration settings and depend on the selected view mode of Monitor.

## 5.1 User Interface

### 5.1.1 Page 1: Settings



Configuration: leaks.scp

Result View
● Memory Usage
○ Method Timing
○ Growing Classes
○ References

Result Filter
Min.CurrSize [bytes]: 1
Min.CpuTime [s/100000]: 1
Min.Elapsed [s/100000]: 1
Min.NrCalls: 1

Picture 4: Monitor Settings

#### 5.1.1.1 Configuration

The configuration area is used to select the configuration file used for monitoring. For every file in the configuration directory[5] there is an entry in the dropdown list. For those configurations carrying a specific description, it is displayed in the text area below. Two buttons control the behavior of Sherlok:

| | |
|---|---|
| Start | Activates the selected configuration and starts profiling. Afterwards the button label is changed to **Stop**. |
| (Stop) | Used to stop profiling again. |
| Reload | Reloads the current configuration from disk and activates it. |

#### 5.1.1.2 Action

The second area contains three buttons for general purposes:

| | |
|---|---|
| Start Tracing | Starts general tracing in Sherlok. When tracing is activated here, all output being delivered to the Monitor display is also recorded in a trace log. After tracing is started this button changes to **Stop Tracing**. |
| (Stop Tracing) | Stops tracing again. |
| Create Sherlog.log | Creates a snapshot of the current trace log and keeps it in temporary storage to be saved later. Then the current trace log is cleared and the button label changes to **Store Sherlog.log**. |
| (Store Sherlok.log) | Stores the snapshot created before in the file `sherlok.log`. |

#### 5.1.1.3 Result View

The Monitor offers four different views on the first page:

| | |
|---|---|
| Memory Usage | Displays the memory usage of individual classes. |
| Method Timing | Displays the execution times and number of calls of methods. |
| Growing Classes | Displays only classes are growing. |
| References | Displays detailed statistics on which objects are allocated by a class. |

#### 5.1.1.4 Result Filter

This section is used to filter the results of the chosen view. Not all filters apply to every view, e.g. a filter on the number of calls will have no effect on the contents of a **Memory Usage** view. The following filters are available:

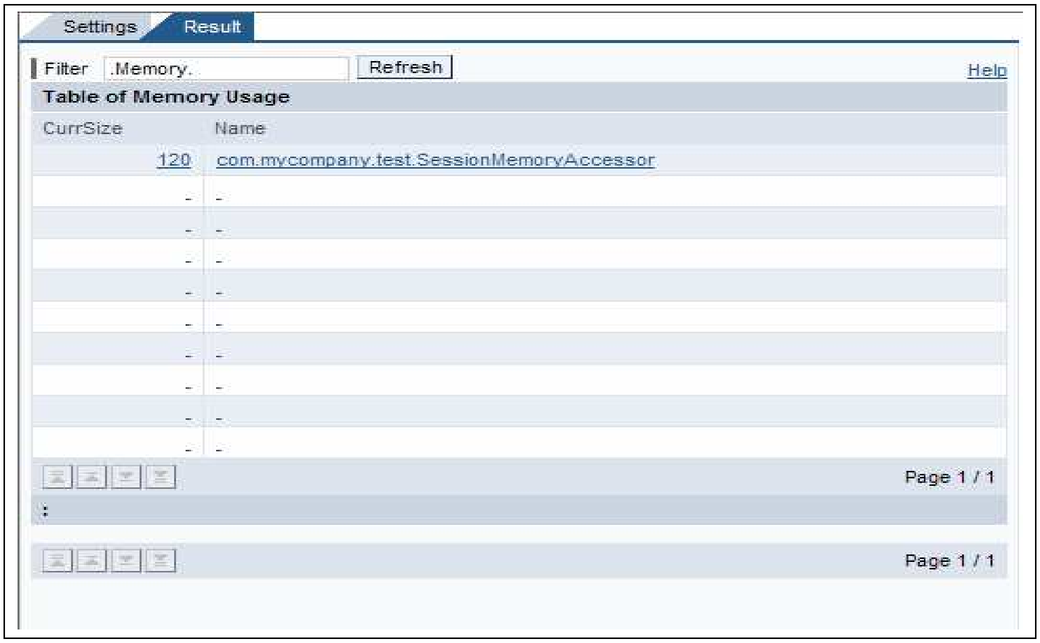
| | |
|---|---|
| Min. CurrSize | Limits the output of the *Memory Usage* and the *References* view to only those classes that claim at least the amount of memory entered here. |
| Min. CpuTime | Limits the output of the *Method Timing* view to only those methods (resp. their classes) that took at least the *CPU-time* (see 4.2) entered here. |
| Min. ElapsedTime | Limits the output of the *Method Timing* view to only those methods (resp. their classes) that took at least the *elapsed time* (see 4.2) entered here. |
| Min. NrCalls | Limits the output of the output of the *Method Timing* view to only those methods (resp. their classes) that were called at least the times entered here. E.g. enter “2” to list only classes with a least one method being called more than once. |

---

[5] see section 11.1.2 for details

## 5.1.2 Page 2: Results

This page contains the results for the selected view. It is divided in a filtering section and the results table itself. The results in the table are filtered by entering a class pattern expression (see 11.1.1) into the textbox and pressing the **Refresh** button.



Filter .Memory.

Table of Memory Usage

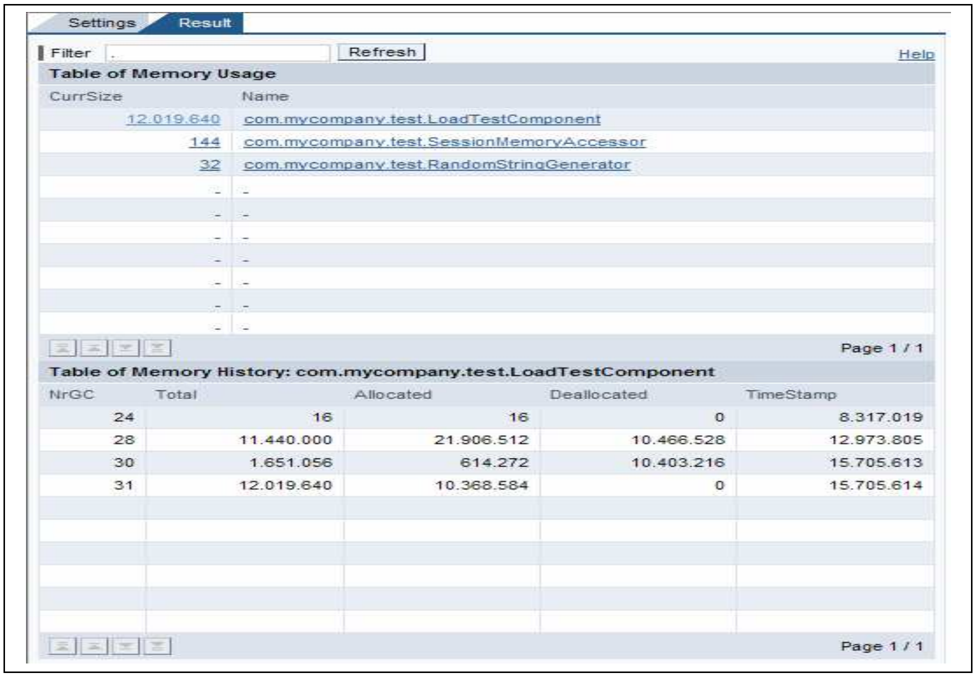
| CurrSize | Name |
|---|---|
| 120 | com.mycompany.test.SessionMemoryAccessor |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |

**Picture 5: Filtering results**

The following sections describe the view types offered in the Results pane.

### *5.1.2.1 Memory Usage Results*

The results for the memory usage view contain a list of classes together with the amount of memory that is currently claimed by the class. By clicking on the display value for an entry, detailed information is shown, containing the history of memory activity for the class.



Filter: .

**Table of Memory Usage**

| CurrSize | Name |
|---|---|
| 12.019.640 | com.mycompany.test.LoadTestComponent |
| 144 | com.mycompany.test.SessionMemoryAccessor |
| 32 | com.mycompany.test.RandomStringGenerator |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |

**Table of Memory History: com.mycompany.test.LoadTestComponent**

| NrGC | Total | Allocated | Deallocated | TimeStamp |
|---|---|---|---|---|
| 24 | 16 | 16 | 0 | 8.317.019 |
| 28 | 11.440.000 | 21.906.512 | 10.466.528 | 12.973.805 |
| 30 | 1.651.056 | 614.272 | 10.403.216 | 15.705.613 |
| 31 | 12.019.640 | 10.368.584 | 0 | 15.705.614 |

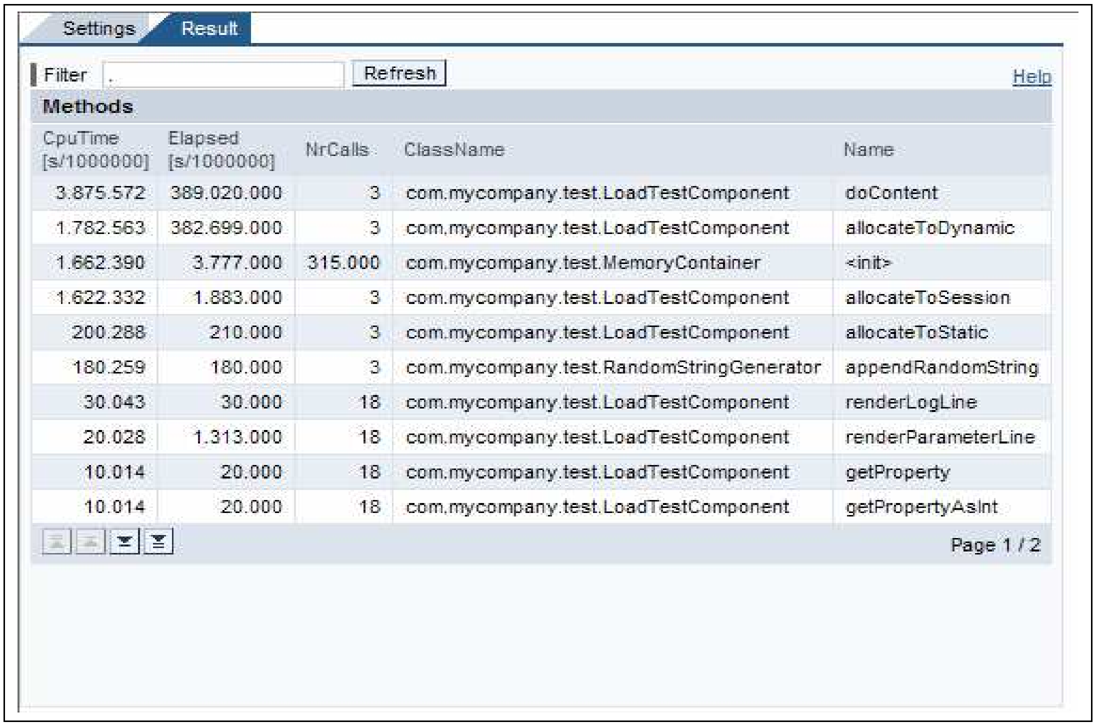
**Picture 6: Memory History view**

The history contains memory allocations and deallocations for the current and privious GC cycles. The size of the table depends on the configuration of Sherlok and can be adjusted with the property **MemoryLimitHistory** (see 11.1.4).

| Column | Description |
|---|---|
| GCNr | Sequence number of the garbage collection.<br>**Note:** This number is assigned by the JVM. Sometimes multiple garbage collections are combined, carrying only the number of the last GC. Therefore there may be gaps in the numbering. |
| Total | The number of bytes that is actually consumed by the class of the application (in bytes). This is the total amount of memory that has been allocated, minus the total of deallocated memory. |
| Allocated | The amount of memory being newly allocated during the GC cycle (in bytes).<br>This number includes all memory that has been requested, no matter if it is still referenced or available for reclaiming. |
| Deallocated | The amount of memory being deallocated during the GC cycle (in bytes).<br>This number includes all memory that has been reclaimed by the garbage collector since the previous garbage collection, no matter if it was allocated in this cycle or in one of the previous ones. |
| TimeStamp | Time stamp value, depending on operating system, with zero meaning current time.<br>The time stamp is a relative value that be used to compare intervals to each other. |

### 5.1.2.2 Memory Timing Results

The memory timing results contain a list of those methods that have been called at least once. Further restrictions may apply according to the filters for CPU-time, elapsed time and number of calls (*Settings* pane).



Filter .

Methods

| CpuTime [s/1000000] | Elapsed [s/1000000] | NrCalls | ClassName | Name |
|---|---|---|---|---|
| 3.875.572 | 389.020.000 | 3 | com.mycompany.test.LoadTestComponent | doContent |
| 1.782.563 | 382.699.000 | 3 | com.mycompany.test.LoadTestComponent | allocateToDynamic |
| 1.662.390 | 3.777.000 | 315.000 | com.mycompany.test.MemoryContainer | <init> |
| 1.622.332 | 1.883.000 | 3 | com.mycompany.test.LoadTestComponent | allocateToSession |
| 200.288 | 210.000 | 3 | com.mycompany.test.LoadTestComponent | allocateToStatic |
| 180.259 | 180.000 | 3 | com.mycompany.test.RandomStringGenerator | appendRandomString |
| 30.043 | 30.000 | 18 | com.mycompany.test.LoadTestComponent | renderLogLine |
| 20.028 | 1.313.000 | 18 | com.mycompany.test.LoadTestComponent | renderParameterLine |
| 10.014 | 20.000 | 18 | com.mycompany.test.LoadTestComponent | getProperty |
| 10.014 | 20.000 | 18 | com.mycompany.test.LoadTestComponent | getPropertyAsInt |

**Picture 7: Method Timing view**

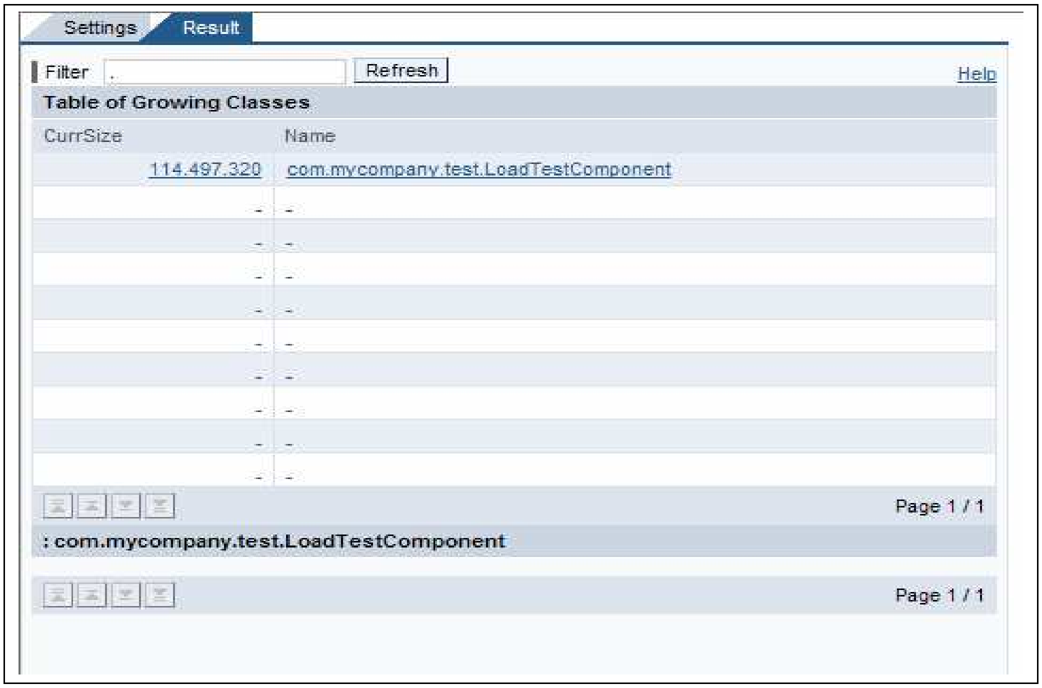
Each of the displayed columns can also be used for sorting by clicking on the column name.

| Column | Description |
|---|---|
| CpuTime | Cumulated CPU time usage<br>This parameter measures the plain *processing* time. This is the time the thread was in *running* state. This is the time it took the method to complete minus the time other threads or the operating system owned the CPU. See 4.2 for details. |
| Elapsed | Cumulated time measured between enter and exit of a method. In short: This is the accumulated execution time of the method.<br>The difference between CPU to elapsed time is the time a thread was in *ready* or *waiting* state. A thread is in waiting state, when it is waiting for a signal of the operating system, like incoming data of an I/O or an object monitor. It is in ready state, when the scheduler has elected other threads to own the CPU. |
| NrCalls | Number of calls to the method |
| ClassName | Name of the class |
| Name | Name of the method |

### 5.1.2.3 Growing Classes

The *Growing Classes* view contains a list of classes that keep on growing from invocation to invocation. These classes are likely candidates for memory leaks.

