

# **JProfiler Manual**

# Index

JProfiler help .....	5
How to order .....	6
A Help topics .....	7
A.1 Background and configuration .....	7
A.1.1 Behind The Scenes .....	7
A.1.2 Configuring profiling settings .....	11
A.1.3 Call tree collection .....	13
A.1.4 Configuring filters .....	15
A.1.5 Remote profiling .....	18
A.1.6 Removing finalizers .....	21
A.2 Memory Profiling .....	24
A.2.1 Recording objects .....	24
A.2.2 Using the difference columns .....	26
A.2.3 Finding a memory leak .....	27
A.3 CPU Profiling .....	31
A.3.1 Time measurements .....	31
A.3.2 Hotspots and filters .....	34
B Reference .....	37
B.1 Getting Started .....	37
B.1.1 Quickstart dialog .....	37
B.1.2 Running the demo sessions .....	37
B.1.3 Overview of features .....	37
B.1.4 JProfiler's start center .....	38
B.1.5 Application server integration .....	39
B.1.6 IDE integration .....	39
B.2 JProfiler setup .....	41
B.2.1 JProfiler setup wizard .....	41
B.2.2 JProfiler licensing .....	41
B.3 IDE integrations .....	42
B.3.1 Overview .....	42
B.3.2 IntelliJ IDEA 3.x .....	42
B.3.3 IntelliJ IDEA 4.x .....	43
B.3.4 Eclipse 2.x / WSAD 5.x .....	46
B.3.5 Eclipse 3.x .....	48
B.3.6 JBuilder .....	50
B.3.7 JDeveloper .....	52
B.3.8 Netbeans 3.x .....	54
B.3.9 Netbeans 4.x .....	56
B.4 Managing sessions .....	59
B.4.1 Overview .....	59
B.4.2 Application settings .....	59
B.4.2.1 Overview .....	59
B.4.2.2 Local session .....	61
B.4.2.3 Remote session .....	62
B.4.2.4 Applet session .....	62
B.4.2.5 Servlet session .....	63
B.4.2.6 Web Start session .....	63
B.4.2.7 Choosing the main class .....	64
B.4.3 Profiling settings .....	64

B.4.3.1 Overview .....	64
B.4.3.2 Call tree collection .....	65
B.4.3.3 Profiling features .....	66
B.4.3.4 Console settings .....	67
B.4.3.5 Miscellaneous settings .....	67
B.4.3.6 Profiling settings templates .....	68
B.4.4 Open session dialog .....	69
B.4.5 Starting remote sessions .....	69
B.4.6 Remote sessions invocation table .....	70
B.4.7 Saving live sessions to disk .....	75
B.5 General settings .....	76
B.5.1 Overview .....	76
B.5.2 Java VMs .....	76
B.5.3 Filter sets .....	77
B.5.4 Default java file path .....	78
B.5.5 IDE integrations .....	79
B.5.6 Miscellaneous options .....	79
B.6 Profiling views .....	80
B.6.1 Overview .....	80
B.6.2 Menu .....	81
B.6.3 Exporting views to HTML .....	84
B.6.4 Undocking views .....	85
B.6.5 Sorting tables .....	85
B.6.6 Using graphs .....	86
B.6.7 Source and bytecode viewer .....	86
B.6.8 Dynamic view filters .....	87
B.6.9 Global filters dialog .....	87
B.6.10 Memory view section .....	88
B.6.10.1 Overview .....	88
B.6.10.2 Class monitor view .....	89
B.6.10.2.1 Overview .....	89
B.6.10.2.2 Settings .....	90
B.6.10.3 Allocation monitor view .....	91
B.6.10.3.1 Overview .....	91
B.6.10.3.2 Settings dialog .....	92
B.6.10.4 Allocation hot spots view .....	94
B.6.10.4.1 Overview .....	94
B.6.10.4.2 Settings dialog .....	95
B.6.10.5 Memory statistics .....	97
B.6.10.5.1 Overview .....	97
B.6.10.5.2 Option dialog .....	97
B.6.10.6 Package selection dialog .....	98
B.6.11 Heap walker view section .....	99
B.6.11.1 Overview .....	99
B.6.11.2 Option dialog .....	100
B.6.11.3 View layout .....	100
B.6.11.4 Classes view .....	103
B.6.11.4.1 Overview .....	103
B.6.11.5 Allocation view .....	104
B.6.11.5.1 Overview .....	104
B.6.11.5.2 Allocation tree .....	104
B.6.11.5.3 Allocation hot spots .....	104
B.6.11.6 Reference view .....	106
B.6.11.6.1 Overview .....	106

B.6.11.6.2 Reference graph .....	106
B.6.11.6.3 Cumulated incoming references .....	108
B.6.11.6.4 Cumulated outgoing references .....	110
B.6.11.6.5 Path to root option dialog .....	111
B.6.11.6.6 Restricted availability .....	112
B.6.11.7 Data view .....	113
B.6.11.7.1 Overview .....	113
B.6.11.7.2 Instance data .....	114
B.6.11.7.3 Class data .....	114
B.6.11.7.4 Restricted availability .....	115
B.6.11.8 View helper dialog .....	116
B.6.11.9 Settings dialog .....	116
B.6.12 CPU view section .....	117
B.6.12.1 Overview .....	117
B.6.12.2 Invocation tree view .....	119
B.6.12.2.1 Overview .....	119
B.6.12.2.2 Settings dialog .....	120
B.6.12.3 Hot spot view .....	121
B.6.12.3.1 Overview .....	121
B.6.12.3.2 Settings dialog .....	122
B.6.12.4 Method graph .....	124
B.6.12.4.1 Overview .....	124
B.6.12.4.2 Method graph wizard .....	125
B.6.12.4.3 Method selection dialog .....	125
B.6.12.4.4 Settings dialog .....	126
B.6.12.5 CPU statistics .....	127
B.6.12.5.1 Overview .....	127
B.6.12.5.2 Option dialog .....	128
B.6.13 Threads view section .....	129
B.6.13.1 Overview .....	129
B.6.13.2 Thread history view .....	130
B.6.13.2.1 Overview .....	130
B.6.13.2.2 Settings dialog .....	131
B.6.13.3 Thread monitor view .....	133
B.6.13.3.1 Overview .....	133
B.6.13.3.2 Settings dialog .....	133
B.6.13.4 Deadlock detection graph .....	134
B.6.13.4.1 Overview .....	134
B.6.13.5 Monitor usage views .....	135
B.6.13.5.1 Overview .....	135
B.6.13.5.2 Current monitor usage .....	135
B.6.13.5.3 Monitor usage history .....	135
B.6.13.6 Monitor usage statistics .....	136
B.6.13.6.1 Overview .....	136
B.6.13.6.2 Option dialog .....	136
B.6.14 VM telemetry view section .....	137
B.6.14.1 Overview .....	137
B.6.14.2 Settings dialog .....	138
B.7 Offline profiling .....	140
B.7.1 Overview .....	140
B.7.2 Profiling API .....	141

## Welcome to JProfiler

Thank you for choosing JProfiler. To help you get acquainted with JProfiler's features, this manual is divided into two sections:

- [Help topics](#) [p. 7]

Help topics present important concepts in JProfiler. They are not necessarily tied to a single view. Help topics are recommended reading for all JProfiler users.

The help topics section does not cover all aspects of JProfiler. Please turn to the reference section for an exhaustive explanation of all features that can be found in JProfiler.

- [Reference](#) [p. 37]

The reference section covers all views, all dialogs and all features of JProfiler. It is highly hierarchical and not optimized for systematic reading.

The reference section is the basis for JProfiler's context sensitive help system. Each view and each dialog have one or more corresponding items in the reference section.

We appreciate your feedback. If you feel that there's a lack of documentation in a certain area or if you find inaccuracies in the documentation, please don't hesitate to contact us at [support@ej-technologies.com](mailto:support@ej-technologies.com).

## How to order

JProfiler licenses can be purchased easily and securely online. We accept Credit cards from Visa, MasterCard/Eurocard, American Express, JCB and Diners Club. You can also pay via bank transfer, via check or in cash.

For pricing information please visit [JProfiler's pricing page](#).

To order JProfiler please visit our [order pages](#).

For large quantities or site licenses please contact [sales@ej-technologies.com](mailto:sales@ej-technologies.com).

## A Help topics

### A.1 Background and configuration

#### A.1.1 Behind The Scenes - How profiling actually works

##### Introduction

Although it is not necessary to know about the internals of profiling to successfully profile your application, it can help you to interpret data that is produced by JProfiler, be more confident when setting up application servers and remote applications for profiling and analyzing problems with profiling in general. You might also just be curious to know what's going on under the hood.

##### Time, space and thread profilers

If you've been profiling C applications, you might know the distinction between time and space profilers. A "time profiler" measures the execution paths of your application on the method level whereas a "space profiler" gives you insight into the development of the heap, such as which methods allocate most memory. Recently, more and more applications are multi-threaded and thread profilers have been developed to analyze thread synchronization issues.

Most of these traditional profilers are "post-mortem" profilers where the profiling wrapper or profiling agent code writes out a data file when the profiled application exits. For an interactive profiler, it makes sense to compare and correlate data from all three domains, so JProfiler combines time, space and thread profilers in a single application.

##### How profilers collect data

A profiler must have some means to collect the data it displays. Profiling data can come from an **interface in the execution environment** or it can be generated by **instrumenting the application** of the application.

One of the most basic common profilers, the Unix shell command `time`, acts as a wrapper to the profiled executable and retrieves post-mortem information about the process from the kernel. Profilers for native applications on Microsoft Windows can attach to running applications and receive available debug information to calculate their profiling data. These are examples of interfaces in the execution environment where the binary of your application are not modified by the profiler.

The `gprof` Unix profiler (part of Unix since 4.2bsd UNIX in 1983) can be hooked into the compilation process by specifying an additional argument to the compiler (`-pg`). In this way, profiling code is added to your application. When the application exits, a data file is written to disk that contains call trees and execution times to be viewed with the `gprof` application. `gprof` is an example of a profiler that instruments your application.

JProfiler takes a mixed approach. It uses the profiling interface of the JVM and instruments classes at load time for tasks where the profiling interface of the JVM doesn't provide any data or adequate performance.

##### The profiling interface of the JVM

The profiling interface of the JVM is intended for profiling agents that are written in C or C++. If you open the `include` directory in your JDK, you will see a number of files with the extension `.h`. Those are the header files that tell a C/C++ library about the interface that is offered by the JVM. The basis for all communication between a native library and the JVM is the Java Native Interface (JNI), defined in `jni.h`.

The JNI allows Java code to call methods in the native library and vice versa. From Java code, you can use the `System.load()` call to load a native library into the same memory space. When you call a method whose declaration contains the "native" modifier, such as `public native String getName()`, a function in the list of loaded native libraries is searched for. The required name pattern

of the corresponding C-function contains the package, the class and the method of the declaration in Java code. JNI also defines how Java data types are represented in a C/C++ library. When the native C-function is called, it gets a "JNI environment" interface as an additional parameter. With this environment interface, it can call Java methods, convert between C and Java data types, and perform other JVM specific operation such as creating Java threads and synchronizing on a Java monitor.

The Java Virtual Machine Profiling Interface (JVMPI) is defined in *jvmpi.h*. It utilizes the JNI for communication with the JVM, but provides an additional interface to configure profiling options. JVMPI is an event-based system. The profiling agent library can register handler functions for different events. It can then enable or disable selected events.

Disabling events is important for reducing the overhead of the profiler. For example, in JProfiler, object allocation recording is switched off by default. When you switch on allocation recording in the GUI, the profiling agent tells the JVMPI interface that the event for object allocations should be enabled. If a lot of objects are created, this can produce a considerable overhead, both in the JVM itself as well in the profiling agent that has to perform bookkeeping operations for each event. During the startup phase of an application server, a lot of objects are created that you're most likely not interested in. Consequently, it's a good idea to leave object allocation recording switched off during that time. It increases the performance of the profiled application and reduces clutter in the generated data. The same goes for the measurement of method calls, called "CPU profiling" in JProfiler.

The JVMPI interface offers the following types of event:

- **Events for the life-cycle of the JVM**

The profiling agent is active before the JVM has been fully initialized. It can monitor how core classes are loaded and what method calls are executed during the initialization phase. When the JVM is initialized just before the main method is called, the profiling agent is notified. Similarly, the impending shutdown of the JVM is reported.

- **Events for the life-cycle of classes**

When a class is loaded and when it is unloaded, the profiling agent can be notified by the JVMPI. All other events, like the object allocation events or the method call events use the integer class ids and the the method ids that are reported with this event. Before a class is loaded, the profiling agent gets a chance to inspect and modify the content of the class file. This is the basis for "dynamic instrumentation" where bytecode is injected into the class file before it is actually loaded by the JVM.

- **Events for the life-cycle of threads**

To be able to show separate call trees for separate threads as well as to analyze monitor contention, the profiling agent must be aware of when threads are created and destroyed. When a thread is started, its identity is established. All other JVMPI events have a pointer that identifies the originating thread.

- **Events for for the life-cycle of objects**

The profiling agent can be notified of when objects are allocated, freed and moved in memory by the garbage collector. At this point, the call stack of the allocation spot can be recorded by the profiling agent. If the object allocation event is switched off, the allocation spot will not be available for the object later on. Such objects show up as "unrecorded objects" in the heap walker.

- **Events for method calls**

The JVMPI can be told to report the entry and the exit for each method. In JProfiler this is called "Full instrumentation". Full instrumentation is generally not recommended, since the overhead of reporting every single method call in the JVM is very large.

- **Events for monitor contention**

Whenever you call synchronized methods, use the `synchronize` keyword or call `Object.wait()`, the JVM uses Java monitors. Events that concern these monitors, such as trying to enter a monitor,



entering a monitor, exiting a monitor or waiting on a monitor are reported to the profiling agent. From this data, the deadlock graph and the monitor contention views are generated in JProfiler.

- **Events for the garbage collector**

Garbage collector activity is reported to the profiling agent. The garbage collector telemetry view in JProfiler is based on these events.

Some information, like references between objects as well as the data in objects are not available from the events that the JVMPI fires. To get exhaustive information on all objects on the heap, the profiling agent can trigger a **"heap dump"**. This command is invoked when you take a snapshot in the heap walker. In a heap dump, the JVMPI packs all the objects on the heap and the references between them into a single byte array and passes it to the profiling agent. That byte array is then parsed by the profiler and converted to an internal representation. Naturally, the memory requirements of this operation are huge: first, the heap is essentially duplicated in the byte array, then the profiling agent must parse it and translated it to data structures. In order to reduce the peak of the memory requirement, JProfiler saves the byte array to a temporary file on disk, releases the array and parses the contents of the temporary file. When profiling an application that maxes out the available physical memory, taking a heap dump can crash the JVM, simply because not enough physical memory is available to allocate the huge required regions of memory.

### **How the profiling agent is activated**

Unlike a JNI library that you load and invoke from Java code, the profiling agent has to be activated at the very beginning of the JVM startup. This is achieved by adding the special JVM parameter

`-Xrunjprofiler`

to the java command line. The `-Xrun` part tells the JVM that a JVMPI profiling agent should be loaded and the remaining characters of the parameter constitute the name of the native library. The canonical name or a native library depends of the platform. For a base name of `jprofiler`, the library name is `jprofiler.dll` on Microsoft Windows, `libjprofiler.so` on Linux and Unix, and `libjprofiler.dylib` on Mac OS X.

Parameters can be passed to the native profiling library by appending a colon to `-Xrunjprofiler` and placing the parameter string behind it. If you pass the `-Xrunjprofiler:port=10000` on the Java command line, the parameter `port=10000` will be passed the the profiling agent.

If the JVM cannot load the specified native library, it quites with an error message. If it succeeds in loading the library it calls a special function in the library to give the profiling agent a chance to initialize itself.

### **Profiling agent and profiling GUI**

Unlike basic profilers that collect data and write out a data file to disk, advanced profilers can display the profiling data at runtime. Although it would be possible to start the GUI directly from the profiling agent, it would be a bad idea to do so, since the profiled process would be disturbed by the secondary application and remote profiling would not be possible. Because of this, the JProfiler GUI is started separately and runs in a separate JVM. The communication between the profiling agent and the GUI is via a TCP/IP network socket. This is also the case if you start applications in JProfiler that are configured as "local" sessions.

In order to profile successfully, it's important to choose the right profiling parameters, especially the filters that limit the extent of the recorded call tree. Since this information is required at startup, the profiling agent stops the JVM and waits for a connection from the GUI where these parameters are configured. Once the connection has been established, the profiled application is allowed to start up.

The recorded profiling data resides in the internal data structures of the profiling agent. Only a small part of the recorded data is actually transferred to the GUI. For example, if you open the invocation

tree or the back-traces in the hotspots views, only the next few levels are transferred from the agent to the GUI. If the entire call tree were transferred to the GUI, potentially big amounts of data would have to be transmitted through the socket. This would make the profiled process slower and remote profiling between different computers would not be feasible. In essence, you could say that the profiling agent keeps a database of the recorded profiling data while the GUI is a client that sends user-initiated queries to the database.

## A.1.2 Configuring profiling settings

### What are profiling settings?

Profiling settings are settings that control the way profiling data is recorded. They must be adjusted according to your personal needs before the session is started. Unlike **view settings**, profiling settings cannot be changed during a running session. The primary distinction between profiling settings and view settings is that profiling settings are **irreversible**. If you wanted to change them during a running session, there would be a loss of data, or an inconsistency between data recorded before and after the change.

Profiling settings are persistent, just like view settings. Every change you make to the profiling settings will be remembered across restarts.

### Limiting the recorded profiling data

Why doesn't JProfiler just record everything it can and show it to the user? The answer is twofold:

- **There's a trade-off between information depth and runtime overhead**

Profiling adds overhead to the profiled application. It runs more slowly and consumes more memory. As an example, consider the call tree. JProfiler records separate call trees for each thread. If all method calls in all classes are recorded, the profiling agent has to do a lot of bookkeeping operations and its internal data structures use a lot of memory.

- **You want to reduce clutter in the recorded data**

Maximum detail doesn't lead to maximum insight. On the contrary, excessive detail will often be in the way. If there's too much information available, you're likely to get lost in it. Let's continue the above example: most of the time, you're not interested in the internal call tree of framework classes. Say, if you call `HashMap.get()`, the sufficiently detailed information will be the duration of this call. When you're not familiar with an implementation or if you're not in control of it, the internal calls structure is not helpful information, but rather just clutter, that you can ignore.

In principle, reducing the information depth can be done after recording. The view filters in the CPU views are such an example: the internal call structure of all classes that do not match the selected view filter is removed from the call tree. However, especially the increased memory consumption of profiling is critical: if you do not have enough physical memory available, the profiled JVM might become unstable or even crash. So in practice, you should record as little data as possible. With appropriate profiling settings you choose the required detail while retaining an acceptable runtime performance.

### Profiling settings templates

At first, the number of profiling settings can be quite overwhelming and the performance implications might not be quite clear. Because of this, JProfiler offers templates for profiling settings. When you start a session, a dialog is displayed where you can select one of several pre-defined templates. Below the combo box, a description and two overhead meters for CPU and memory overhead help you in judging whether the profiling settings are acceptable for you. Please note that the overhead meters do not give any absolute values, that would not even be possible theoretically, as JProfiler has no way of knowing the runtime characteristics of your application. Rather, they are hints that allow you to compare different profiling settings.

Each profiling settings template defines certain values for the profiling settings that can be viewed and modified by clicking the **[Customize profiling settings and filters]** button. When you modify and save those settings, the template combo box displays that the profiling settings are "Customized".

### Accessing the profiling settings

There are three locations where you can access the profiling settings in JProfiler.

- **in the profiling settings dialog that is displayed before a session is started**

When you define a new session, the default profiling settings template is used. Every time a session is started, a dialog is displayed that allows you to change this template or customize the settings in detail (see above).

- **from the menu or tool bar**

When a session is running, you can choose *Edit->Profiling settings* from the main menu or click on the corresponding tool bar button. You can look at the current profiling settings and you can even change them. However, changes in the profiling settings are not applied immediately, they will become effective the next time the session is started.

- **in the application settings dialog**

If you want to compare the profiling settings of two sessions, you can edit them in the start center. This shows the application settings dialog where you click the **[Profiling settings]** button that is located at the bottom. This is intended to let you review the profiling settings of existing sessions.

## **Overview of the various profiling settings**

The most important profiling settings are:

- **the call tree collection method**

This profiling settings determines performance overhead and informational detail in the CPU and memory views that show call trees. A detailed presentation of the various call tree collection method is available in a [separate article](#) [p. 13] .

- **the call tree filters**

The call tree filters determine the detail that is shown in any call tree or call stack in JProfiler. In brief, they define the set of classes whose internal call structure is shown while method calls into all other classes are treated as opaque. Please see the [article on call tree filters](#) [p. 15] for a thorough discussion.

## A.1.3 Call tree collection - collection methods and influence on performance and data

### Call trees and call stacks

At first glance, it might seem that the call tree collection settings only influence the CPU section of JProfiler. However, the memory section as well as the thread section display information that originates from the call tree that is build by the profiling agent of JProfiler: the invocation tree view, the allocation monitor, the stack traces in the monitor contention views as well as many other views all depend on the same call tree. The call tree is always recorded, even if "CPU recording" is switched off in JProfiler.

Selecting the right call tree collection method is crucial for a successful profiling run. As explained in the [article on profiling settings](#) [p. 11], the aim is to get the best runtime performance while retaining an acceptable level of informational detail. While the most important profiling setting in this regard is the [filter configuration](#) [p. 15], the call tree collection method complements this choice. Each call tree collection method has various limitations that you have to bear in mind when configuring call tree filters.

### The different method of call tree collection

There are three different methods for recording the call tree that have different advantages and disadvantages:

- **Dynamic instrumentation**

This is JProfiler's default mode. Before unfiltered classes are loaded by the JVM, JProfiler injects bytecode into the methods of that class that report the entry and exit of a method as well as the invocation of any filtered method. Filtered classes are not touched and run without overhead. If most classes are filtered, this mode causes low overhead while providing highly detailed measurements. Typically, the entire JRE and any framework classes are filtered so that dynamic instrumentation is most often the best choice. Since there are some classes in the `java.*` and `sun.*` packages that the profiling agent does not get a chance to modify, the internal calls of these packages cannot be resolved with dynamic instrumentation. However, for most applications this is not a problem.

- **Sampling**

"Sampling" means to periodically take measurements that are called "samples". In the case of profiling, an additional thread periodically halts the entire JVM and inspects the call stack of each thread. The period is typically 5 ms, so that a large number of method calls can occur between two samples. These method calls cannot be resolved by sampling. The sampler compares the call stacks of two subsequent samples and attributes the elapsed time to the "least common denominator" of the two call stacks. For example, with the call stacks:

Time x:

```
B.subOp()  
B.firstOp()  
B.calculate()  
B.run()  
A.main(String[] args)
```

and

Time x + 5 ms:

```
B.subOp()  
B.secondOp()  
B.calculate()
```

```
B.run()  
A.main(String[] args)
```

the least common denominator is

```
B.calculate()  
B.run()  
A.main(String[] args)
```

This is the invocation path that will be attributed 5 ms of execution time. If there's a very rapid succession of the method calls in `B.calculate()`, its detail cannot be resolved by sampling. You will only know that `B.calculate()` takes a lot of time.

The advantage of sampling is that its performance overhead is not very sensitive to the filter settings. Even without any filters, sampling is still fast since it operates with big granularity in time. You might ask why it is not possible to decrease the sampling time into the microsecond range to achieve a better resolution. The answer is that the process of sampling is a very expensive operation. Halting the entire JVM and querying the call stacks of a threads takes a lot of time. If you do this too often, sampling will actually become slower than dynamic or full instrumentation.

Sampling has two other important informational deficiencies: Since sampling does not monitor the entry and the exit of method calls, there's no invocation count in the CPU views of JProfiler. Furthermore, the allocation spots for objects are only approximate. The actual call stack might always be deeper than the reported one. Consider the above example where objects allocated by `B.subOp()` between time `x` and time `x + 5 ms` are reported as being allocated by `B.calculate()`. The problem is that this informational deficiency is not systematic, but statistical: the confusion sets in when at some later time two subsequent samples both produce the first call stack. Now some objects that are allocated by `B.subOp()` are reported correctly, but not all of them. To get around this deficiency, JProfiler has an option to record the exact allocation spots for sampling. In this case, the profiling agent does not rely on the call tree as recorded by the sampler. Rather, after each object allocation, it queries the JVMPI for the call stack of the current thread. However, this is an expensive operation and if you create a lot of objects the performance of the profiled application may suffer quite a lot.

To conclude, sampling is best suited for performance bottleneck searches with all filters turned off.

- **Full instrumentation**

The profiling agent can ask the JVMPI to report each and every method call, so that the profiling agent can measure it. While this may sound like a good idea at first, in practice, the performance overhead of this "full instrumentation" is too large. Except when you have to display the internal call structure of classes in the `java.*` and `sun.*` packages, this call tree collection mode is not recommended.

## A.1.4 Filters for call tree collection - how they work and how they are configured


### Introduction

Call tree collection filters determine the detail level that JProfiler uses when recording call sequences in the profiled application. Filtering helps to eliminate clutter and decrease the profiling overhead for the profiled application. Also see the article on [profiling settings](#) [p. 11] for a discussion of profiling settings in general.

Since the internal data storage of CPU data in JProfiler is similar to the invocation tree, call tree collection filters are most easily explained while looking at the [invocation tree view](#) [p. 119]. As an example, we profile the "Animated Bezier Curve" demo session that comes with JProfiler. When talking about filters, it is important to define the distinction between your code and framework or library code. Your code should be unfiltered, framework or library code should be filtered. In our example, the `BezierAnim` class is code written by you and the JRE is library code.

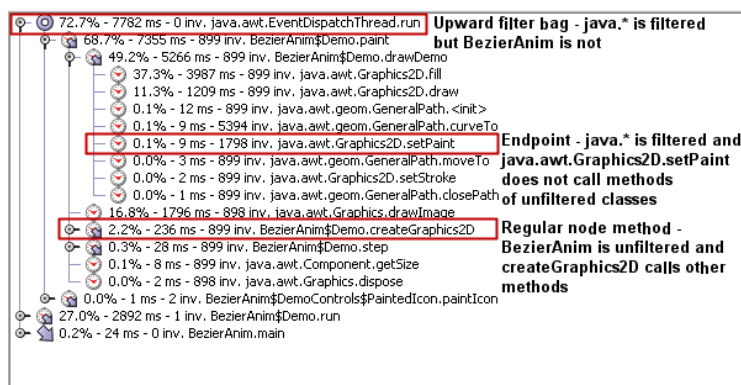
### What are call tree collection filters?

The invocation tree shows call sequences. Each node and each leaf of the invocation tree corresponds to a certain call stack that has existed one or multiple times while CPU recording was switched on. You will notice that there are different icons for nodes in the tree. Among other things, these icons serve to highlight if classes are filtered or not.

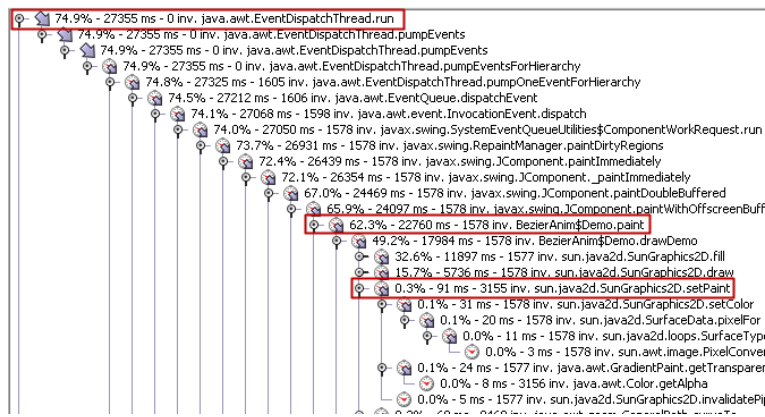
The methods of a filtered class are endpoints in the call tree, i.e. their internal call structure will not be displayed. Also, any methods in other filtered classes that are called subsequently, are not resolved. If, at any later point in the call sequence, the method of an unfiltered class is called, it will be displayed normally. In that case, the invocation tree shows the filtered parent method with a  special icon that indicates that it is from a filtered class and that there may be other method calls in between. The inherent time of those missing method is added to the time of the filtered parent method. In JProfiler's terminology, this is called an "upward filter bag".

### Example with and without filters

The image below illustrates the different node types for a profiling run of the `BezierAnim` class:



In the above invocation tree, the `java.*` package is filtered, so only the first method in the the AWT event dispatch thread is shown. However, the AWT is a complex system and the `java.awt.EventDispatchThread.run()` does not call `BezierAnim.paint()` directly. If we switch to full instrumentation and disable all filters, the invocation tree looks like this:



Now, the entry method into your code - `BezierAnim.paint()` - is substantially more difficult to find. In cases where events are propagated through a complex container hierarchy, the call tree can become many hundreds of levels deep and it becomes next to impossible to interpret the data. In addition, calls like `java.awt.Graphics2D.setPaint()` show their internal structure and implementation classes. As a Java programmer who is not working on the JRE itself, you probably do not know or care that the implementation class is actually `sun.java2d.SunGraphics2D`. Also, the internal call structure is most likely not relevant for you, since you have no control over the implementation. It just distracts from the main goal: how to improve the performance of your code.

Not only is it easier to interpret a call tree that has proper call tree collection filters, but also the profiling overhead of the profiled application is much lower. Recording the entire call tree without filters uses a lot of memory and measuring each call takes a lot of time. Both these considerations especially apply to application servers, where the surrounding framework is often extremely complex and the proportion of executed framework code to your own code might be very big.

### Configuring call tree collection filters

Call tree collection filters are part of the profiling settings of a session. Please see the article on [profiling settings](#) [p. 11] for an explanation on how to change the profiling settings for a session. While call tree collection filters can be changed during a running session, the change will only be effective when the session is restarted.

There are two alternative ways in JProfiler to specify the filtered classes:

- **by defining exclusive filters**

By default, a profiling session uses exclusive filters. "Exclusive filters" means that you specify a list of packages that should be filtered. In order to facilitate working with exclusive filters, you do not have to enter the list of packages in a text field but rather select appropriate "filter sets". Filter sets are named lists of packages that apply to a certain software library, application server or software company. The "Bea WebLogic" filter set contains all packages that are part of the Bea WebLogic application server. Filter sets are defined in JProfiler's general settings and are globally available for all sessions.