**Note:** Though the displayed table looks similar to that of the **Memory Usage** view (see 5.1.2.1), no details are offered on the classes displayed.



Filter .

**Table of Growing Classes**

| CurrSize | Name |
|---|---|
| 114.497.320 | com.mycompany.test.LoadTestComponent |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |
| - | - |

Page 1 / 1

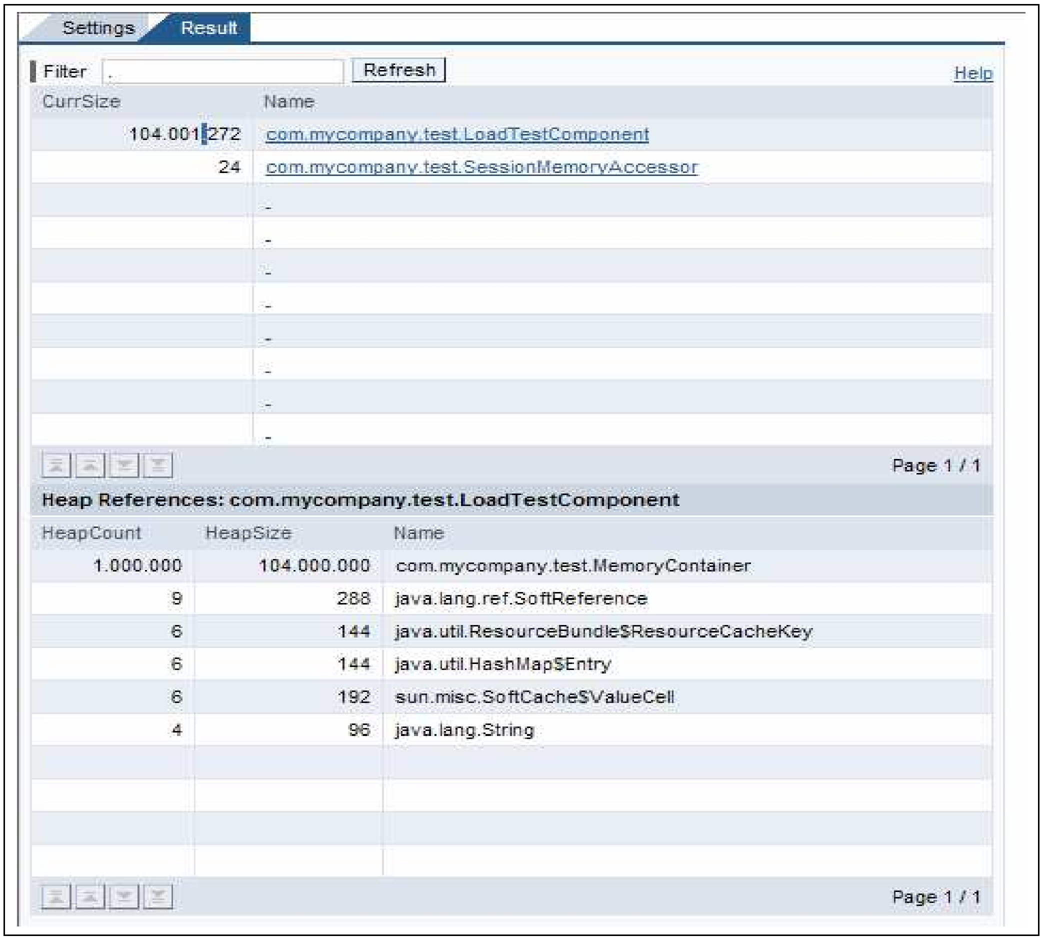
**: com.mycompany.test.LoadTestComponent**

Page 1 / 1

Picture 8: Growing Classes view

### 5.1.2.4 References

This view shows the interdependencies between monitored classes regarding memory allocation. The table contains the list of monitored classes together with the amount of memory they currently claim. After clicking on an entry, the memory consumption is subdivided into individual classes, displayed in a second table below. For every class that was once instantiated by the selected class, the number of created instances plus the total memory is displayed.



| CurrSize | Name |
|---|---|
| 104.001.272 | com.mycompany.test.LoadTestComponent |
| 24 | com.mycompany.test.SessionMemoryAccessor |
| | - |
| | - |
| | - |
| | - |
| | - |
| | - |
| | - |
| | - |

Page 1 / 1

**Heap References: com.mycompany.test.LoadTestComponent**

| HeapCount | HeapSize | Name |
|---|---|---|
| 1.000.000 | 104.000.000 | com.mycompany.test.MemoryContainer |
| 9 | 288 | java.lang.ref.SoftReference |
| 6 | 144 | java.util.ResourceBundle$ResourceCacheKey |
| 6 | 144 | java.util.HashMap$Entry |
| 6 | 192 | sun.misc.SoftCache$ValueCell |
| 4 | 96 | java.lang.String |

Page 1 / 1

**Picture 9: References view**

| Column | Description |
|---|---|
| HeapCount | Number of active instances[6] that have been created by the selected class.<br>This entry shows how many objects of the given class have been allocated and not been deallocated yet. |
| HeapSize | Number of bytes consumed by the active instances referred by **HeapCount** |
| Name | Full qualified class name |

# 5.2 Using Monitor

## 5.2.1 Monitor Memory Requirements

### *5.2.1.1 Prerequisites*

To show the memory consumption of classes, a configuration has to be created that has memory profiling enabled for the classes of interest (see 11.1). Memory profiling is active when

---

[6] of the class denoted by column **Name**

- the configuration property **ProfileMemory** is set to **on** or **all**
- and the *profile scope* is not empty (see 11.1.3).

To obtain a history of memory allocations and deallocations also the type of memory statistics Sherlok generates has to be adjusted. This requires the following settings:

- the configuration property **MemoryStatistic** must be set to *info,*
- the property **MemoryLimitHistory** must be set to the number of GC that Sherlok should remember in the history.

When the history is not enabled then all allocations and deallocations are summarized in a single entry.

#### 5.2.1.2 Procedure

Follow these steps to monitor the memory consumption of classes:

1. Go to the page **Settings**
2. Select a configuration from the dropdown-list **Configuration** supporting memory profiling.
3. Make sure that profiling is active (**Stop** command is displayed in button)
4. Perform some activity to create data to measure (on a page, in an iView, …).
5. Select the Result View **Memory Usage**.
6. Adjust the filter **Min. CurrSize** to prevent small classes from being displayed.
7. Switch to the page **Results** to show the results
8. Filter the results by entering a *class pattern* (see 11.1.1.1) in the text-box an performing **Refresh**.

Use the action **Execute GC** to trigger garbage collections and thus gain additional entries in the history. Go back to 5.1.2.1 for a description of the table.

> The Memory Usage view will not work when memory profiling is turned off in the chosen configuration. Then the results will be empty.

### 5.2.2 Show Method Timing

#### 5.2.2.1 Prerequisites

To show the performance statistics for monitored methods, a configuration has to be created that has the timer facility enabled for the classes of interest (see 11.1). The timer is active when

- the configuration property *Timer* is set to *on*,
- or the property **TimerMethods** contains at least one method that is also part of the *profile scope* (see 11.1.3).

For Windows you can choose to set Timer to hpc (high precision count). This mode implements the RDTC timer and allows you to evaluate times more accurately.

For AIX the timer will not work properly, because the JVM does not implement this feature.

#### 5.2.2.2 Procedure

Follow these steps to measure the performance of methods:

1. Go to the page **Settings**
2. Select a configuration from the dropdown-list **Configuration** that has the timer enabled.
3. Make sure that profiling is active (**Stop** command is displayed in button)
4. Perform the action you want to measure (on a page, in an iView, …).

5. Select the Result View **Method Timing**.
6. Adjust the filters (except for **Min. CurrSize**) to select methods according to your preference.
7. Switch to the page **Results** to show the results
8. Filter the results by entering a *class pattern* (see 11.1.1.1) in the text-box an performing **Refresh**.

See 5.1.2.2 for a description of the result table.

> Note: Performance monitoring will not work when the timer is not activated in the chosen configuration. Then the results will be empty.

## 5.2.3 Find Growing Classes

Classes with steadily growing memory consumption are likely candidates for memory leaks. The Monitor offers a special view that shows only those classes that are supposed to create a memory leak.

### 5.2.3.1 Prerequisites

To see ever-growing classes of classes, a configuration has to be created that has memory alerting enabled for the classes of interest (see 11.1). This is true when

- the configuration property *ProfileMemory* is set to *on* or *all*,
- the *profile scope* is not empty
- and the property *MemoryStatistic* is set to *alert*

### 5.2.3.2 Procedure

Follow these steps to monitor the memory consumption of classes:

1. Go to the page **Settings**
2. Select a configuration from the dropdown-list **Configuration** supporting memory alerting (see 5.2.3.1)
3. Make sure that profiling is active (**Stop** command is displayed in button).
4. Repeat the following steps at least ten times:

a. Use the iView

b. Press **Execute GC**

5. Select the Result View **Growing Classes**.
6. Switch to the page **Results** to show the results
7. When the result list is empty go back to step 4.
8. Filter the results by entering a *class pattern* (see 11.1.1.1) in the text-box an performing **Refresh**.

See 5.1.2.3 for a description of the table.

> Note: The Growing Classes view will not work when memory profiling is turned off in the chosen configuration or the memory statistic is not set to alert mode. Then the results will be empty.

### 5.2.4 Decompose Class Memory Usage

The total memory usage of a class can be decomposed into smaller units. For each new instance of a class, Sherlok remembers the creator (see 4.1).

#### *5.2.4.1 Prerequisites*

To show, where the memory a class actually consumes, goes, a configuration has to be created that has memory profiling enabled for the classes of interest (see 11.1.3). Memory profiling is active when

- the configuration property *ProfileMemory* is set to *on* or *all*
- and the *profile scope* is not empty.

#### *5.2.4.2 Procedure*

Follow these steps to monitor the memory consumption of classes:

1. Go to the page **Settings**
2. Select a configuration from the dropdown-list **Configuration** supporting memory profiling.
3. Make sure that profiling is active (**Stop** command is displayed in button)
4. Perform some activity to create input data to measure (on a page, in an iView, …).
5. Select the Result View **References**.
6. Adjust the filter **Min. CurrSize** to prevent small classes from being displayed.
7. Switch to the page **Results** to show the results
8. Filter the results by entering a *class pattern* (see 11.1.1.1) in the text-box an performing **Refresh**.

Go back to 5.1.2.4 for a description of the table.

> Note: The References view will not work when memory profiling is turned off in the chosen configuration. Then the results will be empty.

# 6 JARM Integration

Since version 1.4.0.10 there is a JARM (Java Application Response-time Measurement) integration. You can choose JARM profiling by the new button on the “Action” pane (see Monitor Settings) or set profile mode in telnet command line. Now memory is accumulated to request level and performance measurement is done on component level. Refer to JARM documentation, how to prepare your sources.

## 6.1 Using JARM Instrumentation

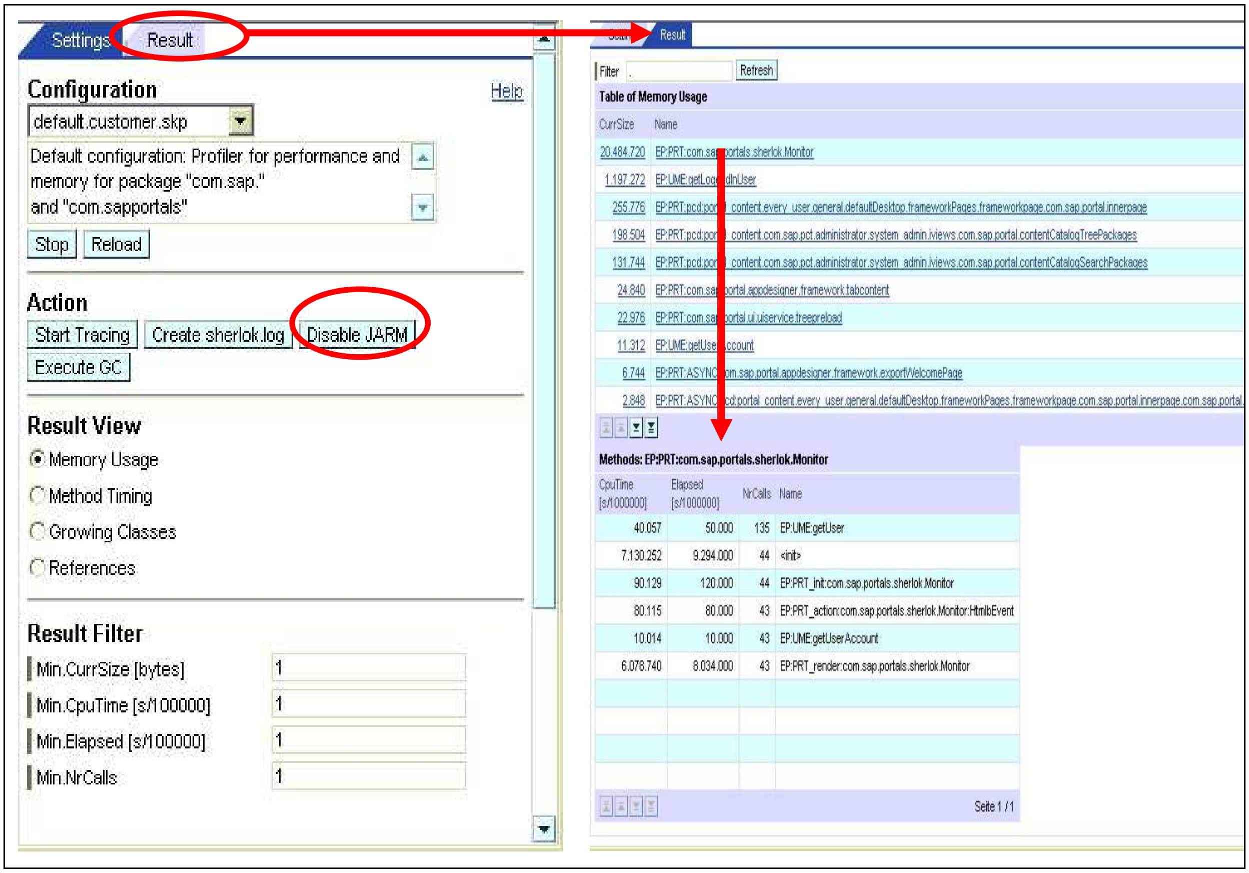
All profile setting will now effective on the nomenclature of JARM replacing “Package” by “Request” and “Method” by “Component”. The following property is valid with JARM:

```
> set ProfileMode=jarm
> set ProfileMethod=.Monitor.EP:PRT_render:.
> reset –s
```

This will filter the following request-component pair

```
Request:        EP:PRT:com.sapportals.sherlok.Monitor
Component:      EP:PRT_render:com.sap.portals.sherlok.Monitor
```

All trace functionality and filter will be applicable to requests and components. Now its possible to run the memory leak detector for JARM instrumented code.



**Configuration**

default.customer.skp

Default configuration: Profiler for performance and memory for package "com.sap." and "com.sapportals"

**Action**

**Result View**

● Memory Usage
○ Method Timing
○ Growing Classes
○ References

**Result Filter**

Min.CurrSize [bytes] 1
Min.CpuTime [s/100000] 1
Min.Elapsed [s/100000] 1
Min.NrCalls 1

Filter .

**Table of Memory Usage**

| CurrSize | Name |
|---|---|
| 20.484.720 | EP:PRT:com.sa portals.sherlok.Monitor |
| 1.197.272 | EP:UME:getLog dInUser |
| 255.776 | EP:PRT:pcd:por l_content.every_user.general.defaultDesktop.frameworkPages.frameworkpage.com.sap.portal.innerpage |
| 198.504 | EP:PRT:pcd:por l_content.com.sap.pct.administrator.system_admin.iviews.com.sap.portal.contentCatalogTreePackages |
| 131.744 | EP:PRT:pcd:por l_content.com.sap.pct.administrator.system_admin.iviews.com.sap.portal.contentCatalogSearchPackages |
| 24.840 | EP:PRT:com.sa portal.appdesigner.framework.tabcontent |
| 22.976 | EP:PRT:com.sa portal.ui.uiservice.treepreload |
| 11.312 | EP:UME:getUse ccount |
| 6.744 | EP:PRT:ASYNC om.sap.portal.appdesigner.framework.exportWelcomePage |
| 2.848 | EP:PRT:ASYNC cd:portal_content.every_user.general.defaultDesktop.frameworkPages.frameworkpage.com.sap.portal.innerpage.com.sap.portal. |

**Methods: EP:PRT:com.sap.portals.sherlok.Monitor**

| CpuTime [s/1000000] | Elapsed [s/1000000] | NrCalls | Name |
|---|---|---|---|
| 40.057 | 50.000 | 135 | EP:UME:getUser |
| 7.130.252 | 9.294.000 | 44 | <init> |
| 90.129 | 120.000 | 44 | EP:PRT_init:com.sap.portals.sherlok.Monitor |
| 80.115 | 80.000 | 43 | EP:PRT_action:com.sap.portals.sherlok.Monitor:HtmlbEvent |
| 10.014 | 10.000 | 43 | EP:UME:getUserAccount |
| 6.078.740 | 8.034.000 | 43 | EP:PRT_render:com.sap.portals.sherlok.Monitor |

Seite 1 / 1

**Picture 10: Monitor Settings**

The button to change modes toggles Enable/Disable JARM. The profile has to be adapted to the naming syntax in the current mode. The following settings where used for the example above:

```
ProfileScope    = .
ProfilePackages = EP:PRT.
```

## 6.2 Using Sherlok-Context API

It’s also possible to use the context API of Sherlok, which is used by JARM, explicit in you program, to get the CPU time for a specific task. To use this interface, you have to collect some JAR files, which are hidden in several distribution packages:

- com.sap.portals.sherlokcore.jar from parts/servlet/iViewEP6/com.sap.portals.sherlok.par

The following context API of class SherlokAts allows you to define requests and subsequent context calls. The interface returns the CPU time in nanoseconds between enter and exit calls of a given component.

```
package com.sap.portals.runtime.profiler;

public SherlokAts
{
   public static native void enterContext(String request, String component);
   public static native long exitContext(String request, String component);
   public static native Boolean jniCommand(String command);
}
```

The following example shows how this interface is used in a user program:

```
import com.sap.portals.runtime.profiler.*;
……
Test () {
   static void main(String[] args) {
      System.loadLibrary(“sherlok”);
      Test t = new Test();
      t.doTest();
   }
}
void doTest() {
   long cpuTime = 0;
   try {
      SherlokAts.enterContext(“myRequest”, null);
      SherlokAts.enterContext(“myRequest”, “ctx1”);
      doSomething();
   finally {
      cpuTime = SherlokAts.exitContext(“myRequest”, “ctx1”);
      SherlokAts.exitContext(“myRequest”, null);
   }
   System.out.println(cpuTime);
}
……
```

A context is given by its unique request- and component name. The request name can be used to sort and filter the results. It’s possible to nest context calls, but you have to take care, that the order of exit statements are kept in reverse order of the enter statements.

To call your program with Sherlok as profiler enter the following command line:

```
java –Xdebug –Xrunsherlok:ProfileMode=jarm,ProfilePackages=. ↲
     –classpath com.sap.portals.sherlokcore.jar;. Test
```

The setting of ProfileMode to **jarm** implies the following parameter as default

- Timer = hpc
- ProfileMemory = off
- ProfileStart = yes

You can save the result statistics in a log file using the SherlokAts class. The jniCommand interface allows you to use the whole set of commands, which are documented for the telnet console.

If you trigger the calls for enterContext and exitContext in different threads, you need to set the ProfileMode to **ats**. In this mode Sherlok uses only one global stack to register context information and not thread local contexts. This mode supports client-server test architectures.

# 7 Tool Integration

Its possible to submit commands to Sherlok with the query part of the URL. This enables the integration into test tools and load generators. For the servlet the following query will write a list of classes to the log file:

```
http://localhost:5100/sherlok/servlet/Monitor?lsc%20-m100
```

# 8 Memory Profiling

Sherlok allows you to see where memory is allocated, specifically which classes are responsible for allocating the memory. You can see the number of objects, total memory space used, and also use a *leak detection* to find potential memory leaks. For basic memory profiling concepts see section 4.1 above. In this section we will now describe how to use the memory monitoring features of Sherlok.

We will first describe what the Sherlok **Monitor** iView can show you and how to use it, then describe some commands that are available through the console mode only. Finally section 8.4 shows how to use all features to track down memory problems.

You may have of several memory related problems in your system:

- Components/classes use far too much memory for *temporary* objects, i.e. generate too much garbage and thereby trigger too many GCs. Also, since they use very much memory during servicing a request, parallel threads increase the likelyhood for an out-of-memory. The focus here must be to find the 'top memory users' and change the code to improve them.
- Components / classes may also *hold* too much memory, i.e. maybe not even generate much garbage, but keep too much memory alive (with strong references), i.e. by letting caches grow without limits, or by having a memory leak. This results in only few components to be able to run in the same JVM (address space). Again we need to identify these and trigger the code changes to improve them.
- Finally you can have an *out-of-memory* situation that is caused by a combination of 'bad components' as described above and several (too many?) threads needing memory at the same time. Here the focus of Sherlok is to find out who exactly caused the OOM by holding too much memory. (note that the class asking for the few bytes that triggered the OOM is generally not the culprit).

For the first two cases, we recommend to measure and optimize memory consumption (static and dynamic) during development and testing – one iView at a time.

In case you have to 'find the bad iView' in a running installation, you may try 'identifying the bad component' as described in section 8.1.1 below. The out-of-memory situation and how to deal with it is described in detail in section …

**Classes Responsible for Memory**

Remember that Sherlok always associates the memory with the monitored class that allocated this memory, not the one that may be holding it. This is important to keep in mind when interpreting measurements.

### 8.1.1 Identifying the “Bad Component”

Normally the best way to measure an iView is to activate memory profiling and then just test this iView (see Testbench above). Sometimes one gets to a situation however, where a system is running 100+ iViews, you have memory problems, and you have to find out in a running system which iView is 'the bad one'.

The normal profile mode (=profile, i.e. by events) is too slow to activate in a running productive system since Sherlok intercepts all methods (entry&exit) to record memory usage. If you can focus the measurement effort on just a few iView classes, you can use the **interrupt** mode. Here Sherlok inserts an interrupt into the bytecode so that only at a few places methods are intercepted and define the memory aggregation points (monitored classes).

To activate this mode, you should perform the following settings in the config file (that you are using):

- set **ProfileMode** to interrupt
- define a as narrow as possible package scope, i.e. list all class/package names that uniquely identify the iViews to be measured
- if you use simple iViews (not HTMLB), the **doContent** method is the perfect entry point to observe
- take the standard excludes as a precaution – it shouldn't be necessary if **ProfilePackages** is defined narrowly enough

```
ProfileMode     = interrupt
ProfilePackages = com.mycompany.iViews.
ProfileMethods  = .doContent
ProfileExcludes = com.sap., com.sapportals., com.inqmy., java., javax., sun., org.xml.
```

Then 'reset' to reload the new config file, and activate monitoring. Now you should see memory summaries being shown for exactly the iViews in focus.

## 8.2 Memory Leak Finder

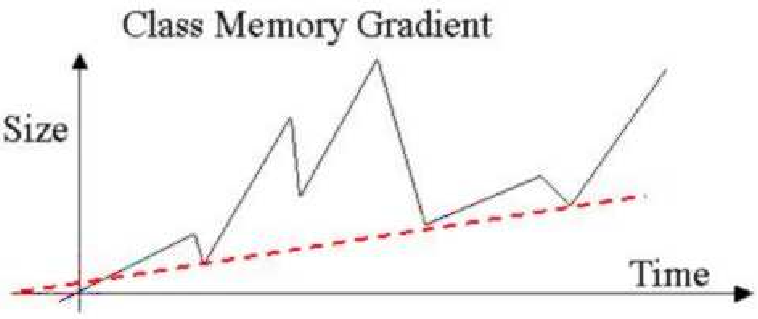
Also in Java it is possible to create memory leaks. A transient component might store data into a static global section like the user sessions and just forget to remove them after terminating. Inefficient cache implementations show the same behavior as memory leaks and should also be in your considerations. In both cases you will lose your most precious resource: memory. And in both cases you want to know who created the memory and put it into the static context, which is shown by Sherlok.

We assume that you already know how to use the memory monitoring and how to get history information on memory usage of classes. The history shows how much memory (allocated, deallocated, retained) a class is responsible for (allocated it) by checking these values after each GC.

A potential memory leaking class will keep more and more memory, i.e. on average deallocate less than it allocates (note again that Sherlok always observes the 'responsible' class that allocated the memory, not another class that may be holding it). Sherlok can automatically observe all monitored classes and report those that are 'growing'.

To just immediately report a growing class is not useful however. For one, a class can allocate some amounts of memory, and only after a few GCs (maybe even a full GC) come back to the initial / stable memory level. Also, depending on how many active threads are just using a class, the 'current memory usage' at a GC event can vary quite a bit. Thus we must only report classes that 'on average' show a growing memory usage.

The intervals (number of GCs) where a class exhibits an equilibrium memory usage can be quite large. Therefore, Sherlok also employs the following heuristic to compress the GC history and only focus on the relevant points: Consider the following memory levels after GCs for a class A of 10, 11, 12, 15, 12, 14, 17, 13 MB (i.e. the class is responsible for that much memory). Then Sherlok only keeps the values 10, 11, 13 in the compressed history, since these are the only 'leak relevant' points. The following diagram shows a typical memory behavior like this.



For leak analysis Sherlok evaluates the gradient of memory size after GC for every profiled class in scope. After a transient oscillation the gradient should be zero for all classes. In any other case the memory will be exhausted at a certain point.

### Interpretation of the results

Note that not all classes reported really have a memory leak. In some cases you simply have a cache that gets filled very slowly and the behavior also depends on the program load (user requests). You must look at each case carefully. In general however, a reported leak that is consistent over a longer running time is most likely a real leak. (Note again that Sherlok reports the class that allocated the memory, i.e. is responsible, not the class that is holding it.)

### 8.2.1 Activating the Memory Leak Finder Mode

To activate the memory leak detector, you have to set config property **MemoryStatistic** to **alert.** It is recommended to leave the **MemoryLimitHistory** at 10 (the default). This is enough to hold the data for the compressed history.

```
MemoryHistory      = alert
MemoryLimitHistory = 10
```

The memory leak detector will only work properly with a sufficient number of garbage collection events. It will store only values in the memory history table, which are relevant for gradient calculation. This is far less, than the number of GC (see details on algorithm above). Therefore also the GCNr field values in the output are not consecutive.

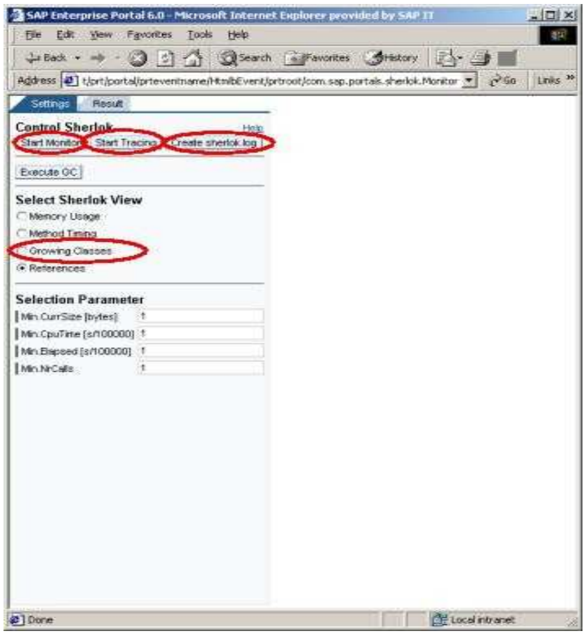
### 8.2.2 Usage via Console

When you have the console open, the Memory Leak Finder writes output for each growing class where it assumes a memory leak according to the algorithm described above. You should always have a log active so you can observe long-running tests.

### 8.2.3 Usage with UI

In the iView Monitor press the buttons **StartMonitor**, **StartTrace** (it is quite unclear why you have to press these buttons to get a memory leak report when you select GrowingClasses below – but for now we can live with that) and **Create sherlok.log** (creates a new log file, i.e. overwrites the current one; where is this documented elsewhere?). Now you can select radio button **GrowingClasses** and the **Result** pane will show all growing classes detected so far.

Update with new UI and screen shot



t/prt/portal/prteventname/HtmlbEvent/prtroot/com.sap.portals.sherlok.Monitor

○ Memory Usage
○ Method Timing
○ Growing Classes
● References

Min.CurrSize [bytes] 1
Min.CpuTime [s/100000] 1
Min.Elapsed [s/100000] 1
Min.NrCalls 1

**Picture 11: iView Monitor Memory Leak Analysis**

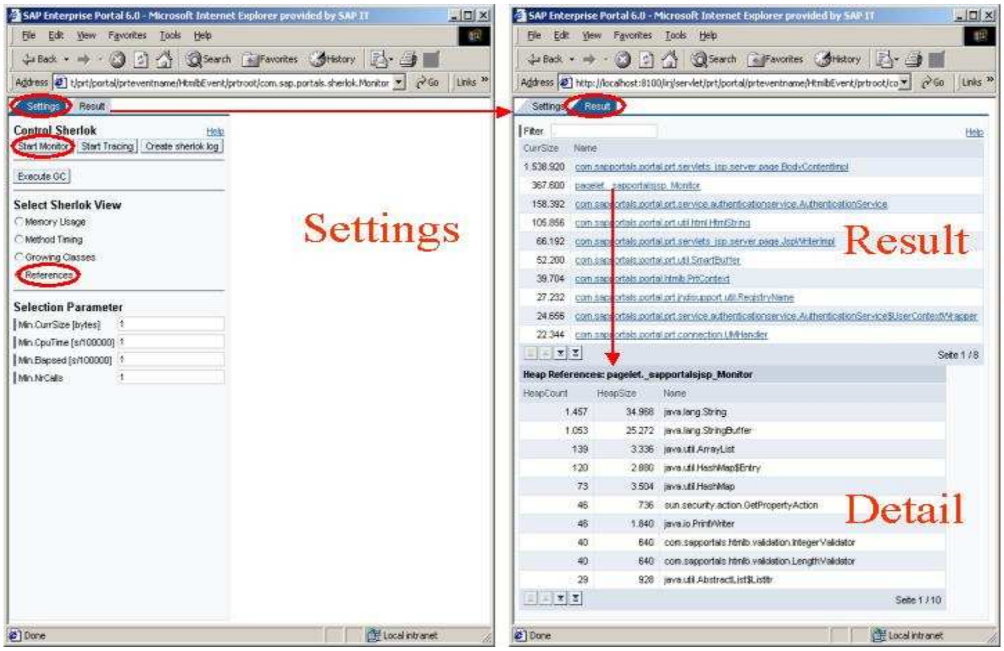
## 8.3 Quick Heap Dump

Once you have found a growing class, you might want to find out, which kind of objects are involved. 'Involved' means which objects were allocated by the class that is 'responsible'.

When you use the Monitornig iView you can select the class you want to see the 'allocated by this class objects' and then get the result shown in the table below.