For a new profiling session, all filter sets are activated. If you want to resolve classes in a filtered package, you have to deselect the corresponding filter set in the profiling settings.

Exclusive filters are most appropriate for profiling application servers and regular applications where you're interested in all classes except a set of well-defined framework and library classes. If you have more specific requirements with respect to filtering, inclusive filters might be the better choice.

- **by defining inclusive filters**



With inclusive filters you define a list of packages and classes that should **not** be filtered. The call tree is only resolved for packages and classes that you have specified, all other classes are filtered.

This approach is recommended if you have a lot of different library or framework classes that are not contained in JProfiler's default list of filter sets, or if your code base is very large and you're only interested in certain parts of it.

### **View filters**

In addition to the call tree collection filters, there is a view filters control at the bottom of all views that display call trees. View filters are similar to inclusive filters and can be changed during a session. However, they can only **reduce** the recorded information by taking out classes that do not correspond to the selected view filter.

In the invocation tree, they have a similar behavior like the call tree collection filters. In the hot spot views, they simply hide all classes that do not correspond to the filter selection. This is very different from call tree collection filters, where the hot spots themselves change with different filter settings.

## A.1.5 Remote profiling - application servers and standalone applications

### Introduction

Although it is easiest to profile applications and application servers that are running on your local machine, sometimes it is not possible to replicate the execution environment on your computer. If you have no physical access to the remote machine or if the remote machine has no GUI where you could run JProfiler, you have to set up remote profiling.

Remote profiling means that the profiling agent is running on the remote machine and the JProfiler GUI is running on your local machine. Profiling agent and JProfiler GUI communicate with each other through a socket. As explained in the [background article on JProfiler](#) [p. 7], this situation is fundamentally the same as running a "local session", just that the socket communication socket connects between different machines. The main difference for you is that for local sessions you don't have to worry about the location of native libraries and that the startup sequence can be managed by JProfiler.

### The remote integration wizard

All integration wizards in JProfiler can help you with setting up remote profiling. After choosing the integration type or application server, the wizard asks you where the profiled application is located. If you choose the remote option, there will be additional questions regarding the remote machine.

When the remote integration wizard asks you for startup scripts or other files of the application server on the remote machine it brings up a standard file selector. If the file system of the remote machine is accessible as a network drive or mounted into your file system, you can select those files and JProfiler will directly write modified files to the right location.

If you do not have direct access to the file system of the remote machine, you can still copy the required files to the local machine. However, you must then transfer the modified or new files back to the remote machine after the integration wizard has completed.

### Requirements for remote profiling

Although the integration wizards in JProfiler give you all required information, it's always a good idea to have a little more inside knowledge about the mechanics and the requirements of remote profiling. When trouble-shooting a failed integration, you should check that the requirements below are fulfilled correctly.

The following requirements have to be satisfied for remote profiling:

1. JProfiler has to be installed on the local machine **and** on the remote machine. If the remote machine is a Unix machine, you might not be able to run the GUI installer of JProfiler. In this case, please use the `.tar.gz` archive to install JProfiler.

You do **not** have to enter a license key on the remote machine, the license key is always provided by the JProfiler GUI. Because of that, it is sufficient to unpack JProfiler to any directory where you have write permission.

2. The operating system and the architecture of the remote machine must be explicitly supported by JProfiler. Please see the [list of supported platforms](#) for more information. JProfiler is not a pure Java application, it contains a lot of native code which is not easily portable to unsupported platforms.
3. The native library path on the remote machine must contain the platform-specific directory in the `bin` directory of the JProfiler installation. The "native library path" is defined by a different environment variable on each platform. For example, on Windows, it is simply the `PATH` environment variable, on Linux it is `LD_LIBRARY_PATH`. The help page on [remote sessions](#) [p. 69] in the reference section tells you the corresponding environment variables for all platforms.
4. On the remote machine, you have to add a number of VM parameters to the java invocation of your application server or your standalone application. The fundamental VM parameters are `-Xrunjprofiler`, which tells the JVM to load the native profiling agent and

`-Xbootclasspath/a:{path to agent.jar}` which adds required Java classes to the bootclasspath. `agent.jar` is located in the `bin` directory of your JProfiler installation.

Depending on your JVM and your platform, you have to add further VM parameters to your java invocation. The [remote session invocation table](#) [p. 70] in the reference section gives you the exact parameter sequence for your configuration.

- 5 On the local machine, you have to define a remote session whose "host" entry points to the remote machine.

### Starting remote profiling

If you run the integration wizard for a local application server, JProfiler will be able to start it and connect to it. JProfiler has no way to start the application server if it is located on a remote machine. For remote applications and application servers, you have to perform **two** actions to start the profiling session:

- 1 Execute the modified start script on the remote machine. The application or application server will **not** start up, it prints a few lines of information and tells you that it is waiting for a connection.
- 2 Start the remote session in the JProfiler GUI on the local machine. The remote session will connect to the remote computer and the remote application or application server will then start up.

### Trouble-shooting

When things don't work out as expected, please have a look at the terminal output of the profiled application or application server on the remote machine. For application servers, the stderr stream might be written to a log file. Depending on the content of the stderr output, the search for the problem takes different directions:

- If stderr contains "Waiting for connection ...", the configuration of the remote machine is ok. The problem might then be related to the following questions:
  - Did you forget to start the remote session in the JProfiler GUI on your local machine?
  - Is the host name or the IP address correctly configured in the remote session?
  - Is there a firewall between the local machine or the remote machine?
- If stderr contains an error message about not being able to bind a socket, the port is already in use. The problem might then be related to the following questions:
  - Did you start JProfiler multiple times on the remote machine? Each profiled application needs a separate communication port. Please see below on how to change that port.
  - Are there any zombie java processes of previous profiling runs that are blocking the port? In this case please kill these processes.
  - Is there a different application on the remote machine that is using the JProfiler port? Please see below on how to change the port for JProfiler.

The communication port is defined as a parameter to the `-Xrunjprofiler` VM parameter. To define a communication port of 25000, please change this VM parameter to `-Xrunjprofiler:port=25000`. Also, please make sure that the same port is configured in the remote session in the JProfiler GUI on your local machine. Please note that this port has nothing to do with HTTP or other standard port numbers and must not be the same as any port that's already in use on the remote machine.

- If stderr contains an error message about not being able to load native libraries, the native library path is not configured correctly. Please see the requirements above on how to configure the native library directory. If the problem persists, it might be a problem with dependencies. On Unix platforms, you can execute

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH ldd libjprofiler.so
```

in the native library directory to get information about missing dependencies. On Microsoft Windows, you can download the dependency walker from <http://www.dependencywalker.com> to analyze the problem.

Please note that it is **not** a good idea to define the VM parameter `java.library.path`. If you absolutely have to do that, please make sure that the definition contains the appropriate native library directory for JProfiler.

- If stderr contains a `NoClassDefFound` exception for a class in the `com.jprofiler.agent` package, the bootclasspath has not been configured correctly. Please see the requirements above on how to configure the bootclasspath. Putting `agent.jar` in the regular classpath does **not** help and may actually be harmful.
- If there are no lines in stderr that are prefixed with `JProfiler>` and your application or application server starts up normally, the `-Xrunjprofiler` VM parameter has not been included in the java call. Please find out which java call in your startup script is actually executed and add the VM parameters there.

## A.1.6 Replacing finalizers with phantom references

### Why finalizers are bad

Sometimes one must perform pre-garbage collection actions such as freeing resources. In a JDBC driver, for example, a database connection may be held by a connection object. Before the connection object is garbage collected, the actual database connection must be closed. In such a case, one typically cannot rely on the `close()` method being called by the user application code.

Most often, **finalizers** are used to solve this problem. A finalizer is created by overriding the `finalize()` method of `java.lang.Object`. In that case, before the object is garbage collected, this `finalize` method will be called. Unfortunately, there are severe problems with the design of this finalizer mechanism. Using finalizers has a negative impact on the performance of the garbage collector and can break data integrity of your application if you're not very careful since the "finalizer" is invoked in a random thread, at a random time. If you use a lot of finalizers, the finalizer system may be completely overwhelmed which can lead to `OutOfMemoryErrors`. In addition, you have no control about when a finalizer will be run, so it can create problems with locking, the shutdown of the JVM and other exceptional circumstances.

The solution is to **eliminate finalizers** where they are not strictly required and **replace the necessary ones with phantom references**.

### What are phantom references?

Phantom references can be used to perform actions before an object is garbage collected in a safe way. In the constructor of a `java.lang.ref.PhantomReference`, you specify a `java.lang.ref.ReferenceQueue` where the phantom reference will be enqueued once the referenced object becomes "phantom reachable". Phantom reachable means unreachable other than through the phantom reference. The initially confusing thing is that although the phantom reference continues to hold the referenced object in a private field (unlike soft or weak references), its `getReference()` method always returns `null`. This is so that you cannot make the object strongly reachable again.

From time to time, you can poll the reference queue and check if there are any new phantom references whose referenced objects have become phantom reachable. In order to be able to do anything useful, one can for example derive a class from `java.lang.ref.PhantomReference` that references resources that should be freed before garbage collection. The referenced object is only garbage collected once the phantom reference becomes unreachable itself.

### How to replace finalizers with phantom references

Let's continue with the example of the JDBC driver above: Before a connection object is garbage collected, the actual database connection must be closed. The following steps are necessary to achieve this with phantom references:

- **Add data structure that holds phantom references**

The JDBC driver class gets a data structure that holds phantom references to the connection objects. A private field

```
private LinkedList phantomReferences = new LinkedList();
```

would be appropriate. This is necessary to ensure that phantom references are not garbage collected as long as they have not been handled by the reference queue.

- **Create reference queue**

Before a connection object will be garbage collected, its phantom reference will be enqueued into the associated reference queue. The JDBC driver thus gets an additional private field

```
private ReferenceQueue queue = new ReferenceQueue();
```

- **Derive a class from PhantomReference that references resources**

You will not be able to access the original object from a phantom reference. Therefore, you have to add the resources that must be freed to the phantom reference itself. In our example JDBC driver this could be a class named `DatabaseConnection`. The phantom reference class will thus look like:

```
public class ConnectionPhantomReference extends PhantomReference {
    private DatabaseConnection databaseConnection;

    public MyPhantomReference(ConnectionImpl connection, ReferenceQueue queue)
    {
        super(connection, queue);
        databaseConnection = connection.getDatabaseConnection();
    }

    public void cleanup() {
        databaseConnection.close();
    }
}
```

The custom phantom reference extracts the resource object from the implementation class of the connection and saves it in a private field. It additionally provides a `cleanup()` method that can be invoked once after the phantom reference is taken out of the reference queue.

- **Create and remember phantom references when objects are created**

When a connection object is created, a corresponding `ConnectionPhantomReference` must be created as well and added to the `phantomReferences` list:

```
phantomReferences.add(new ConnectionPhantomReference(connection, queue));
```

- **Create reference queue handler thread**

When a phantom reference is added to the queue by the garbage collector, no further action is taken. You have to handle and empty the reference queue yourself. It's best to create a separate daemon thread that removes phantom references from the queue and invokes the `cleanup` method:

```
Thread referenceThread = new Thread() {
    public void run() {
        while (true) {
            try {
                ConnectionPhantomReference ref =
                    (ConnectionPhantomReference)queue.remove();
                ref.close();
                phantomReferences.remove(ref);
            } catch (Exception ex) {
                // log exception, continue
            }
        }
    }
};
```