The detail view shows the following entries:

| ***8.3.1.1 Column*** | ***8.3.1.2 Description*** |
|---|---|
| HeapCount | The number of objects of a kind in all memory heap, for which the component is responsible |
| HeapSize | Cumulated size of all objects of a kind |
| Name | Class name |


SAP Enterprise Portal 6.0 - Microsoft Internet Explorer provided by SAP IT
File Edit View Favorites Tools Help
Back Search Favorites History
Address t/prt/portal/prteventname/HtmlbEvent/prtroot/com.sap.portals.sherlok.Monitor
Go Links
Settings
Result
Control Sherlok
Help
Start Monitor
Start Tracing
Create sherlok log
Execute GC
Select Sherlok View
Settings
Done
Local intranet
SAP Enterprise Portal 6.0 - Microsoft Internet Explorer provided by SAP IT
Address http://localhost:8100/irj/servlet/prt/portal/prteventname/HtmlbEvent/prtroot/co
Filter
Result
Seite 1 / 8
Detail
Seite 1 / 10
Done
Local intranet

Select Sherlok View

○ Memory Usage
○ Method Timing
○ Growing Classes
● References

Selection Parameter

| | |
|---|---|
| Min.CurrSize [bytes] | 1 |
| Min.CpuTime [s/100000] | 1 |
| Min.Elapsed [s/100000] | 1 |
| Min.NrCalls | 1 |

| CurrSize | Name |
|---|---|
| 1.538.920 | com.sapportals.portal.prt.servlets_jsp.server.page.BodyContentImpl |
| 367.600 | pagelet._sapportalsjsp_Monitor |
| 158.392 | com.sapportals.portal.prt.service.authenticationservice.AuthenticationService |
| 105.856 | com.sapportals.portal.prt.util.html.HtmlString |
| 66.192 | com.sapportals.portal.prt.servlets_jsp.server.page.JspWriterImpl |
| 52.200 | com.sapportals.portal.prt.util.SmartBuffer |
| 39.704 | com.sapportals.portal.htmlb.PrtContext |
| 27.232 | com.sapportals.portal.prt.jndisupport.util.RegistryName |
| 24.656 | com.sapportals.portal.prt.service.authenticationservice.AuthenticationService$UserContextMapper |
| 22.344 | com.sapportals.portal.prt.connection.UMHandler |

**Heap References: pagelet._sapportalsjsp_Monitor**

| HeapCount | HeapSize | Name |
|---|---|---|
| 1.457 | 34.968 | java.lang.String |
| 1.053 | 25.272 | java.lang.StringBuffer |
| 139 | 3.336 | java.util.ArrayList |
| 120 | 2.880 | java.util.HashMap$Entry |
| 73 | 3.504 | java.util.HashMap |
| 46 | 736 | sun.security.action.GetPropertyAction |
| 46 | 1.840 | java.io.PrintWriter |
| 40 | 640 | com.sapportals.htmlb.validation.IntegerValidator |
| 40 | 640 | com.sapportals.htmlb.validation.LengthValidator |
| 29 | 928 | java.util.AbstractList$ListItr |

**Picture 12: iView Monitor Heap Dump**

Note: This view obviously doesn't show which reference exactly holds a certain object that is a 'leaked object'. To see this you must use another Profiling tool like e.g. JProbe, OptimizeIt or 'Yourkit'.

### Console Mode

When you use the console mode, there is currently no way to select the class to focus on, thus the console shows all objects of all classes, regardless of who allocated them.

Note: Sherlok uses internal data for heap analysis that is already maintained for monitoring, and is therefore many times faster than the heap dump of typical other tools.

## 8.4 Investigating Memory Problems

This section describes how to analyze an out-of-memory (OOM) situation. This assumes that you are in a support situation where the preventive measurements have not helped to find a 'bad component'.

### 8.4.1 Trapping the Out-of-Memory Situation

Often OOMs happen in a productive system under heavy load. In these situations it is generally not possible to run Sherlok in full memory monitoring mode (that would show memory consumption for all monitored classes) due to the performance impact.

The first task is to find which components were active when the OOM happened, to get an idea of what components were using memory at that time. The problem is that the default behavior of the Java VM for an OOM is as follows: When an OOM first happens, the JVM by itself first does a FullGC to recover memory so that it can continue. If not enough memory can be reclaimed, the JVM throws the 'real OOM' exception. At this point Sherlok requests a full heap dump from the JVM and dumps some more memory statistics to the log file.

Sherlok offers a “OOM trap mode” where it basically incurs no runtime overhead and simply dumps relevant information when an OOM happens. Sherlok does this before the default JVM handling of the OOM happens, i.e. when most relevant information is still available. Moreover, you should also let Sherlok trace garbage collection events and collect the information about them.

To enter this mode, open the console and add these commands:

```
> start trace
> trace add exception
> trace add gc
```

**Note:** The exception handling is independent from the monitor, so you have virtually no impact on performance .

When the application raises an **OutOfMemoryError** exception

- Sherlok requests a full JVM thread dump (like **dt -a** command). This shows you what requests were active at the time of the OOM.
- Sherlok dumps the location and the thread, which caused the exception.
- Sherlok dumps the monitored thread and methods to console (like **dt –c** command)

This information gives you an idea what was happening just before the OOM. The thread (class in the causing thread) causing the OOM is often not the 'bad component' but just the one that hit the wall in the end. The job is to find the classes that are responsible for an extraordinary amount of memory needed.

There are different situations causing **OutOfMemoryError**. To define actions in this situation, you have to figure out, which is the one:

- Heap overflow
- PermSize overflow
- Maximum number of threads reached
- Exceeded addressable application memory (JVM + shared libraries)

## 8.4.2 Heap Overflow

Heap Overflow is the most frequent memory problem occurring within Java applications. There are several reasons why the heap can be exhausted by an application:

- Memory leaks: Classes allocate memory / objects and do not release them (the references to them), so that these objects cannot be collected.
- High peak memory demand: Memory-intensive applications can exhaust available memory in peak-situations without having a substantial need for it.

### *8.4.2.1 High peak memory demand*

Applications with excessive peak memory demand create an overly high number of objects during their processing work without actually requiring them in the long run. Due to the high number of temporary objects (called *garbage*), the memory requirements can rise above acceptable levels, especially when those memory intensive tasks are executed concurrently. The memory is bound by the concurrent threads that do the processing work. The memory footprint could be reduced by (1) reducing the need of dynamic memory (discarded right after use) per request, and (2) by serializing the memory intensive tasks, i.e. reduce the number of parallel tasks that need a lot of memory during processing.

Note: The number of parallel application threads can be set in the SAP J2EE visual admin. The value should not exceed 40 for most application cases.

Excessive memory demand can be spotted by looking at the amount of garbage that is created on each GC cycle:

- Memory intensive applications show high values for both allocated and deallocated memory during each GC cycle.
- The allocation rate of those applications is over-average.

#### *8.4.2.2 Memory leaks*

Memory leaks are created when references to unused objects are kept in some place like a class or another object instance. Another way to create memory - or general resource leaks is by not freeing resources with manual lifecycle management like connections or pool entries. As all these resources need some memory and when the execution of such an application goes forward the free memory is eaten up until it is exhausted. The memory is bound by the data that is held in the application.

In all cases, not freeing a resource means in the end not removing a certain reference that holds the objects. Since these references must be kept somewhere, the prime candidates for memory leaks are *collections* (sets, hashmaps, …) of objects or references.

Memory leaks can be identified by looking for steadily increasing memory consumption, i.e. the gap between newly allocated memory and memory being deallocated during garbage collections.

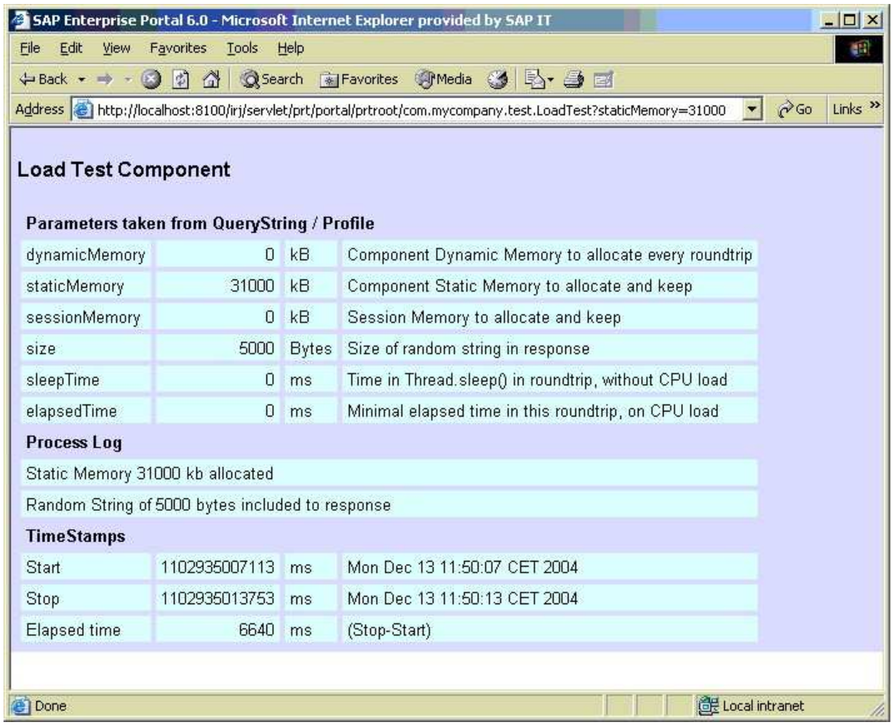
This is shown per monitored class, i.e. the class that is responsible for the allocation. If a class A allocates objects and then hands them over to another class B, which keeps them in some state / collections, class A is still responsible for the memory leak and therefore the right 'first clue'.

To check your application for eventually memory leaks, choose your packages for profiling and start the analysis.

```
> set ProfilePackages=com.sap.,com.mycompany.
> set MemoryStatistic=alert
> set TraceMethods=.LoadTestComponent.doContent{request.getServletRequest.getQueryString}
> reset –s
> start trace
> trace add parameter
> start monitor
```

You need at least ten steps in the following schema:

1. Choose an action on your interface
2. Execute gc command in Sherlok.


SAP Enterprise Portal 6.0 - Microsoft Internet Explorer provided by SAP IT
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go
Links
Done
Local intranet

http://localhost:8100/irj/servlet/prt/portal/prtroot/com.mycompany.test.LoadTest?staticMemory=31000

**Load Test Component**

**Parameters taken from QueryString / Profile**

| | | | |
|---|---|---|---|
| dynamicMemory | 0 | kB | Component Dynamic Memory to allocate every roundtrip |
| staticMemory | 31000 | kB | Component Static Memory to allocate and keep |
| sessionMemory | 0 | kB | Session Memory to allocate and keep |
| size | 5000 | Bytes | Size of random string in response |
| sleepTime | 0 | ms | Time in Thread.sleep() in roundtrip, without CPU load |
| elapsedTime | 0 | ms | Minimal elapsed time in this roundtrip, on CPU load |

**Process Log**

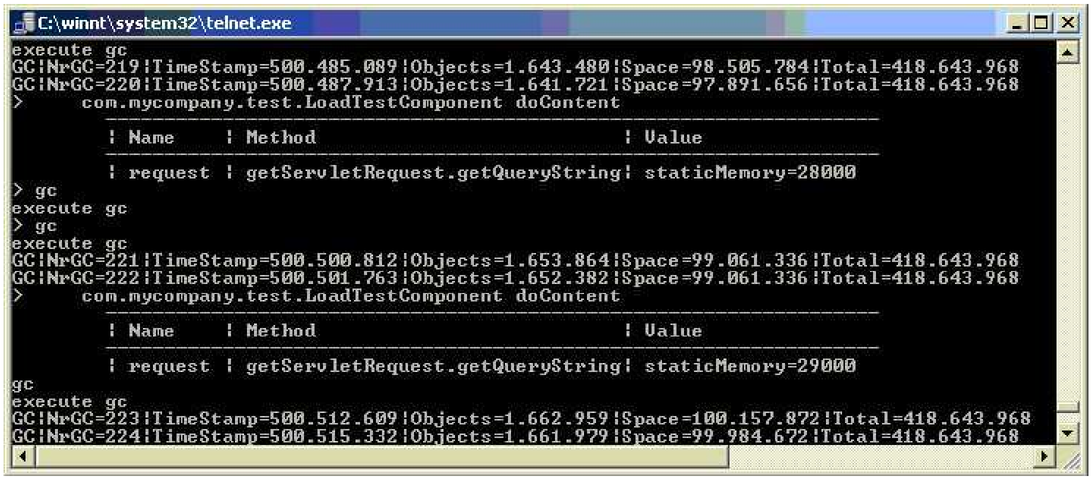
Static Memory 31000 kb allocated

Random String of 5000 bytes included to response

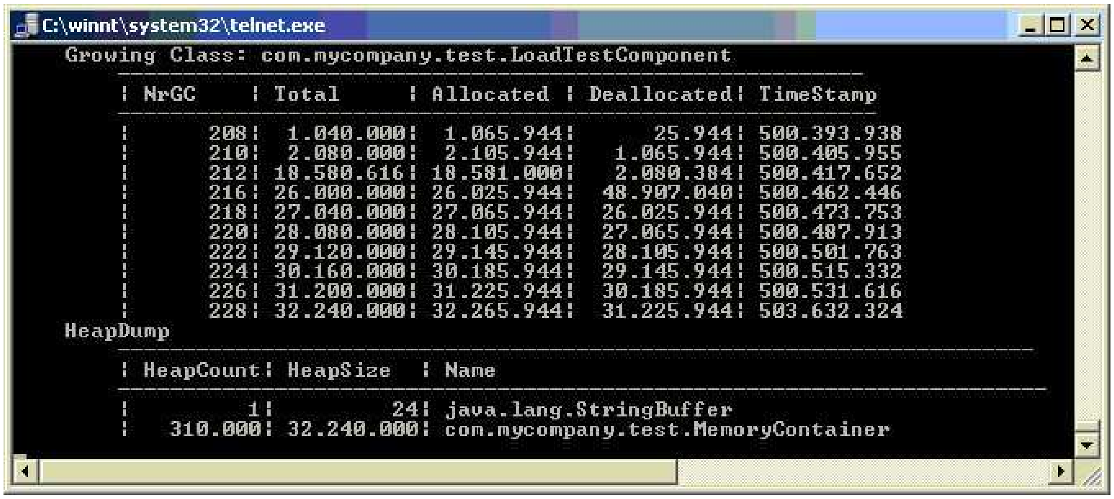
**TimeStamps**

| | | | |
|---|---|---|---|
| Start | 1102935007113 | ms | Mon Dec 13 11:50:07 CET 2004 |
| Stop | 1102935013753 | ms | Mon Dec 13 11:50:13 CET 2004 |
| Elapsed time | 6640 | ms | (Stop-Start) |

This is the load test component. Increasing the “staticMemory” for each step will result into the following output:



```
execute gc
GC|NrGC=219|TimeStamp=500.485.089|Objects=1.643.480|Space=98.505.784|Total=418.643.968
GC|NrGC=220|TimeStamp=500.487.913|Objects=1.641.721|Space=97.891.656|Total=418.643.968
>      com.mycompany.test.LoadTestComponent doContent
        ------------------------------------------------------------------
        | Name     | Method                             | Value
        ------------------------------------------------------------------
        | request  | getServletRequest.getQueryString| staticMemory=28000
> gc
execute gc
> gc
execute gc
GC|NrGC=221|TimeStamp=500.500.812|Objects=1.653.864|Space=99.061.336|Total=418.643.968
GC|NrGC=222|TimeStamp=500.501.763|Objects=1.652.382|Space=99.061.336|Total=418.643.968
>      com.mycompany.test.LoadTestComponent doContent
        ------------------------------------------------------------------
        | Name     | Method                             | Value
        ------------------------------------------------------------------
        | request  | getServletRequest.getQueryString| staticMemory=29000
gc
execute gc
GC|NrGC=223|TimeStamp=500.512.609|Objects=1.662.959|Space=100.157.872|Total=418.643.968
GC|NrGC=224|TimeStamp=500.515.332|Objects=1.661.979|Space=99.984.672|Total=418.643.968
```

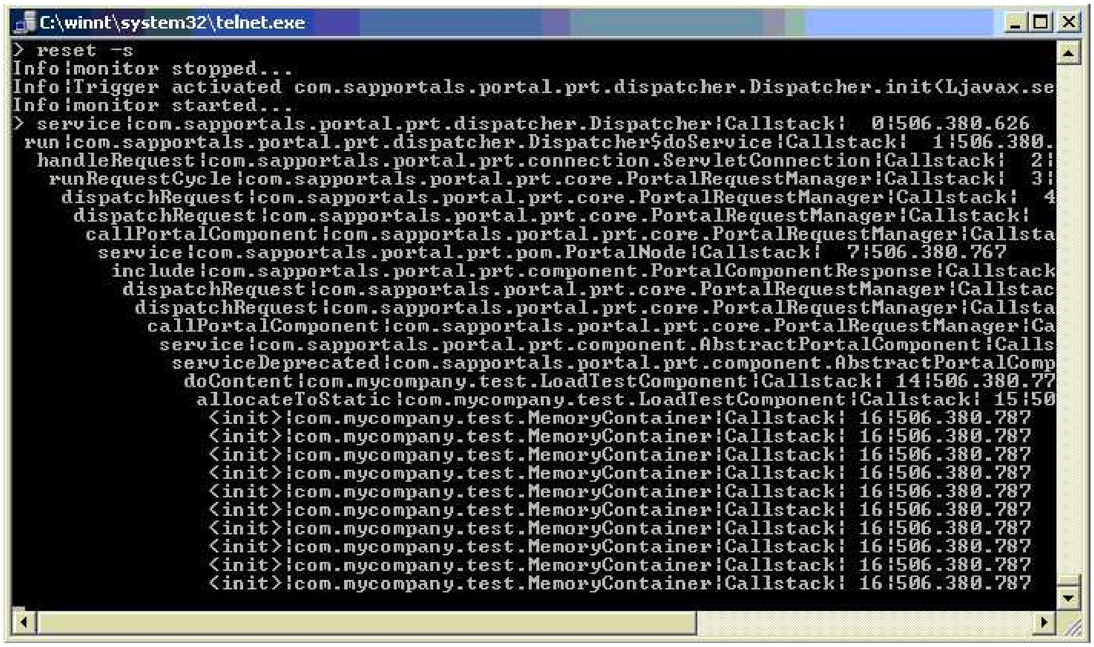


```
    Growing Class: com.mycompany.test.LoadTestComponent
        -----------------------------------------------------------------
        ¦ NrGC     ¦ Total      ¦ Allocated ¦ Deallocated¦ TimeStamp
        -----------------------------------------------------------------
        ¦       208¦  1.040.000¦  1.065.944¦      25.944¦ 500.393.938
        ¦       210¦  2.080.000¦  2.105.944¦   1.065.944¦ 500.405.955
        ¦       212¦ 18.580.616¦ 18.581.000¦   2.080.384¦ 500.417.652
        ¦       216¦ 26.000.000¦ 26.025.944¦  48.907.040¦ 500.462.446
        ¦       218¦ 27.040.000¦ 27.065.944¦  26.025.944¦ 500.473.753
        ¦       220¦ 28.080.000¦ 28.105.944¦  27.065.944¦ 500.487.913
        ¦       222¦ 29.120.000¦ 29.145.944¦  28.105.944¦ 500.501.763
        ¦       224¦ 30.160.000¦ 30.185.944¦  29.145.944¦ 500.515.332
        ¦       226¦ 31.200.000¦ 31.225.944¦  30.185.944¦ 500.531.616
        ¦       228¦ 32.240.000¦ 32.265.944¦  31.225.944¦ 503.632.324
    HeapDump
        ------------------------------------------------------------------------------
        ¦ HeapCount¦ HeapSize   ¦ Name
        ------------------------------------------------------------------------------
        ¦         1¦          24¦ java.lang.StringBuffer
        ¦   310.000¦ 32.240.000¦ com.mycompany.test.MemoryContainer
```

After ten steps Sherlok finds the memory leak and gives this output, which tells the user that LoadTestComponent created 310.000 objects of type MemoryContainer. The short heap dump shows all scalar objects created in the context of the growing class.

Now the next step is to find the context, where objects of type “MemoryContainer” where created. We search for the method “<init>”, which is normally called by the JVM for all scalar objects. We can choose the context, if these elements where created in different sections of the application.

```
> set TraceMethods=/.Dispatcher./.MemoryContainer.<init>
> set TraceTrigger=..Dispatcher.service
> reset –s
> start trace
> trace rem parameter
> trace add stack –tree -p
> start monitor
```



```
> reset -s
Info¦monitor stopped...
Info¦Trigger activated com.sapportals.portal.prt.dispatcher.Dispatcher.init(Ljavax.se
Info¦monitor started...
> service¦com.sapportals.portal.prt.dispatcher.Dispatcher¦Callstack¦  0¦506.380.626
 run¦com.sapportals.portal.prt.dispatcher.Dispatcher$doService¦Callstack¦  1¦506.380.
  handleRequest¦com.sapportals.portal.prt.connection.ServletConnection¦Callstack¦  2¦
   runRequestCycle¦com.sapportals.portal.prt.core.PortalRequestManager¦Callstack¦  3¦
    dispatchRequest¦com.sapportals.portal.prt.core.PortalRequestManager¦Callstack¦  4
     dispatchRequest¦com.sapportals.portal.prt.core.PortalRequestManager¦Callstack¦
      callPortalComponent¦com.sapportals.portal.prt.core.PortalRequestManager¦Callsta
       service¦com.sapportals.portal.prt.pom.PortalNode¦Callstack¦  7¦506.380.767
        include¦com.sapportals.portal.prt.component.PortalComponentResponse¦Callstack
         dispatchRequest¦com.sapportals.portal.prt.core.PortalRequestManager¦Callstac
          dispatchRequest¦com.sapportals.portal.prt.core.PortalRequestManager¦Callsta
           callPortalComponent¦com.sapportals.portal.prt.core.PortalRequestManager¦Ca
            service¦com.sapportals.portal.prt.component.AbstractPortalComponent¦Calls
             serviceDeprecated¦com.sapportals.portal.prt.component.AbstractPortalComp
              doContent¦com.mycompany.test.LoadTestComponent¦Callstack¦ 14¦506.380.77
               allocateToStatic¦com.mycompany.test.LoadTestComponent¦Callstack¦ 15¦50
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
                <init>¦com.mycompany.test.MemoryContainer¦Callstack¦ 16¦506.380.787
```

The result shows, that doContent calls the method “allocateToStatic” which allocates “MemoryContainer”. A look to the source “LoadTestComponent.java” shows the reason: All memory is stored in a list which has a static root.

```
…
private static MemoryContainer m_staticMemory;
…
public void doContent(
          IPortalComponentRequest  request,
          IPortalComponentResponse response)
{
          …

          m_staticMemory = allocateToStaic(staticMemoryVal, m_staticMemory);
          …
 }
```

Check the number of elements and the number of calls to this method with the lsm command:

```
> lsm -f.MemoryContainer.<init>
> lsm –f.allocateToStatic
> lsm –f.doContent
```

### 8.4.3 PermSize Overflow

The PermSize is the space for permanent static class attributes. The PermSize is allocated as extra application memory space, which is not correlated or interfered with the Java memory heap. Its size is set by the JVM parms PermSize as shown below, typically between 64 and 256 MB.

```
-XX:PermSize=192M –XX:MaxPermSize=192M
```

The space needed is proportional to the number of classes loaded, i.e. the number and size of the applications running on that server node. You can use Sherlok to watch the number of classes with the following command:

```
> lss
```

Some applications allow hot deployment (like iViews, …). In this case the class loader is set to zero when a new version of the application is loaded and thus makes the old code (in Perm space) victim for the garbage collector. A new class loader will reload all its classes.

Sherlok allows to watch the state of unloaded classes to keep track of the memory, they are still responsible for:

```
> lsd –m<minimum number of bytes to display>
```

### 8.4.4 Maximum number of Threads

On different operating systems there are different restrictions on the number of threads, which can be handled by one application. Each thread will need by default 500K on memory, so this resource should be used very carefully.

Use the following command to list the actual number of threads. Make sure, that the number does not increase to values bigger than 300, because this will cause also major impact on performance.

```
> lss
```

Note: iView programmers should generally not create threads, definitely not simply create threads with `new Thread()` but always use thread pools.

### 8.4.5 Maximum address Space, DLLs

On different operating systems there are different restrictions on the addressable application memory. On Windows this address space is less than 2GB including all DLLs, PermSize, threads and Java heap. Increasing for example the PermSize will reduce the maximum possible Java heap size.

DLLs also live in the same memory space as the Java process and thus take memory away from the Java heap. This can become a problem when the address space is limited as is the case on Windows (heap size max 1.3 GB). In that case the address space used by DLLs must be taken away from the java heap size.

Sherlok itself is a DLL and also needs memory to run, typically 10-15% of the monitored application if full memory monitoring is used.

It can happen that an OOM occurs when the Java heap is still not 'full' but when a DLL tries to allocate too much memory and thus eats away at the allotted space for the Java heap.

Note: It is very important to make sure the Java process never runs out of physical memory. Make sure that the JVM with heap size + perm size + DLLs all together fits into physical memory. This should be carefully monitored on the operating system level.

# 9 Performance Profiling

For basic performance profiling concepts see section … above.

## 9.1 Latency

Most of the problems come from high latencies the application is suffering. These can be spotted by comparing the elapsed time with the CPU time. When the elapsed time is significantly higher than the CPU time, the reason for the loss of performance does not lie in the computations performed inside the application’s thread. Instead the application is waiting for another party to complete.

### 9.1.1 Contention

Sometimes a method requires exclusive access to some object or resource that it does not own. Depending on the competition, it may spend most of the time in waiting for another thread to release it. This kind of bottleneck is a performance bug that prevents an application from being scalable. When the object is shared with the infrastructure or other applications the entire portal installation may suffer. In both cases the iView sourcecode has to be inspected in detail in order to find and remove the bottleneck.

### 9.1.2 Broad Queries

When an external database or R/3 repository is queried for information, a lot depends on efficiency of the query. When the query (like an SQL statement) is too general, the time needed to perform it is very long. As a result the calling thread is spending most of the time waiting for the results to come in from the external system. The method takes too long to execute and the connected system may be overloaded with queries. The iView sourcecode has to be inspected for complex queries using JDBC, JCo or other middleware technology.

### 9.1.3 Bad System Configuration

High latencies during the execution of a method may come from slow connections to an external system or from an overloading of the system itself. The iView *and* the environment have to be inspected in order to find the responsible subsystem. Every stage of the portal request has to be examined for performance leaks.

## 9.2 Computational Complexity

Applications that perform complex computations can be found by comparing the elapsed time with the CPU time. When the CPU time is close to the elapsed time, the method is spending most of the time being busy. While this may be normal for scientific applications, expensive computations are a rare exception in portal applications. So More likely is a defect in the iView. Examples of such defects are:

### 9.2.1 Busy Waiting

A status is queried over and over to observe a change. Correct implementations would register for some event or signal in order to be notified and sleep in the meantime. In broken implementations the sleeping phase is missing or broken, leading to continuous requeries (polling).

### 9.2.2 Inefficient algorithms

There are many ways to waste time in a program. Inefficient algorithms waste time because they fail to use the fastest path to the solution of a problem. There are many ways to sort a list, but there are only a few ways that offer superior performance.

Sherlok can be used to detect slow methods, but it cannot tell if there is a better solution available. To find inefficient algorithms, two parameters for a method are important:

- **CPU time:** It tells the overall costs of the method.
- **No. of calls:** It can be used to see if some operation is performed more often than expected.

Common cases of inefficient algoriths are:

- **Inappropriate choice of data structures** (e.g. Java Collection Classes**):** Often these can be discovered by looking at the number of calls to the *equals* method. High numbers indicate inefficient algorithms.
- **Repeated evaluation of invariant expressions in a loop**: A high number of calls may point to this problem, too.
- **Back and forth transformations**: These can sometimes be observed by looking at the number of calls to the constructor.

## 9.3 Missing caches

Some methods are inherently complex. They require a lot of time to be executed and there is no way of accelerating them substantially. In this case caching is a helpful strategy to avoid repeated execution of these expensive methods. When an expensive operation is called several times with the same parameters yielding the same result, a missing cache is a defect that may seriously degrade the overall performance of the system.

## 9.4 Garbage Collection Delays

### 9.4.1 High memory allocation and deallocation rate (Full garbage collections)

In Java all objects (complex types) are created on the heap using dynamic memory management, and released automatically some time after the last reference is dropped. This allows for a very simple memory model that avoids the risk of dangling references. On the downside, the simplicity is paid with high costs for memory management. Dynamic allocation of memory is a complex operation, much slower than the creation of variables on the stack. As all objects are created on the heap, operations that create a lot of temporary data suffer a high penalty for dynamic memory management and put high load on the garbage collector.

On a system with high load, the garbage collector may fail to catch up on releasing unused objects, because its thread runs with a lower priority. At some point, the available memory is exhausted and the garbage collector is forced to do a *full garbage collection*[7], a lengthy operation that may interrupt normal program execution for several seconds.

See http://developers.sun.com/techtopics/mobility/midp/articles/garbage/ for complete coverage.

---

[7] as opposed to a *partial garbage collection* that uses heuristics to improve performance

# 10 Application Tracing

Beyond the profiling functionality of Sherlok which allows to take an overall picture of the software, tracing allows us to monitor the *execution* of a program itself, i.e. the dynamic behavior. You can observer certain events and then have certain actions performed at these events. All output goes to the `sherlok.log` and/or web UI.

The most important are these:

- Trace Methods: You can trace methods to see when they are called and even print parameters.
- Trace contentions: You can trace a thread contention, if the thread has to wait on a monitor.
- Stack Traces: When a method is entered / reached you can get a stack trace for that call to see who calls the method
- You can trace exceptions. All unhandled exceptions are traced with additional information. For out-of-memory there are additional actions performed to collect relevant data.
- Trace Trigger: You also have the option to show all calls (program execution) 'underneith' a certain method. This can be triggered by a threshold of 'uses more time than …' and 'more memory than ..'

The trace functions can generate more data than you can possibly analyze ont-the-fly. So its important to have direct control and user input, which makes it unavoidable to use the telnet console interface.

Tracing can be started and stopped in the console with the short command “s”.

## 10.1 Startup

### 10.1.1 Telnet Console

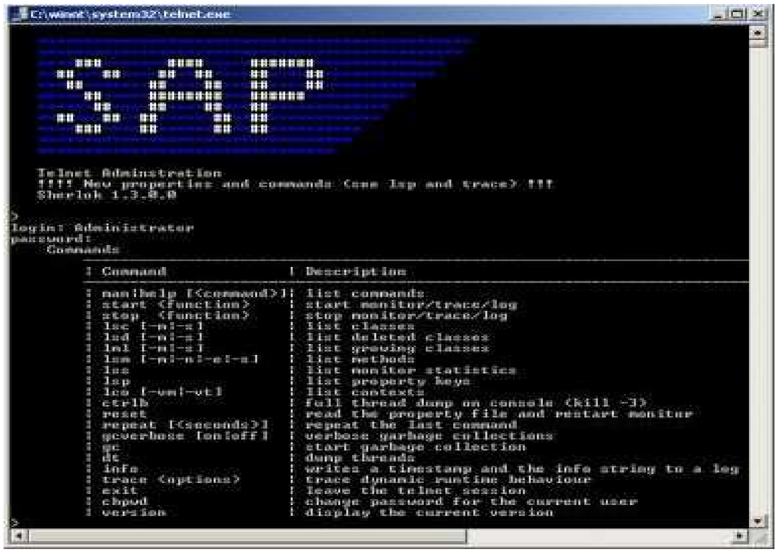
Operation of the Sherlok console and all its commands, as well as configuration files is described in detail below. In this section we only show what is needed for the trace examples.

Launch a command line terminal in your operating system and enter the following command line:

```
$ telnet localhost 2424
```

If you didn’t keep the defaults, modify the line and insert the current values of **TelnetHost** and **TelnetPort**.