```
referenceThread.setDaemon(true);  
referenceThread.start();
```

The phantom reference is removed from the `phantomReferences` list. Now the phantom reference is unreferenced itself and the referenced object can be garbage collected.

## A.2 Memory Profiling

### A.2.1 Recording objects

#### Introduction

By default, JProfiler does not track the creation of all objects. This reduces the runtime overhead of the profiling agent regarding execution speed as well as memory consumption.

However, allocation recording is not only a way to increase runtime performance, it also helps you to focus on important parts of your application and to reduce clutter in the memory views. Imagine you have a web application that's started in the framework of an application server. The server allocates a huge number of objects in a great number of classes. If you want to focus on the objects created by your web application, the objects from the server startup will be in the way. In JProfiler, you can start allocation recording before you perform a certain action and so reduce the displayed objects to those that are allocated as a direct consequence of that action.

#### Starting and stopping allocation recording

The profiler menu as well as the toolbar allow you to start and stop allocation recording. If no allocations have ever been recorded, the dynamic memory views show placeholders with the corresponding "record" button. If you wish to enable allocation recording for the entire application run, you can do so in the profiling settings dialog

When you stop allocation recording, the garbage collection of the recorded objects will still be tracked by the dynamic memory views. In this way you can observe if the objects created during a certain period of time are actually garbage collected at some point. Please note that the manual garbage collection button in JProfiler just invokes the `System.gc()` method. This leads to a full GC in 1.3 JREs where the garbage collector makes the best effort to remove all unreferenced objects. However, 1.4 and 1.5 JREs perform incremental garbage collection, so full garbage collection is not available when working with such a recent JRE. To check if the remaining objects are really referenced, or if the garbage collector just doesn't feel like collecting them yet, you can take a heap snapshot. The heap walker offers the option "Remove unreferenced and weakly referenced objects" which is the equivalent of a full GC.

JProfiler also keeps statistics on garbage collected objects. All dynamic memory views have a mode selector where you can choose whether to display only live objects on the heap, only garbage collected objects, or both of them.

When you have stopped allocation recording and you restart it, the previous contents of the dynamic memory views will be deleted. In this way, allocation recording gives you the ability to do differencing of the heap between two points in time.

If you have very specific requirements as to where allocation recording should start and stop, you can use the [offline profiling API](#) [p. 141] to control allocation recording programmatically.

#### Implications of unrecorded objects

For "unrecorded" objects there are the following implications:

- JProfiler does not know the allocation spot for an unrecorded object. This becomes apparent in the heap walker. The heap walker takes a heap snapshot and is able to show all objects on the heap, however, the allocation information is not available from the JVMPi and the "Allocations" view will contain top-level method nodes that are labeled as "Unrecorded objects".
- JProfiler does not know the class name for an unrecorded object. This influences the monitor contention views where JProfiler is only able to display the name of a monitor object if the object has been recorded.

The object graph in the VM telemetry views is not affected by allocation recording.



### **Allocation recording and the heap walker**

In the heap walker options dialog, that is displayed before a heap snapshot is taken, the first option is labeled "Select recorded objects". This allows you to work with a set of objects that has been created during a certain period of time. This is just an initial selection step and does not mean that the heap walker will discard all unrecorded objects. In the reference view you can still reach all referenced and referencing objects and create a new object set with unrecorded objects.

If you use the "take heap snapshot with selection" action in the dynamic memory views, the number of selected objects will only match approximately, if "Select recorded objects" is checked and "Remove unreferenced and weakly referenced objects" is not checked in the heap walker options dialog. The numbers might still not match exactly since the dynamic memory views can change in time while a heap snapshot is fixed.

## A.2.2 Using the difference column in the memory views

### Introduction

In contrast to [allocation recording](#) [p. 24] , where you can restrict the displayed objects to a certain period of time, a common situation is that you want to retain all recorded objects but still see the difference of object allocations with respect to a certain point in time. In particular, you might be interested in which classes have a decreasing allocation count, something that would not be possible with allocation recording.

### Memory views with differencing

By default the difference column is not displayed. Only when you choose *Edit->Mark current values* or the corresponding toolbar button, the difference column is shown as the last column. The following views in JProfiler have an optional difference column:

- **class monitor**

In the class monitor, the difference column displays the number of currently allocated objects of a class minus the number at the point when the values were marked.

- **allocations hotspot view**

In the allocations hotspot view, the difference column is similar to the class monitor, just that the number of allocations in a method are measured. If you select a class for the hotspots view with the "Change selection" button, the number of allocations is additionally for a single package or class only.

In most cases you'll be interested in sorting the view by the values in the difference column. There are two sort modes that can be adjusted in the view settings dialog:

- **absolute ordering**

With absolute ordering, the absolute value of the difference will be used for sorting. This is appropriate if you're interested in the biggest changes.

- **normal ordering**

With normal ordering, you'll have positive differences at the top, then a usually long list of zero differences and finally the negative differences. This is the right setting if you're looking for a memory leak and are only interested in positive differences.

### Differencing and the heap walker

The difference column only shows a calculation, there's no fixed set of objects behind this number. Because of that, it is not possible to select the "difference objects" and work with them in the heap walker. To select objects based on their time of creation, please see the [article on allocation recording](#) [p. 24] .

### A.2.3 Finding a memory leak

#### Introduction

Unlike C/C++, Java has a garbage collector that eventually frees all unreferenced instances. This means that there are no classic memory leaks in Java where you forget to delete an object or a memory region. However, in Java you can forget something else: to remove all references to an instance so that the object can be garbage collected. If an object is only ever held in a single location, this may seem simple, but in many complex systems objects are passed around through many layers, each of which can add a permanent reference to the object.

Sometimes it appears to be clear that an object should be garbage collected when looking at the local environment of where the object is created and discarded. However, any call to a different part of a system that passes the object as a parameter can cause the object to "escape" if the receiver intentionally or by mistake continues to hold a reference to the object after the call has completed. Often, over-eager caching with the intention to improve performance or design mistakes where parallel access structures are built are the reason for memory leaks.

#### Recognizing a memory leak

The first step when suspecting a memory leak is to look at the heap and object telemetry views. When you have a memory leak in your application, these graphs must show a linear positive trend with possible oscillations on top.

If there's no such linear trend, your application probably simply consumes a lot of memory. This is not a memory leak and the strategy for that case is straightforward: Find out which classes or arrays use a lot of memory and try to reduce their size or number or instances.

#### Using differencing to narrow down a memory leak

The first stop when looking for the origin of a memory leak is the [differencing action](#) [p. 26] of the class monitor. Simple memory leaks can sometimes be tracked down with the differencing function alone.

First, you observe the differences in the class monitor and find out which class is causing the problems. Then you switch to the allocation hotspots view, select the problematic class and observe in the difference column in which method the problematic instances are allocated. Now you know the method in which these instances were created.

An analysis of the code for this method and the methods to which these instances are passed may already yield the solution to the memory leak. If not, you have to continue with the heap walker.

#### The heap walker and memory leaks

When you take a heap snapshot, you first have to create an object set with those object instances or arrays that should be freed by the garbage collector but are still referenced somewhere. If you've already narrowed down the origin of the memory leak in the dynamic memory views, you can use the "Take heap snapshot for selection" action to save you some work and to start in the heap walker right at the point where you left off in the dynamic memory views.

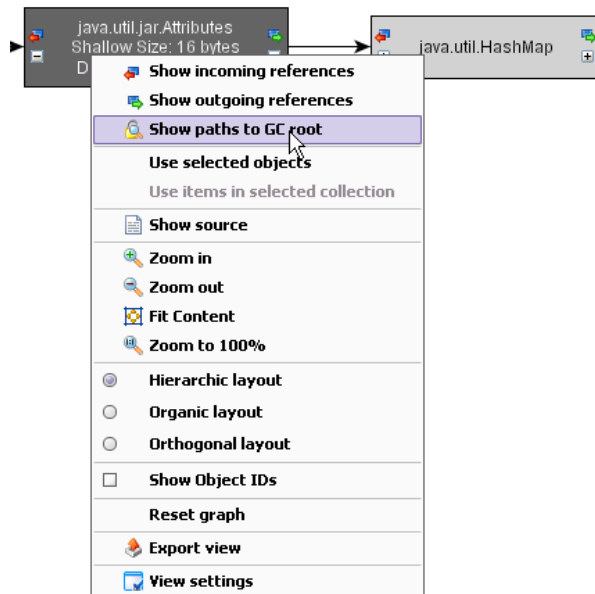
By default, the heap walker cleans a heap snapshot from objects that are unreferenced but are still not collected by the garbage collector. This behavior can be controlled by the "Remove unreferenced and weakly referenced objects" option in the heap walker options dialog. When searching for a memory leak, this "full garbage collection" is desirable, since unreferenced objects are a temporary phenomenon without any connection to a memory leak.

If necessary, you can now further narrow down the memory leak by adding additional selection steps. For example, you can go to the data view and look at the instance data to find out a number of instances that definitely should have been freed. By flagging these instances and creating a new set of objects you can reduce the number of objects that are in your focus.

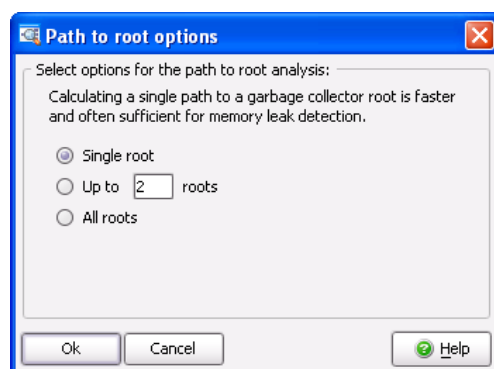
#### Using the reference graph to find the reason for a memory leak

The core instrument for finding memory leaks is the reference graph in the heap walker. Here you can find out how single objects are referenced and why they're not garbage collected. By successively opening incoming references you may spot a "wrong" reference immediately. In complex systems this is often not possible. In that case you have to find one or multiple "garbage collector roots". Garbage collector roots are points in the JVM that are not subject to garbage collection. These roots emanate strong references, any object that is linked by a chain of references to such a root cannot be garbage collected.

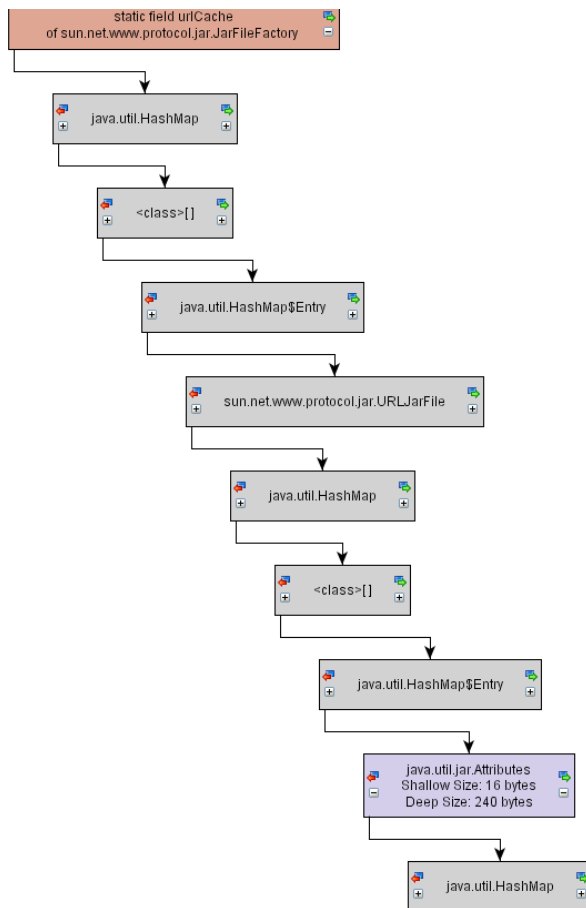
When you right-click on an object in the reference view, the context menu offers the option to search for paths to the garbage collector roots:



Potentially there are very many garbage collector roots and displaying them all can lead to the situation that a sizable fraction of the entire heap has to be shown in the reference graph. Also, looking for garbage collector roots is computationally quite expensive, and if thousands of roots can be found, the computation can take very long and use a lot of memory. In order to prevent this, it is recommend to start with a single garbage collector root and search for more roots if required. An option dialog is displayed after you trigger the search:



As you can see in this example, the chain to a garbage collector root can be quite long:



The reason for a memory leak can be anywhere along this chain. It is of a semantic nature and cannot be found out by JProfiler, but only by the programmer. Once you have found the faulty reference, you can work on your code to remove it. Unless there are other references, the memory leak will be gone.

### Using the cumulated references views to find the reason for a memory leak

In some cases, you might not succeed in narrowing down the object set to a reasonable size. Your object set might still contain a large number of instances that are OK and using the reference graph might not provide any insight in this situation.

If such a situation arises, the cumulated reference tables available in the reference view of the heap walker can be of help. The cumulated incoming reference table shows all possible **reference types** into the current object set:

Current object set: 231 instances of java.util.jar.Attributes  
2 selection steps, 4 kB shallow size, [calculate deep size](#)

Cumulated incoming references Use selected

Reference type	Reference count	Size
field <b>value</b> of java.util.HashMap\$Entry	223	5352
field <b>attr</b> of java.util.jar.Manifest	8	128
field <b>superAttr</b> of sun.net.www.protocol.jar.URLJarFile	1	56

From the reference type, you may be able to narrow down the object set. For example, you may know that one type of reference is OK, but another is not. As a hypothetical example, the reference from `HashMap$Entry` in the table above might be OK, but the reference from `java.util.jar.Manifest` might be suspicious. By selecting the 8 objects who are referenced in this way, you can discard the other 224 instances and use the reference graph to show the path to a garbage collector root.

## A.3 CPU Profiling

### A.3.1 Time measurements in different CPU views

#### Wall clock time and CPU time

When the duration of a method call is measured, there are two different possibilities to measure it:

- Most likely you'll be interested in the **wall clock time**, that is the duration between the entry and the exit of a method as measured with a clock. For the profiling agent this is a straightforward measurement. While it might seem at first glance that measuring times should not have any significant overhead, this is not so if you need a high resolution measurement. Operating systems offer different timers with different performance overheads.

For example, on Microsoft Windows, the standard timer with a granularity of 10 milliseconds is very fast, because the operating system "caches" the current time. However, the duration of method calls can be as low as a few nanoseconds, so a high resolution timer is needed. A high resolution timer works directly with a special hardware device and carries a noticeable performance overhead. In JProfiler, CPU recording is disabled by default, however, call tree collection is always enabled. If you compare the duration of the startup sequence of an application server with and without CPU recording, you will notice the difference.

Wall clock time is measured separately for each thread. In CPU views where the thread selection includes multiple threads, the displayed times can be larger than the total execution time of the application. If you have 10 parallel threads of the same class `MyThreadClass` whose `run()` method take 1 second and "All threads" is selected in the invocation tree, the `MyThreadClass.run()` node in the invocation tree will display 10 seconds, even though only one second has passed.

- Since the CPU might be handling many threads with different priorities, the wall clock time is not the time the CPU has actually spent in that method. The scheduler of the operating system can interrupt the execution of a method multiple times and perform other tasks. The real time that was spent in the method by the CPU is called the **CPU time**. In extreme cases, the CPU time and the wall clock time can differ by a large factor, especially if the executing thread has a low priority.

The standard time measurement in JProfiler is wall clock time. If you wish to see the CPU time in the CPU views, you can change the measurement type in the profiling settings. The problem with CPU time measurement is that most operating systems provide this information with the granularity of the standard timer - high resolution measurements would carry too much overhead. This means the CPU times are only statistically valid for method that have a CPU time bigger than the typical granularity of 10 milliseconds.

#### Thread statuses

The notion of time measurement must be refined further, since not all times are equally interesting. Imagine a server application with a pool of threads that waiting to perform a task. Most of the time would then be in the method that keeps the threads waiting while the actual task will only get a small part of the overall time and will be hard to spot. The necessary refinement is done with the concept of **thread status**. There are 4 different thread statuses in JProfiler:

- **Runnable**

In this case the thread is ready to execute code. The reason that this is not called "Running" is that it may actually not be running due to the scheduler of the operating system. However, if given a chance, the thread will execute instructions.

- **Waiting**

This means that the thread has deliberately decided to enter into hibernation until a certain event occurs. This happens when you call `Object.wait()` and the current thread will only become runnable again when some other thread calls `Object.notify()` on the same object.

- **Blocking**

Whenever synchronized blocks of code or synchronized methods occur, there can be monitor contention. If one thread is in the synchronized area all other threads trying to enter it will be blocked. Frequent blocking can reduce the liveness of your application.

- **Net I/O**

During network operations, many calls in the Java standard libraries can block because they're waiting for more data. This kind of blocking is called "Net I/O" in JProfiler. JProfiler knows the list of methods in the JRE that lead to blocked net I/O and instruments them at load time.

When looking for performance bottlenecks, you're mostly interested in the "Runnable" thread state although it's always a good idea to have a look at the "Net I/O" and "Blocking" thread states in order to check if the network or synchronization issues are reducing the performance of your application.

### **Times in the invocation tree**

Method nodes in the invocation tree are sorted by **total time**. This is the sum of all execution times of this method on the particular call path as given by the ancestor nodes. Only threads in the current thread selection are considered and only measurements with the currently selected thread status are shown.

Optionally, the invocation tree offers the possibility to show the **inherent time** of a method. The inherent time is defined as the total time of a method minus the time of its child nodes. Since child nodes can only be unfiltered classes, calls into filtered classes go into the inherent time. If you change your [call tree collection filters](#) [p. 13], the inherent times in the invocation tree can change.

### **Times in the hotspots view**

While the invocation view shows all call stacks in your application, the hotspot view shows the methods that take most of the time. Each method can potentially be called through many different call stacks, so the invocation counts in the invocation tree and the hotspots view do not have to match. The hotspot view shows the inherent time rather than the total time. In addition, the hotspot view offers the option to include calls to filtered classes into the inherent time. Please see the article on [hotspots and filters](#) [p. 34] for a thorough discussion of this topic.

When you open a hotspot node, you see a reverse invocation tree. However, the times that are displayed in those **backtraces** do not have the same meaning as those in the invocation tree, since they do not express a time measurement for the corresponding method node. Rather, the time displayed at each method node indicates how much time that particular call tree contributes to the hot spot. If there is only one backtrace, you will see the hotspot time at each node.

### **Times in the method graph**

The times that are shown for **method nodes** in the method graph are the same as those in the hotspots view, since the method graph is also method-centric. The times that are associated with the **incoming arrows** are the same as those in the first level of the hot spot backtrace, since they show all calling methods and the cumulated duration of their calls. The time on the **outgoing arrows** is a measurement that cannot be found in the invocation tree. It shows the cumulated duration of calls from this method, while the invocation tree shows the cumulated duration of calls from the current call stack.

### **Times in the CPU statistics**

The CPU statistics offers three types of statistics: method, class and package statistics. Time measurements in the method statistics are the same as those in the hotspot view, only all recorded methods are shown and not only hotspot methods. The class and package statistics are cumulations



of the method statistics. It is important to remain aware of the fact that the filter settings influence the class and package statistics [just as they influence the hotspot times](#) [p. 34] .

### A.3.2 The influence of call tree collection filters on hotspots

#### Introduction

The notion of a performance hot spot is not absolute but relative to your point of view. The total execution time of a method is not the right measure, since in that case your main method or the `run()` method of the AWT event dispatch thread would be the biggest hotspots in most cases. Such a definition of a hotspot would not be very useful.

As the other extreme one could use the unfiltered inherent time of the execution of a method for the ranking of hotspots. The unfiltered inherent time is the total time minus the time spent in all other method calls. This would not be very useful either, since the biggest hotspots will most likely always be core methods in the JRE, like string manipulation, I/O classes or core drawing routines in obscure implementation classes of the AWT.

As the above considerations make clear, the definition of a hotspot is not trivial and must be carefully considered.

#### Definition of a hotspot

Only with filters is it possible to come up with a useful definition of a hotspot. Usually, your [call tree collection filters](#) [p. 15] will be set up in such a way that all library classes and framework classes will be filtered out. In the following discussion, we're going to assume that this is the case.

In order to be useful a hot spot must be

- **a method in your own classes**

This can be obtained by measuring hotspots with the inherent time of a method call **plus** the calls into filtered classes.

- **a method in a library class that you call directly**

Since filters are endpoints for the measurement of methods, the inherent time of a filtered method is equal to its total execution time. When simply measuring hotspots with this filtered inherent time, filtered classes will likely be the hotspots.

Which one of these viewpoints is more helpful depends on the actual situation. JProfiler's hotspot view offers both modes with the combo box in the top-right corner. The allocation hotspots views also offer this mechanism of adjusting the definition of a hotspot.

#### Example

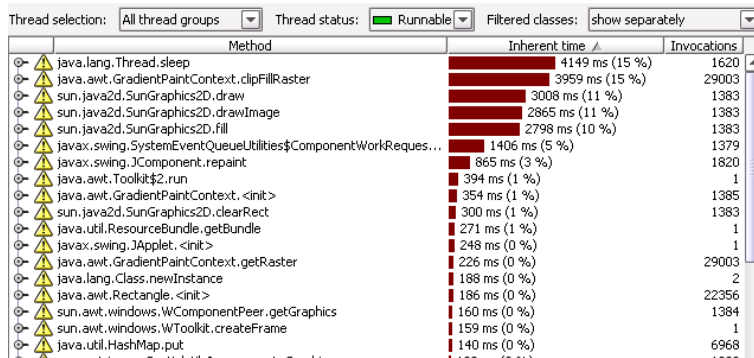
Let us profile the animated Bezier curve demo that comes with JProfiler. We will try out different filter settings and check how they influence the list of hotspots. In this case we consider the `BezierAnim` class to be "our" code, while the JRE classes are library classes.

We start by using the "Maximum detail" profiling setting template. Here, full instrumentation is used and no filters are applied.

Method	Inherent time	Invocations
java.lang.Thread.sleep	4004 ms (15 %)	1603
java.awt.GradientPaintContext.clipFillRaster	3056 ms (11 %)	23176
sun.java2d.loops.Blit.Blit	2149 ms (8 %)	3075
sun.dc.pr.PathFiller.writeAlpha8	1197 ms (4 %)	45341
sun.java2d.pipe.DuctusShapeRenderer.renderPath	1010 ms (3 %)	2216
sun.java2d.loops.MaskBlit.MaskBlit	451 ms (1 %)	21207
java.lang.Math.min	436 ms (1 %)	227815
sun.dc.pr.PathFiller.setOutputArea	344 ms (1 %)	2216
sun.java2d.pipe.AlphaPaintPipe.renderPathTile	320 ms (1 %)	23176
sun.java2d.loops.MaskFill.MaskFill	318 ms (1 %)	24133
sun.dc.pr.Rasterizer.getState	317 ms (1 %)	78623
java.awt.GradientPaintContext.<init>	308 ms (1 %)	1109
sun.java2d.pipe.DuctusRenderer.createShapeRasterizer	299 ms (1 %)	2216
sun.awt.windows.Win32BitLoops.Blit	226 ms (0 %)	1107
sun.dc.pr.PathFiller.writeAlpha	210 ms (0 %)	45341
sun.java2d.pipe.DuctusRenderer.getAlpha	203 ms (0 %)	45341
sun.dc.pr.Rasterizer.writeAlpha	187 ms (0 %)	45341
sun.awt.AppContext.get	186 ms (0 %)	14352

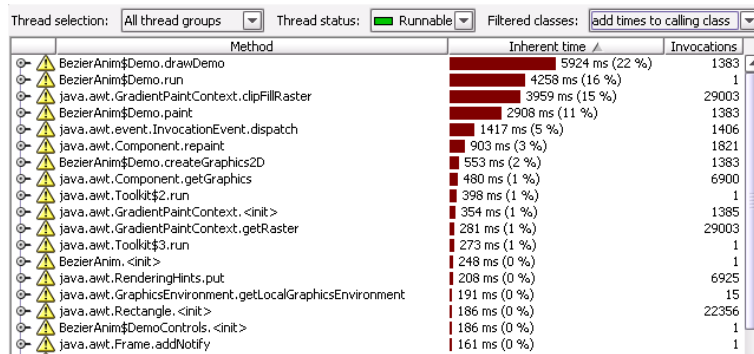
As we can see, most of the hotspots are implementation classes in `sun.*` implementation packages. These classes are never called by our code. While we could open the backtraces and see how they have been invoked, this is cumbersome and produces no insight into any performance problems that we might be able to solve.

In the next step, we restrict our filters so that only `BezierAnim` and the `java.awt.*` packages are unfiltered. This viewpoint is a little strange, somewhat as if the `java.awt.*` belonged to our code, too. But we want to show how the inclusion of filters changes the hotspots, so we take this middle step. We do this by customizing the "Maximum detail" profiling setting template and entering `BezierAnim`, `java.awt.*` as inclusive filters.



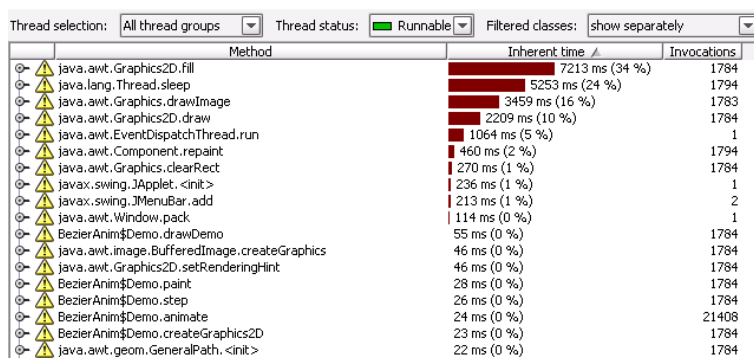
Again, there are a lot of filtered classes in the list of hotspot, but while the first two hotspots have stayed the same, the list below them is completely different to the unfiltered case.

Since we now have filters, we can change the viewpoint further by choosing the "add times to calling class" option in the top-right combo box labeled "Filtered classes".



Now, the list has changed completely and we only see unfiltered classes.

Finally, we profile with the "All features enabled, high CPU profiling detail" option, where all classes in the JRE are filtered.



Any of the methods that appear in the list of hotspots have been called from our code. This is great for finding performance bottlenecks but sometimes we only want to see our own methods. Again we choose the "add times to calling class" option in the top-right combo box labeled "Filtered classes".

Thread selection: All thread groups Thread status: ■ Runnable Filtered classes: add times to calling class

	Method	Inherent time ▲	Invocations
⊖	BezierAnim\$Demo.drawDemo	9590 ms (45 %)	1784
⊖	BezierAnim\$Demo.run	5729 ms (27 %)	1
⊖	BezierAnim\$Demo.paint	3512 ms (16 %)	1784
⊖	java.awt.EventQueueThread.run	1064 ms (5 %)	1
⊖	BezierAnim\$Demo.createGraphics2D	414 ms (1 %)	1784
⊖	BezierAnim\$DemoControls.<init>	286 ms (1 %)	1
⊖	BezierAnim.<init>	237 ms (1 %)	1
⊖	BezierAnim.main	139 ms (0 %)	1
⊖	BezierAnim\$Demo.step	26 ms (0 %)	1784
⊖	BezierAnim\$Demo.animate	26 ms (0 %)	21408
⊖	BezierAnim\$DemoControls.<clinit>	14 ms (0 %)	1
⊖	BezierAnim.init	13 ms (0 %)	1

All but one method are directly from the BezierAnim class. The `java.awt.EventQueueThread.run()` method is an upward filter bag. It contains framework calls that are executed **before** any method in our own code is called. This is why it cannot be included into any of our methods.

From the above example you can see how important the filter sets and the definition of a hotspot are for the actual results in the hotspot view. The same considerations apply to the allocation hotspot view.

## B Reference

### B.1 Getting Started

#### B.1.1 Quickstart dialog

By default, the quickstart dialog is shown when JProfiler is started. It contains a number of shortcuts that help to get started with profiling your application. The manual configuration dialog as well as all integration wizards are also available on the "New session" tab of the [start center](#) [p. 38]. Once you're familiar with JProfiler you can turn off the quickstart dialog by deselecting the check box `show quickstart at startup` at the bottom.

You can access the quickstart dialog at any later time by pressing `SHIFT-F1` or by choosing *Help->Show quickstart dialog* from JProfiler's main menu.

#### B.1.2 Running the demo sessions

For a quick tour of JProfiler's features, please run the **demo sessions**:

1. Start up JProfiler and wait for the [start center](#) [p. 38] to appear.
2. Choose one of the demo sessions from the list of available sessions.
3. Click **[Ok]**.
4. The profiling settings dialog appears. To accept the default settings, just click **[Ok]**.
5. A terminal window is opened for the demo process and the main window of JProfiler starts displaying [profiling information](#) [p. 80].

The Java source code for the demo sessions can be found in "`{JProfiler install directory}/demo/`" and "`{JProfiler install directory}/tomcat/webapps/examples`".

#### B.1.3 Overview of features

JProfiler's features are ordered into view sections. A view section can be made visible by selecting in JProfiler's sidebar. JProfiler offers the following view sections:

- [Memory profiling](#) [p. 88]  
Keep track of your objects and find out where the problem spots are.
- [a heap walker](#) [p. 99]  
Use the drill down capabilities of JProfiler's unique heap walker to find memory leaks.
- [CPU profiling](#) [p. 117]  
Find out where your CPU time is going and zero in on performance bottlenecks.
- [Thread profiling](#) [p. 129]  
Check the activity of your threads, resolve deadlocks and get detailed information on your application's monitor usage.
- [VM telemetry information](#) [p. 137]  
Unfold the statistical history of your application with JProfiler's virtual machine telemetry monitors.

In order to help you find JProfiler's features which are most important to you, we present a situational overview. There are two types of uses for a profiler which arise from different motivations:

- [Problem solving](#)

If you turn to a profiler with a problem in your application, it most likely falls into one of the following three categories:

- **Performance problem**

To find performance related problem spots in your application, turn to JProfiler's [CPU section](#) [p. 117] . Often, performance problems are caused by excessive creation of temporary objects. For that case, the [dynamic memory views](#) [p. 88] with their view mode set to "garbage collected objects" will show you where efforts to reduce allocations make sense.

- **Excessive memory consumption**

If your application consumes too much memory, the [dynamic memory views](#) [p. 88] will show you where the memory consumption comes from. With the [reference views](#) [p. 106] in the [heap walker](#) [p. 99] you can find out which objects are unnecessarily kept alive in the heap.

- **Memory leak**

If your application's memory consumption goes up linearly with time, you likely have a memory leak which is show stopper especially for application servers. The "mark current values and show differences" feature in the [memory section](#) [p. 88] and the [heap walker](#) [p. 99] will help you to find the cause.

- **Deadlock**

If you experience a deadlock, JProfiler's [deadlock detection graph](#) [p. 134] will help you to find the cause even for complex locking situations.

- **Hard to find bug**

A often overlooked but highly profitable use of a profiler is that of debugging. Many kinds of bugs are exceptionally hard to find by hand or by using a traditional debugger. Some bugs revolve around complex call stack scenarios (have a look at the [CPU section](#) [p. 117] ), others around entangled object reference graphs (have a look at the [heap walker section](#) [p. 99] ), both of which are not easy to keep track of.

Particularly JProfiler's [thread views](#) [p. 129] are of great help in multi-threaded situations, where race-conditions and deadlocks are hard to track down.

- **Quality assurance**

During a development process, it's a good idea to regularly run a profiler on your application to assess potential problem spots. Even though an application may prove to be "good enough" in test cases, an awareness for performance and memory bottlenecks enables you adapt your design decisions as the project evolves. In this way you avoid costly re-engineering when real-world needs are not met. Use the information presented in JProfiler's [telemetry section](#) [p. 137] to keep an eye on the evolution of your application. The ability to [save profiling snapshots](#) [p. 75] enables you to keep track of your project's evolution. The [offline profiling](#) [p. 140] capability allows you to perform automated profiling runs on your application.

#### **B.1.4 JProfiler's start center**

When JProfiler is started, the **start center** window appears. The start center is composed of three tabs:

- **Open session**

All sessions configured by you or the preconfigured demo sessions can be started by double clicking on a session or by selecting a session and clicking **[Ok]** at the bottom of the start center. In addition, sessions can be [edited](#) [p. 59] , copied or deleted by using the buttons on the right hand side of the dialog or by invoking the context menu.

- **New session**

Sessions can be created in one of two ways:

- **By manual configuration**

Use the **[New session]** button to [manually configure](#) [p. 59] a new session. After you finish configuring your session, it will be started.

- **Through an integration wizard**

Use the **[New server integration]** button to invoke the [integration wizard](#) [p. 39] selector. The **[New remote integration]** and **[New applet integration]** buttons are convenience shortcuts. After you finish configuring your session, you can either start the session immediately or the "open session" tab will be displayed with the new session selected.

- **Convert session**

Here, you can convert existing local sessions to remote sessions or [offline profiling sessions](#) [p. 140]. The existing local session that is chosen for conversion will not be modified.

- **Open snapshot**

Previously [saved sessions](#) [p. 75] can be opened from this tab by selecting the desired \*.jps file and clicking **[Ok]** at the bottom of the start center.

When you choose not to open a profiling session for an empty window and exit the start center by clicking the **[Cancel]** button, all of JProfiler's views are disabled and only the general settings (*Edit->General settings*) and the *Session* and *Help* menus are enabled.

The start center can be invoked at any later time

- by choosing *Session->Start center* or clicking on the corresponding  toolbar button.

If a session is currently active upon opening a session, it will be stopped after a confirmation dialog and the new session will replace all profiling data of the old session.

- by choosing *Session->Start center in new window*. A new main window of JProfiler will be opened, other active sessions will not be affected.

### B.1.5 Application server integration

JProfiler's application server integration wizard makes profiling application servers especially easy. It can be invoked in one of two ways:

- from the [start center](#) [p. 38] on the "new session" tab.
- by selecting *Session->New server integration* from JProfiler's main menu.

During the first step of the wizard you are asked to specify the product which is to be integrated. The second step asks you, whether the profiled application or application server is running on the local computer or on a remote machine. The subsequent steps depend on this choice. Please follow the instructions presented by the wizard.

If you miss support for a particular product, please don't hesitate to contact us through the [support request form](#)

### B.1.6 IDE integration

JProfiler [integrates seamlessly into several popular IDEs](#) [p. 42]. To bring up the integration dialog, please select *Session->IDE integrations* from JProfiler's main menu.

Select the desired IDE from the drop down list and click on **[Integrate]**. After completing the instructions, you can invoke JProfiler from the integrated IDE without having to specify class path, main class, working directory, used JVM and other options again. Also, source code navigation will be performed in the IDE.

See [here](#) [p. 42] for specific explanations regarding each IDE integration.



## B.2 JProfiler setup

### B.2.1 JProfiler setup wizard

If you run JProfiler for the first time, a setup wizard will guide you through the steps to collect all required information in order to create and run profiling sessions. Everything you enter here can be changed at a later time through the menus of a running instance of JProfiler. The setup wizard inquires about:

- **Importing settings from an older version of JProfiler**

If you would like to import your settings, please select the config file of your old JProfiler installation. The name of the config file is `config.xml`. This file is located in

- `{JProfiler installations directory}/config` for JProfiler <= 2.1.1
- `{User home directory}/.jprofiler2` for JProfiler >= 2.2
- `{User home directory}/.jprofiler3` for JProfiler >= 3.0

**Note:** If a JProfiler >= 2.2 installation is detected, it is imported automatically. This step can be used to migrate a JProfiler configuration to a different computer.

- **License information**

You are required to [enter your key and your personal information](#) [p. 41] before proceeding to the next step. **Note:** If a JProfiler >= 3.0 installation is detected, this step is omitted.

- **Java virtual machines installed on your system**

JProfiler will search your local fixed drives for installed JVMs. You may stop the search at any time and edit found JVMs or add new JVMs manually in the following screen. The "Check found JVMs" step of the wizard works like the ["Java VMs" tab](#) [p. 76] in JProfiler's [general settings](#) [p. 76] where JVMs may be changed later on. **Note:** If a JProfiler >= 2.2 installation is detected, this step is omitted.

- **IDE integration**

JProfiler can be fully integrated into [a number of popular IDEs](#) [p. 42]. By selecting the desired integration and clicking **[Integrate]** button you start the integration process. You can also perform the integrations later on by choosing *Session->IDE integrations* from the main menu.

### B.2.2 JProfiler licensing

Without a valid license, JProfiler cannot be started. If you don't have a key, visit [www.jprofiler.com](http://www.jprofiler.com) to get an evaluation key or to buy a license. If you have already obtained an evaluation key and were not able to evaluate JProfiler, please write to [sales@ej-technologies.com](mailto:sales@ej-technologies.com) to request a new key. JProfiler 3 does not work with a JProfiler 2 license key. Please upgrade your license on our website.

You can enter your license key in one of two ways:

- In JProfiler's [setup wizard](#) [p. 41]
- Through the menu of JProfiler's main screen: *Help->Enter license key*

Together with your license key, you are asked for your name and - if applicable - for the name of your company.

Please read the included file `license.txt` to learn about the scope of the license.

To make it easier for you to enter the license key, you may use the **[Paste from clipboard]** button, after copying any text fragment which contains the license key to your system clipboard. If a valid license key can be found in the clipboard content, it is extracted and displayed in the dialog.

## B.3 IDE integrations

### B.3.1 JProfiler IDE integrations

JProfiler can be integrated into the IDEs listed [here](#) [p. 42] . Installation is done either

- **Automatically (recommended)**

Select *Session->IDE integrations* from JProfiler's main menu or go to the [IDE integrations tab](#) [p. 79] in the [general settings dialog](#) [p. 76] . Now select the desired IDE from the drop down list, click on **[Integrate]** and [follow the instructions](#) [p. 79] .

- **Manually**

The directory *integrations* in the JProfiler install directory holds a number of archives which can be used for manually integrating JProfiler with any of the supported IDEs. See the file *README.txt* in the above directory for detailed instructions.

After completing the instructions, you can invoke JProfiler from the integrated IDE without having to specify class path, main class, working directory, used JVM and other options again.

All integrations insert toolbar buttons and menu entries into the respective IDE that run the application in the IDE with profiling enabled. On Windows and Mac OS X, the IDE reuses an already running instance of JProfiler to present profiling data. If JProfiler is not running, it will be started automatically.

Navigation to source code from JProfiler will be performed in the IDE, i.e. if you choose the "Show source" action for a class or a method, it will be displayed in the IDE and not in JProfiler's integrated source code viewer.

### B.3.2 JProfiler as an IntelliJ IDEA 3.x plugin

With JProfiler integrated into [JetBrain's IntelliJ IDEA](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** JProfiler requires at least IDEA 3.0 For IDEA 4.x, a [different plugin](#) [p. 43] with more capabilities is available.

The installation of the IntelliJ IDEA plugin is started by selecting "IntelliJ IDEA 3.x" on the

- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close IntelliJ IDEA while performing the plugin installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41] , please complete the entire setup first before starting IntelliJ IDEA.

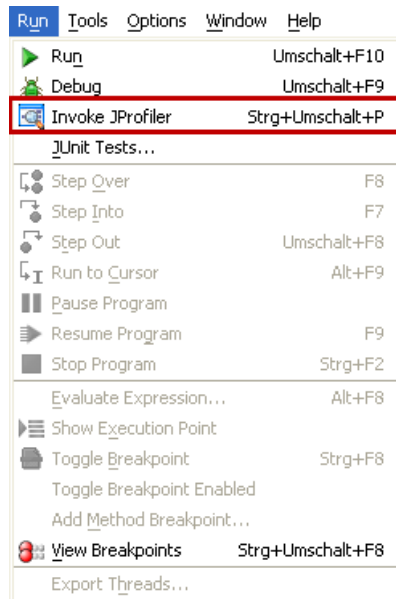
A file selector will then prompt you to locate the installation directory of IntelliJ IDEA.

After acknowledging the completion message, you can start IntelliJ IDEA and check whether the installation was successful. You should now see a menu entry *Run->Invoke JProfiler* in IDEA's main menu.

To profile your application from IntelliJ IDEA, choose *Run->Invoke JProfiler* from IDEA's main menu or click on the corresponding toolbar button. In the launch configuration selection dialog, you can decide whether you want to open a new window in JProfiler for the profiling session or if you wish to reuse the last window to accommodate the profiling session.



main toolbar with "JProfiler" button



"Run" menu with "JProfiler" action

If no instance of JProfiler is currently running, JProfiler is started, otherwise the running instance of JProfiler will be used for starting the application and for presenting profiling data. The information contained in the launch configuration is transmitted to JProfiler. With this information, JProfiler immediately starts a new profiling session. When you close the window, JProfiler asks you if you want to save the session for standalone execution. If you answer with yes, you can enter a name for the session. You will then be able to start it from the [start center](#) [p. 38] or from the [open session dialog](#) [p. 69] if you open JProfiler as a standalone application.

Only run configurations of type **Application** can be profiled with the IntelliJ IDEA 3.x integration. For profiling servers, please consider the IntelliJ IDEA 4.x integration where this is possible. Alternatively, you can profile any application and application server with the standalone JProfiler GUI.

When JProfiler is started from IntelliJ IDEA, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in IDEA and not in JProfiler's integrated source code viewer.

All profiling settings and view settings changes are persistent across session restarts.

**Note:** To configure a native library path, please define the VM parameter `-Djava.library.path` in IntelliJ IDEA, it will be translated to the native library path by JProfiler.

### B.3.3 JProfiler as an IntelliJ IDEA 4.x plugin

With JProfiler integrated into [JetBrain's IntelliJ IDEA](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** The IDEA 4.x plugin requires at least IDEA 4.0 For IDEA 3.x, a [different plugin](#) [p. 43] is available.

The installation of the IntelliJ IDEA plugin is started by selecting "IntelliJ IDEA 4.x" on the

- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]

- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close IntelliJ IDEA while performing the plugin installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41], please complete the entire setup first before starting IntelliJ IDEA.

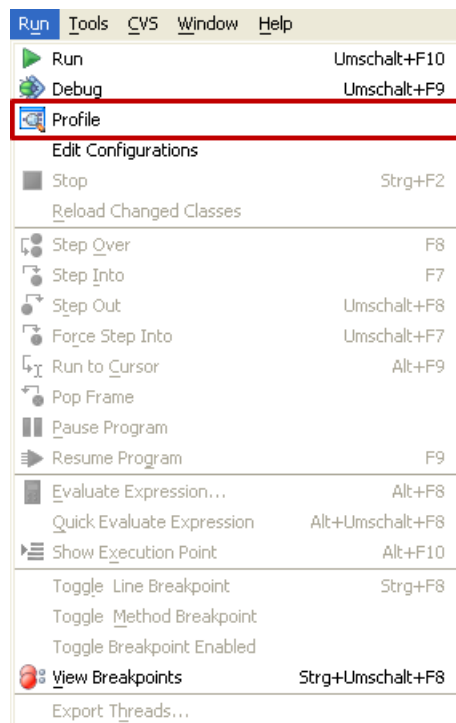
A file selector will then prompt you to locate the installation directory of IntelliJ IDEA.

After acknowledging the completion message, you can start IntelliJ IDEA and check whether the installation was successful. You should now see a menu entry *Run->Profile* in IDEA's main menu.

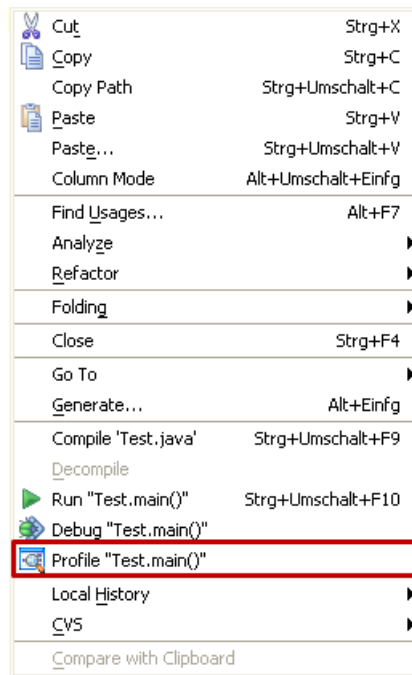
To profile your application from IntelliJ IDEA, choose one of the profiling commands in the *Run* menu, the context menu in the editor, or click on the corresponding toolbar button.



Main toolbar with "Profile" button

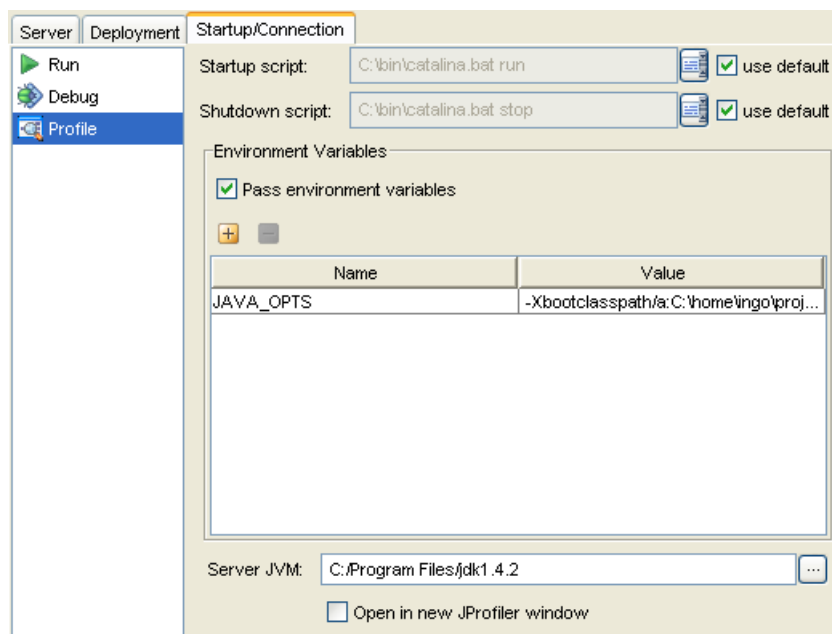


"Run" menu with "Profile" action



Editor context menu with "Profile" action

JProfiler can profile all run configuration types from IDEA, also applications servers. To configure further settings, please edit the run configuration, choose the "Startup/Connection" tab, and select the "Profile" entry. The screenshot below shows the startup settings for a local server configuration. Depending on the run configuration type, you can adjust JVM options or retrieve profiling parameters for remote profiling.



Startup settings for profiling of a local server configuration

For all run configuration types you can decide whether you want to open a new window in JProfiler for the profiling session or if you wish to reuse the last window to accommodate the profiling session.

The profiled application is then started just as with the usual "Run" commands. If no instance of JProfiler is currently running, JProfiler is also started, otherwise the running instance of JProfiler will be used for presenting profiling data.

When JProfiler is started from IntelliJ IDEA, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in IDEA and not in JProfiler's integrated source code viewer.

### B.3.4 JProfiler as an eclipse 2.x / WSAD 5.x plugin

When JProfiler is integrated into the [eclipse 2.x IDE](#) or into [WSAD 5.x](#) (which is based on eclipse 2.1), JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** The eclipse 2.x plugin works with eclipse 2.0, eclipse 2.1 and WSAD 5.x. In the following text, the IDE will always be called "eclipse". For eclipse 3, a [different plugin](#) [p. 48] with more capabilities is available.

#### Profiling a J2EE application from WSAD:

With the IDE integration for WSAD, only run configurations of type "Java application" can be profiled.

To profile a J2EE application from within WSAD, please choose *Session->Integration wizards->New server integration* from JProfiler's main menu and select the server integration type **IBM Websphere started from WSAD**. The [integration wizard](#) [p. 39] will lead you step by step through the required modifications to profile your server.

The installation of the eclipse plugin is started by selecting "eclipse 2.x" or "IBM WSAD 5.x" on the


- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close eclipse while performing the plugin installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41], please complete the entire setup first before starting eclipse.

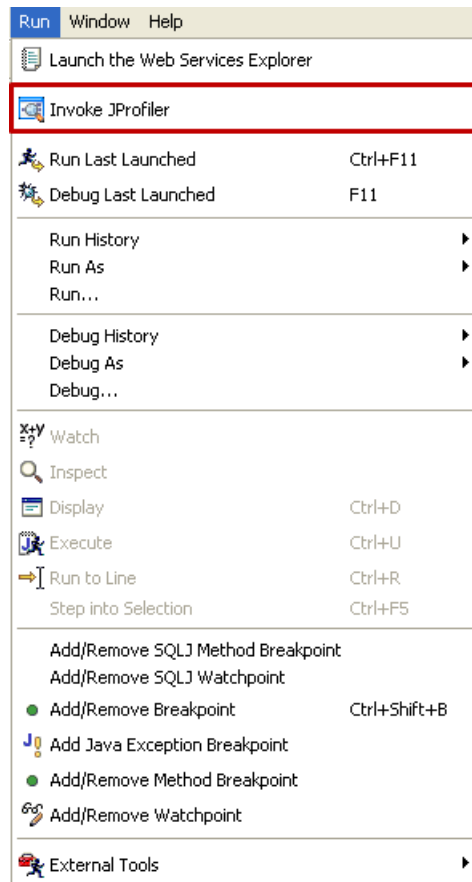
A file selector will then prompt you to locate the **installation directory** of eclipse. For WSAD, this is the directory that contains the "eclipse" subdirectory.

After acknowledging the completion message, you can start eclipse and check whether the installation was successful. If the menu item *Run->Invoke JProfiler* does not exist in the **Java perspective**, please enable the JProfiler plugin for this perspective under *Window->Customize perspective* by opening the **Other** section and checking "JProfiler".

To profile your application from eclipse, choose *Run->Invoke JProfiler* from eclipse's main menu or click on the corresponding  toolbar button.



Main toolbar with "JProfiler" button



"Run" menu with "JProfiler" action

A dialog with the available launch configurations will be displayed. Choose the desired configuration and press **[Ok]**. If JProfiler has already been opened from eclipse, you can check the *Open in new window* option to open a new window of JProfiler for the profiling session. Otherwise the last used main window will accommodate the profiling session.

If no instance of JProfiler is currently running, JProfiler is started, otherwise the running instance of JProfiler will be used for starting the application and for presenting profiling data. The information contained in the launch configuration is transmitted to JProfiler. With this information, JProfiler immediately starts a new profiling session. When you close the window, JProfiler asks you if you want to save the session for standalone execution. If you answer with yes, you can enter a name for the session. You will then be able to start it from the [start center](#) [p. 38] or from the [open session dialog](#) [p. 69] if you open JProfiler as a standalone application.

All profiling settings and view settings changes are persistent across session restarts.

When JProfiler is used with the eclipse integration, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in eclipse and not in JProfiler's integrated source code viewer.

**Note:** To configure a native library path, please define the VM parameter `-Djava.library.path` in eclipse, it will be translated to the native library path by JProfiler.

The used JProfiler installation can be changed by repeating the integration from JProfiler or by adjusting the JProfiler executable in eclipse under *Window->Preferences->JProfiler*. When you upgrade to a newer version of JProfiler, make sure to repeat the integration, since the plugin has to be updated, too.

### B.3.5 JProfiler as an eclipse 3.x plugin

When JProfiler is integrated into the [eclipse 3.x IDE](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** The eclipse 3.x plugin works with eclipse 3.0 and eclipse 3.1. For eclipse 2.x, a [different plugin](#) [p. 46] is available.

The installation of the eclipse plugin is started by selecting "eclipse 3.x" on the

- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close eclipse while performing the plugin installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41], please complete the entire setup first before starting eclipse.

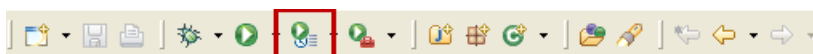
A file selector will then prompt you to locate the **installation directory** of eclipse.

After acknowledging the completion message, you can start eclipse and check whether the installation was successful. If the menu item *Run->Profile ...* does not exist in the **Java perspective**, please enable the JProfiler plugin for this perspective under *Window->Customize perspective* by bringing the **Command** tab to front and selecting the "Profile" checkbox.

eclipse provides shared infrastructure for profiling plugins that allows only one active profiler at a time. If another profiler has registered itself in eclipse, JProfiler will show a collision message dialog at startup. Please go to the *plugin* directory in your eclipse installation and delete the plugins that are specified in the warning message in order to guarantee that JProfiler will be used when you click on one of the profiling actions.

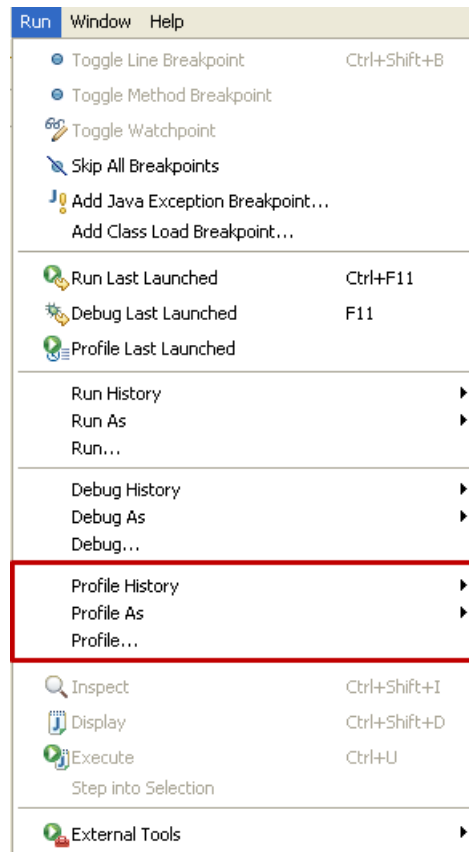
If you are upgrading the integration from JProfiler <=3.2, please delete your Eclipse "configuration" directory except the config.ini file before restarting Eclipse. This is to avoid a common Eclipse 3.x plugin cache bug.

To profile your application from eclipse, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button. The profile commands are equivalent to the debug and run commands in eclipse and are part of eclipse's infrastructure.



Main eclipse toolbar with "Profile" button





eclipse "Run" menu with "Profile" actions

The profiled application is then started just as with the usual "Run" commands. If no instance of JProfiler is currently running, JProfiler is also started, otherwise the running instance of JProfiler will be used for presenting profiling data.

Every time a run configuration is profiled, a dialog box is brought up that asks you whether a new window should be opened in JProfiler. To get rid of this dialog, you can select the "Don't ask me again" checkbox. The window policy can subsequently be configured in the JProfiler settings in eclipse (see below).

All profiling settings and view settings changes are persistent across session restarts.

When JProfiler is used with the eclipse integration, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in eclipse and not in JProfiler's integrated source code viewer.

Several JProfiler-related settings can be adjusted in eclipse under *Window->Preferences->JProfiler*.

The used JProfiler installation can be changed by repeating the integration from JProfiler or by adjusting the JProfiler executable in the corresponding text field. When you upgrade to a newer version of JProfiler, make sure to repeat the integration, since the plugin has to be updated, too.

The window policy can be configured as

- **Ask each time**

Every time you profile a run configuration, a dialog box will ask you whether a new window should be opened in JProfiler. This is the default setting.

- **Always new window**

Every time you profile a run configuration, a new window will be opened in JProfiler.

- **Reuse last window**

Every time you profile a run configuration, the last window will be reused in JProfiler.

You can manually repeat the collision detection that is performed at startup. With the corresponding checkbox, you can also switch off collision detection at startup.

### **B.3.6 JProfiler as a JBuilder OpenTool**

With JProfiler integrated into Borland's [JBuilder](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** JProfiler requires at least JBuilder 7.0

The installation of the JBuilder OpenTool is started by selecting "JBuilder 7 to 2005" on the

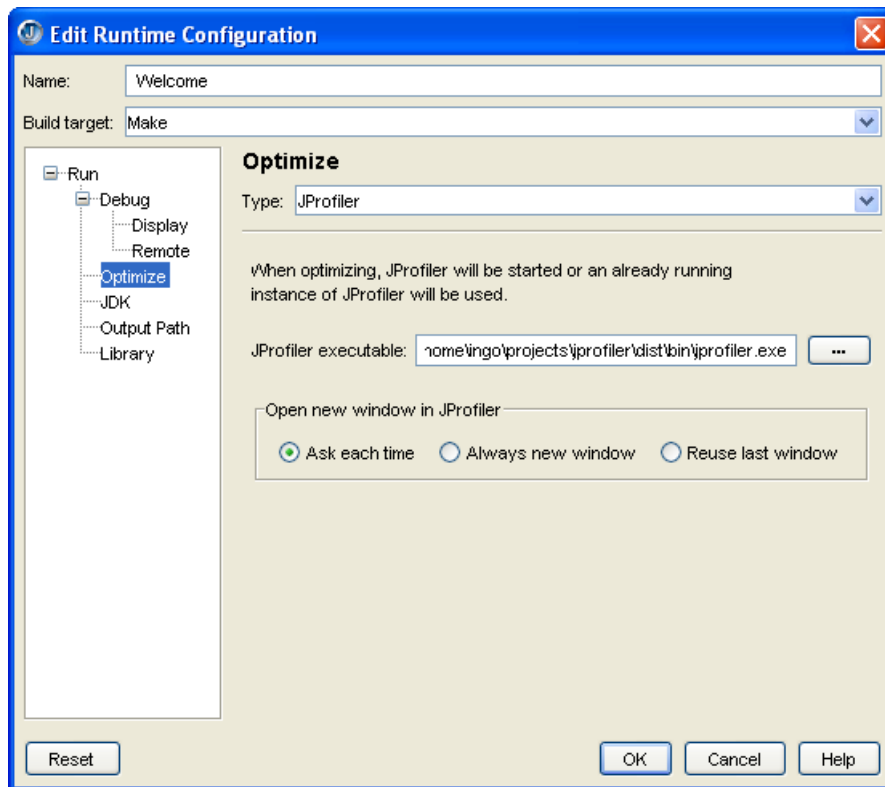
- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close JBuilder while performing the OpenTool installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41] , please complete the entire setup first before starting JBuilder.

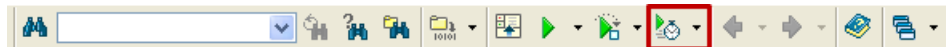
A file selection box will then prompt you to locate the installation directory of JBuilder.

After acknowledging the completion message, you have to start JBuilder and set JProfiler as the optimizer for your project. Invoke *Run->Configurations* from JBuilder's main menu, select a runtime configuration, press **[Edit]** and select the "Optimize" tab in the resulting runtime properties dialog. If an optimizer type with the name "JProfiler" exists on this tab, the OpenTool was recognized correctly. Activate this optimizer and then click **[Ok]**.

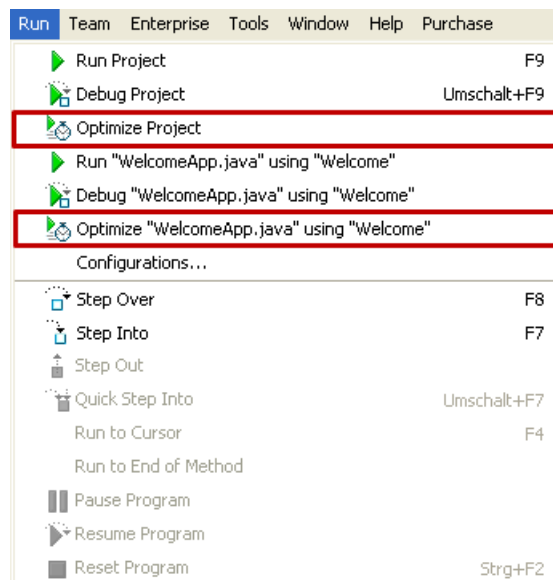


Optimizer configuration dialog

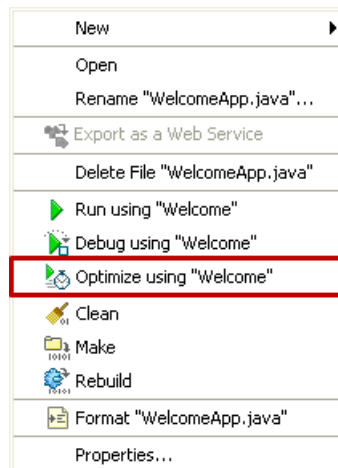
To profile your application from JBuilder, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button.



Main toolbar with "Optimize" button



"Run" menu with "Optimize" actions



Project explorer context menu with "Optimize" action

The profiled application is then started just as with the usual "Run" commands. If no instance of JProfiler is currently running, JProfiler is also started, otherwise the running instance of JProfiler will be used for presenting profiling data.

Every time a run configuration is profiled, a dialog box is brought up that asks you whether a new window should be opened in JProfiler. To get rid of this dialog, you can select the "Don't ask me again" checkbox. The window policy can subsequently be configured in the optimizer settings in JBuilder (see below).

All profiling settings and view settings changes are persistent across session restarts.

When JProfiler is started from JBuilder, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in JBuilder and not in JProfiler's integrated source code viewer.

Several JProfiler-related settings can be adjusted in JBuilder under *Run->Configurations->Edit->Optimize*:

The used JProfiler installation can be changed by repeating the integration from JProfiler or by adjusting the JProfiler executable in the corresponding text field. When you upgrade to a newer version of JProfiler, make sure to repeat the integration, since the OpenTool has to be updated, too.

The window policy can be configured as

- **Ask each time**

Every time you profile a run configuration, a dialog box will ask you whether a new window should be opened in JProfiler. This is the default setting.

- **Always new window**

Every time you profile a run configuration, a new window will be opened in JProfiler.

- **Reuse last window**

Every time you profile a run configuration, the last window will be reused in JProfiler.

### B.3.7 JProfiler as a JDeveloper Addin

With JProfiler integrated into Oracle's [JDeveloper](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** JProfiler requires at least JDeveloper 10g.

The installation of the JDeveloper addin is started by selecting "JDeveloper 10g" on the

- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close JDeveloper while performing the addin installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41] , please complete the entire setup first before starting JDeveloper.

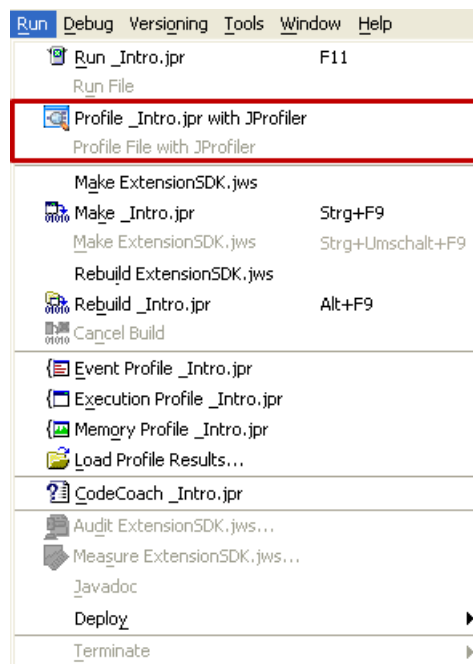
A file selection box will then prompt you to locate the installation directory of JDeveloper.

After acknowledging the completion message, you can start JDeveloper and check whether the installation was successful. You should now see a menu entry *Run->Profile with JProfiler* in JDeveloper's main menu.

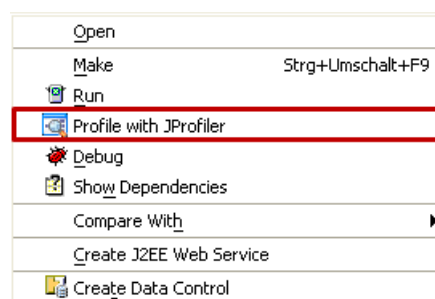
To profile your application from JDeveloper, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button.



Main toolbar with "JProfiler" button



"Run" menu with "JProfiler" actions



Project explorer context menu with "JProfiler" action

The profiled application is then started just as with the usual "Run" commands. If no instance of JProfiler is currently running, JProfiler is also started, otherwise the running instance of JProfiler will be used for presenting profiling data.

Every time a run configuration is profiled, a dialog box is brought up that asks you whether a new window should be opened in JProfiler. To get rid of this dialog, you can select the "Don't ask me again" checkbox. The window policy can subsequently be configured in the "JProfiler" node in the settings dialog of JDeveloper (see below).

All profiling settings and view settings changes are persistent across session restarts.

When JProfiler is started from JDeveloper, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in JDeveloper and not in JProfiler's integrated source code viewer.

Several JProfiler-related settings can be adjusted in JDeveloper under *Tools->Preferences->JProfiler*.

The used JProfiler installation can be changed by repeating the integration from JProfiler or by adjusting the JProfiler executable in the corresponding text field. When you upgrade to a newer version of JProfiler, make sure to repeat the integration, since the addin has to be updated, too.

The window policy can be configured as

- **Ask each time**

Every time you profile a run configuration, a dialog box will ask you whether a new window should be opened in JProfiler. This is the default setting.

- **Always new window**

Every time you profile a run configuration, a new window will be opened in JProfiler.

- **Reuse last window**

Every time you profile a run configuration, the last window will be reused in JProfiler.

### **B.3.8 JProfiler as a Netbeans 3.x /Sun ONE Studio module**

With JProfiler integrated into Sun Microsystems' [Netbeans\(TM\)](#) or [Sun ONE Studio\(TM\)](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** The Netbeans 3.x plugin needs at least Netbeans 3.3 or Sun ONE Studio 4. In the following text, the IDE will always be called "Netbeans". For Netbeans 4, a [different module](#) [p. 56] with more capabilities is available.

The installation of the Netbeans module is started by selecting "Netbeans IDE 3.3 to 3.x" or "Sun ONE studio 4 to 9" on the

- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**

**Reminder:** Please close Netbeans while performing the module installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41], please complete the entire setup first before starting Netbeans.

A file selection box will then prompt you to locate the installation directory of Netbeans. In the next step, you are asked whether the installation should be performed globally, or for a single user only. A single user installation is mostly of interest in network installations where the user cannot write to the Netbeans installation directory. If you decide for a single user installation, another file selection

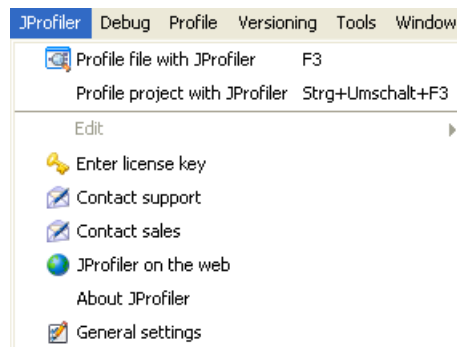
box will then prompt you to locate your Netbeans user directory. This is a version-specific directory under `.netbeans` in your user home directory.

The Netbeans updater is then invoked and the module is installed. After acknowledging the completion message, you can start Netbeans and check whether the installation was successful. You should now see a menu entry *JProfiler* top-level menu in Netbeans' main menu.

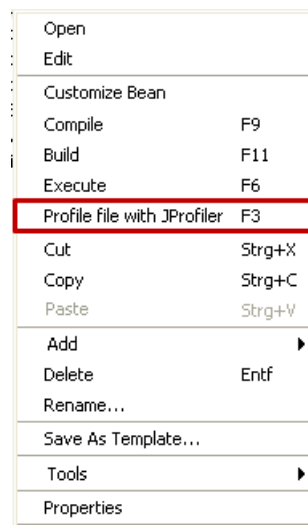
To profile your application from Netbeans, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button.



Main toolbar with "JProfiler" button



"JProfiler" menu



Explorer context menu with "JProfiler" action

The profiled application is then started just as with the usual "Run" commands. When a profiling session is started,

- **in all versions except Netbeans 3.6**

the current workspace is switched to the `Profiling` workspace, which contains an output window, an execution task list and - once a profiling session is started - a tabbed JProfiler window with one tab for each profiling session.

- **in Netbeans 3.6**

a new tab with a JProfiler window is created.


Profiling sessions are closed by closing the corresponding tab (in all versions except Netbeans 3.6 via its context menu). Apart from the excluded tool bar buttons for "Attach/Detach" and "Session settings", the JProfiler window and its views are exactly the same as in the [standalone version](#) [p. 80].

All profiling settings and view settings changes are persistent across session restarts.

When JProfiler is used with the Netbeans integration, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in Netbeans and not in JProfiler's integrated source code viewer.

The *JProfiler* menu in Netbeans' main menu bar contains all actions required to run JProfiler from within Netbeans:

- **Profile file with JProfiler**

-  Start profiling the currently selected class.


- **Profile project with JProfiler**

- Start profiling the main class of the current project.


- **Edit**

- Contains the JProfiler's view specific *Edit* menu which is active only during profiling.


- **Enter license key**

-  Allows you to [enter your license key](#) [p. 41].


- **Contact sales**

-  Brings up your default mail client to write an e-mail to ej-technologies' sales department.

- **Contact support**

-  Brings up your default mail client to write an e-mail to ej-technologies' support department. The license key is automatically included in the subject of the e-mail.

- **JProfiler on the web**

-  Connects to JProfiler's web site in the default web browser.

- **About JProfiler**

- Shows general information about your copy of JProfiler and its license status.

### B.3.9 JProfiler as a Netbeans 4.x

With JProfiler integrated into Sun Microsystems' [Netbeans\(TM\)](#) JProfiler can be invoked from within the IDE without any further need for session configuration.

**Requirements:** The Netbeans 4.x plugin needs at least Netbeans 4.0. For Netbeans 3, a [different module](#) [p. 54] is available.

The installation of the Netbeans module is started by selecting "Netbeans IDE 4.0" on the

- IDE integration tab of JProfiler's [setup wizard](#) [p. 41]
- [miscellaneous options tab](#) [p. 79] of JProfiler's [general settings](#) [p. 76] (use *Session->IDE integrations* in JProfiler's main menu as a shortcut).

and clicking on **[Integrate]**



**Reminder:** Please close Netbeans while performing the module installation. If you are performing the installation from JProfiler's [setup wizard](#) [p. 41] , please complete the entire setup first before starting Netbeans.

A file selection box will then prompt you to locate the installation directory of Netbeans. In the next step, you are asked whether the installation should be performed globally, or for a single user only. A single user installation is mostly of interest in network installations where the user cannot write to the Netbeans installation directory. If you decide for a single user installation, another file selection box will then prompt you to locate your Netbeans user directory. This is a version-specific directory under `.netbeans` in your user home directory.

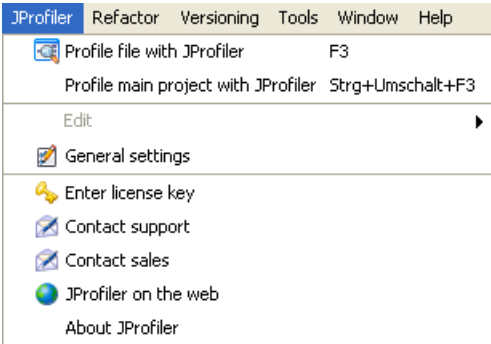
The Netbeans updater is then invoked and the module is installed. After acknowledging the completion message, you can start Netbeans and check whether the installation was successful. You should now see a menu entry *JProfiler* top-level menu in Netbeans' main menu.

You can profile **web applications** with the integrated Tomcat or with any other Tomcat server configured in Netbeans. When your main project is a web project, selecting "Profile project with JProfiler" (see below) starts the Tomcat server with profiling enabled. Please make sure to stop the Tomcat server before trying to profile it.

To profile your application from Netbeans, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button.



Main toolbar with "JProfiler" button



"JProfiler" menu

Open	
Compile File	F9
Run File	Umschalt+F6
Profile file with JProfiler	F3
Debug File	Strg+Umschalt+F5
Cut	Strg+X
Copy	Strg+C
Paste	Strg+V
Add	
Delete	Entf
Save As Template...	
Find Usages...	Alt+F7
Refactor	
Tools	
Properties	

The profiled application is then started just as with the usual "Run" commands. When a profiling session is started, a new tab with a JProfiler window is created.

Profiling sessions are closed by closing the corresponding tab. Apart from the excluded tool bar buttons for "Attach/Detach" and "Session settings", the JProfiler window and its views are exactly the same as in the [standalone version](#) [p. 80] .

All profiling settings and view settings changes are persistent across session restarts.

When JProfiler is used with the Netbeans integration, the "Show source" action for a class or a method in one of JProfiler's view will show the source element in Netbeans and not in JProfiler's integrated source code viewer.

The *JProfiler* menu in Netbeans' main menu bar contains all actions required to run JProfiler from within Netbeans:

- **Profile file with JProfiler**



Start profiling the currently selected class.

- **Profile project with JProfiler**

Start profiling the main class of the current project.