Now you can login with user “Administrator” and password “sherlok” (default login or other user you have set up). The Sherlok telnet console shows up with a table of commands and a short description. The **help** command can generate this list of commands as well and with `help <cmd>` you get detailed help on *`<cmd>.`*



```
Telnet Administration
!!!! New properties and commands (see lsp and trace) !!!
Sherlok 1.3.0.0

>
login: Administrator
password:
    Commands
        ---------------------------------------------------------------------
        | Command                | Description
        ---------------------------------------------------------------------
        | man|help [<command>]|  | list commands
        | start <function>       | start monitor/trace/log
        | stop  <function>       | stop monitor/trace/log
        | lsc [-n|-s]            | list classes
        | lsd [-n|-s]            | list deleted classes
        | lnl [-n|-s]            | list growing classes
        | lsm [-n|-n|-e|-s]      | list methods
        | lss                    | list monitor statistics
        | lsp                    | list property keys
        | lcs [-vm|-vt]          | list contexts
        | ctrlb                  | full thread dump on console (kill -3)
        | reset                  | read the property file and restart monitor
        | repeat [<seconds>]     | repeat the last command
        | gcverbose [on|off]     | verbose garbage collections
        | gc                     | start garbage collection
        | dt                     | dump threads
        | info                   | writes a timestamp and the info string to a log
        | trace <options>        | trace dynamic runtime behaviour
        | exit                   | leave the telnet session
        | chpwd                  | change password for the current user
        | version                | display the current version
>
```

Picture 13: Telnet Console

### 10.1.2 Logging

First of all you should consider to log your session. Everything you see on the console can be logged into the file `sherlok.log`, including the commands you launch an their timestamp of execution. Check if you have an old `sherlok.log` you want to keep and rename it. Then enter:

```
> start log
```

Each start log command will delete `sherlok.log` for rewrite. Later enter `stop log` to close the log file.

### 10.1.3 Activate the Tracer

Some trace functions will only work if the monitor is running:

- `trace add trigger`
- `trace add thread`
- `trace add method`

If you stop the monitor all data will be frozen. You can re-read a changed configuration file and restart the monitoring session with the **reset** command.

```
> start monitor
> stop monitor
```

(edit configuration file; now re-read:)

```
> reset
```

You have to start the tracer to have any output

```
> start trace
> trace add gc
```

You can stop the tracer, if you don’t want any output anymore. Some of the traces might result in a mass output. These traces can be stopped by just pressing the return key:

- `trace add trigger`
- `trace add contention`
- `trace add thread`

The output for all these traces is handled special to prohibit spoiling the log file:

- No output parameter: Output only on console
- `-f<file>`: Output to a file located in ConfigPath
- `-p`: Output to Sherlok.log

The tracer has different output volume for different settings and shows different behavior:

- small volume as “trace add gc” or “trace add classes” will write output into the sherlok.log file
- huge volume as “trace add stack” need additional parameter to define output file. Option –p to write output into the sherlok.log and –f<name> to write the output into the specified file (no output to console for good performance).

## 10.2 Trace Garbage Collector

Use to GC Trace see more details about GCs in the sherlok log. The GC trace is triggered by the GC activation and is activated by the following command:

```
> start trace
> trace add gc
```

The GC trace is independent from monitor. The trace writes following information to the console and log file

| *10.2.1.1 Column* | *10.2.1.2 Description* |
|---|---|
| Trigger | Trigger is **GC** |
| GCNr | The count of garbage collections since start of the application |
| TimeStamp | Relative operating system dependent time |
| Objects | Number of objects in JVM |
| Space | Space allocated by all objects |
| Total | Available heap space (why not call it 'Available' or 'free' instead of 'total') |

## 10.3 Trace Thread Contentions

The thread contention tracer is a fast and lightweight function to check the performance of a multithreaded application. It shows you the bottlenecks of the application, where threads have to wait for each other. This function is lightweight with minimal impact to the performance.

In the properties file specify the following entries

```
Timer          = on
ProfileMemory  = off
```

On the telnet console enter the following commands

```
> start log
> start trace
> trace add contention –e30 –a -p
```

## 10.4 Trace Methods

Method tracing is useful to show if certain methods are called and with what parameters. Significant methods can e.g. be a login method (parm: what user), a http request (parm: user and URL), etc. All methods specified in TraceMethods property will be activated.

```
TraceMethods    = .Dispatcher.service; .doContent
```

The method trace is triggered by method enter and exit events, both are recorded. This feature replaces the typical print statement in your code. The trace writes following information to the console and log file

| Column | Description |
|---|---|
| Trigger | Trigger is **Trace** |
| Event | Name of the event is **Enter** or **Exit** |
| Time | Timestamp of this event |
| Class | Class and package of the method |
| Method | Method name |
| Thread | Name of the executing thread. Useful if you want to check if there are any pending requests. |
| Info | (for **Exit** events only) CPU time in nano seconds by this method call |

## 10.5 Trace Parameters

Sometimes you want to not only see a method being called but also the parameters passed with the call. Sherlok can show you parameters for basic types and trace them together with the method output. In contrast to the method trace, this event is only activated an method entry and is independent from the state of the monitor.

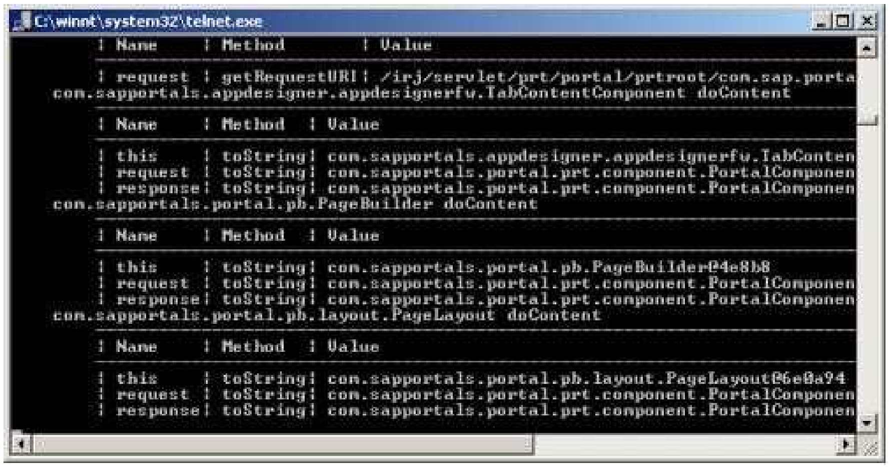
The parameter trace is triggered by the method enter event. It is independent from monitor. To use this trace, you have to set the **TraceMethods** property in the active configuration file which defines the methods to be traced. Please note: The list of methods are semicolon separated. Example:

```
TraceMethods    = .Dispatcher.service {request.getRequestURI}; .doContent
```

In curly brackets you can access methods of the call parameter object, which have void parameter and returns an object.

For output the method **toString** is called. If you do not specify any curly brackets, the output is done for all call parameters. Its possible to specify the method **getClass** to retrieve the class information. You can get the method signature with the command “**lsm –a –m0 –n0 –e0 –f.Dispatcher.service**”.

| Column | Description |
|---|---|
| Name | Call parameter name |
| Method | Requested method |
| Value | Result for execution method **toString** on the dereferenced object |



```
        | Name      | Method          | Value
        ----------------------------------------------------------------------------
        | request  | getRequestURI| /irj/servlet/prt/portal/prtroot/com.sap.porta
   com.sapportals.appdesigner.appdesignerfw.TabContentComponent doContent
        ----------------------------------------------------------------------------
        | Name      | Method   | Value
        ----------------------------------------------------------------------------
        | this      | toString| com.sapportals.appdesigner.appdesignerfw.TabConten
        | request  | toString| com.sapportals.portal.prt.component.PortalComponen
        | response| toString| com.sapportals.portal.prt.component.PortalComponen
   com.sapportals.portal.pb.PageBuilder doContent
        ----------------------------------------------------------------------------
        | Name      | Method   | Value
        ----------------------------------------------------------------------------
        | this      | toString| com.sapportals.portal.pb.PageBuilder@4e8b8
        | request  | toString| com.sapportals.portal.prt.component.PortalComponen
        | response| toString| com.sapportals.portal.prt.component.PortalComponen
   com.sapportals.portal.pb.layout.PageLayout doContent
        ----------------------------------------------------------------------------
        | Name      | Method   | Value
        ----------------------------------------------------------------------------
        | this      | toString| com.sapportals.portal.pb.layout.PageLayout@6e0a94
        | request  | toString| com.sapportals.portal.prt.component.PortalComponen
        | response| toString| com.sapportals.portal.prt.component.PortalComponen
```

**Picture 14: Trace Parameter**

## 10.6 Trace HotSpots

Sometimes you wonder “what takes the system so long”. This can be because a certain low level activity is slow and affects all programs. More often, there are few certain functions that take very long and slow down the execution of a request. Sherlok can show all methods that take longer than a certain time or use more memory than a certain limit.

To find these slow functions you should use the “hotspot method”, i.e. look for the slowest first, then the next and so forth.

### 10.6.1 Setup

The HotSpot trace is triggered by an method exit event and a the elapsed response time or the allocated memory.

**Example:** Write trace for all methods in scope and filter that take more than 100 msecs.

```
> start monitor
> start trace
> trace add trigger –e100 -p
```

To use HotSpot, you have to set the **TraceTrigger** property in the config file (***.skp**) and start the tracer. For a portal application the following entry would be possible:

```
TraceTrigger = .Dispatcher.service
```

This trace point is an example that is relevant in the portal. You can find many possible trigger points in the file `EP_trace.skp`. For other applications, you should create another file that contains the proper 'important methods' for tracing.

The tracer is active during the execution of method specified with **TraceTrigger**, i.e. underneath the **TraceTrigger** call, methods are traced.

It may happen, that there are two methods with the same name. In this case you can specify the signature separated with a colon. If there are methods with the same name and signature, you can specify the row number in the output of the reset command.

```
TraceTrigger = .doTest,(Ljava/lang/String;)V,2
```

## 10.6.2 Options

| Option | Description |
|---|---|
| -ascii<br>-tree<br>-xml | Output formats. The xml format can be used in any browser, but will be finished only by the command **stop trace**. The tree format will write indent spaces to represent the call hierarchy. Default is ascii. |
| -e<time> | Elapsed time to trigger event. Any method, that takes more time will trigger a callstack dump. unit = msecs |
| -m<memory> | Allocated memory to trigger event. Any time the application allocates this amount of memory it will trigger a callstack dump. In the output only the monitored classes are listed. |
| -f<file name> | Redirect output to the given file |
| -p | Redirect trace output to sherlok.log |

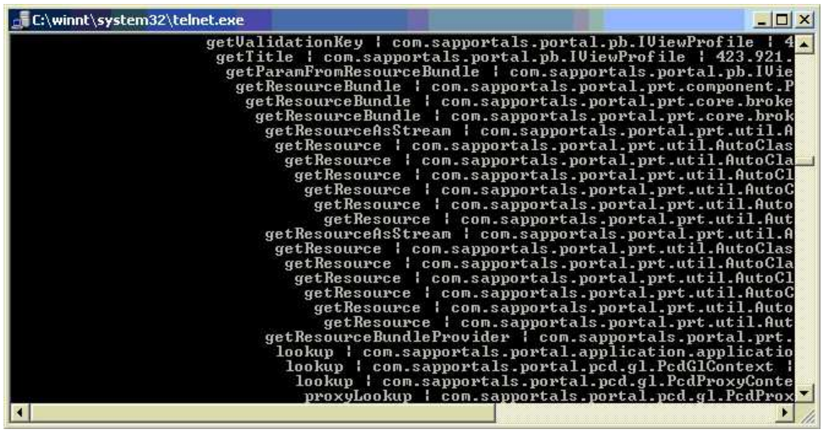
## 10.6.3 Output Option -ascii

The ascii output writes the result into an unformatted text line, separated by vertical slashes. This kind of output is useful if you have post processors, which can extract the data you need like awk or Excel. Each line contains the following information:

| *10.6.3.1 Column* | *10.6.3.2 Description* |
|---|---|
| Trigger | Trigger name is constant: **Trace** |
| Level | Level in call stack counting from the entry point **TriggerMethod** |
| Class | Class and package of the method |
| Method | Method name |
| Event | Name of the event is **Call** or **Time** |
| Info | For **Call** event: CPU time in nanoseconds by this method call<br>For **Time** event: Elapsed time in milliseconds |

## 10.6.4 Output Option -tree

The tree output indents the lines to represent the call hierarchy. This allows pattern recognition at a glance. The most right method in a jigsaw pattern is the method, which triggered the event. The caller is the leftmost method.



```
getValidationKey ¦ com.sapportals.portal.pb.IViewProfile ¦ 4
 getTitle ¦ com.sapportals.portal.pb.IViewProfile ¦ 423.921.
  getParamFromResourceBundle ¦ com.sapportals.portal.pb.IVie
   getResourceBundle ¦ com.sapportals.portal.prt.component.P
    getResourceBundle ¦ com.sapportals.portal.prt.core.broke
     getResourceBundle ¦ com.sapportals.portal.prt.core.brok
      getResourceAsStream ¦ com.sapportals.portal.prt.util.A
       getResource ¦ com.sapportals.portal.prt.util.AutoClas
        getResource ¦ com.sapportals.portal.prt.util.AutoCla
         getResource ¦ com.sapportals.portal.prt.util.AutoCl
          getResource ¦ com.sapportals.portal.prt.util.AutoC
           getResource ¦ com.sapportals.portal.prt.util.Auto
            getResource ¦ com.sapportals.portal.prt.util.Aut
      getResourceAsStream ¦ com.sapportals.portal.prt.util.A
       getResource ¦ com.sapportals.portal.prt.util.AutoClas
        getResource ¦ com.sapportals.portal.prt.util.AutoCla
         getResource ¦ com.sapportals.portal.prt.util.AutoCl
          getResource ¦ com.sapportals.portal.prt.util.AutoC
           getResource ¦ com.sapportals.portal.prt.util.Auto
            getResource ¦ com.sapportals.portal.prt.util.Aut
      getResourceBundleProvider ¦ com.sapportals.portal.prt.
       lookup ¦ com.sapportals.portal.application.applicatio
        lookup ¦ com.sapportals.portal.pcd.gl.PcdGlContext ¦
         lookup ¦ com.sapportals.portal.pcd.gl.PcdProxyConte
          proxyLookup ¦ com.sapportals.portal.pcd.gl.PcdProx
```

**Picture 15: Trace HotSpot Tree**

In the upper example you can see, that the method

getResourceBundle calls

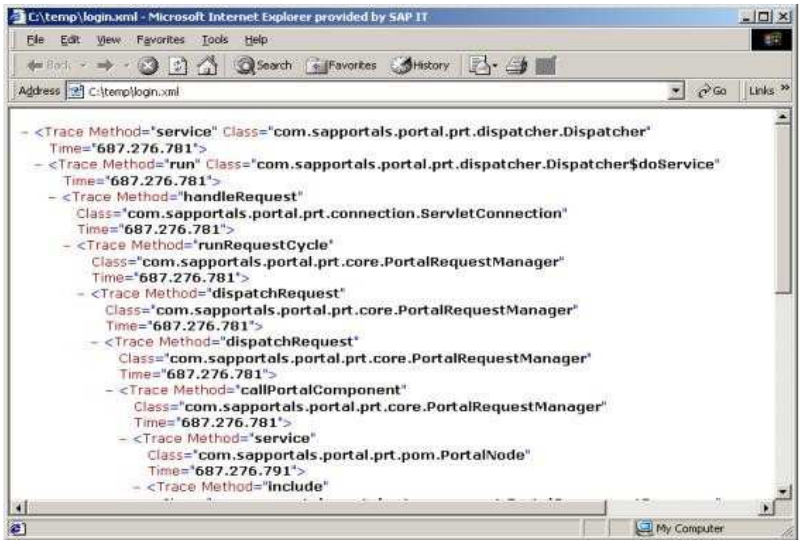
getResourceAsStream twice and getResourceBundleProvider afterwards.

| ***10.6.4.1 Column*** | ***10.6.4.2 Description*** |
|---|---|
| Method | Method name |
| Class | Class and package of the method |
| Event | Name of the event is **Trigger** |
| Level | Level in call stack counting from the entry point **TriggerMethod** |
| Info | Timestamp of the entry of the method |
| Thread | Name of the thread |

All entries are separated by the ProfileOutputSeparator character.

## 10.6.5 Output Option -xml

The xml output is generated in two paths. The finalize process, which closes all open tags is only called with the command **stop trace**. The limit for browsers to process this kind of input is about 2MB of data. The caller is the root (left most / top )node.