- **Edit**

Contains the JProfiler's view specific *Edit* menu which is active only during profiling.

- **Enter license key**



Allows you to [enter your license key](#) [p. 41] .

- **Contact sales**



Brings up your default mail client to write an e-mail to ej-technologies' sales department.

- **Contact support**



Brings up your default mail client to write an e-mail to ej-technologies' support department. The license key is automatically included in the subject of the e-mail.

- **JProfiler on the web**



Connects to JProfiler's web site in the default web browser.

- **About JProfiler**

Shows general information about your copy of JProfiler and its license status.

## B.4 Managing sessions

### B.4.1 Sessions overview

The information required to start a profiling run is called a **session**. Sessions are saved in the file `{User home directory}/.jprofiler3/config.xml` and can be easily migrated to a different computer by importing this file in the [setup wizard](#) [p. 41] . When upgrading JProfiler, your settings of older installations are imported automatically.

Sessions are created

- on the "new session" tab of JProfiler's [start center](#) [p. 38] .
- by selecting *Session->New session* from JProfiler's main menu.
- automatically by JProfiler's [application server integration wizard](#) [p. 39] .

Sessions are edited, deleted and opened

- in JProfiler's [start center](#) [p. 38] .
- through the [open session dialog](#) [p. 69] which is accessible from JProfiler's main menu via *Session->Open session*.

The settings for a session are collected in 2 steps:

- **application settings**

The [application settings dialog](#) [p. 59] collects all information that is required to start your application with profiling enabled. This dialog is presented when you create a new session. From a running session it is available through JProfiler's main menu via *Edit->Application settings*. If you use an [IDE integration](#) [p. 42] , this information will be provided by the IDE.

- **profiling settings**

In the [profiling settings dialog](#) [p. 64] you can configure the way your application is profiled and change the focus of a profiling run toward performance or accuracy, CPU or memory profiling. When profiling, there is a general trade-off between profiling overhead and information depth. Most likely your personal requirements will change from profiling run to profiling run, so these settings are displayed every time before your application is started.

Changes in the parameters of a session that are specified in the [application settings dialog](#) [p. 59] or the [profiling settings dialog](#) [p. 64] are not made effective while that session is running. To change session parameters, you need to detach JProfiler from your application and restart the session. On the other hand, **view parameters** are adjustable during a running session and are saved separately for each session.

### B.4.2 Application settings

#### B.4.2.1 Application settings dialog

The application settings dialog collects all information that is required to start your application with profiling enabled. This dialog is presented when you [create a new session](#) [p. 59] . From a running session it is available through JProfiler's main menu via *Edit->Application settings*.

In the application settings dialog you configure the following details:

- **Session name**


Every session has a unique name that is presented in the "Open session" pane of the [start center](#) [p. 38] and in the [open session dialog](#) [p. 69] . It is also used for the title of the main window and

the terminal window. Next to the name text field you see an ID which is used for choosing the session in offline profiling.


- **Session type**

There are four different session types. Depending on this choice, the middle part of the tab will display different options.


- [Local sessions](#) [p. 61]

 A local session starts your application when the session is opened. You need to specify the virtual machine, as well as your application's class path, main class, parameters and working directory. Your application will be started in a separate terminal window and is prevented from exiting while JProfiler is attached to it. Local sessions are most convenient for profiling GUI and console applications where you have written the main class yourself.

- [Remote sessions](#) [p. 62]


 A remote session connects to a running application which has been [started with JProfiler's profiling agent](#) [p. 69]. The profiling agent listens on the default port of 8849 which can be changed in the agent's initialization parameters. Remote sessions are most convenient for profiling server applications on remote machines and application servers where you write classes which are loaded and invoked within the application server's framework.

- [Applet sessions](#) [p. 62]

 Applet sessions are used for profiling applets with Sun's applet viewer which is shipped with every JDK. You only need to supply the URL to a HTML page containing the applet.


**Note:** If the applet view is too restrictive for your applet, please use the **Java plugin integration wizard** available on the `New session` tab of the [start center](#) [p. 38].

- [Servlet sessions](#) [p. 63]

 JProfiler comes with an integrated Apache Tomcat servlet engine. To quickly profile servlets and Java server pages, you just need to supply the path to the `web.xml` file associated with your project.

**Note:** This session type is for demonstration purposes only. To profile your own installation of Tomcat or other application servers, please use the **application server integration wizards** available on the `New session` tab of the [start center](#) [p. 38].

- [Web Start sessions](#) [p. 63]

 JProfiler can profile [Java Web Start](#) applications. You only need to supply the URL for the JNLP file or select a cached application.

- **Additional Java file path**

With the two radio buttons on the left you choose whether to set the

- **Additional class path**

The session-specific additional class path allows you to enter directories and jar files which are prepended to the [default class path](#) [p. 78] (see there for more details on choosing the class path). The class path is also used by the [bytecode viewer](#) [p. 86] to find class files for display.

- **Additional source path**

The session-specific additional source path allows you to enter directories which are prepended to the [default source path](#) [p. 78] (see there for more details on choosing the source path). Note that the sources of the selected JDK contained in `src.jar` or `src.zip` will be automatically appended if they are installed. The source path is used by the [source code viewer](#) [p. 86] to display Java sources.

- **Additional native library path**


The session-specific native library path allows you to enter directories which are prepended to the [default native library path](#) [p. 78] (see there for more details on choosing the native library path). You only need the native library path for loading native libraries with calls to `java.lang.System.loadLibrary()` or for resolving dependent libraries that have to be dynamically loaded by your native libraries.

Adjusting the class and source path during an active session is effective for the [source code and bytecode viewer](#) [p. 86] only.

#### **B.4.2.2 Local session**

If the session type in the [application settings dialog](#) [p. 59] is set to "Local", the following settings are displayed in the middle part of the dialog:

- **Java VM**

Choose the Java VM to run your application. Java VMs are configured on the ["Java VMs" tab](#) [p. 76] of JProfiler's [general settings](#) [p. 76] which are accessible by clicking the  **[General settings]** button on the bottom of the dialog.

- **Working directory**

Choose the directory in which your **java** process will be started either manually or by clicking on the **[...]** button to bring up a file chooser. As long as you have not selected a particular directory, this option is set to `[startup directory]` which means that JProfiler's startup directory will also be your application's working directory.

- **VM arguments**

If your application needs virtual machine arguments of the form `-Dproperty=value`, you can enter them here. Parameters that contain spaces must be surrounded with double quotes (like `"-Dparam=a parameter with spaces"`).

- **Main class or executable JAR**

Enter the fully qualified name of your main class or the path to an executable JAR file here. If you enter a main class, it has to be contained in either in the [default class path](#) [p. 78] or in the additional class path (see above).

Clicking on the **[...]** button brings up menu that lets you

- **Search the classpath**

If you have already configured your classpath, this option will search for classes with a main method and present them in the [main class selection dialog](#) [p. 64] .

- **Browse for an executable JAR file**

This brings up a file chooser where you can select an executable JAR file.

- **Browser for a .class file**

This brings up a file chooser where you can select the `*.class` file of the desired main class. A dialog box will ask you whether to add the associated class path root directory to the class path.

- **Arguments**

This is the place to enter any arguments you want to supply to the main class of your application. Arguments that contain spaces must be surrounded with double quotes (like `"a parameter with spaces"`).

#### B.4.2.3 Remote session

If the session type in the [application settings dialog](#) [p. 59] is set to "Remote", the following settings are displayed in the middle part of the dialog:

- **Host**

Enter the host on which the application you want to profile is running either as a DNS name or as an IP address. If this is your local computer, you may enter `localhost`.

- **Port**

Choose the port on which the remote profiling agent is listening. If you have not supplied a [port parameter](#) [p. 70], the default port 8849 is the correct choice. This default can be restored by clicking the **[Default]** button on the right side of the text field.

- **Timeout**

Choose the timeout in seconds after which JProfiler will give up trying to connect to the remote application.

- **Start command**

If you enable the "start command" checkbox and enter the path to an executable in the text field to the right, JProfiler will execute this command before trying to connect to the remote application. The output of that command will be displayed in a terminal window similar to the ["local" session type](#) [p. 59]. In this case JProfiler has full control over the life cycle of the profiled application. If the terminal window is closed, the stop button is clicked or JProfiler is exited, the process will be killed if it is still alive.

The application server integration wizard uses start commands to make it easy to profile application servers. Should you want to take control of the launching of the application server you can temporarily uncheck the "start command" checkbox while preserving the suitable start command.

- **Stop command**


If you enable the "stop command" checkbox and enter the path to an executable in the text field to the right, JProfiler will execute this command when disconnecting from the remote application, i.e. when the terminal window is closed, the stop button is clicked or JProfiler is exited.

The application server integration wizard uses stop commands where possible.

#### B.4.2.4 Applet session

If the session type in the [application settings dialog](#) [p. 59] is set to "Applet", the following settings are displayed in the middle part of the dialog:

- **Java VM**

Choose the Java VM to run your applet. The main class `sun.applet.AppletViewer` from the `tools.jar` of the selected JVM will be used to show the applet. Java VMs are configured on the ["Java VMs" tab](#) [p. 76] of JProfiler's [general settings](#) [p. 76] which are accessible by clicking the  **[General settings]** button on the bottom of the dialog.

- **URL**


Enter a URL pointing to an HTML page which contains the applet. By clicking on the **[...]** button you can bring up a file chooser to select an HTML file on your file system.

**Note:** If the applet view is too restrictive for your applet, please use the **Java plugin integration wizard** available on the `New session` tab of the [start center](#) [p. 38].

#### B.4.2.5 Servlet session

If the session type in the [application settings dialog](#) [p. 59] is set to "Servlet", the following settings are displayed in the middle part of the dialog:

- **Java VM**

Choose the Java VM to run JProfiler's integrated Apache Tomcat. Java VMs are configured on the ["Java VMs" tab](#) [p. 76] of JProfiler's [general settings](#) [p. 76] which are accessible by clicking the  **[General settings]** button on the bottom of the dialog.

- **Tomcat HTTP port**

Select the port on which the integrated Apache Tomcat will listen for HTTP requests. After you open a servlet session, a web browser pointing to `http://localhost:[port]` will be opened.

- **Path to web.xml**

Select the `web.xml` file associated with your web project. Your web project will be mounted as the Tomcat root context. If you leave this path empty, the default root context of JProfiler's integrated Apache Tomcat will be profiled. By clicking on the [...] button you can bring up a file chooser to select a `web.xml` file.

- **Context path**

If your web application requires a different context path, you can enter it here.

The integrated Tomcat is located in the `tomcat` directory of your JProfiler installation. JProfiler modifies `conf/server.xml` on every start of a servlet. The first `Context` that is found in this file is modified.

**Note:** This session type is for demonstration purposes only. To profile your own installation of Tomcat or other application servers, please use the **application server integration wizards** available on the [New session tab](#) of the [start center](#) [p. 38] .

#### B.4.2.6 Web start session

If the session type in the [application settings dialog](#) [p. 59] is set to "Web Start", the following settings are displayed in the middle part of the dialog:

- **URL of the JNLP file**

Every Web Start application is launched by means of a launch descriptor called a JNLP file. Enter the URL of the JNLP file in the text field. By clicking on the [...] button you can bring up a dialog which shows the JNLP URLs of all applications which have already been downloaded by Java Web Start. Choose one in the list and press **[Ok]** to transfer the URL to the text field.

- **JVM for running Java Web Start**

Choose the Java VM to run Java Web Start. This applies only to the Web Start launcher, **not** to the actual JVM your application will be run with. For Java  $\geq 1.4.2$  this automatically selects the Java Web Start installation directory.

- **JVM for profiling**

Choose the Java VM that will be used by Web Start to run the application. To make really certain which JVM will be used, it is recommended to deactivate all other JVMs in the Java control panel, or to specify an exact version in the JNLP file.

The selected JVM does not necessarily have to be the exact same JVM that Web Start will choose to launch the application, however, it should at least be of the same minor version. If there's a version mismatch, the profiling parameters might be unsuitable and a "Could not create JVM message" might be the consequence.

- **Web Start version**


Since Java 1.4.2, Java Web Start is integrated into the JRE, before that it has a separate version number and it was possible to install it to an arbitrary location. Please choose whether you wish to profile

- **Java Web Start bundled with JRE >= 1.4.2**

If this option is selected, the Java Web Start installation is determined by the above setting "JVM for running Java Web Start".

- **Java Web Start version 1.2**

If this option is selected, you have to choose the directory where Java Web Start is installed. On Windows, this would usually be *C:\Program Files\Java Web Start*.

Note: Java VMs are configured on the ["Java VMs" tab](#) [p. 76] of JProfiler's [general settings](#) [p. 76] which are accessible by clicking the  **[General settings]** button on the bottom of the dialog.

#### **B.4.2.7 Main class selection dialog**


This dialog is relevant for the configuration of a [local session](#) [p. 61]. It is shown if you click on the [...] button next to the `Main class or executable JAR` text field and choose the `Search classpath` option from the drop down menu.

In order to select a main class, please double-click on it or select it first and click **[Ok]**. The main class will then be shown in the `Main class or executable JAR` text field of the local session panel. You can also leave the dialog with the **[Cancel]** to avoid making any changes to the configuration.

### **B.4.3 Profiling settings**

#### **B.4.3.1 Profiling settings dialog**

The profiling settings dialog can be invoked from


- the [the profiling settings templates dialog](#) [p. 68] that is displayed just before a session is started.
- the [application settings dialog](#) [p. 59].
- JProfiler's main menu and the toolbar. The  toolbar button and the menu item *Edit->Profiling settings* open the profiling settings dialog. To apply changes in the profiling settings, you must restart the session.

Please see the detailed discussion in the [help topic on profiling settings](#) [p. 11] to get a background understanding of the various available settings.

The dialog is grouped into several tabs:

- [Call tree collection](#) [p. 65]  
Set call tree collection options for the session.
- [Profiling features](#) [p. 66]  
Disable profiling features to increase execution speed.
- [Console](#) [p. 67]  
Configure console settings.
- [Miscellaneous](#) [p. 67]  
Set miscellaneous options for profiling.



Other settings, which concern the presentation of profiling data are called **view settings** and are accessible from the main toolbar  as well as from context sensitive menus in each view. View settings are persistent as well and are saved automatically for each session.

#### B.4.3.2 Adjusting call tree collection options

On this tab of the [profiling settings dialog](#) [p. 64], you can adjust the call tree collection method and the [filter sets](#) [p. 77] which are active for the session being edited. These settings influence the detail level and the execution speed of the profiled application.

##### Call tree collection method

Select the call tree collection method for CPU profiling as one of

- **dynamic instrumentation**

With dynamic instrumentation selected, JProfiler modifies the bytecode of all unfiltered classes on the fly as they are loaded by the JVM to include hooks for CPU profiling. Java core classes (`java.*`) cannot be profiled this way and are filtered automatically. Dynamic instrumentation is considerably faster than full instrumentation.

- **sampling**

When sampling is enabled JProfiler inspects the call stacks of all threads periodically. The period between to measurements for each thread is booked to the greatest common denominator of the two class stacks. Sampling is fast even without any filters, but accuracy is low and no invocation count is displayed.

If sampling is enabled, the **sampling frequency** can be adjusted with a slider. The possible range is 1 - 20 ms. A lower sampling frequency incurs a higher CPU overhead when profiling.

**record exact allocations** is available only in combination with sampling. If this option is not selected, allocations will be reported according to the call traces recorded by the sampling procedure. This may lead to incorrect allocation spots. When `record exact allocations` is enabled, allocation spots are determined with an alternative method. However, there is a considerable increase in CPU and memory overhead if this option is selected.

- **full instrumentation**


With full instrumentation selected, all method calls are traced by JProfiler. This allows for profiling Java core classes (`java.*`), but is considerably slower than dynamic instrumentation, especially for 1.4 JVMs.

**subclassmethods** is available only in combination with full instrumentation and shows the concrete classes on which the methods are invoked rather than the classes where the methods are implemented.

The **active filter sets** panel allow you to choose whether the specified filters are to be

- **exclusive**

In the table below, you choose the [filter sets](#) [p. 77] which are active for the session being edited. These filters specify the end points for CPU profiling. In packages or classes matching one of the filter sets, no further calls into other filtered classes will be resolved. If such a call sequence should reach an unfiltered class again, it will be displayed with an upward filter bag in the [invocation view](#) [p. 119]. In other words, methods in filtered classes are treated as "opaque methods".

Filter sets can be edited on the ["Filter sets" tab](#) [p. 77] of JProfiler's [general settings](#) [p. 76] which are accessible by clicking the  **[General settings]** button on the bottom of the dialog or invoking the context menu on the filter sets table.

Click on the checkboxes in the "Apply" column to toggle filter sets. And select or deselect all filter sets at once with the buttons **[Select all]** and **[Deselect all]** or the corresponding entries in the context menu.

- **inclusive**

Enter comma separated filters for packages and classes in the text box. Only calls into matching packages and classes will be resolved for CPU profiling.

For example, if you specify `com.mycorp.`, `com.othercorp.` as the inclusive filters, a call into `com.mycorp.MyClass.myMethod()` and all calls it makes will be measured and displayed. All calls from `com.mycorp.MyClass.myMethod()` that don't belong to the packages `com.mycorp` or `com.othercorp` are treated as opaque and will be displayed as endpoints in the [invocation view](#) [p. 119]. If such a call sequence should reach an class included in the inclusive filters again, it will be displayed with an upward filter bag.

#### B.4.3.3 Disabling profiling features in session configuration

On this tab of the [profiling settings dialog](#) [p. 64], you can adjust profiling features of JProfiler in order to speed up program execution and reduce memory consumption.

##### Disabled profiling features

- **Disable call tree collection**

If you don't need [CPU profiling](#) [p. 117] data as well as the [allocation view](#) [p. 104] in the heap walker, the [allocation monitor view](#) [p. 91] and the stack trace information in the [monitor usage views](#) [p. 135], you can switch off the call tree collection. This will speed up profiling considerably and reduce memory usage.

- **Disable monitor contention views**

if you are not interested in monitor contention events, you may switch data collection off by selecting this checkbox to lower the memory consumption of the profiled application. If monitor contention views are **enabled**, the following settings govern the level of detail for the monitor contention views:

- **Monitor contention threshold**

Select the minimum time threshold in microseconds ( $\mu$ s) for which a monitor contention (i.e. when a thread is blocking) is displayed in the [monitor usage history view](#) [p. 135].

- **Monitor waiting threshold**

Select the minimum time threshold in microseconds ( $\mu$ s) for which a monitor wait state (i.e. when a thread is waiting) is displayed in the [monitor usage history view](#) [p. 135].

##### Allocation monitor

The information depth of the [allocation monitor view](#) [p. 91] is governed by this setting.

- **Show cumulated allocations of all classes**

The **[Change Selection]** in the [allocation monitor](#) [p. 91] and the [allocation hot spots view](#) [p. 94] will not work in this setting, only the cumulated allocations of all classes and array types can be displayed.

- **Show allocations resolved for each class**

The **[Change Selection]** in the [allocation monitor](#) [p. 91] and the [allocation hot spots view](#) [p. 94] is enabled this setting, single classes, array types and packages can be displayed by the allocation monitor view. Memory consumption is increased by this setting.

- **Disable allocation monitor**

The [allocation monitor](#) [p. 91] and the [allocation hot spots view](#) [p. 94] will be disabled. This setting speeds up the profiling of your application and reduces the memory overhead.

## Invocation tree

By default, JProfiler does not resolve line numbers in call trees. If you enable `show line numbers` for sampling and dynamic instrumentation, line number resolution will be enabled for the [call tree collections modes](#) [p. 65] of "Sampling" and "Dynamic instrumentation". For "Full instrumentation", line number resolution is not available.

When a method calls another method multiple times in different lines of code, line number resolution will show these invocations as separate method nodes in the [invocation tree](#) [p. 119] and the [allocation monitor view](#) [p. 91]. Backtraces in the hotspot views will also show line numbers.

Note that a line number can only be shown if the call to a method originates in an unfiltered class.

### B.4.3.4 Configure console settings

On this tab of the [profiling settings dialog](#) [p. 64], you can set options for the console that is displayed for locally started programs. This includes local sessions, applets, web start applications and remote sessions with a configured start command.

JProfiler offers two types of consoles:

- **Java Console**

This is a cross-platform console, that supports text input, sending CTRL-C to the profiled application, text selection and clipboard operations. For the Java console you can set the following options:

- **Buffer size**

The number of most recent lines of output that are held by the console. Default is 1000.

- **Window size**

The initial size (width x height) of the console in characters. Note that the console does not wrap text. Default is 80 x 25.

This console integrates with JProfiler's *Window* menu.

- **Native Console**

On Microsoft Windows, you also have the option to use the native console. This console does not integrate with JProfiler's *Window* menu.

### B.4.3.5 Miscellaneous options in session configuration

On this tab of the [profiling settings dialog](#) [p. 64], you can adjust various profiling options which govern the detail of the collected profiling data and the execution speed of the profiled application.

## Time settings

- **CPU time measurement**

Select whether you want times shown in the [CPU view section](#) [p. 117] to be measured in

- **elapsed time**

With elapsed time selected, the clock time difference between method entry and method exit will be shown. Note that if the [thread state selector](#) [p. 117] is set to its standard setting (Runnable). Waiting, blocking and Net IO thread states are not included in the displayed times.

- **estimated CPU time**

With estimated CPU time selected, the CPU time used between method entry and method exit will be shown. On Windows and Mac OS the system supplies CPU times with a 10 ms resolution which are used to calculate the estimated CPU times. On Linux and Solaris the VM does not supply a CPU time and the estimated CPU times are roughly estimated by looking at the number of runnable threads.


## VM life cycle control

- **Keep VM alive**

Installs a security manager which intercepts your application's calls to `System.exit()` and executes them after JProfiler's GUI front end disconnects. This option allows you to profile code sections which are close to a forced termination of the virtual machine. Don't use this feature when you profile an application server which installs its own security manager. If you get security related exceptions when profiling your applications, try unchecking this option.

## Dynamic views

- **Start with all views frozen**

To disable dynamic updates in JProfiler's views, you can check **Start with all dynamic views frozen**. Click on the unfreeze button in the toolbar if you want to start dynamic data to be displayed or fetch data manually with the  fetch data button which is visible only in the frozen state.

- **Transmission periods**

Based on the varying degree of computing expenses required for the different views, the transmission periods for the dynamic views have been split into two separate settings:

- **Call stack trees and thread history**

This setting influences the update interval of the

- [CPU view section](#) [p. 117]
- [allocation monitor](#) [p. 91]
- [allocation hot spots view](#) [p. 94]
- [thread history view](#) [p. 130]

- **Tables and graphs**

This setting influences the update interval of the

- [class monitor](#) [p. 89]
- [VM telemetry view section](#) [p. 137]

### B.4.3.6 Profiling settings template dialog

Before a [session is started](#) [p. 59], the profiling settings template dialog is displayed. This dialog displays a list of pre-configured profiling settings templates that are targeted at a variety of situations. As different templates in the drop down list are selected, the description box and the performance indicators below it are updated accordingly. Both description and performance indicators should help

you choose the best template for your task at hand. If you click on the **[Customize profiling settings and filters]** button below the drop down box, the [profiling settings dialog](#) [p. 64] is opened.

In the **Startup** section dialog you can choose whether recording of CPU or allocation data should be started immediately. For many profiling use cases the startup phase of an application is not of interest. For large applications servers, you can save a lot of memory and speed up the startup phase by not recording allocations from the beginning.

- **record CPU data on startup**

Both the [invocations view](#) [p. 119] and the [hot spots view](#) [p. 121] will display data immediately.

- **record allocations on startup**






Both the [class monitor](#) [p. 89], the [allocation monitor](#) [p. 91] and the [allocation hot spots view](#) [p. 94] will display data immediately.

When you click on **[Ok]**, the session will be started.

#### B.4.4 Open session dialog

The open session dialog serves two functions:

- To open [profiling sessions](#) [p. 59]. Double click on an existing session or choose a session and click **[Open]** to start a profiling session.
- To [edit](#) [p. 59], copy and delete existing sessions.

The list of available session configurations displays the session name which can be changed when [editing](#) [p. 59] a session. In addition, the associated icon to the left of the session name show whether the session is  a local session,  a remote session,  an applet session  a servlet session or  a Java Web Start session

The facility to open sessions is also embedded in JProfiler's [start center](#) [p. 38].

#### B.4.5 Starting remote sessions

To start your application or application server in such a way that you can connect to it with a remote session from JProfiler's GUI front end, the following steps are required:

- **Windows 98/NT/2000/XP**

1. **Adjust your startup command**

Add the following command line parameters to your startup command:

- `-Xrunjprofiler`
- `-Xbootclasspath/a:{JProfiler install directory}/bin/agent.jar`
- other JVM-specific options found in the [remote session invocation table](#) [p. 70]

2. **Adjust PATH**

Add `{JProfiler install directory}\bin\windows` to the `PATH` environment variable.

- **Linux and Solaris**

1. **Adjust your startup command**

Add the following command line parameters to your startup command:

- -Xrunjprofiler
- -Xbootclasspath/a:{JProfiler install directory}\bin\agent.jar
- other JVM-specific options found in the [remote session invocation table](#) [p. 70]

## 2 Adjust LD\_LIBRARY\_PATH

Add {JProfiler install directory}/bin/{linux-x86 | solaris-sparc} to the LD\_LIBRARY\_PATH environment variable.

## • Mac OS

### 1 Adjust your startup command

Add the following command line parameters to your startup command:

- -Xrunjprofiler
- -Xbootclasspath/a:{JProfiler install directory}\bin\agent.jar
- other JVM-specific options found in the [remote session invocation table](#) [p. 70]

## 2 Adjust DYLD\_LIBRARY\_PATH

Add {JProfiler install directory}/bin/macos to the DYLD\_LIBRARY\_PATH environment variable.

The [remote session invocation table](#) [p. 70] shows the complete command for all supported JVMs.

### B.4.6 Remote session invocation table

Please look at the help page on [starting remote sessions](#) [p. 69] for a complete sequence of steps that need to be taken for remote profiling. Below you find the condensed instructions on how to modify your startup command for a remote profiling session. The table shows all supported JVM vendors and versions. Square braces like [your path to agent.jar] are to be replaced according to the textual description. \${PARAM} is to be replaced by the parameters you would like to pass to the profiling agent. The following parameters are available:

- **port=nnnnn** chooses the port on which the agent listens for remote connections. Be sure to use the same value in JProfiler's GUI front end.
- **offline** enables the [offline profiling](#) [p. 140] mode. You cannot connect with the GUI front end when using the offline profiling mode. The parameters **id** and **config** have to be supplied as well.
- **id=nnnnn** chooses the session used for [offline profiling](#) [p. 140] .
- **config={path to JProfiler config file}** supplies the path to JProfiler's configuration file. This parameter is **only needed for offline profiling** [p. 140] .
- **verbose-instr** prints the names of all instrumented classes to stderr. This is a debugging option.

Multiple parameters are separated by commas such as in

"offline,id=172,config=~/.jprofiler3/config.xml".

Vendor: **Sun Microsystems Inc.**

Version 1.2.2

- classic mode:

```
java -classic -Xrunjprofiler:${PARAM} [solaris: -native]
-Djava.compiler=none -Xbootclasspath/a:[path to
agent.jar] [your JVM parameters] -classpath [class path]
[main class] [parameters]
```

**Note:** is not available on Solaris

- default mode:

```
java -Xrunjprofiler:${PARAM} [solaris: -native]
-Djava.compiler=none -Xbootclasspath/a:[path to
agent.jar] [your JVM parameters] -classpath [class path]
[main class] [parameters]
```

**Note:** is only available on Solaris

#### Version 1.3.0

- classic mode:

```
java -classic -Xrunjprofiler:${PARAM} [solaris:
-Xboundthreads] -Xbootclasspath/a:[path to agent.jar]
[your JVM parameters] -classpath [class path] [main
class] [parameters]
```

**Note:** is not available on Solaris

- default mode:

```
java -Xrunjprofiler:${PARAM} [solaris: -Xboundthreads]
-Xbootclasspath/a:[path to agent.jar] [your JVM
parameters] -classpath [class path] [main class]
[parameters]
```

**Note:** is only available on Solaris

#### Version 1.3.1

##### Unsupported releases with known problems: 1.3.1, 1.3.1\_01

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM} [solaris:
-Xboundthreads] -Xbootclasspath/a:[path to agent.jar]
[your JVM parameters] -classpath [class path] [main
class] [parameters]
```

- mixed mode:

```
java -Xmixed -Xrunjprofiler:${PARAM} [solaris:
-Xboundthreads] -Xbootclasspath/a:[path to agent.jar]
[your JVM parameters] -classpath [class path] [main
class] [parameters]
```

**Note:** does not work with full instrumentation

- classic mode:

```
java -classic -Xrunjprofiler:${PARAM} [solaris:
-Xboundthreads] -Xbootclasspath/a:[path to agent.jar]
```

```
[your JVM parameters] -classpath [class path] [main class] [parameters]
```

**Note:** is not available on Solaris

#### Version 1.4.0

**Unsupported releases with known problems:** 1.4.0-beta, 1.4.0-beta2, 1.4.0-beta3, 1.4.0-rc, 1.4.0

- mixed mode:

```
java -Xmixed -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

**Note:** does not work with full instrumentation

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

#### Version 1.4.1

- mixed mode:

```
java -Xmixed -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

**Note:** does not work with full instrumentation

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

#### Version 1.4.2

*see version 1.4.1*

#### Version 1.5.0

- mixed mode:

```
java -Xmixed -Xrunjprofiler:${PARAM} -Xshare:off  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

**Note:** does not work with full instrumentation

- interpreted mode:



```
java -Xint -Xrunjprofiler:${PARAM} -Xshare:off
-Xbootclasspath/a:[path to agent.jar] [your JVM
parameters] -classpath [class path] [main class]
[parameters]
```

Vendor: **IBM Corporation**

Version 1.3.0

- default mode:

```
java -Xrunjprofiler:${PARAM} -Djava.compiler=none
-Xbootclasspath/a:[path to agent.jar] [your JVM
parameters] -classpath [class path] [main class]
[parameters]
```

Version 1.3.1

*see version 1.3.0*

Version 1.4.0

*see version 1.3.0*

Vendor: **Apple Computer, Inc.**

Version 1.3.1

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM}
-Xbootclasspath/a:[path to agent.jar] [your JVM
parameters] -classpath [class path] [main class]
[parameters]
```

Version 1.4.1

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM} -XX:-UseSharedSpaces
-Xbootclasspath/a:[path to agent.jar] [your JVM
parameters] -classpath [class path] [main class]
[parameters]
```

Version 1.4.2

*see version 1.4.1*

Vendor: **BEA Systems, Inc.**

Version 1.4.1

- default mode:

```
java -Xjvmpi:entryexit=off -Xrunjprofiler:${PARAM}
-Xbootclasspath/a:[path to agent.jar] [your JVM
parameters] -classpath [class path] [main class]
[parameters]
```

**Note:** does not work with full instrumentation

- fullinstr mode:

```
java -Xrunjprofiler:${PARAM} -Xbootclasspath/a:[path  
to agent.jar] [your JVM parameters] -classpath [class  
path] [main class] [parameters]
```

**Note:** choose only when working with full instrumentation

- noopt mode:

```
java -Xnoopt -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

Vendor: **Hewlett-Packard Co.**

Version 1.3.1

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

- mixed mode:

```
java -Xmixed -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

**Note:** does not work with full instrumentation

- classic mode:

```
java -classic -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

Version 1.4.1

- mixed mode:


```
java -Xmixed -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

**Note:** does not work with full instrumentation

- interpreted mode:

```
java -Xint -Xrunjprofiler:${PARAM}  
-Xbootclasspath/a:[path to agent.jar] [your JVM  
parameters] -classpath [class path] [main class]  
[parameters]
```

#### B.4.7 Saving live sessions to disk


Snapshots of live profiling sessions can be saved to disk by selecting *Session->Save snapshot* from JProfiler's main menu or by clicking on the corresponding  tool bar entry in JProfiler's main tool bar.

A file chooser will be brought up where you can select the name and directory of the snapshot file to be written. The standard extension of JProfiler's snapshot files is *\*.jps*. Once JProfiler has finished writing the snapshot to disk, a message box informs you that the snapshot was saved.

Saved snapshots can be loaded

- by selecting *Session->Open snapshot* from JProfiler's main menu.
- by selecting a file from the "Open snapshot" tab in JProfiler's [start center](#) [p. 38] .

After a snapshot has been loaded, the functionality of all views is identical to a live profiling session with the exception of the [heap walker view](#) [p. 99] : The heap walker overview will be shown if a heap snapshot was taken at the time of saving, otherwise the heap walker will be unavailable.

The status bar indicates that a snapshot is being viewed by displaying the message  **Snapshot** in its rightmost compartment.

## B.5 General settings

### B.5.1 General settings

JProfiler's general settings apply to all sessions and are divided into four parts:

- [Java VMs](#) [p. 76]  
Configure the Java VMs available to launch local profiling sessions.
- [Filter sets](#) [p. 77]  
Configure the filter sets available to sessions which determine the information depth and package based data consolidation.
- [Default java file path](#) [p. 78]  
Configure the default class path, source path and native library path to be used for all sessions.
- [IDE integrations](#) [p. 79]  
Install IDE integrations for JProfiler.
- [Miscellaneous](#) [p. 79]  
Configure miscellaneous options for JProfiler.

### B.5.2 Configuring JVMs in general settings

This tab in JProfiler's [general settings](#) [p. 76] allows you to add, edit and delete Java VMs that can be used to launch local sessions, applet sessions and servlet sessions. Any JVM added here will be available from the "Java VM" combo box in the [application settings dialog](#) [p. 59] for the configuration of local, applet, servlet or Web Start sessions. Any changes you make to an existing Java VM configuration directly affect the sessions which already use it.

**Adding a new Java VM** is done by clicking the **[New]** button on the right hand side of the tab. A file chooser is brought up and prompts you for the selection of the home directory of the JVM (e.g. *c:\Program Files\jdk1.4.0* or */usr/lib/java*). The selected virtual machine will be checked for usability by JProfiler and depending on success, a new entry in the Java VMs table is added. Proceed as described below for editing a Java VM. If JProfiler reports that the JVM is not usable and you feel that your JVM should be supported, please don't hesitate to contact us through the [support request form](#)

**Editing an existing Java VM** is done by entering information directly into the columns of the Java VMs table.

- Double-click on the "Name" column to change the name of the JVM. This name is used for JVM selection in the [application settings dialog](#) [p. 59] .
- Double-click on "JVM home directory" and enter a directory manually or click on the button labeled **[...]** to change the home directory of an existing JVM. The new directory will be accepted only if the directory contains a JVM which is usable by JProfiler.
- Choose the JVM Mode from the "JVM Mode" combo box. This setting is initially set to the preferred value for the chosen JVM. See the [remote session invocation table](#) [p. 70] for details on this option.

**Deleting an existing Java VM** is done by selecting the Java VM which is to be deleted and clicking the **[delete]** button on the right hand side of the tab or choosing *Delete* from the context menu. Please note that all local sessions which currently use the deleted JVM will remain without an associated JVM and will be unusable until you assign them a new one in the [application settings dialog](#) [p. 59] .

**Setting the default Java VM** is done by selecting the Java VM which is to be the default Java VM and clicking the **[Set default]** button on the right hand side of the tab or choosing *Set default* in the

context menu. For new sessions, this Java VM will be initially selected in the [application settings dialog](#) [p. 59] .

**Searching for Java VMs** is done by clicking the **[Search]** button on the right hand side of the tab. This invokes the search wizard which corresponds to the functionality found in [JProfiler's setup wizard](#) [p. 41] . While the search wizard shows all JVMs found on your local fixed drives, only new JVMs will be merged into the Java VMs tab in the [general settings](#) [p. 76] dialog.

### B.5.3 Configuring filter sets in general settings

This tab in JProfiler's [general settings](#) [p. 76] allows you to add, edit and delete filter sets which can be activated individually for **exclusive filters** on the [call tree collection](#) [p. 65] tab of the [profiling settings dialog](#) [p. 64] . Active filter sets influence the information depth for CPU profiling and the execution speed of the profiled applications.

#### What are filter sets?

Data collected from classes defined in packages and classes matching a filter set for exclusive filters or failing to match the inclusive filters, are consolidated for the [CPU profiling views](#) [p. 117] , the [allocation monitor view](#) [p. 91] , the [heap walker allocation view](#) [p. 104] and the call stack information in the [monitor usage views](#) [p. 135] and are shown without further detail. Due to the considerable depth and complexity of external class libraries, an unfiltered view of invocation and allocations trees as well as hot spot lists is often unmanageable and would prevent access to the focal points of your application. From the point of view of an unfiltered class, filter sets work in both directions:

- **upward filtering**

When calling a method in an unfiltered class through methods of a number of filtered classes, upward filtering consolidates the path through which an unfiltered method is called into a single step. This is of utmost importance when profiling servlets or Enterprise Java Beans through an application server, where the entry point of your class might be buried several hundred methods deep in the application server's framework. Similarly, for GUI applications, the AWT call sequence is container based which makes it next to impossible to find callbacks like `MyAction.actionPerformed` or `MyComponent.paint` without upward filtering of the `java.awt.*` and `javax.swing.*` classes.

- **downward filtering**

When using external class libraries, most of the time the inner details are not important in performance analysis and optimizations, as they are delivered in binary form and are subject to a vendor controlled update cycle. Downward filtering consolidates a call into a filtered class into a single opaque call. This greatly reduces unwanted detail and directs the focus to the most important parts of your application.

**Adding new filter sets** is done by clicking the **[New]** button on the right hand side of the tab. A new row is added to the filter sets table and you proceed as described below for editing filter sets.

**Editing an existing filter set** is done by directly entering information into the columns of the filter sets table.

- Double-click on the "Name" column to change the name of the filter sets. This name is used for filter set selection in the [profiling settings dialog](#) [p. 65] .
- Double-click on "Filtered packages" to change the list of filtered packages. Entries designate the start of a package name such as `javax.ejb` and are separated by commas such as in `javax.ejb, javax.naming`.

**Deleting an existing filter set** is done by selecting the filter set which is to be deleted and clicking the **[delete]** button on the right hand side of the tab or choosing *Delete* from the context menu. Note

that sessions which previously had this filter activated will lose this filter set and not regain it if you re-add a filter set with the same name.

#### B.5.4 Configuring the default java file path in general settings

On this tab you can edit the default class path and source path for all sessions. With the two radio buttons on the top you choose whether to set the

- **Default class path**

The default class path is be added to the launch command of every local session, applet session or servlet session. The **additional class path** which is entered in the [application settings dialog](#) [p. 59] is **prepended** to the default class path and can be used to **override** it. The default class path is also used by the [bytecode viewer](#) [p. 86] to find class files for display.


- **Default source path**


The default source path is is used by the [source code viewer](#) [p. 86] to display Java sources. The **additional source path** which is entered in the [application settings dialog](#) [p. 59] is **prepended** to the default source path and can be used to **override** it. You do not need to add a source path for JDK sources, as the sources of the selected JDK contained in *src.jar* or *src.zip* will be automatically appended if they are installed.

- **Default native library path**

The default native library path is used for all local sessions. You only need the native library path for loading native libraries with calls to `java.lang-System.load()`. The **additional native library path** which is entered in the [application settings dialog](#) [p. 59] is **prepended** to the default native library path and can be used to **override** it.

To edit the path you can

**add a new entry** by clicking  on the right side of the tab. In the following file chooser select a directory (or a jar file for the class path) to add to the path.

**copy a path from the system clipboard** by clicking  on the right side of the tab. The path must consist of

- a single path entry
- multiple path entries separated by the standard path separator (";" on Windows, ":" on UNIX) or by line breaks.

Each path entry can be


- **absolute**



The path entry is added as it is.

- **relative**

On the first occurrence of a relative path, JProfiler brings up a directory chooser and asks for the root directory against which relative paths should be interpreted. All subsequent relative paths will be interpreted against this root directory.

JProfiler will only add unique path entries into the list. If no new path entry could be found, a corresponding error message is displayed.

**remove an existing entry** by clicking  on the right side of the tab or choosing *Remove* from the context menu. You will not be prompted for confirmation.

**change the position of an existing entry** by clicking  for moving it upward or  for moving it downward. The context menu entries *Move up* and *Move down* are equivalent to the image buttons.

### B.5.5 IDE integrations

All IDE integrations of JProfiler can be set up from this tab. JProfiler [integrates seamlessly into several popular IDEs](#) [p. 42]. See [here](#) [p. 42] for specific explanations regarding each IDE integration.

Select the desired IDE from the drop down list and click on **[Integrate]**. After completing the instructions, you can invoke JProfiler from the integrated IDE without having to specify class path, main class, working directory, used JVM and other options again.

JProfiler caches the location of integrated IDEs. If you repeat the installation of a particular integration, JProfiler will ask you whether to reuse the known location of the IDE. This is useful when updating to a newer version of JProfiler or for repairing a broken IDE integration.

### B.5.6 Configuring miscellaneous options in general settings

Miscellaneous options are collected on this tab.

The **look and feel** of JProfiler can be chosen as one of

- **Alloy look and feel**

Alloy look and feel is an elegant look and feel which is developed by [Incors GmbH](#). Our thanks go to Incors for this stylish addition to the Java platform.

- **Java look and feel**

Standard cross platform look and feel.

- **Native look and feel**

Native look and feel on your platform.

When you switch to a different look and feel, you have to restart JProfiler for the new setting to take effect.

**Note:** This setting is not available on Mac OS X.

Warning messages can be disabled by clicking the **Don't show this dialog again** checkbox in the warning dialog. To **enable all warning messages again**, click the "Reset warning messages button".

The **browser start command** for your default browser can be adjusted here. Use `$URL` as a variable for the URL to be displayed. This setting is important for [exporting views to HTML](#) [p. 84] and [servlet sessions](#) [p. 59]. If you leave the text box empty, JProfiler will use the system defaults on Windows and Mac OS X and try to invoke *netscape* on Linux and Solaris.

By default, exported files are opened immediately after they have been [exported](#) [p. 84]. In order to change this behavior, you can deselect the **Open exported files** checkbox in the "Export options" section.

## B.6 Profiling views

### B.6.1 Views overview

JProfiler organizes profiling data into **view sections** which collect similar or connected views. The view section chooser is located on the left side of JProfiler's main window, while the single views of a view section can be selected by choosing the tabs on the bottom of the window. View sections can also be switched via JProfiler's *Views* menu or the keyboard shortcuts which are indicated below.

- [Memory views](#) [p. 88]



(CTRL-1) The memory view section contains views which are concerned with the details of object allocations.

- [Heap walker](#) [p. 99]



(CTRL-2) The heap walker view section allows you to take a snapshot of the heap and analyze it in detail.

- [CPU views](#) [p. 117]



(CTRL-3) The CPU view section contains views which are concerned with method calls and time measurements.

- [Thread views](#) [p. 129]









(CTRL-4) The thread view section contains views which are concerned with the details of thread status, monitor contentions and wait states.

- [Telemetry views](#) [p. 137]



(CTRL-5) The telemetry view section contains views which are concerned with historical characteristics of cumulated virtual machine variables.

The functionality of the various views is strongly dependent on the **state of the current session**.

- If the session is  **attached**, the complete functionality of all views is available. In that case, the session may be either
  -  **working**, where the information in all dynamic views is continuously updated. This state is indicated in the status bar.
  -  **frozen**, where the information in all dynamic views remains static. This state is indicated in the status bar. You can fetch the current data for all dynamic views by clicking on the  fetch data button which is visible only in the frozen state.
- If the session is  **detached**, the functionality of most views is incomplete. Any information which is not already stored in JProfiler's GUI front end and would have to be queried from the profiled application is unavailable.
- If a [profiling snapshot](#) [p. 75] is opened, the status bar displays  **Snapshot**. The functionality of all views is identical to the **working state** with the exception of the [heap walker view](#) [p. 99] : The heap walker overview will be shown if a heap snapshot was taken at the time of saving, otherwise the heap walker will be unavailable.

Common properties of profiling views include

- [Exporting views to HTML](#) [p. 84]



- [Undocking views from the main window](#) [p. 85]
- [Sorting tables](#) [p. 85]
- [Source and bytecode viewer](#) [p. 86]
- [Dynamic view filters](#) [p. 87]

## B.6.2 JProfiler's menu


JProfiler's toolbar and menu contain actions applicable to all views as well as actions which are view-sensitive or appear for certain views only. The common menu and toolbar entries fall into six categories:

The **session menu** contains actions to create, open and close sessions and snapshots.


- **Start center**

(CTRL-O) Brings up JProfiler's [start center](#) [p. 38] . If there already is an open session in the current window, it will be discarded once a new session is opened. This action is also available from JProfiler's toolbar.

- **New window**

 (CTRL-ALT-O) Open in a new instance of JProfiler's main window and brings up JProfiler's [start center](#) [p. 38] .

- **New session**

 (CTRL-N) Creates a new session and brings up the [application settings dialog](#) [p. 59] . The new session will be started after leaving the dialog with **[OK]**. If there is already an open session in the current window, it will be discarded.

- **Integration wizards**

This submenu contains the starting points for the [application server integration wizards](#) [p. 39] , just like the "New session" tab on the [start center](#) [p. 38] .


- **Conversion wizards**

This submenu contains the starting points for the conversion wizards, just like the "Convert" tab on the [start center](#) [p. 38] .

- **Open session**

Brings up the [open session dialog](#) [p. 69] . If there already is an open session in the current window, it will be discarded once a new session is opened.

- **Save snapshot**

 (CTRL-S) Brings up a file chooser to select a [snapshot file](#) [p. 75] to be written. A dialog box informs about the successful completion of the operation. This action is also available from JProfiler's toolbar.


- **Open snapshot**

Brings up a file chooser to select a [snapshot file](#) [p. 75] to be opened. If there already is an open session in the current window, it will be discarded.

- **IDE integrations**

Short cut to the IDE integrations tab of the [general settings dialog](#) [p. 79] where you can integrate all supported IDEs.

- **Close window**

 (CTRL-W) Closes the current window. If there is an open session in the current window, you will be asked for confirmation.

- **Exit JProfiler**

(CTRL-ALT-X) After confirmation, closes all open main windows and exits JProfiler.

The **edit menu** contains view-specific actions and gives access to JProfiler's settings. View specific actions are described in the help page of the [corresponding view](#) [p. 80] .


- **General settings**

 Brings up the [general settings dialog](#) [p. 76] .


- **Application settings**

Brings up the [application settings dialog](#) [p. 59] . Note that any changes here become effective only when the session is restarted.

- **Profiling settings**

 (CTRL-ALT-T) Brings up the [profiling settings dialog](#) [p. 64] . Please note that any changes here become effective only when the session is restarted. This action is also available from JProfiler's toolbar.





- **View settings**

 (CTRL-T) Brings up the view settings dialog for the corresponding view. If disabled, the currently active view has no particular settings. This action is also available from JProfiler's toolbar.

The **profiler menu** contains actions which change the window or session as a whole.



- **Stop/Detach/Start/Attach session**

(F11) This action is also available from JProfiler's toolbar.

-  Stops the session (all [session types](#) [p. 59] except remote session), i.e. the process is destroyed. In a stopped session, the profiling views are [not fully functional](#) [p. 80] (visible if currently started and not remote session).
-  Detaches the current [remote session](#) [p. 59] . The profiled JVM will be detached from JProfiler's front end and continues to run undisturbed. In a detached session, the profiling views are [not fully functional](#) [p. 80] (visible if currently attached and remote session).
-  Starts the application configured in the current session if it is a [local session, applet session or servlet session](#) [p. 59] (visible if currently detached and not remote session).
-  Attaches the current [remote session](#) [p. 59] to a remote application or reconnects to it. (visible if currently detached and remote session).







- **Record allocation data**


This action is also available from JProfiler's toolbar. The [memory views](#) [p. 88] and some [telemetry views](#) [p. 137] rely on allocation data.

-  Start recording allocation data. (visible if allocations are currently not recorded).
-  Stop recording allocation data. (visible if allocations are currently recorded).



- **Record CPU data**

This action is also available from JProfiler's toolbar. The [CPU views](#) [p. 117] rely on CPU data.

-  Start recording CPU data. (visible if CPU data is currently not recorded).
-  Stop recording CPU data. (visible if CPU data is currently recorded).
- **Freeze/Unfreeze session**  
(F12) This action is also available from JProfiler's toolbar.
  -  [Freezes all views](#) [p. 80] for the current session (visible if currently not frozen).
  -  [Unfreezes all views](#) [p. 80] for the current session (visible if currently frozen).
- **Get current data**  
(F5)  [Updates all views](#) [p. 80] with the current data. This action is only enabled if the current session is frozen. This action is also available from JProfiler's toolbar if the current session is frozen.
- **Run garbage collector**  
 Run the garbage collector in the profiled JVM. This action is also available from JProfiler's toolbar.
- **Show global filters for call tree collection**  
Show a dialog with a tree view of all [exclusive or inclusive filters](#) [p. 87] that JProfiler uses when recording the method call tree. This action is also available at the bottom of several views that show call trees.

The **views menu** provides one-click access to all of JProfiler's profiling views, grouped into the four  [view sections](#) [p. 80] .


The **window menu** allows you to keep track of all [tops level windows created by JProfiler](#) [p. 85] .

- **Undock/Dock view**  
(CTRL-E) This action is also available from the context menu when right clicking the view in the tab selector at the bottom of the window.
  -  Undocks the view and shows it in a separate top level window. (visible if the currently active view is docked into the main window)
  -  Docks the view and returns it to the main window. (visible if the currently active view is undocked)
- **Dock all floating views**  
Docks all currently undocked views into their main windows.
- **Cycle to previous window**  
(CTRL-F2) Activate the previous window in the window list and bring it to the front.
- **Cycle to next window**  
(CTRL-F3) Activate the next window in the window list and bring it to the front.
- **Tile all undocked views**  
Tile the desktop with all undocked views.
- **Stack all undocked views**


Resize all undocked views to a standard size and stack them regularly on the desktop.

At the bottom of the window menu you can directly navigate to a window by selecting it from the list. The **help menu** gives access to help, web sites, and useful e-mail addresses for JProfiler.


- **Help contents**

 (F1) Brings up context sensitive help. This action is also available from JProfiler's toolbar.

- **JProfiler on the web**

 Connects to JProfiler's web site in the web browser configured under [general settings](#) [p. 79] .

- **Contact sales**

 Brings up a sales contact form in the web browser configured under [general settings](#) [p. 79] .

- **Contact support**

 Brings up a support contact form in the web browser configured under [general settings](#) [p. 79] .


- **Enter license key**

 Allows you to [enter your license key](#) [p. 41] .

- **About JProfiler**

Shows general information about your copy of JProfiler and its license status.

### B.6.3 Exporting views

All views can be exported to external formats by selecting **Export** from the *Edit* menu or context menu or clicking on the corresponding  toolbar button. A file chooser will be brought up allowing you to specify the output file and the export format.

The export format is chosen with the "file type" combo box in the file chooser. The following export formats are available:

- **HTML**

Available for all views. The view will be exported to an HTML file. Besides the HTML file, several image files might be written to a subdirectory *jprofiler\_images*. If the [option to open files after export](#) [p. 79] is enabled, the web browser configured in the [general settings](#) [p. 79] is opened and the exported HTML file is displayed.

- **CSV data**

Available for tabular views and graphs with a time axis. CSV data suitable for Microsoft Excel is written to a file. If the [option to open files after export](#) [p. 79] is enabled, the registered application for CSV is opened and the exported CSV file is displayed.

- **XML data**

Available for tree views. XML data with a self-explanatory format is written to a file. If the [option to open files after export](#) [p. 79] is enabled, the registered application for XML is opened and the exported XML file is displayed.

If you export the same view multiple times to the same directory under the same name, a running number will be appended to the filename. The export directory location is persistent and remembered across restarts.

With the HTML export functionality you can **print** all views from JProfiler via your web browser.


#### B.6.4 Integrated source code and bytecode viewer

All views in JProfiler can be undocked and promoted to a separate top level window by

- choosing *Window->Undock view* from JProfiler's main menu
- right clicking the view in the tab selector at the bottom of the window and choosing *Undock view* from the context menu.

An undocked view has a reduced main menu that contains only the *Edit* and *Window* menus from the [main menu](#) [p. 81] as well as a reduced toolbar. With *Window->Show main window for this session* (CTRL-H) the corresponding main window can be activated.




If a view has been undocked, a placeholder is shown in the corresponding tab in the main window. An undocked view can be re-docked into its main window by

- choosing *Window->Dock view* from the main menu of the undocked view.
- closing the window.
- clicking the  dock button in the placeholder for the view.
- choosing *Window->Dock view* from JProfiler's main menu
- right clicking the view in the tab selector at the bottom of the window and choosing *Undock view* from the context menu.

Undocked views can be tiled or stacked with the *Window->Tile all undocked view* and *Window->Stack all undocked view* menu entries. Note that undocked views of all main windows are treated uniformly.

To dock all undocked views with a single action, please choose *Window->Dock all floating view*. Note that undocked views of all main windows are docked.

JProfiler keeps track of all created top level windows in the window list available at the bottom of the *Window* menu. These windows include

-  main windows
- undocked views
-  console windows
-  source and bytecode viewers

This list does not include

- native console windows
- windows opened by the profiled application

To navigate to a window in the window list, click on it or use the *Window->Cycle to previous window* (CTRL-F2) and *Window->Cycle to next window* (CTRL-F3) menu entries.

#### B.6.5 Sorting tables in profiling views

Many of JProfiler's profiling views are displayed as tables. These tables can be sorted by any column in three ways:

- Choose the sort column from the **context menu**.





- Choose the sort column from the *Edit->Sort* menu which appears for table views.
- Click on the column header of the sort column.

Performing one of these operations multiple times alternates between ascending and descending sort order. The current sort column and sort order is indicated graphically in the column headers as well as in the relevant menus.

### B.6.6 Zooming and navigating in graphs

Some of JProfiler's profiling views are displayed as graphs.



The zoom level for these graphs can be adjusted in the following ways:

- **Zoom in** by rolling the mouse wheel toward you, clicking on the  zoom in toolbar button or choosing the corresponding entry from the context menu.
- **Zoom out** by rolling the mouse wheel away from you, clicking on the  zoom out toolbar button or choosing the corresponding entry from the context menu.
- **Zoom to 100%** by clicking on the  zoom 100% toolbar button or choosing the corresponding entry from the context menu.
- **Fit graph to window** by clicking on the  fit content toolbar button or choosing the corresponding entry from the context menu.

To zoom in **on a particular object**, you can select it first and then use the zoom in action described above.

Besides using the scrollbars to navigate to other parts of the graph you can drag the graph with the mouse to move it.

### B.6.7 Integrated source code and bytecode viewer

Wherever applicable, JProfiler provides access to the source code as well as the bytecode of profiled classes and displays them in a source and bytecode viewer frame. The source and bytecode viewer has two tabs, one for source code and the other for bytecode. Both tabs display the same class. Invoking the source and bytecode viewer through the  **Show source** action in the *Edit* menu or context menu displays the frame with the source tab activated, the  **Show bytecode** action activates the bytecode tab first.



To be able to show the source code of a class, the source must be available from the [source path](#) [p. 59] of the session. To be able to jump directly to the chosen method in the source code viewer and to display the bytecode of a class, the class file must be available from the [class path](#) [p. 59] of the session. Changes in class path and source path for an active session are recognized immediately by the source and bytecode viewer.

The source code tab has a method selector combo box displaying all methods of the displayed class. Note that if an inner class was displayed, only the methods of the inner class are shown in the method selector. When selecting a method, the bytecode viewer opens the class file tree at the corresponding position.

The bytecode of a class is displayed in a tree showing

- General information
- Constant pool
- Interfaces
- Fields

- Methods
- Class file attributes

If you look for the bytecode, select the "Code" child of the desired method. The bytecode viewer is extensively hyperlinked, allowing you to navigate easily to constant pool entries or branch targets and go back and forth in your navigation history with the   navigation controls at the top of the tab.

**Note:** When JProfiler is started through an [IDE integration](#) [p. 42], the integrated source code viewer is not used and the source element is displayed in the IDE.

### B.6.8 Dynamic view filters

For many dynamic views, **view filters** can be set at the bottom of the view. Enter a comma separated list of packages into the combo box and hit enter to dynamically filter the view.

All dynamic views with a view filter box at the bottom share the same current view filter. To reset the view filter and show the entire content of the view again, click on **[Reset view filters]** in the lower right corner of the view. The combo box holds view filters that have been entered during the current session. Selecting an entry from the combo box enables the view filter immediately.

View filters have an effect similar to the [inclusive filters](#) [p. 65] that are set for the session. These are configured in the [profiling settings dialog](#) [p. 64] and are not adjustable while the session is active. However, the active filter sets of the session strongly influence the speed and memory consumption of the profiled application while the view filters don't. It is therefore advisable to activate as many filter sets as possible in the [call tree collection options](#) [p. 65] and use the view filters for dynamic drill down only.


### B.6.9 Global view filters for call tree collection

This dialog can be shown by


- choosing *Profiler->Show global filters for call tree collection* from JProfiler's main menu
- clicking on the **[Global filters]** button that is shown in the bottom right corner of views that show call trees or time measurements of method calls.

Depending on your [profiling settings](#) [p. 64] in the [call tree collection tab](#) [p. 65], the dialog shows a tree of

- **excluded packages**

 end points defined by exclusive filters show packages whose internal call structure is not recorded.

- **included packages**

 starting points defined by inclusive filters show packages whose internal call structure is recorded. The call structure of other packages is not resolved.

**Note:** All sub-packages are included in end points and starting points.

This dialog is only intended to present information on the filtered packages while a session is running. Filters cannot be modified in this dialog. Please see the [call tree collection settings](#) [p. 65] on how to modify filters.

## B.6.10 Memory view section

### B.6.10.1 Memory view section

The memory view section contains the

- [class monitor view](#) [p. 89]

The class monitor view shows the dynamic class-resolved statistics for current heap usage and garbage collected objects.

- [allocation monitor view](#) [p. 91]


The allocation monitor view shows the dynamic allocation tree for the current heap usage and garbage collected objects.


- [allocation hot spots view](#) [p. 94]

The dynamic allocation hot spots view shows which methods are responsible for creating objects of a selected class.


- [memory statistics](#) [p. 99]

The memory statistics view shows statically calculated statistics for package resolved memory consumption and allocations.


Unless "Record allocations on startup" has been selected in the *Startup* section of the [profiling settings dialog](#) [p. 64], data acquisition has to be started manually by clicking on  **Record allocation data** in the tool bar or by selecting *Profiler->Record allocation data* from JProfiler's main menu.

Allocation data acquisition can be stopped by clicking on  **Stop recording allocation data** in the tool bar or by selecting *Profiler->Stop recording allocation data* from JProfiler's main menu.

**Restarting** data acquisition **resets** all data in the the [class monitor view](#) [p. 89], the [allocation monitor view](#) [p. 91] and the [allocation hot spots view](#) [p. 94]

When you  stop recording allocations, the recorded objects will still be tracked for garbage collection. For example, if all recorded objects are garbage collected, both class monitor and allocation monitor will be empty in their default view mode (live objects only). You can then still display all recorded objects if you switch to one of the other two view modes (garbage collected only or both live and garbage collected).

The [heap walker](#) [p. 99] will be able to display allocation information only for recorded objects, otherwise the entire heap is displayed in the heap walker.

The dynamic memory views are integrated with the heap walker. The  [take heap snapshot with selection](#) [p. 99] action on the toolbar, in the *Edit* and context menus takes a heap snapshot and creates an object set with the currently selected objects.



## B.6.10.2 Class monitor view

### B.6.10.2.1 Class monitor view

The class monitor view shows the list of all loaded classes and arrays types together with the number of instances which are allocated on the heap. There are three columns shown in the table, which can be [sorted](#) [p. 85] .

- **Name**

Shows the name of the class or the array type. The notation `<class>[]` stands for non-primitive arrays of any class type. (e.g. the array might be of type `String[]` or `Object[]`). A further distinction is not possible due to the nature of Java bytecode.

- **Instance count**


Shows how many instances are currently allocated on the heap. This instance count is displayed graphically as well.

- **Size**

Shows the total size of all allocated instances. Note that this is the **shallow size** which does not include the size of referenced arrays and instances but only the size of the corresponding pointers. The size is in bytes and includes only the object data, it does not include internal JVM structures for the class, nor does it include class data or local variables.

Only recorded objects will be displayed in this view. See the [memory section overview](#) [p. 88] for further details on allocation recording.

By **marking** the current state, you can follow the evolution of the heap. Marking the current values can be achieved by

- choosing *Edit->Mark current values* from JProfiler's main menu
- choosing the corresponding  toolbar entry
- choosing *Mark current values* from the context menu

Upon marking, a fourth column labeled **Difference** appears with all values initially set to zero. With each subsequent update, the column's values track the difference of the instance count with respect to the point in time where the mark was set. The graphical representation of the instance count column shows the marked state in green and positive differences in red.

By default, the difference column is sorted on the **absolute values** in it, this can be changed in the [class monitor view settings dialog](#) [p. 90] .

You can remove the mark by

- choosing *Edit->Remove mark* from JProfiler's main menu
- choosing *Remove mark* from the context menu

The class monitor can filter objects according to their liveness status:

- **Live objects**

 Only objects which are currently in memory are shown.



- **Garbage collected objects**


 Only objects which have been garbage collected are shown.

- **Live and garbage collected objects**

 All created objects are shown.

To switch between the three modes, you can click on the toolbar entry displaying the current mode and chose the new desired mode. Also, JProfiler's main menu and the context menu allow the adjustment of the view mode via *Edit->Change view mode*.


If the garbage collected objects are shown, you can reset the accumulated data by clicking on the  reset action in the toolbar or choosing the *Reset garbage collector for this view* menu item in the *Edit* or context menu. All garbage collector data will be cleared and the view will be empty for the "Garbage collected objects" mode until further objects are garbage collected. Note that you can force garbage collection by clicking on the garbage collector  tool bar button or by selecting *Profiler->Run garbage collector* from JProfiler's main menu.

The *Edit->Take heap snapshot for selection* menu item and the corresponding  toolbar entry take a new snapshot, switch to the [heap walker view](#) [p. 99] and create an object set with the currently selected class.

The update frequency for the class monitor can be set on the [miscellaneous tab](#) [p. 67] in the [profiling settings dialog](#) [p. 64] .

You can [stop and restart allocation recording](#) [p. 88] to clear the class monitor and [freeze all views](#) [p. 80] to ensure that the class monitor remains static.

#### **B.6.10.2.2 Class monitor view settings dialog**

The class monitor view settings dialog is accessed by bringing the [class monitor view](#) [p. 89] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog.

The **sorting of the difference column** can be toggled between absolute value ordering or normal ordering.


### B.6.10.3 Allocation monitor view

#### B.6.10.3.1 Allocation monitor view


The allocation monitor view shows a top-down call tree cumulated for all threads and filtered according to the [active filter sets](#) [p. 65] which is similar to the one shown in the [invocation view](#) [p. 119] in JProfiler's [CPU section](#) [p. 117] except that it shows allocations of class instances and arrays instead of time measurements.

The entries in the allocation tree have different meanings which are indicated by the displayed icons:


- **leaf method where allocations were performed**

 This points to a method where at least one allocation of a class instance or an array has taken place and no other unfiltered allocating methods were called.

- **node method where allocations were performed**

 This points to a method at least one allocation of a class instance or an array has taken place and other allocating methods were called.

- **node method where no allocations were performed**

 This points to a method where no allocation of a class instance or an array has taken place. However, there are other methods called from this method which perform allocations so this method must be shown to complete the tree up to the root.

Every entry in the allocation tree has textual information attached which - depending on the [allocation monitor view settings](#) [p. 92] shows

- a **percentage number** which is calculated with respect to the root of the tree or calling method.
- a **size measurement** in bytes or kB which displays the shallow size of those objects which were allocated here (see below).
- an **allocation count** which shows how many instances of classes and arrays have been allocated here (see below).
- a **method name** which is fully qualified or relative with respect to the calling method.
- a **line number** which is only displayed if line number resolution has been enabled in the [profiling settings](#) [p. 66] and if the calling class is unfiltered. Note that the line number shows the line number of the invocation and not of the method itself.

The size and the allocation count are either cumulated for all method calls below the associated method or not, depending on the corresponding [view setting](#) [p. 92]. Note that allocations performed in calls to filtered classes are consolidated in the first method call into a filtered class.

Only recorded objects will be displayed in this view. See the [memory section overview](#) [p. 88] for further details on allocation recording.


By default, the allocation monitor displays allocations of all classes and array types. The allocation monitor can display allocations for a selected class, package or array type if class resolved allocations are enabled in the "Allocation monitor" section on the [features tab](#) [p. 66] of the [profiling settings dialog](#) [p. 64].

The header above the tree view shows the current selection, the **[Change selection]** button brings up the [package selection dialog](#) [p. 98]. If class-resolved allocations are not enabled, a corresponding message is displayed instead of the package selection dialog. Note that you can still get class-resolved allocations in the heap walker unless the allocation monitor has been completely disabled.

The **[Show all]** button clears the selection and reverts the allocation monitor to its default state of showing all allocations.

The allocation monitor can filter objects according to their liveness status:

- **Live objects**

 Only objects which are currently in memory are shown.



- **Garbage collected objects**


 Only objects which have been garbage collected are shown.

- **Live and garbage collected objects**

 All created objects are shown.


To switch between the three modes, you can click on the toolbar entry displaying the current mode and chose the new desired mode. Also, JProfiler's main menu and the context menu allow the adjustment of the view mode via *Edit->Change view mode*.

If the garbage collected objects are shown, you can reset the accumulated data by clicking on the  reset action in the toolbar or choosing the *Reset garbage collector for this view* menu item in the *Edit* or context menu. All garbage collector data will be cleared and the view will be empty for the "Garbage collected objects" mode until further objects are garbage collected. Note that you can force garbage collection by clicking on the garbage collector  tool bar button or by selecting *Profiler->Run garbage collector* from JProfiler's main menu.

The *Edit->Take heap snapshot for selection* menu item and the corresponding  toolbar entry take a new snapshot, switch to the [heap walker view](#) [p. 99] and create an object set with the currently selected class and allocation spot.

You can [stop and restart allocation recording](#) [p. 88] to clear the allocation monitor and [freeze all views](#) [p. 80] to ensure that the allocation monitor remains static.

#### **B.6.10.3.2 Allocation monitor view settings**

The allocation monitor view settings dialog is accessed by bringing the [allocation monitor view](#) [p. 91] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog.

The view mode can be toggled with the **cumulate allocations** checkbox. This sets whether allocations should be cumulated to show all allocations below any method or not.

The **allocations description** options control the amount of information that is presented in the description of the method call.

- **Always show fully qualified names**

If this option is not checked, class name are omitted in intra-class method calls which enhances the conciseness of the display.

- **Always show signature**

If this option is not checked, method signatures are shown only if two methods with the same name appear on the same level.

The **percentage calculation** determines against what allocation numbers percentages are calculated.

- **Absolute**

Percentage values show the contribution to the total number of allocations.

- **Relative**

Percentage values show the contribution to the parent method.

Whether the contribution is cumulated or not depends on the `Cumulate allocations` setting.

## B.6.10.4 Allocation hot spots view

### B.6.10.4.1 Allocation hot spots view


The allocation hot spots view shows a list of methods where objects of a selected class have been allocated. Only methods are included which contribute at least 0.1% of the total number of allocations. The methods are filtered according to the [active filter sets](#) [p. 65]. This view is similar to the [hot spots view](#) [p. 121] in JProfiler's [CPU section](#) [p. 117] except that it shows allocations of class instances and arrays instead of time measurements.

**Note:** The notion of a hot spot is relative. Hot spots depend on the filter sets that you have enabled on the [call tree collection tab](#) [p. 65] of the [profiling settings dialog](#) [p. 64]. Filtered methods are opaque, in the sense that they include allocations performed in calls into other filtered methods. If you change your filter sets you're likely to get different hot spots since you are changing your point of view. Please see [the help topic on hotspots and filters](#) [p. 34] for a detailed discussion.

Every hot spot is described in several columns:

- the **method name**
- the **percentage** of all allocations together with a bar whose length is proportional to this value.
- the **number of allocations**.

The hot spot list can be [sorted on all columns](#) [p. 85].

If you click on the  handle on the left side of a hot spot, a tree of backtraces will be shown. Every entry in the backtrace tree has textual information attached to it which depends on the [allocation hot spots view settings](#) [p. 95].


- the **percentage** of all allocations. This value is calculated with respect either to the parent hot spot or the called method. The percentage base can be changed in the [allocation hot spots view settings](#) [p. 95].
- the **number of allocations** that are contributed to the hot spot along this call path.  
**Note:** This is **not** the number of allocations in this method.
- the **method name** which is fully qualified or relative with respect to the calling method.
- a **line number** which is only displayed if line number resolution has been enabled in the [profiling settings](#) [p. 66] and if the calling class is unfiltered. Note that the line number shows the line number of the invocation and not of the method itself.

The combo box at the top-right corner of the view allows you to treat allocations of filtered classes in two different ways:

- **show separately**  
Filtered classes can be hotspots of their own. This is the default mode.
- **add to calling class**  
Allocations of filtered classes are always added to the calling class. In this mode, a filtered class cannot be a hotspot, except if it is a top-level upward filter bag, i.e. if it is not called by any unfiltered class, but calls unfiltered classes itself.

With these two modes you can change your viewpoint and the definition of a hotspot. Please see [the help topic on hotspots and filters](#) [p. 34] for a detailed discussion of this topic.

By **marking** the current state, you can follow the evolution of the allocation hotspots. This is particularly useful for quickly finding the origin of memory leaks. Marking the current values can be achieved by



- choosing *Edit->Mark current values* from JProfiler's main menu
- choosing the corresponding  toolbar entry
- choosing *Mark current values* from the context menu

Upon marking, a fourth column labeled **Difference** appears with all values initially set to zero. With each subsequent update, the column's values track the difference of the allocation count with respect to the point in time where the mark was set. The graphical representation of the percentage column shows the marked state in green and positive differences in red.

By default, the difference column is sorted on the **absolute values** in it, this can be changed in the [allocation hot spots view settings dialog](#) [p. 95] .

You can remove the mark by

- choosing *Edit->Remove mark* from JProfiler's main menu
- choosing *Remove mark* from the context menu


When **navigating** through the hot spots backtraces tree by opening method calls, JProfiler automatically expands methods which are only called by one other method themselves. To quickly **expand larger portions** of the hot spots backtraces tree, select a method and choose  *Edit->Expand 10 levels* from the main window's menu or choose the corresponding menu item from the context menu. If you want to **collapse an opened part** of the hot spots backtraces tree, select the topmost method that should remain visible and choose  *Edit->Collapse all* from the main window's menu or the context menu.

Only recorded objects will be displayed in this view. See the [memory section overview](#) [p. 88] for further details on allocation recording.

By default, the allocation hot spots displays allocations of all classes and array types. The allocation hot spots view can display allocations for a selected class, package or array type if class resolved allocations are enabled in the "Allocation monitor" section on the [features tab](#) [p. 66] of the [profiling settings dialog](#) [p. 64] .


The header above the tree view shows the current selection, the **[Change selection]** button brings up the [package selection dialog](#) [p. 98] . If class-resolved allocations are not enabled, a corresponding message is displayed instead of the package selection dialog. Note that you can still get class-resolved allocations in the heap walker unless the allocation monitor has been completely disabled.

The **[Show all]** button clears the selection and reverts the allocation hot spots view to its default state of showing all allocations.

The *Edit->Take heap snapshot for selection* menu item and the corresponding  toolbar entry take a new snapshot, switch to the [heap walker view](#) [p. 99] and create an object set with the currently selected class and allocation hot spot.

You can [stop and restart allocation recording](#) [p. 88] to clear the allocation hot spots view and [freeze all views](#) [p. 80] to ensure that the allocation hot spots view remains static.

#### B.6.10.4.2 Allocation hot spots view settings

The allocation hot spots view settings dialog is accessed by bringing the [allocation hot spots view](#) [p. 94] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog.

The **allocations description** options control the amount of information that is presented in the description of the method call.

- **Always show fully qualified names**

If this option is not checked, class name are omitted in intra-class method calls which enhances the conciseness of the display.

- **Always show signature**

If this option is not checked, method signatures are shown only if two methods with the same name appear on the same level.

The **percentage calculation** determines against what allocation numbers percentages are calculated for the hot spot backtraces.

- **Absolute**

Percentage values show the contribution to the total number of allocations.

- **Relative**

Percentage values show the contribution relative to the called method.

Whether the contribution is cumulated or not depends on the `Cumulate allocations` setting.


The **sorting of the difference column** can be toggled between absolute value ordering or normal ordering.




## B.6.10.5 Memory statistics

### B.6.10.5.1 Memory statistics

The memory statistics view shows statically calculated statistics for package resolved memory consumption and allocations.

To calculate a statistics, click  **Calculate statistics** in the tool bar or select *Edit->Calculate statistics* from JProfiler's main menu. If a statistics has been calculated, the context menu also provides access to this action.

Before a statistics is calculated, the [memory statistics options dialog](#) [p. 97] is brought up. The resulting statistics table is static and can be re-calculated by executing  **Calculate statistics** again. The statistics options dialog remembers your last selection.

The package level statistics table displays two columns:

- **Package**  
the name of the package. If `cumulate packages` has been selected in the [options dialog](#) [p. 97], every possible package in the package hierarchy will be displayed, even if there are no classes in it.
- **Instance count**  
the number of objects **allocated from this package** or **allocated in this package** depending on the statistics type selected in [options dialog](#) [p. 97] together with a percentage bar and a percentage value.

### B.6.10.5.2 Memory statistics options

The memory statistics options dialog sets parameters for the output of the [memory statistics view](#) [p. 97].

- **Statistics type**  
One of
  - **Package statistics of objects**  
Show allocated objects of classes in each package.
  - **Package statistics of allocations**  
Show allocation performed in each package.
- **Heap type**  
One of
  - Live objects
  - Garbage collected objects
  - Live and garbage collected objects
- **Cumulate packages**  
If selected, allocations are cumulated for sub-packages in the package hierarchy. Also, every possible package in the package hierarchy will be displayed, even if there are no classes in it. Otherwise, packages are treated as separate entities without hierarchy.

#### B.6.10.6 Package selection dialog

The package selection dialog is shown when clicking on **[Change selection]** in the [allocation monitor view](#) [p. 91] or the [allocation hot spots view](#) [p. 94] provided that class resolved allocations are enabled on the [features tab](#) [p. 66] of the [profiling settings dialog](#) [p. 64] .

The tree view displays all arrays and classes in a hierarchical package tree. You can select

- **Classes**

A single class can be chosen by double-clicking on it or selecting it in the tree and clicking **[Ok]** or pressing the `Enter` key.

- **Packages**

An entire package **and all its recursively contained sub-packages** can be chosen by selecting the desired package in the tree and clicking **[Ok]** or pressing the `Enter` key.

- **Arrays**

An array type can be chosen by opening the `<Arrays>` top level node and double-clicking on the desired array type or selecting it and clicking **[Ok]** or pressing the `Enter` key.

You can leave the dialog by pressing `Escape` or clicking **[Cancel]**.

## B.6.11 Heap walker view section


### B.6.11.1 Heap walker view section

With the heap walker, you can find memory leaks, look at single instances and flexibly select and analyze objects in several steps.

Important notions are

- **the current snapshot**


The heap walker operates on a static snapshot of the heap which is taken by

- choosing *Edit->Take heap snapshot* from JProfiler's main menu
- clicking on the corresponding  toolbar button
- using the "Take heap snapshot for selection" action in the dynamic [memory views](#) [p. 88] .

If a snapshot has already been taken, it will be discarded after confirmation. If the [current session](#) [p. 59] is [detached](#) [p. 80] , it is not possible to take a new snapshot, Taking a snapshot may take a few seconds depending on the size of the profiled application.

- **the initial object set**

After a snapshot has been fully prepared, you are taken to the the [classes view](#) [p. 103] and all objects in the snapshot are displayed. You can return to this view at any later point by


- choosing *Edit->Heap walker start view* from JProfiler's main menu
- clicking on the the corresponding  toolbar button

- **the current object set**

After each selection step a new object set is created which then becomes the current object set. Starting with the initial object set, you add selection steps and change the current object set to drill down toward your objective. The contents of the current object set (any number of instances of classes and arrays) are described in the title area of the heap walker.

You can calculate the deep size of the entire object set by clicking on the "Calculate deep size" hyperlink in the title area. Once the deep size has been calculated, the hyperlink is replaced with the result.

The history of your selection steps can be shown at the bottom by

- choosing *Edit->Show selection steps* from JProfiler's main menu
- clicking on the the corresponding  toolbar button



- **the view on the current object set**

All views share the same [basic layout](#) [p. 100] . There are 4 views, all show the current object set:

- the [classes view](#) [p. 103]
- the [allocation view](#) [p. 104]
- the [reference view](#) [p. 106]
- the [data view](#) [p. 113]

The view is chosen either

- using the view selector at the bottom of the heap walker.
- or from the [view helper dialog](#) [p. 116] that is displayed each time a new object set is created. You can suppress this dialog in the [heap walker view settings](#) [p. 116] .

The   history controls of the heap walker in JProfiler's toolbar allow you to go backward and forward in the **history of your view changes**. View changes where selection steps were performed, as well as those performed through the view selector are recorded in the history.

Changing the current object set is done by clicking on the **[Use selected]** button in the heap walker views. You first select objects of interest and then use this button to create a new object set that contains only these objects. In many cases you can double click on an item to create a new object set with it.

The heap walker will be able to display allocation information only for recorded objects. See the [memory section overview](#) [p. 88] for further details.

#### B.6.11.2 Heap walker option dialog

The heap walker options dialog is displayed each time before the actual [heap snapshot](#) [p. 99] is taken.

- **Select recorded objects**

If this option is checked, the heap walker will restrict the heap snapshot to recorded objects only. In this way you can focus on the objects that have been created during a selected time span. If the option is unchecked, all objects on the heap will be shown (excluding any objects removed by the next option).

- **Remove unreferenced and weakly referenced objects**

If this option is checked, JProfiler will remove all objects from the heap that are **not strongly referenced**. These include:

- unreferenced objects that are eligible for garbage collection
- objects that are referenced only through soft, weak and phantom references
- objects that are in the finalizer queue and will be garbage collected as soon as the finalizers have been run

For strongly referenced objects, the heap walker will not display soft, weak and phantom references.

This mode is preferable for memory leak detection, and is especially helpful to obtain useful information when showing the [path to root](#) [p. 111] for selected objects. Unchecking this option allows you to analyze the heap "as-is".

1.3 JVMs perform a full garbage collection before the heap snapshot is taken, so unreferenced objects can never occur even if the option is unchecked. With 1.4 JVMs, no garbage collection is performed prior to the snapshot.