C:\temp\login.xml - Microsoft Internet Explorer provided by SAP IT
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\temp\login.xml
Go Links
My Computer

```
- <Trace Method="service" Class="com.sapportals.portal.prt.dispatcher.Dispatcher"
    Time="687.276.781">
  - <Trace Method="run" Class="com.sapportals.portal.prt.dispatcher.Dispatcher$doService"
      Time="687.276.781">
    - <Trace Method="handleRequest"
        Class="com.sapportals.portal.prt.connection.ServletConnection"
        Time="687.276.781">
      - <Trace Method="runRequestCycle"
          Class="com.sapportals.portal.prt.core.PortalRequestManager"
          Time="687.276.781">
        - <Trace Method="dispatchRequest"
            Class="com.sapportals.portal.prt.core.PortalRequestManager"
            Time="687.276.781">
          - <Trace Method="dispatchRequest"
              Class="com.sapportals.portal.prt.core.PortalRequestManager"
              Time="687.276.781">
            - <Trace Method="callPortalComponent"
                Class="com.sapportals.portal.prt.core.PortalRequestManager"
                Time="687.276.781">
              - <Trace Method="service"
                  Class="com.sapportals.portal.prt.pom.PortalNode"
                  Time="687.276.791">
                - <Trace Method="include"
```

**Picture 16: Trace HotSpot xml**

## 10.7 Trace Java Exceptions

The exception trace is triggered by unhandled Java exceptions and writes some information to the log for each unhandled exception. It is independent from the monitor.

**OutOfMemoryError** exceptions are handled in a special way:

- Sherlok requests a full JVM thread dump (like **dt -a** command). This shows you what requests were active at the time of the OOM.
- Sherlok dumps the location and the thread, which caused the exception.
- Sherlok dumps the monitored thread and methods to console (use `dt –c`)

```
> start trace
> trace add exception
```

## 10.8 Trace Thread Contentions

A thread contention event is raised whenever a thread has to wait on a monitor. Sherlok gives you the time the thread waited and the callstack for detailed analysis.

```
> start trace
> trace add contention [-a]
```

If you specify option –a, you get also contention information in packages, which are not in your ProfileScope.

Later on you can view the accumulated results for contentions in your scope:

```
> lsm –m0 –n0 –e0 –c1
```

## 10.9 Trace Stack

The stack trace shows the full  callstack for a given method call. All methods given in property TraceTrigger will be activated for tracing.

```
> start trace
> trace add stack [ -a ][ -tree ][ -p ][ -f<file> ]
```

With stack trace its possible to find allocation points for specific objects tracing its <init> method. The following statement will trace all string allocations in a given context:

```
> set TraceTrigger = /…/.myPackage./…/java.lang.String.<init>
> reset -s
> start trace
> trace add stack
```

With the option –a the trace shows the source code line of each statement.

## 10.10 Trace Threads

Sherlok can trace activities of a specific thread. This is useful if you want to evaluate an activity on waiting or sleeping threads. Sometimes you see some hundred thread sleeping if you request a full thread dump, but a huge activity in your system performance viewer. The thread tracer can show how often a thread is activated in a certain time span and what load results. This is especially useful, if you want to monitor communication on an idle node, or if you don’t know the entry points of a foreign application.

Use the command “dt” for a list of threads and “dt –c” to list Sherlok callstacks.

```
> start trace
> trace add trace [ -n<name of the thread ] [ -tree | –xml | –ascii ] [ -f<file> ]
```

if you don’t specify a name, Sherlok will start to dump activities of all threads. The trace result is in the given format and can be redirected into a file.

# 11 Reference

## 11.1 Configuration Reference

### 11.1.1 General Config Concepts

#### *11.1.1.1 Class Patterns*

During the operation of Sherlok there is often the need to restrict some facility to a limited set of classes to control monitoring or tracing. To define a subset of classes we use strings we call *class-pattern*. A class-pattern is a fully qualified name of a class. To denote a set of classes the dot "." is supported as a wildcard character. When used as the first or the last character of a class-pattern it is interpreted as an arbitrary sequence of characters. In all other cases (infix) it is treated like an ordinary dot-character.

Examples:

- The pattern "java.util.Tree." is matched by **java.util.TreeSet**, **java.util.TreeMap** any other class whose full qualified name starts with "java.util.Tree".
- ".List" matches any class "*List" in any package, like **java.util.LinkedList**, **java.util.ArrayList** or **org.w3c.dom.NodeList** .
- "." matches any class.
- "java.util.TreeSet" matches only **java.util.TreeSet**, but it does not match **java.util.TreeSet.Iterator**.

#### *11.1.1.2 Method Patterns*

In addition to specifying entire classes for monitoring, it is also possible to limit data to a certain method call. This is done by specifying a pattern for method names in the same manner as it was done for classes before, simply by adding the method name.

#### Examples

- The pattern ".get" matches any method that is called "get" in any class, for example **java.util.HashMap.get**.
- The pattern ".Entry.get." matches any method whose name starts with "get" and whose class is called "Entry", like **java.util.HashMap.Entry.getKey**.

#### *11.1.1.3 Call Stack Patterns*

In order to specify the targeting method of some Sherlok function more precisely, it is possible to define not only a pattern for methods but even for method-calls in conjunction with a certain state of the call-stack. The call-stack pattern specifies a matching rule for the call-stack contents at some point in time. It consists of sequence of nested method-calls, separated with a slash.

Call-stack matching is enabled by starting the method expression with a slash "/". The slash at the beginning denotes the top of the call-stack, like the **main** method or the *run* method of a thread. For example the call-stack resulting from method **f** calling **g** and **g** calling **h** (**f → g → h**) would be expressed as "/f/g/h". Each component in such an expression is interpreted as a simple *method pattern*, as explained above.

### Example

- "/…/.set./.get." is matched by any setter-method that directly calls a getter[8].
- "/MyApp.main/.save" is matched by calls to any method named **save** as long as this call happens directly in the static method **main** of executable class **MyApp**.

The *call-stack patterns* support another wildcard: "…". The ellipse "…" stands for an arbitrary number of subsequent calls[9]. A specific method call can be described with a substring of the full qualified method name.

### Example

- *"/…/.doContent/.../.createRepository"* denotes the call to some method **createRepository**, as long as it is executed as part of a call to another method **doContent**. This would be the case for the call-stack … → **doContent → myDispatcher → DataAccess → createRepository**

## 11.1.2 Handling Multiple Configuration Files

Sherlok has a lot of settings that determine its behavior. These are specified in configuration files. One configuration is active at a time and the user can select a configuration to be activated in the web UI and console mode.

This allows to define multiple profiles, each one with a specific purpose, e.g. one for catching out-of-memory issues, one for portal operation tracing, one for catching time or memory hotspots etc. Also, one development group can write one or more config files that are geared towards their set of classes and situation. Sherlok comes with a set of predefined config files for typical situations. These can be adjusted to a application area e.g. by defining the classes that are relevant.

Configuration files have the suffix ".scp" and reside by default in a subdirectory `sherlok` of where the Sherlok DLL is. This config directory is specified at startup time with a JVM parameter **ConfigPath** (see section 2.2.4).

### *11.1.2.1 Loading configuration files*

You can load a configuration file in the UI by selecting the **Configuration** combobox In the console you can specify which configuration is to be loaded with the command `set configfile =` … and then load it with **reset**.

### *11.1.2.2 List of predefined config files*

Sherlok comes with a set of predefined config files. The most important ones are:

| File | Description |
|---|---|
| default.skp | Default properties for portal namespace |
| default.customer.skp | Default properties for customer namespace |
| default.drilldown,skp | Drill down for memory problems |
| monitor.leak.skp | Configuration for memory leak detection |
| trace.skp | Configuration for performance analysis |

---

[8] Getters and setters are a pattern used to set public attributes of an object in Java. A getter method starts with "get", a setter method starts with "set": **getName**, **setName**, **getCity**, …

[9] Roughly speaking: "." and "…" are for call-stacks like "?" and "*" for characters sequences.

## 11.1.3 Defining the Profile Scope

Several config parameters work together to define exactly which classes and methods are considered for profiling. It is important to understand the exact effect of these parameters so that you on one hand can limit the scope of Sherlok (and overhead) and also define it wide enough to catch possible trouble sources. The relevant parameters are defined as class / method patterns.

Fundamentally one always selects methods to be monitored – or excluded. Specifying 'by class', as it is done in Packages and Excludes, selects or deselects *all methods* of the matching classes.

Class patterns are generally defined by prefix, e.g. “com.sap.km.” denotes all classes underneith **com.sap.km.** Method patterns are generally specified as suffix, e.g. “.doContent” selects all **doContent** methods in all packages/classes.

### *11.1.3.1 Definition*

Profile Scope = Packages + Methods - Excludes

| | |
|---|---|
| Packages | matches on *classes* and selects all methods from all classes matching patterns |
| Methods | matches on *methods* and selects all methods matching the method patterns |
| Excludes | matches on *classes* and removes all classes matching the patterns, and thereby also all methods in those classes |

## 11.1.4 Configuration Parameters

A configuration file is a simple text file with name-value pairs, where “#” is used as comment prefix. Each row consists of one assignment, i.e. `name = value`.

All configuration items listed here can be changed dynamically, i.e. when Sherlok runs. In the console mode you can use the 'set' command to change a value, in the UI there are special UI elements for certain settings, but not for all.

Some of the configuration settings done in the console mode / UI take effect immediately. For the other items, you must *reset* Sherlok. The **reset** command reloads the specified configuration files, and clears internal tables, history etc.

### *11.1.4.1 Global Settings*

| Property Name | Description | Default |
|---|---|---|
| ProfileInfo | Textual description of this profile. | (empty) |
| ProfileStart | Possible values are **yes** or **no**. Activates the monitor at the very start of Sherlok, i.e. at the startup of the JVM *and* when reloading a configuration.<br>When you have a problem during the initialization / startup phase of a JVM / application, i.e. before you can do any user input, you should set this value to yes". | no |

<table>
<tr><td>ProfileMode</td><td>Possible values are <b>profile</b> or <b>interrupt</b>.<table>
<tr><td>profile</td><td>In profile mode every method invocation is intercepted by Sherlok and <i>all</i> involved methods and classes are recorded according to the other Profile scope settings (see below). This mode is appropriate when you have no predefined few classes to start you investigation from. This mode is slower than interrupt mode but looks at all classes.<br>Profile mode is required for tracing.</td></tr>
<tr><td>interrupt</td><td>The interrupt mode is appropriate when you have a set of predefined components (classes) to be considered for monitoring (e.g. memory). It works by Sherlok inserting interrupts into the bytecode code and is much faster than profile mode. However, the JVM can only handle up to 1000 interrupts gracefully, so this mode can only be used for a few defined classes as entry points e.g. for all iViews.</td></tr>
<tr><td>jarm</td><td>The jarm mode allows to attach Sherlok to JARM instrumented java code. Sherlok will generate the memory information for JARM contexts</td></tr>
<tr><td>ats</td><td>In ATS mode Sherlok uses only one global stack for context information. This allows to create client-server test suits.</td></tr>
</table></td><td>profile</td></tr>
<tr><td>ProfileLimitOutput</td><td>Limit the length output for one command (in console an UI lists). This is useful to limit the output of commands that can create very long lists such as lsm.</td><td>2000</td></tr>
<tr><td>ProfileLimitHash</td><td>Internal Sherlok parameter: Limits the amount of memory Sherlok needs internally. It is the number of entities (objects, classes, ...) that Sherlok can monitor, where each entity needs 20 bytes.<br>When this internal space allocated to Sherlok is exceeded, Sherlok issues an error message and stops monitoring. You must increase this parameter and restart the JVM.</td><td>4000000</td></tr>
<tr><td>ProfileOutputType</td><td>Format of output used for the output socket. The console mode is attached to the output socket as well.<br>Possible values are <b>ascii</b> or <b>xml</b>. The XML output could be used to link the telnet port to other applications, e.g. a custom Sherlok UI.</td><td>ascii</td></tr>
<tr><td>Tracer</td><td>Enables trace functions from the start of the program. You can specify any trace option and its parameters (preceding with two minus signs):<br><code>Tracer=trigger--e100--tree,gc,methods</code></td><td></td></tr>
</table>

### 11.1.4.2 Selecting Profiling/Monitoring Scope

| Property Name | Description | Default |
|---|---|---|
| ProfilePackages | Defines the set of classes considered for profiling. It is a comma-separated list of *class patterns*. Each class that matches one of the patterns is monitored – unless it also matches **ProfileExcludes**. (see below).<br>**Example**<br>`ProfilePackages = com.mycompany., .myconsult.,`<br>`de.meinefirma.portal.MyIView` | (no classes) |
| ProfileMethods | Comma-separated list of Method Patterns. The matching method calls will be profiled for performance and memory (memory aggregated on class level),.<br>**Example:**<br>Activate all doContent methods in all classes.<br>`ProfileMethods = .doContent` | (no methods) |
| ProfileExcludes | Comma-separated list of *class-patterns*. All methods from the matching packages/classes are *not* monitored even if they match **ProfilePackages** or **ProfileMethods**. This is important to exclude system components.<br>Example:<br>`ProfileExcludes = com.sap., com.sapportals.,`<br>`com.inqmy., java., javax., sun., org.xml.` | (no classes) |

#### 11.1.4.3 Memory Monitoring

| Property Name | Description | Default |
|---|---|---|
| ProfileMemory | Controls whether Sherlok monitors memory allocations and releases. There are three settings:<br>off No profiling of memory<br>on Profile all memory activities within the defined scope<br>all Same as 'on' but in addition also all memory activities outside of the scope are collected into one **unknown** entry. Useful to see if a problem occurs *outside* of the defined scope (e.g. a memory leak). | off |
| MemoryLimitHistory | Number of GC cycles Sherloks remembers for memory profiling. For each remembered cycle Sherlok stores the total, allocated and deallocated size together with a timestamp per class. This way you can see how the memory needs of a class developed over time.<br>Note: The history uses 'smart compression'. See section xxx! | 10 |
| MemoryStatistic | Controls how Sherlok processes memory allocation and release information for each monitored class. Possible values:<br>(empty) For each monitored class just the accumulated allocated and released memory is kept.<br>info For each monitored class a history is kept (default: last 10 entries) that shows how the memory values behaved over time.<br>alert Selects the 'leak detector mode' which keeps a compressed history and watches for increases. For details see section 8.2 | (empty) |

#### 11.1.4.4 Performance / Timing Measurements

| Property Name | Description | Default |
|---|---|---|
| Timer | Activates *all* methods in profile scope for time measurement. Possible values are **on** or **off**. | off |
| TimerMethods | Comma separated list of methods in profile scope, which are activated for time measurement (as opposed to 'Timer' which activates all methods). | (no methods) |

#### 11.1.4.5 Trace Parameters

| Property Name | Description | Default |
|---|---|---|
| TraceMethods | *Semicolon* separated list of methods, which will be activated for method trace. These methods can be used to monitor enter / exit events, observe parameter values etc.<br>The method tracer will output the matching methods when they are called. When the timer is activated for these methods then also their execution times will be reported. | (no methods) |
| TraceGC | Activates tracing of garbage collections when turned on. Will output detailed info about each GC into the `sherlok.log`. Possible values are **on** and **off**. | off |

| | | |
|---|---|---|
| TraceTrigger | Method as activation point for trace events. For details on usage see section 10 | (no trigger) |
| TraceVerbose | This option will write additional field header information to the trace output for each line. Possible values are **on** or **off**. | on |

## 11.2 Telnet Console Reference

This section describes the commands and output for the Sherlok console interface. All Sherlok features are accessible via the console. This is mostly used by 'power users'.

### 11.2.1 Starting the Telnet Client

Launch a command line terminal in your operating system and enter the following command line:

```
$ telnet localhost 2424
```

If you didn’t keep the defaults, modify the line and insert the current values of **TelnetHost** and **TelnetPort**. (specified as command line args on the JVM call).

Now you can login with username “Administrator” and password “sherlok” (unless you changed the password).

### 11.2.2 Accounts

With the first startup Sherlok will create a telnet account in the text file **sherlok.pwd** with the following entry:

```
Administrator=<encrypted_password_sherlok>
```

You can add new accounts by copying an entire line and renaming the user. The password is encrypted and can be changed using the telnet console command **chpwd**.

Note: User name and password are case sensitive!

Note: Adding a user will not become effective until a restart of the java process (not just 'reset').

### 11.2.3 Command: man | help

Sherlok offers the following commands on the console. You get the list from the 'help' or 'man' command. You can specify a command as argument to get detail help for this command.

```
> man
    Commands
        --------------------------------------------------------------------------------
        | Command             | Description
        --------------------------------------------------------------------------------
        | man|help [<command>]| list commands
        | start <function>    | start monitor/trace/log
        | stop  <function>    | stop monitor/trace/log
        | lsc [-m|-s|-h]      | list classes
        | lsd [-m|-s|-h]      | list deleted classes
        | lml [-m|-s|-h]      | list growing classes / memory leaks
        | lsm [-m|-n|-e|-s]   | list methods
        | lss                 | list monitor statistics
        | lsp                 | list property keys and values
        | lco [-vm|-vt]       | list contexts
        | lhd                 | list heap dump
        | reset               | read the property file and restart monitor
        | repeat [<seconds>]  | repeat the last command
        | gc                  | start garbage collection
        | dt                  | dump threads
        | info                | writes info string to a log file with a timestamp
        | trace <options>     | trace dynamic runtime behaviour
        | lcf                 | list available configuration files
        | set name = val      | set options
        | exit                | leave the telnet session
        | chpwd               | change password for the current user
        | version             | display the current version
```

### 11.2.4 Command: lsp

The **lsp** command shows you all the config parameters and their current values. See section 11.1 for a detailed description on parameter semantics.

```
> lsp
  Properties
      -------------------------------------------------------------------------------
      | Property              | Value              | Description
      -------------------------------------------------------------------------------
      | ProfileStart          | no                 | Inital startup: [yes|no]
      | ProfileMode           | profile            | Profiling mode: interrupt or profile(event)
      | ProfilePackages       |                    | List of classes added to profile
      | ProfileExcludes       |                    | List of classes removed from profile
      | ProfileMethods        | .doContent         | List of methods added to profile
      | ProfileLimitOutput    | 500                | Maximum number of output lines for any cmd
      | ProfileLimitHash      | 131.071            | Maximum number of objects for profiler
      | ProfileOutputType     | ascii              | Sets the output type [xml|ascii]
      | ProfileOutputSeparator| |                  | Output separator for traces
      | ProfileMemory         | on                 | Switch memory profiling [on|off|all]
      | MemoryLimitHistory    | 10                 | Number of entries for memory history
      | MemoryStatistic       | alert              | Memory leak detection for alert [alert|info]
      | Timer                 | on                 | Activates the timer for all methods [on|off]
      | TimerMethods          |                    | Methods to activate for time measurement
      | TraceMethods          | .Dispatcher.service| Methods to activate for tracing
      | TraceTrigger          | .Dispatcher.service| Trigger method for trace
      | TraceVerbose          | on                 | trace output for GC
      | TraceGC               | off                | Trace GC information [on|off]
      | TelnetPort            | 2.222              | Port to connect
      | TelnetHost            | localhost          | Hostname for remote access
      | ConfigFile            | sherlok\default.scp| Active config file
      | Tracer                |                    | Startup configuration for trace
```

### 11.2.5 Command: start | stop monitor

Activates or deactivates the Java profile interface, which inserts or remove hooks to

- Method enter and exit events
- Memory allocation, deallocation and move events

Memory tracking and methods performance measurement is possible only if the monitor is running.

```
> start monitor
```

### 11.2.6 Command: start | stop trace

The Tracer allows output triggered by the application execution. You can add or remove trigger conditions after you started the tracer (see command trace). The tracer can be used in combination with the monitor. The result is an application execution trace. The trace can be started and stopped with the short command “s”.

```
> start trace
> trace add methods
```

### 11.2.7 Command: start | stop log

Starting the logger will create a new `sherlok.log` file and will pipe all console output also into this file.

Note: If you want to save your old log file, rename it before you do a 'start log'.

```
> start log
```

### 11.2.8 Command: lsc

The command **lsc** lists monitored classes. The possible command options are as follows:

```
> man lsc
    lsc [-m<number>][-s<column name>][-f<pattern>][-h]: list monitored classes
        ---------------------------------------------------------------------
        | Attribute      | Description
        ---------------------------------------------------------------------
        | -h             | output with GC history
        | -m<number>     | select classes with allocated bytes > <number>
        | -f<pattern>    | filter classes
        | -s<column name>| sort by column name
```

The default is to show all registered classes, even if they currently don't hold memory (equivalent to `lsc –m0`). Use `lsc –m10000` to list all classes that are responsible for at least 10 KB.

### 11.2.9 Command: lsd

The command **lsd** lists *deleted classes*. These are classes that have been expired by the VM, i.e. cannot be used anymore, but that are still responsible for memory being kept and therefore still shown in Sherlok. This allows to trap classes that allocate some memory, pass it on / store it in another class (e.g. global session) and then get removed.

Classes will be reloaded for hot deployment or for the JSP compiler results.

When a class appears multiple times in **lsd**, it means that an expired copy of the class is still in the **permanent space** (in the code part of the JVM memory) and not unloaded by the GC yet. For JDK 1.3.1. these classes are only removed from the JVM by a full garbage collection.

You can also monitor / trace the behavior of classes, i.e. loading and unloading, by using `trace add class` command. See the **trace** command.

```
> man lsd
    lsd [-m<number>][-s<column name>][-h]: list deleted classes
        ---------------------------------------------------------------------
        | Attribute      | Description
        ---------------------------------------------------------------------
        | -h             | output with GC history
        | -m<number>     | select classes with allocated bytes > <number>
        | -s<column name>| sort by column name
```

### 11.2.10 Command: lml

The command **lml** lists potential memory leaks, i.e. 'growing classes'.

```
> man lml
    lml [-m<number>][-s<column name>][-h]: list growing classes
        ---------------------------------------------------------------------
        | Attribute      | Description
        ---------------------------------------------------------------------
        | -h             | output with GC history
        | -m<number>     | select classes with allocated bytes > <number>
        | -s<column name>| sort by column name
```

### 11.2.11 Command: lsm

The command **lsm** lists all monitored methods with CPU time, elapsed time, and number of calls, ordered by CPU time (default).

```
> man lsm
    lsm [-m<number>][-n<number>][-e<number>][-s<column name>]: list monitored methods
        -------------------------------------------------------------
        | Attribute      | Description
        -------------------------------------------------------------
        | -m<number>     | select methods with CpuTime > <number>
        | -n<number>     | select methods with NrCalls > <number>
        | -e<number>     | select methods with Elapsed > <number>
        | -s<column name>| sort by column name
```

## 11.2.12 Command: lsp

The command **lsp** lists the current values of the Sherlok properties. These can be set in the config file or partly by the set command.

```
    -----------------------------------------------------------------------------------------
    | Property              | Value              | Description
    -------------------------------------------------------------------------_---------------
    | ProfileScope          | .                  | List of classes
    | ProfileMode           | profile            | Profile Mode: [profile|interrupt]
    | ProfilePackages       | com.sap..          | List of classes added to profiler
    | ProfileExcludes       |                    | List of classes removed from profiler
    | ProfileMethods        |                    | List of methods added to profiler
    | ProfileStart          | no                 | Initial startup: [yes|no]
    | ProfileLimitOutput    | 500                | Maximum number of output lines for any command
    | ProfileLimitHash      | 131.071            | Maximum number of objects for profiler
    | ProfileOutputType     | ascii              | [xml|ascii] Sets the output type
    | TraceMethods          | .Dispatcher.service| Methods to activate for tracing
    | TraceVerbose          | on                 | trace output for GC
    | ProfileMemory         | on                 | Switch memory profiling [on|off|all]
    | TraceGC               | off                | Trace GC information [on|off]
    | MemoryLimitHistory    | 10                 | Number of entries for memory history buffer
    | MemoryStatistic       | alert              | [alert|info] Memory leak detection for alert
    | Timer                 | on                 | Activtes the timer for all methods [on|off]
    | TimerMethods          |                    | Methods to activate for time measurement
    | ProfileOutputSeparator| |                  | Output separator for traces
    | TraceTrigger          | .Dispatcher.service| Trigger method for trace
    | TelnetPort            | 2.222              | Port to connect
    | TelnetHost            | localhost          | Hostname for remote access
    | ConfigFile            | sherlok\default.scp| Active config file
```

## 11.2.13 Command: lss

The command **lss** lists overall statistics, i.e. number of classes, threads, function calls, and a few other values.

```
> lss
    Monitor Statistic
        -----------------------------
        | Name         | Value
        -----------------------------
        | NewFktCalls  |        0
        | NewObjects   |        0
        | NewAllocation|        0
        | NrThreads    |      180
        | NrClasses    |    5.409
        | Monitor      | idle
        | Trace        | stopped
        | Logging      | true
```

## 11.2.14 Command: lhd

The command **lhd** means “list heap dump”. It lists all instances maintained by Sherlok grouped by class name. The command will only list scalar object instances, no arrays and no basic types. You can sort the output with the **-s** option and filter out classes (e.g. **java.lang**) with the **-f** option.

```
> man lhd
    lhd [-s<column name>][-f<pattern>]: list heap dump [sorted by column]
```

Use the command lhd together with the command lsc. The command lsc lists the number of bytes a class is responsible for and lhd shows what kind of objects these are and how many of them where allocated.

```
> lhd -sHeapSize
    Classes
        ------------------------------------------------------------------------------
        | HeapCount| HeapSize | Name
        ------------------------------------------------------------------------------
        |    64.113| 1.538.712| java.lang.String
        |    15.236|   365.664| java.util.HashMap$Entry
        |    10.978|   263.472| java.lang.StringBuffer
        |     3.925|   188.400| java.util.HashMap
        |    11.048|   176.768| java.util.jar.Attributes$Name
        |     2.541|   142.296| sun.io.CharToByteCp1252
        |     2.436|    97.440| java.util.Hashtable
        |       909|    72.720| com.sap.portals.jdbc.sqlserver.tds.TDSRPCParameter
        |       824|    72.512| java.util.jar.JarFile$JarFileEntry
        |       824|    59.328| java.util.zip.ZipEntry
        |       524|    58.688| com.sap.portals.jdbc.base.BaseColumn
        |     2.194|    52.656| java.lang.ClassNotFoundException
…
```

## 11.2.15 Command: reset

The command **reset** reloads the config file with the currently set config file name (set by the `set configfile=`... command). It also resets the internal Sherlok state and all statistics.

```
> man reset
    reset Relead the configuration file and clear all Sherlok statistics
```

## 11.2.16 Command: repeat

The command **repeat** repeats the last command every **n** seconds, with **n** being the parameter (default ist 1 second). This is very useful for automatic traces or **dt** commands etc.

```
> man repeat
    repeat [<seconds>]: repeat the last command
        ----------------------------------------------------------------------
        | Attribute| Description
        ----------------------------------------------------------------------
        | <seconds>| repeater time intervall in seconds (default is 1 sec)
        Any command will stop the repeater
```

## 11.2.17 Command: dt

The command **dt** (=dump threads) shows the threads known by Sherlok. It shows the threads with their names. (Sherlok doesn’t 'see' all threads of the JVM like the finalizer or startup thread)

The command **`dt –c`** shows all threads that have currently have a monitored class active in them. For all these, the current call stack is shown, showing only the methods that are in the monitoring scope.

The Sherlok-internal thread list is reset by the **reset** command. 'Threads know to Sherlok' will them again be accumulated from scratch (according to activity of monitored classes).

The command **dt -a** creates a full thread dump.

The command **dt –vm** sends a “kill –3” to the process group and **dt –jl** a “kill –3” to the process. The full thread dump is written to the stdout console (not the sherlok.log!). This shows all threads that are defined in the system, even if they are inactive. Most will be having the status “waiting on monitor” which means that they wait for a request and currently do nothing.

This command **dt –vm** should not be used if you started within a startup framework, since this may cause the JVM to terminate and then restart (by the startup framework).

```
> man dt
    dt [-c][-s<column>][-a|-vm|-jl] dump threads
        -------------------------------------------------------------------
        | Attribute | Description
        -------------------------------------------------------------------
        | -c        | dump callstack (if not empty)
        | -s<column>| sort output
        | -a        | full heap dump
        | -jl       | full heap dump (kill -3 to process)
        | -vm       | kill -3 CAUTION: THIS MIGHT TERMINATE APPLICATION
```

### 11.2.18 Command: dex

The command dex lists the statistic for all exceptions and prints the name and the number of events, which causes the exception..

### 11.2.19 Command: info

The command **info** allows to write a comment into the sherlok log file. A timestamp is attached.

### 11.2.20 Command: trace

The command trace adds new triggers to the tracer. For details on how to use the options, see section 10.

```
>  man trace
    trace [-verbose] [add|remove <trace-option>]
      -----------------------------------------------------------------------------------
      | Attribute                            | Description
      ----------------------------------------------------------------------------------
      | -verbose                             | add additional information to the console output
      | gc                                   | trace garbage collection
      | parameter                            | trace call parameters for TraceMethods
      | exception                            | trace exceptions: stop on OutOfMemoryError
      | contention -e<elapsed-time>
      |            -a -ascii|-tree|-xml      | trace thread contentions
      | stack                                | trace callstack for TraceTrigger method
      | method                               | trace enter and exit events for TraceMethods
      | class                                | trace class load and unload events
      | thread -n<thread-name>               | trace method enter events for <thread-name>
      | trigger <options>                    | trace triggered by TraceTrigger
      | trigger -ascii|-xml|-tree            | set output to ascii, xml or tree view
      | trigger -e<elapsed-time>             | trace all methods, which ecceed given elapsed-time
      | trigger -a                           | trace all method enter events
      | trigger -c                           | count up method enter events
      | trigger -f<file-name>                | redirect output to <file-name>
```

### 11.2.21 Command: lcf

The command **lcf** lists all available config files. They can then be loaded by set …. and activated by reset.

### 11.2.22 Command: set

The command **set** allows to set config parameters directly for the running Sherlok session. Changes are not propagated to the config file. Currently the only supported value is 'ConfigFile'  (case sensitive!). The use case is to switch to another config file and the load it with **reset**.

```
> set ConfigFile = default.skp
```

### 11.2.23 Command: exit

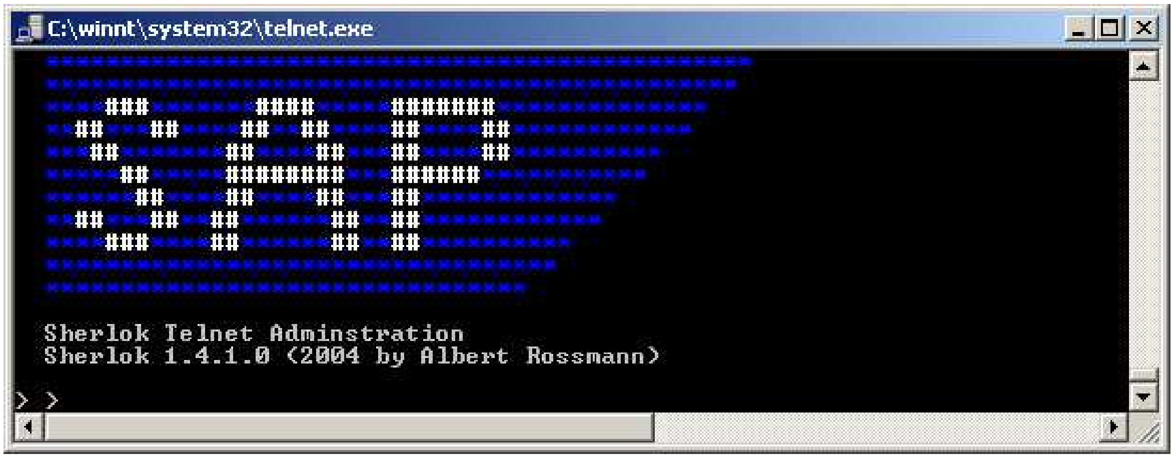
The command **exit** closes the Sherlok console. Sherlok itself will continue to run in the mode it had set last. You can always reconnect with the telnet call.

### 11.2.24 Command: chpwd

The command **chpwd** lets you change the password for your current user. The encrypted password is (over) written into the file sherlok.pwd.

### 11.2.25 Command: version

The command 'version' shows the version of Sherlok.



```
Sherlok Telnet Adminstration
Sherlok 1.4.1.0 (2004 by Albert Rossmann)

> >
```