#### B.6.11.3 Heap walker view layout

All [heap walker views](#) [p. 99] share the same basic layout:

Current object set: 7087 objects in 336 classes, 3641 arrays  
1 selection step, 773 kB of memory

Use selected Selection button Description of current object set

Name	Instance count	Size
java.lang.String	2601	62424
char[ ]	2579	146888
<class>[ ]	951	45688
java.util.HashMap\$Entry	772	18528
java.util.Hashtable\$Entry	762	18288
java.lang.ref.Finalizer	370	11840
sun.java2d.loops.Blit	187	7480
java.lang.Integer	153	2448
sun.java2d.loops.ScaledBlit	152	6080
java.util.jar.Attributes\$Name	87	1392
sun.java2d.loops.SurfaceType	66	1584
java.security.AccessControlContext	63	1512

Selection step 1 : Recorded objects after full GC  
7087 objects in 336 classes, 3641 arrays

Selection history View selector

Classes Allocations References Data

The description of the current object set shows

- **what kind of objects** are in the current object set. If there is more than one class or array type in the current object set, a cumulative count will be given separately for class instances and arrays. As it is often the case, if all objects are of a single class or array type, the class name or array type will be displayed.
- **how many selection steps** have occurred so far. This gives an idea of the complexity of the current selection.
- **how much space** the current object set uses on the heap. Note that this is the shallow size which does not include the sizes of referenced arrays and class instances.

With the selection button you can add another selection step. A new object set that contains only the currently selected objects will be created. Some views have other view specific controls in this area.

The main portion of the view displays the content which depends on the current view type.

The selection history shows all selection steps that have occurred so far. The selection history pane is a vertical split pane and can be resized to the most convenient size. You can toggle the visibility of the selection history window by

- choosing *Edit->Show selection steps* from JProfiler's main menu
- clicking on the corresponding toolbar button

The view selector allows you to switch between the four different views **without changing the current object set**. The views show

- the [classes](#) [p. 103] in the current object set

- the [allocation spots](#) [p. 104] of the current object set
- the [references](#) [p. 106] of the current object set
- the [class and instance data](#) [p. 113] of the current object set

## B.6.11.4 Classes view

### B.6.11.4.1 Heap walker - classes

The heap walker classes view conforms to the [basic layout](#) [p. 100] of all heap walker views. Also see the [help on key concepts](#) [p. 99] for the entire heap walker.

The functionality of the classes view is identical to that of the [class monitor view](#) [p. 89] except that it is static with respect to the current snapshot and only instances of classes and arrays in the current object set are shown.

The classes view shows the list of all classes and arrays types together with the number of instances which are allocated on the heap. There are three columns shown in the table, which can be [sorted](#) [p. 85] .

- **Name**

Shows the name of the class or the array type. The notation `<class>[]` stands for non-primitive arrays of any class type. A further distinction is not possible due to the nature of Java bytecode.

- **Instance count**

Shows how many instances are currently allocated on the heap. This instance count is displayed graphically as well.

- **Size**

Shows the total size of all allocated instances. Note that this is the **shallow size** which does not include the size of referenced arrays and instances but only the size of the corresponding pointers. The size is in bytes and includes only the object data, it does not include internal JVM structures for the class, nor does it include class data or local variables.

No specific view settings apply to the classes view.

To add a selection step from this view you can

- select one or multiple classes or array types from the table and click the **[Use selected]** button above the table.
- double click on a single class or array type.

A new object set will be created that contains only the instances of the selected classed or array types. After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

## B.6.11.5 Allocation view

### B.6.11.5.1 Heap walker - allocations

The heap walker allocation view conforms to the [basic layout](#) [p. 100] of all heap walker views. Also see the [help on key concepts](#) [p. 99] for the entire heap walker.

The allocation view of the heap walker offers two **view modes** that can be changed in the combo box at the top of the view:

- [Cumulated allocation tree](#) [p. 104]  
Shows the allocation tree for the current object set. Each method node includes the allocations from all descendent method nodes.
- [Allocation tree](#) [p. 104]  
Shows the allocation tree for the current object set. Each method node only includes the allocations in that particular method.
- [Allocation hot spots](#) [p. 104]  
Shows the allocation hot spots for the current object set.

### B.6.11.5.2 Heap walker allocation view - Allocation tree

The allocation tree is one of the **view modes** in the [allocation view](#) [p. 104] of the [heap walker](#) [p. 99].

The contents and functionality of the allocation tree view mode correspond to those of the [allocation monitor](#) [p. 91] in the [memory view section](#) [p. 88]. Contrary to the allocation monitor, only allocations in the current object set are shown. You can customize this view through the [heap walker view settings](#) [p. 116].

The heap walker will be able to display allocation information only for recorded objects, unrecorded objects are summed up in a top-level entry called `unrecorded objects`. See the [memory section overview](#) [p. 88] for further details.

To add a selection step from this view you can

- select one or multiple allocation spots from the table and click the **[Use selected]** button above the table.
- double click on a single allocation spot.

A new object set will be created that contains

- all instances of classes and arrays allocated in the selected allocation spots **and in allocation spots below** for the `cumulated allocation tree` view mode.
- only the instances of classes and arrays allocated in the selected allocation spots for the `allocation tree` view mode.

After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

**Note:** If you wish to see the allocations performed in a method regardless on what call sequence has lead to this method, you can switch to the [allocation hot spots view mode](#) [p. 104].

### B.6.11.5.3 Heap walker allocation view - Allocation hot spots

The allocation hot spots list is one of the **view modes** in the [allocation view](#) [p. 104] of the [heap walker](#) [p. 99].



The contents and functionality of the allocation hot spots list are similar to those of the [allocation hot spots view](#) [p. 94] in the [memory view section](#) [p. 88] . Contrary to that view, only allocations in the current object set are shown. Also, back traces are not available. You can customize this view through the [heap walker view settings](#) [p. 116] .

The heap walker will be able to display allocation information only for recorded objects, unrecorded objects are summed up in a top-level entry called `unrecorded objects`. See the [memory section overview](#) [p. 88] for further details.

To add a selection step from this view you can

- select one or multiple allocation hot spots from the table and click the **[Use selected]** button above the table.
- double click on a single allocation hot spot.

A new object set will be created that contains all instances of classes and arrays allocated in the selected methods. After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

## B.6.11.6 Reference view

### B.6.11.6.1 Heap walker - Reference view

The heap walker reference view conforms to the [basic layout](#) [p. 100] of all heap walker views. Also see the [help on key concepts](#) [p. 99] for the entire heap walker.

The reference view of the heap walker offers three **view modes** that can be changed in the combo box at the top of the view:



- [Reference graph](#) [p. 106]  
Shows graphs of references for all objects in the current object set.
- [Cumulated incoming references](#) [p. 108]  
Shows a table of the cumulated references that hold the objects in the current object set.
- [Cumulated outgoing references](#) [p. 110]  
Shows a table of the cumulated references that originate from objects in the current object set.

The reference view helps you to find memory leaks. Please note the "Show path to GC root" functionality in the [reference graph](#) [p. 106] for this purpose.

### B.6.11.6.2 Heap walker reference view - Reference graph

The reference graph list is one of the **view modes** in the [reference view](#) [p. 106] of the [heap walker](#) [p. 99] .

The reference graph shows the incoming and outgoing references of all instances of classes and arrays which are contained in the current object set. There is always one instance visible at a time.

Above the upper right corner of the table,   navigation controls allow to move back and forth through all the instances or arrays in the current object set.

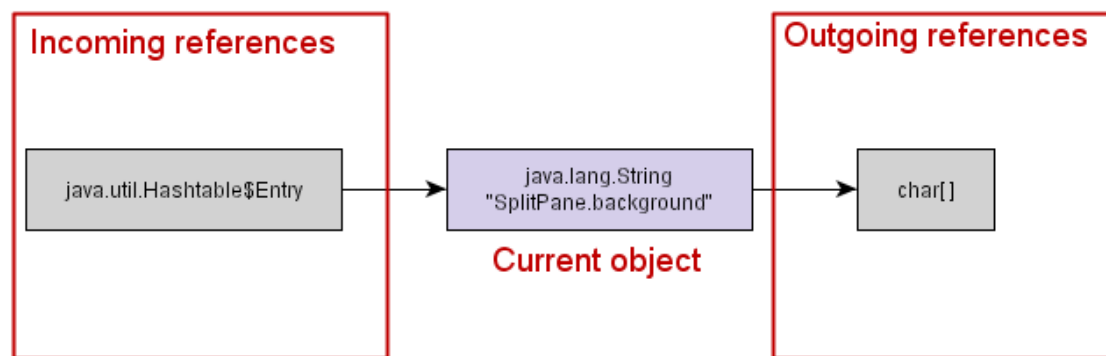
The order of the instances in the object set can be adjusted to

- **unsorted**  
The objects are in a random order. This is the default setting.
- **sorted by shallow size**  
Objects with a larger shallow size are displayed first.
- **sorted by deep size**  
Objects with a larger deep size are displayed first.

After changing the sort order, the displayed index is set to one.

The instance navigation is linked with the [data view](#) [p. 113] of the [heap walker](#) [p. 99] to allow you to easily switch back and forth between these views while keeping the focus on the same object.



No specific view settings apply to the reference graph.





The references graph has the following properties:

- Instances are painted as rectangles with the class name of the instance written inside the rectangle.
- References are painted as arrows, the arrowhead points from the holder toward the holdee. If you move the mouse over the reference, a **tooltip window** will be displayed that shows details for the particular reference.
- The current instance has a violet background. In the current instance, the shallow size as well as the deep size of the object are shown.
- Garbage collector roots have a red background.
- String values are shown directly in the `java.lang.String` instance rectangle.

By default, the reference graph only shows the direct incoming and outgoing references of the current instance. You can expand the graph by **double clicking on any object**. This will expand either the direct incoming or the outgoing references for that object, depending on the direction you're moving in. Selective actions for expanding the graph are available in the view-specific toolbar and the context menu:

-  Show outgoing references
-  Show incoming references


If applicable, an instance has plus and minus signs at the left and the right side to show or hide incoming and outgoing references. The controls at the left side are for incoming, the controls at the right side for outgoing references. The plus signs have the same effect as the  Show outgoing references and the  Show incoming references actions. A minus sign hides all outgoing references and all objects that are not connected to the central instance. This can have the effect that the object on which you click the minus sign is hidden as well.

Additionally, the plus and minus signs give you the following indications:

- **plus sign**  
There might be references to display. You have not yet tried to expand them.
- **minus sign**  
You have expanded all references, there are no more references to expand.
- **no sign**  
You have tried to expand references, but there were none.


To reset the graph to its original state, you can choose *Reset graph* from the context menu.

The reference graph offers a number of [navigation and zoom options](#) [p. 86] .

To check why an instance is not garbage collected, you can select it and use the  Show paths to GC root button in the view-specific toolbar or the corresponding entry in the context menu.

A [dialog](#) [p. 111] will ask you whether to search for a single garbage collector root or for all roots. After that, the paths to root are searched. This is a computationally expensive operation and can take some time. A progress dialog is shown while the paths to root are calculated.

- If the object is **not** referenced by a garbage collector root, a message box will be displayed. Note that this case is only possible if the "Remove unreferenced and weakly referenced objects" option in the [heap walker option dialog](#) [p. 100] is unchecked.
- Otherwise the graph is then expanded up to the garbage collector roots that were found. The garbage collector roots themselves are displayed with a red background.

There are three layout strategies for showing the reference graph which can be chosen by clicking on  in the toolbar or choosing the layout strategy from the context menu.

- **Hierarchic layout**

Standard layout that tries to layout the graph from left to right. This is suitable for most purposes.

- **Organic layout**

Layout that tries to layout instances for optimal proximity. This layout is suitable for complex situations and can visualize clusters.

- **Orthogonal layout**

Layout that tries to layout instances on a rectangular grid. This layout is suitable if your objects form a matrix.

To add a selection step from this view you can

- select one or multiple objects and click the **[Use ...]** button above the graph and choose *selected objects* in the popup menu. There is a corresponding entry in the context menu. A new object set will be created that contains only the selected instances. Multiple objects are selected by keeping the SHIFT key pressed during selection.
- select an array of objects or a standard collection from the `java.util` package and click the **[Use ...]** button above the graph and choose *items in selected collection* in the popup menu. There is a corresponding entry in the context menu. A new object set will be created that contains the objects in the array or collection. If you select a map collection, you are prompted if you want to include the key objects as well.

After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

#### **B.6.11.6.3 Heap walker reference view - Cumulated incoming references**

The cumulated incoming references table is one of the **view modes** in the [reference view](#) [p. 106] of the [heap walker](#) [p. 99] .

The cumulated incoming references table shows the list of all reference types through which the instances of classes and arrays in the current object set are held. This view has two display modes that determine how the "Object count" and the "Size" column have to be interpreted:

- **Show counts and sizes of reference holders**

The "Object count" and the "Size" columns refer to the objects that reference any objects in the current object set through a certain reference type.

- **Show counts and sizes of referenced objects**

The "Object count" and the "Size" columns refer to the objects in the current object set that are referenced through a certain reference type.

There are three columns shown in the table, which can be [sorted](#) [p. 85] :

- **Reference type**

Shows the type of the incoming reference which is one of

- **field**

some of the objects or arrays in the current object set are held in the indicated field of an instance of the indicated class.

- **static field**

some of the objects or arrays in the current object set are held in the indicated static field of the indicated class.

- **constant**

some of the objects or arrays in the current object set are held in the constant pool of the indicated class. These references mostly stem from constants declared as `private static final`.

- **class array content**

some of the objects in the current object set are held in an array of instances of classes. The arrays are of types or supertypes of the held objects. A further distinction is not possible due to the nature of Java bytecode.

- **JNI global/local reference**

some of the objects or arrays in the current object set are held through the Java Native Interface. Generally global references are persistent across a number of native calls which local references are only valid for the duration of one native call. These references are of interest to JNI programmers only. If you do not use any extra native libraries and encounter these reference types nonetheless, they can be attributed to the internal state of the JVM. In that case, there won't be any accessible objects behind these references and the `Size` column will show a zero value.

- **java stack**

some of the objects in the current object set are held in a stack frame of a thread.

- **sticky class, thread block, unknown type**

internal references in the JVM.

Note that for static fields, constants, java stack references and the internal references in the JVM the origin of the reference do not belong to accessible objects. The `Size` column shows a zero value and a filter selection is not possible for these incoming reference types.

- **Object count**

Depending on the display mode, shows

- **Show counts and sizes of reference holders**

How many objects are holding on to any object in the current object set through this reference type.

- **Show counts and sizes of referenced objects**

How many objects in the current object set are referenced through this reference type.

The reference count is displayed graphically as well.

- **Size**

Depending on the display mode, shows

- **Show counts and sizes of reference holders**

The total size of all objects that are holding on to any object in the current object set through this reference type.

- **Show counts and sizes of referenced objects**

The total size of all objects in the current object set that are referenced through this reference type.

Note that this is the **shallow size** which does not include the size of referenced arrays and instances but only the size of the corresponding pointers.

No specific view settings apply to the cumulated incoming reference table.

To add a selection step from this view you can

- select one or multiple references from the table and click the **[Use ...]** button above the table and choose *reference holders* in the popup menu. A new object set will be created that contains all objects that hold any object in the current object set by way of the selected reference types.
- select one or multiple references from the table and click the **[Use ...]** button above the table and choose *referenced objects* in the popup menu. A new object set will be created that contains all objects in the current set that are held by a reference of one of the the selected types.
- double click on a reference. Depending on the display mode, either the reference holders or the referenced objects are selected as the new object set.

A new object set will be created that contains all instances of classes and arrays that reference objects in the current object set via the selected references. After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

#### **B.6.11.6.4 Heap walker reference view - Cumulated outgoing references**

The cumulated outgoing references table is one of the **view modes** in the [reference view](#) [p. 106] of the [heap walker](#) [p. 99] .

The cumulated outgoing reference table shows the list of all reference types which originate from the instances of classes and arrays in the current object set.

There are three columns shown in the table, which can be [sorted](#) [p. 85] :

- **Reference type**

Shows the type of the outgoing reference which is one of

- **field**

the referenced object or array is held in the indicated field of an instance of the indicated class.

- **static field**

the referenced object or array is held in the indicated static field of the indicated class.

- **constant**

the referenced object or array is held in the constant pool of the indicated class. These references mostly stem from constants declared as `private static final`.

- **class array content**

the referenced object or array is held in an array of instances of classes (e.g. the array might be of type `String[]` or `Object[]`). A further distinction is not possible due to the nature of Java bytecode.

- **Object count**

Shows how many references of this outgoing reference type are present in the current object set. The reference count is displayed graphically as well.

- **Size**

Shows the total size of the object set which would result if this reference type was added as a filter step. Note that this is the **shallow size** which does not include the size of referenced arrays and instances but only the size of the corresponding pointers.


No specific view settings apply to the cumulated outgoing reference table.

To add a selection step from this view you can

- select one or multiple references from the table and click the **[Use selected]** button above the table.
- double click on a reference.

A new object set will be created that contains all instances of classes and arrays that are referenced by objects in the current object set via the selected references. After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

#### **B.6.11.6.5 Path to root option dialog**

The path to root option dialog is displayed after clicking the  Show path to GC root toolbar button in the [reference graph](#) [p. 106] of the [heap walker](#) [p. 99].

The path to root analysis can calculate:

- **a single root**

Only a single garbage collector root will be found. When searching for a memory leak, this option is often appropriate since any path to a garbage collector root will prevent the instance from being garbage collected.

- **up to a certain number of roots**

A specified maximum number of roots will be found and displayed. If a single root is not sufficient, try displaying one root more at a time until you get a useful result.

- **all roots**

All paths to garbage collector roots will be found and displayed. This analysis takes much longer than the single root option and can use a lot of memory.

After completing the dialog with the **[Ok]** button, the analysis will be calculated and the result will be shown in the reference graph.

With the **[Cancel]** button, the path to root option dialog is closed and no analysis is performed.

#### **B.6.11.6.6 Restricted availability of the reference view**

If the initial data set of the [heap walker](#) [p. 99] is displayed, the reference view is not available. You have to perform one selection step first. This can be one of

- [selection of one or several classes](#) [p. 103]
- [selection of one or several allocation spots](#) [p. 104]



After such a selection step, the reference view will be available.



## B.6.11.7 Data view

### B.6.11.7.1 Heap walker - Data view

The heap walker data view conforms to the [basic layout](#) [p. 100] of all heap walker views. Also see the [help on key concepts](#) [p. 99] for the entire heap walker.

The heap walker instance data view shows the instance data and the class data of all instances of classes and arrays which are contained in the current object set. There is always one instance visible at a time whose class name or array type is given above the field table. Above the upper right corner of the table,   navigation controls allow to move back and forth through all the instances or arrays in the current object set.

The order of the instances in the object set can be adjusted to

- **unsorted**  
The objects are in a random order. This is the default setting.
- **sorted by shallow size**  
Objects with a larger shallow size are displayed first.
- **sorted by deep size**  
Objects with a larger deep size are displayed first.


After changing the sort order, the displayed index is set to one.

The instance navigation is linked with the [reference graph](#) [p. 106] of the [heap walker](#) [p. 99] to allow you to easily switch back and forth between these views while keeping the focus on the same object.

The data view of the heap walker offers two or more **view modes** that can be changed in the combo box at the top of the view:

- [Instance data](#) [p. 114]  
Shows the instance data for the current instance.
- [Class data](#) [p. 114]  
Shows the class data of the class or a particular super class of the current instance. There is one such view mode for each class in the class hierarchy.

Above the main table, the shallow size and the deep size of the displayed instance are shown together with the class name. The deep size is calculated as the shallow size plus total size of all referenced objects. In extreme cases, this value may be a significant percentage of the entire heap.

To be able to select multiple objects from the current object set and create a new object set from them, you can **flag** each instance with the checkbox at the top right corner of the data view. The  menu button to the right of the checkbox allows you to

- clear all flags
- flag all instances up to the currently displayed instance

The **[Use ...]** button above the table shows a popup where you can select all flagged as well as all unflagged instances (see below).

To add a selection step from this view you can

- select an entry from the table that is either a `java.lang.String` object or if it has a `[reference]` value and click the **[Use ...]** button above the table and choose *this instance* from the popup menu.

Alternatively, you can double click on the entry. An object set will be created that contains only the selected object.

- click the **[Use ...]** button above the table and choose *this instance* from the popup menu. An object set will be created that contains only the currently displayed instance.
- click the **[Use ...]** button above the table and choose *flagged instance* or *unflagged instance* from the popup menu. An object set will be created that contains only the flagged or unflagged instances.

After your selection, the [view helper dialog](#) [p. 116] will assist you in choosing the appropriate view for the new object set.

#### B.6.11.7.2 Heap walker data view - Instance data

The instance data table is one of the **view modes** in the [data view](#) [p. 113] of the [heap walker](#) [p. 99].

The main table in the instance data table lists all fields of the current instance or all array elements of the current array. There are three columns shown in the table, which can be [sorted](#) [p. 85].

- **Number**  
Shows the number of the field in the class file.
- **Field name**  
Shows type and name for the field if a class instance is displayed or `array element` if an array is displayed.
- **Value**  
Shows the value of the field as
  - the explicit contents of the field for primitive field types and instances of `java.lang.String`.
  - `[reference]` for non-primitive field types which hold a live reference.
  - `null` for non-primitive field types which are empty.

No specific view settings apply to the class data view.

Please see the help on the [data view](#) [p. 113] for how to perform selection steps from this view.

#### B.6.11.7.3 Heap walker data view - Class data

The class data table is one of the **view modes** in the [data view](#) [p. 113] of the [heap walker](#) [p. 99].

The main table in the class data table first lists the static fields of the current class, then those constant pool entries which are references. There are three columns shown in the table, which can be [sorted](#) [p. 85]. Note that sorting by number always keeps static fields and constant pool references together.

- **Number**  
Shows the number of the static field in the class file or the ordinal number of the constant pool reference.
- **Field name**  
Shows type and name for the static field or the number of the constant pool entry in the class file.
- **Value**  
Shows the value of the field as
  - the explicit contents of the field for primitive field types and instances of `java.lang.String`.

- `[reference]` for non-primitive field types which hold a live reference.
- `null` for non-primitive field types which are empty.

No specific view settings apply to the class data view.

Please see the help on the [data view](#) [p. 113] for how to perform selection steps from this view.

#### **B.6.11.7.4 Restricted availability of the data view**

If the initial data set of the [heap walker](#) [p. 99] is displayed, the data view is not available. You have to perform one selection step first. This can be one of

- [selection of one or several classes](#) [p. 103]
- [selection of one or several allocation spots](#) [p. 104]

After such a selection step, the data view will be available.

### B.6.11.8 Heap walker view helper dialog

The view helper dialog is displayed each time a new object is created. New object sets are created by choosing objects in the [heapwalker views](#) [p. 99] and clicking on the **[Use selected]** button.

The view helper dialog is intended to assist you in choosing the view that is most interesting for the new object set. You can switch to desired view by selecting the corresponding radio button and closing the dialog with the **[Ok]** button. On the right hand side of the dialog a short description of the selected view is displayed.


The view helper dialog automatically suggest a view based on the contents of the new object set.

To discard the new object set you can leave the dialog with the **[Cancel]** button. You will then be returned to the previous heap walker view.

You can suppress this dialog by clicking the *Do not show this dialog again* checkbox at the bottom of the dialog. In this case the view change to the automatically suggested view will be performed without confirmation.

To show the dialog again at a later time, you can adjust this setting in the [heap walker view settings](#) [p. 116] .

### B.6.11.9 Heap walker view settings dialog

The heap walker view settings dialog is accessed by bringing the [heap walker view](#) [p. 99] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The various context menus also give access to the view settings dialog.

The **General** tab of the view settings dialog controls aspects which apply to all heap walker views.

- **Show selection steps**

If checked, the selection history window at the bottom of the heap walker is shown.

- **Show view helper dialog for new object sets**

If checked, the [view helper dialog](#) [p. 116] will be displayed when a new object set is created.

The **Allocations** tab applies to the [allocation view](#) [p. 104] only. It is analogous to the [allocation monitor view settings](#) [p. 92] .

**Note:** Unlike for the allocation monitor view, there is no "cumulate allocations" option since the view mode combo box in the allocations view of the heap walker offers both an "allocation tree" and a "cumulated allocation tree".

The **References** tab applies to the [references view](#) [p. 106] only.

- **Show object IDs**




If checked, all objects in the [reference graph](#) [p. 106] are annotated with object IDs. This can help you in checking if two objects in two different reference graphs are the same or not.

## B.6.12 CPU view section

### B.6.12.1 CPU view section

The CPU view section contains several views which are **thread resolved**. Directly above those views you can see the current selection of thread and thread state.

The thread selection can be one of

-  thread groups
-  active threads
-  dead threads


Next to the thread selector you find information about the **thread state** which is one of


- **All states**  
No filtering is performed.
- **Runnable**  
Only runnable thread states will be shown. This is the standard setting.
- **Waiting**  
Only waiting thread states will be shown.
- **Blocked**  
Only blocked thread states will be shown.
- **Net I/O**  
Only blocking network operations of the java library will be shown.

In the dynamic views thread selection and thread state are displayed in combo boxes. After changing the selection in the thread selector or the thread state selector, the dynamic views are updated immediately with the new settings. The thread selector applies to all dynamic views simultaneously. Initially it is set to `All thread groups` and may be switched to specific threads or thread groups as soon as they come into existence.

Please turn to the [thread view section](#) [p. 129] for more detailed information on threads.

The update frequency can be set on the [miscellaneous tab](#) [p. 67] in the [profiling settings dialog](#) [p. 64] for all dynamic views of the CPU view section.

Unless "Record CPU data on startup" has been selected in the `Startup` section of the [profiling settings dialog](#) [p. 64], data acquisition has to be started manually by clicking on  **Record CPU data** in the tool bar or by selecting *Profiler->Record CPU data* from JProfiler's main menu.

CPU data acquisition can be stopped by clicking on  **Stop recording CPU data** in the tool bar or by selecting *Profiler->Stop recording CPU data* from JProfiler's main menu.

**Restarting** data acquisition **resets** the CPU data in all dynamic views of the CPU view section.

The CPU view section contains the

- [invocation tree view](#) [p. 119]  
The invocation tree view shows top down call trees for the selected thread or thread group.
- [hot spots view](#) [p. 121]  
The hot spots view shows the methods where most of the time is spent in the profiled application.

- [method graph](#) [p. 124]

The method graph shows call graphs for selected methods and threads.

- [CPU statistics](#) [p. 127]

The statistics view shows statically calculated statistics for package and method resolved CPU time.


## B.6.12.2 Invocation tree view

### B.6.12.2.1 Invocation tree view


The invocation tree view shows a [thread resolved](#) [p. 117] top-down call tree which is [downward and upward filtered](#) [p. 77] for the [selected filter sets](#) [p. 65] .

The entries in the invocation tree have different meanings which are indicated by the displayed icons:


- **node method above threshold**

 This points to a method whose time usage is above the threshold set in the [invocation view settings](#) [p. 120] . There are other methods called from this method which are above the threshold as well.


- **node method below threshold**

 This points to a method whose time usage is below the threshold set in the [invocation view settings](#) [p. 120] . However, there are other methods called from this method which are above the threshold so this method must be shown to complete the tree up to the root.

- **leaf method above threshold**



 This points to a method whose time usage is above the threshold set in the [invocation view settings](#) [p. 120] . There are no other unfiltered methods called from here on. If this is a top level node, it may inform you about direct calls to other filtered classes which never call a method from an unfiltered class.

- **upward filter bag**

 This points to a section of the invocation tree which traverses methods from [filtered classes](#) [p. 65] , such as the AWT dispatch mechanism with java core classes filtered or servlets and Enterprise Java Beans called from a filtered application server framework. Between the shown method and the unfiltered children there may be any number of method invocation that are filtered and thus not shown.

Every entry in the invocation tree has textual information attached which - depending on the [invocation view settings](#) [p. 120] shows


- a **percentage number** which is calculated with respect to either the root of the tree or the calling method.
- a **total time measurement** in ms or  $\mu$ s. This is the total time that includes calls into other methods.
- an **inherent time measurement** in ms or  $\mu$ s. This is the inherent time that does not include calls into unfiltered methods.
- an **invocation count** which shows how often the method has been invoked on this path.
- a **method name** which is fully qualified or relative with respect to to the calling method.
- a **line number** which is only displayed if line number resolution has been enabled in the [profiling settings](#) [p. 66] and if the calling class is unfiltered. Note that the line number shows the line number of the invocation and not of the method itself.

When **navigating** through the call tree by opening method calls, JProfiler automatically expands methods which only call one other method themselves. To quickly **expand larger portions** of the invocation tree, select a method and choose  *Edit->Expand 10 levels* from the main window's menu or choose the corresponding menu item from the context menu. If you want to **collapse an opened part** of the invocation tree, select the topmost method that should remain visible and choose  *Edit->Collapse all* from the main window's menu or the context menu.

You can set **change the root** of the invocation tree to any method by selecting that method and choosing *Edit->Set as root* from the main window's menu or by choosing the corresponding menu item from the context menu. Percentages will now be calculated with respect to the new root if the percentage base has been set to "total thread time" in the [view settings dialog](#) [p. 120]. To **return to the full view** of all methods called in the current thread or thread group, select *Edit->Show all* from the main window's menu or the context menu.

You can [stop and restart CPU data acquisition](#) [p. 117] to clear the invocation tree and [freeze all views](#) [p. 80] to ensure that the invocation tree remains static.


#### B.6.12.2.2 Invocation tree view settings dialog

The invocation view settings dialog is accessed by bringing the [invocation tree view](#) [p. 119] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog.

The **invocation description** options control the amount of information that is presented in the description of the method call.

- **Show time**  
Show the total time that was spent in the method call.
- **Show inherent time**  
Show the inherent time (excluding calls to unfiltered methods) that was spent in the method call.
- **Show invocation count**  
Show how many time the method was called in this particular call sequence.
- **Always show fully qualified names**  
If this option is not checked, class name are omitted in intra-class method calls which enhances the conciseness of the display.
- **Always show signature**  
If this option is not checked, method signatures are shown only if two methods with the same name appear on the same level.

The **time scale** options determine the unit of time measurement, which may be **milliseconds** (ms) or **microseconds** (μs)

The **threshold** below which method invocations are ignored is entered in **microseconds** (μs). Method calls which are below this threshold are not shown in the invocation tree except for the case where they are part of a call sequence which leads to a method call with a cumulated duration above the given threshold. Those method calls are indicated by a  downward arrow.

The **percentage base** determines against what time span percentages are calculated.

- **Absolute**  
Percentage values show the contribution to the total time.
- **Relative**  
Percentage values show the contribution to the calling method.



### B.6.12.3 Hot spot view

#### B.6.12.3.1 Hot spots view

The hot spots view shows a list of all method calls which use at least 0.1% of the total time spent in the current thread or thread group and lie above the threshold defined in the [hot spots view settings](#) [p. 122] . See the help on the [estimated CPU time/elapsed time setting](#) [p. 67] and take into account the selection of the [thread state selector](#) [p. 117] to properly assess the meaning of this total time. By opening a hot spot method entry, the tree of backtraces leading to that method call are calculated and shown.

**Note:** The notion of a hot spot is relative. Hot spots depend on the filter sets that you have enabled on the [call tree collection tab](#) [p. 65] of the [profiling settings dialog](#) [p. 64] . Filtered methods are opaque, in the sense that calls into other filtered methods are attributed to their own time. If you change your filter sets you're likely to get different hot spots since you are changing your point of view. Please see [the help topic on hotspots and filters](#) [p. 34] for a detailed discussion.


Every hot spot is described in several columns:

- the **method name**
- the **inherent time** in ms or  $\mu$ s of how much time has been spent in the hot spot together with a bar whose length is proportional to this value. All calls into this method are summed up regardless of the particular call sequence.


If the method belongs to an unfiltered class, this time does not include calls into other methods. If the method belongs to a filtered class, this time includes calls into other filtered methods.

- the **invocation count** of the hot spot.


The hot spot list can be [sorted on all columns](#) [p. 85] .

If you click on the  handle on the left side of a hot spot, a tree of backtraces will be shown. The entries in the hot spots backtraces tree have different meanings which are indicated by the displayed icons:


- **method above threshold**

 This points to a method whose time usage is above the threshold set in the [hot spots view settings](#) [p. 122] .

- **method below threshold**

 This points upward from a method whose time usage is below the threshold set in the [invocation view settings](#) [p. 120] , but is needed to display the whole backtrace.

- **invocation root**

 The root of the invocation.

Every entry in the backtrace tree has textual information attached to it which depends on the [hot spots view settings](#) [p. 122] .

- a **percentage number** which is calculated with respect either to the total time or the called method.
- a **time measurement** in ms or  $\mu$ s of how much time has been contributed to the parent **hot spot** on this path.
- an **invocation count** which shows how often the **hot spot** has been invoked on this path.

**Note:** This is **not** the number of invocations of this method.



- a **method name** which is fully qualified or relative with respect to the calling method.

- a **line number** which is only displayed if line number resolution has been enabled in the [profiling settings](#) [p. 66] and if the calling class is unfiltered. Note that the line number shows the line number of the invocation and not of the method itself.

The combo box at the top-right corner of the view allows you to treat calls to filtered classes in two different ways:


- **show separately**  
Filtered classes can be hotspots of their own. This is the default mode.
- **add to calling class**  
Calls to filtered classes are always added to the calling class. In this mode, a filtered class cannot be a hotspot, except if it is a top-level upward filter bag, i.e. if it is not called by any unfiltered class, but calls unfiltered classes itself.

With these two modes you can change your viewpoint and the definition of a hotspot. Please see [the help topic on hotspots and filters](#) [p. 34] for a detailed discussion of this topic.

When **navigating** through the hot spots backtraces tree by opening method calls, JProfiler automatically expands methods which are only called by one other method themselves. To quickly **expand larger portions** of the hot spots backtraces tree, select a method and choose  *Edit->Expand 10 levels* from the main window's menu or choose the corresponding menu item from the context menu. If you want to **collapse an opened part** of the hot spots backtraces tree, select the topmost method that should remain visible and choose  *Edit->Collapse all* from the main window's menu or the context menu.

You can [stop and restart CPU data acquisition](#) [p. 117] to clear the invocation tree and [freeze all views](#) [p. 80] to ensure that the invocation tree remains static.


#### B.6.12.3.2 Hot spots view settings dialog

The hot spots view settings dialog is accessed by bringing the [hot spots view](#) [p. 121] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog.

The **invocation description** options control the amount of information that is presented in the description of the method call.

- **Show time**  
Show the total time that was spent in the method call.
- **Show invocation count**  
Show how many time the method was called in this particular call sequence.
- **Always show fully qualified names**  
If this option is not checked, class name are omitted in intra-class method calls which enhances the conciseness of the display.
- **Always show signature**  
If this option is not checked, method signatures are shown only if two methods with the same name appear on the same level.

The **time scale** options determine the unit of time measurement, which may be **milliseconds** (ms) or **microseconds** (µs)

The **threshold** below which hot spots are ignored and method invocations in hot spots backtraces are shown as being below threshold (indicated by an  upward arrow) is entered in **milliseconds** (ms).

The **percentage calculation** determines against what time span percentages are calculated.

- **Absolute**

Percentage values show the contribution to the total recorded time.


- **Relative**


Percentage values shows the contribution to the invoked method.

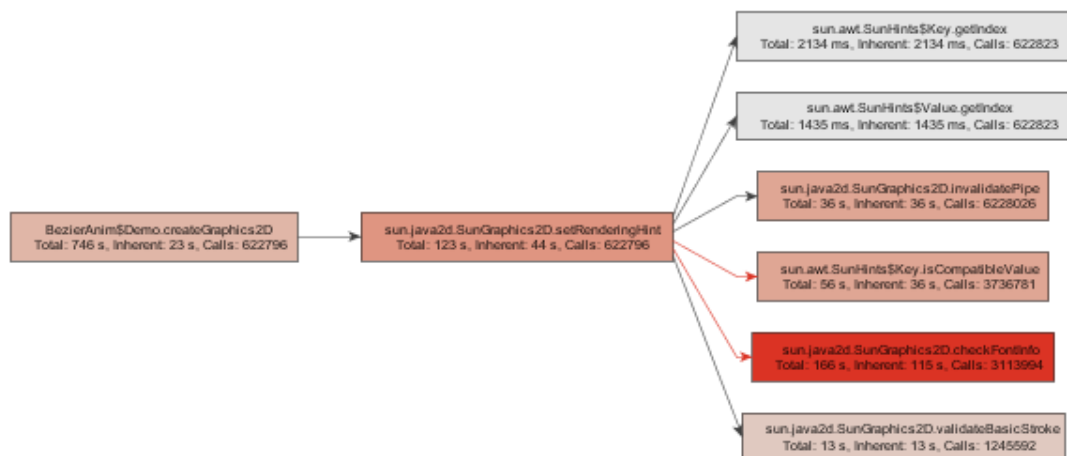
## B.6.12.4 Method graph

### B.6.12.4.1 Method graph view

The method graph view shows a statically calculated [thread resolved](#) [p. 117] call graph for selected methods.

To calculate a call graph, click  **Generate graph** in the tool bar or select *Edit->Generate graph* from JProfiler's main menu. If a graph has been calculated, the context menu also provides access to this action.

Before a graph is calculated, the [method graph wizard](#) [p. 125] is brought up. The resulting graph is static and can be re-calculated by executing  **Generate graph** again. The method graph wizard remembers your last selection.



The references graph has the following properties:




- Methods are painted as rectangles. The rectangle includes information about
  - The method name without parameters. In order to see the parameters of a method, switch on **signature tooltips** in the [method graph view settings](#) [p. 126] or select the corresponding check item in the context menu.
  - The total time (including calls into unfiltered methods)
  - The inherent time (excluding calls into unfiltered methods)
  - The number of calls into this method
- The method rectangles have a background coloring which - depending on the [method graph view settings](#) [p. 126] - is taken from a gray to red scale for increasing
  - inherent time
  - **or** total time

The percentage base is

- the time spent in the displayed methods only
- **or** the time spent in all methods

- Method calls are painted as arrows, the arrowhead points from the caller toward the callee. If you move the mouse over the call arrow, a **tooltip window** will be displayed that shows details for the particular call.
- Method call arrows have a color which is taken from a black to red scale for an increasing percentage in execution time. In this way you can spot the most important calls of a method without checking their tooltips one by one.

By default, the method graph only shows the direct incoming and outgoing calls of the initially selected methods. You can expand the graph by **double clicking on any method**. This will expand the direct incoming and outgoing calls for that method. Selective actions for expanding the graph are available in the toolbar, the *Edit* menu and the context menu:

-  Show calling methods
-  Show called methods
-  Add methods to graph, to add other unrelated methods to the graph. The [method selection dialog](#) [p. 125] will then be displayed.

You can **hide methods** by selecting them and pressing the delete key. You can select multiple methods and delete them together.

The reference graph offers a number of [navigation and zoom options](#) [p. 86] .

#### B.6.12.4.2 Method graph wizard

The method graph wizard is displayed before a [method graph](#) [p. 124] is calculated and sets parameters for the method graph.

##### 1. Thread selection

Similar to the [dynamic views of the CPU view section](#) [p. 117] , you can select a thread or thread group and a thread state for which the method graph will be calculated.

##### 2. Initially displayed methods

The method graph initially displays a number of selected methods and their immediate call environment. Select one or multiple methods in this step. The method table shows

- method name
- inherent time
- total time
- invocations

and can be [sorted](#) [p. 85] on all columns. Initially it is sorted by inherent time to show the most interesting hot spots at the top of the table.

You can add further methods later on with the [method selection dialog](#) [p. 125] .


After you click **[Finish]** in the last step, the method graph will be calculated, if you leave the wizard with **[Cancel]**, you are returned to the old method graph.

#### B.6.12.4.3 Method selection dialog

The method selection dialog is displayed when adding new methods to the [method graph](#) [p. 124] .

The method selection dialog offers a list of methods similar to step 2 in the [method graph wizard](#) [p. 125] . If you leave the dialog with **[Ok]**, the selected methods and their immediate call environments will be shown in the method graph. If you leave the dialog with **[Cancel]**, the method graph will not be changed.

#### B.6.12.4.4 Method graph view settings dialog

The method graph settings dialog is accessed by bringing the [method graph](#) [p. 124] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog.

- **Show signature tooltips**

If checked, the signature of a methods will be shown in a **tooltip window** when you move the mouse over it.

- **Color information**

This setting determines the meaning of the gray to red scale of the background color of method rectangles. It can be one of

- Inherent time
- Total time

- **Color scale base**


This setting determines the percentage base for calculating the background color of method rectangles. It can be one of


- Displayed methods only. If this setting is checked, the coloring of method changes as new methods are expanded or added.
- All methods. If this setting is checked, the coloring stays the same as new methods are expanded or added.

## B.6.12.5 CPU statistics

### B.6.12.5.1 CPU statistics

The CPU statistics view shows statically calculated [thread resolved](#) [p. 117] statistics for package and method resolved CPU times.

To calculate a statistics, click  **Calculate statistics** in the tool bar or select *Edit->Calculate statistics* from JProfiler's main menu. If a statistics has been calculated, the context menu also provides access to this action.

Before a statistics is calculated, the [CPU statistics options dialog](#) [p. 128] is brought up. The resulting statistics table is static and can be re-calculated by executing  **Calculate statistics** again. The statistics options dialog remembers your last selection.

The package level statistics table displays two columns:

- **Package**  
the name of the package. If `cumulate packages` has been selected in the [options dialog](#) [p. 128], every possible package in the package hierarchy will be displayed, even if there are no classes in it.
- **Time**  
the time spent in the package together with a percentage bar and a percentage value. If `cumulate packages` has been selected in the [options dialog](#) [p. 128], times are cumulated for sub-packages. If the package is [filtered](#) [p. 77], calls into other filtered classes are included in this time measurement.

The class level statistics table displays two columns:

- **Class**  
the fully qualified name of the class.
- **Time**  
the time spent in the method of the class together with a percentage bar and a percentage value. If the class is [filtered](#) [p. 77], calls into other filtered classes are included in this time measurement.

The method statistics table displays three columns:

- **Name**  
the fully qualified name of the method.
- **Time**  
the time spent in the method together with a percentage bar and a percentage value. If the method is from a [filtered class](#) [p. 77], the internal calls into other filtered methods are cumulated in this value. If the method belongs to a filtered class, calls into other filtered classes are included in this time measurement.
- **Invocations**  
the number of times this method was invoked.

**Note:** If you would like to see **all** invoked methods in the JVM, please [de-select all filter sets](#) [p. 65] and [switch to full instrumentation or sampling](#) [p. 65].

#### B.6.12.5.2 CPU statistics options

The CPU statistics options dialog sets parameters for the output of the [CPU statistics view](#) [p. 127] .

Choose one of the following statistics:

- **Packages statistics**

Show times spent in each package. If **Cumulate packages** is selected, times are cumulated for sub-packages in the package hierarchy. Also, every possible package in the package hierarchy will be displayed, even if there are no classes in it. If **Cumulate packages** is not selected, packages are treated as separate entities without hierarchy.

- **Methods statistics**

Show times spent in each method and the invocation count for each method.

With the **Thread** and **Thread status** combo boxes you can choose for which thread and thread state the statistic will be calculated. This selection is similar to the one for the [dynamic CPU views](#) [p. 117] .



## **B.6.13 Threads view section**

### **B.6.13.1 Thread view section**

The thread view section contains the

- [threads history view](#) [p. 130]

The threads history view shows detailed historic information about the status of all threads in the JVM.

- [threads monitor view](#) [p. 133]

The threads monitor view shows dynamic information about the currently running threads.

- [deadlock detection graph](#) [p. 134]

The deadlock detection graph paints a graphic visualisation of all deadlocks in the JVM.

- [current monitor usage view](#) [p. 135]

The current monitor usage view shows monitors that are currently involved in a waiting or blocking operation.

- [monitor usage history view](#) [p. 135]

The monitor usage history view shows waiting and blocking operations on monitors.

- [monitor usage statistics](#) [p. 136]


The statistics view shows statically calculated statistics for monitor usage.

### B.6.13.2 Thread history view

#### B.6.13.2.1 Thread history view

The thread history view shows the list of all threads in the JVM in the order they were started. On the left hand side of the view, the names of the threads appear as a fixed column, the rest of the view is filled with a scrollable measuring tool which shows time on its horizontal axis. The origin of the time axis coincides with the starting time of the first thread in the JVM. Each alive thread is shown as a colored line which starts when the thread is started and ends when the thread dies. The color indicates a certain thread status and is one of


- **green**

 Green color means that the thread is **runnable** and eligible for receiving CPU time by the scheduler. This does not mean that the thread has in fact consumed CPU time, only that the thread was ready to run and was not blocking or sleeping. How much CPU time a thread is allotted, depends on various other factors such as general system load, the thread's priority and the scheduling algorithm.


- **orange**

 Orange color means that the thread is **waiting**. The thread is sleeping and will be woken up either by a timer or by another thread.

- **red**

 Red color means that the thread is **blocking**. The thread has is trying to enter a `synchronized` code section or a `synchronized` method whose monitor is currently held by another thread.

- **blue**

 Light blue color means that the thread is in **Net I/O**. The thread is waiting for a `network operation` of the java library to complete. This thread state occurs if a thread is listening for socket connections or if it is waiting to read or write data to a socket.

At the top of the view, there is a thread filter selector. You can use it to filter the displayed threads by

- **liveness status**

From the combo box you can choose if you wish to display

- Both alive and dead threads
- Alive threads only
- Dead threads only

- **name**

In the text box you can enter the full name of a thread or only a part of it. Only threads whose names begin with this fragment are displayed. You can also use wildcards ("\*" and "?") to select groups of threads. Please note that if you use wildcards, you have to manually append a trailing "\*" if desired. You can display the union of multiple filters at the same time by separating multiple filter expressions with commas, e.g. `AWT- , MyThreadGroup-* -Daemon`.


The selection is performed once you press the enter key. The combo box contains all entries performed during the current session. The **[Reset filters]** button can be used to remove all filters.

When you move the mouse across the thread history view, the time at the position of the mouse cursor will be shown in JProfiler's status bar.



The thread history view has two different display modes. The display mode is a persistent view setting and is thus also accessible through the [thread history view settings dialog](#) [p. 131] .

- **fixed time scale**

If you currently are in the "scale to fit window" mode, you can switch to this mode by

- choosing the  scale mode selector button in the lower right corner of the view.
- choosing *Edit->Scale to fit window* from JProfiler's main menu.
- choosing *Scale to fit window* from the context menu.
- checking *Scale to fit window* in the [thread history view settings dialog](#) [p. 131] .


In this mode, the time scale on the time axis does not change with time and the time axis can be scrolled with the scrollbar on the bottom which appears if the total time span does not fit into the current view size. If the current time is visible, the view is in **auto-follow mode** where the time axis is scrolled automatically when new data arrives to always show the current time. If you are not in auto-follow mode, because you scrolled back in time, just move the scrollbar to the right end of the time scale to re-enable auto-following.

You can adjust the scale of the time axis by **zooming in or out**.  Zooming in increases the level of detail while  zooming out decreases it. You change the zoom level by

- using the zoom controls in the lower right corner of the view.
- choosing *Edit->Zoom in* and *Edit->Zoom out* from JProfiler's main menu.
- choosing *Zoom in* and *Zoom out* from the context menu.

- **scale to fit window**


If you currently are in the "fixed time scale" mode, you can switch to this mode by

- choosing the  scale mode selector button in the lower right corner of the view.
- choosing *Edit->Continue at fixed scale* from JProfiler's main menu.
- choosing *Continue at fixed scale* from the context menu.
- unchecking *Scale to fit window* in the [thread history view settings dialog](#) [p. 131] .

The time scale on the time axis is adjusted continuously in order to show the total time span in the current size of the view. Zooming is not possible in this mode.

Grid lines and background of the thread history view can be configured in the [thread history view settings dialog](#) [p. 131] .

#### B.6.13.2.2 Thread history view settings dialog

The thread history view settings dialog is accessed by bringing the [thread history view](#) [p. 130] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog. The following options are available:

- **Scale to fit window**

Determines whether the view operates in the "fixed time scale" or "scale to fit window" mode. These modes are described in the thread history view [help page](#) [p. 130] .

- **Grid lines for time axis**

Controls on what ticks grid lines will be shown along the time axis.

- **Background**

Controls the appearance of the background of the thread history view.

### B.6.13.3 Thread monitor view

#### B.6.13.3.1 Thread monitor view

The thread monitor view shows the filtered list of all threads in the JVM together with associated information on times and status. There are a maximum of six columns shown in the table, which can be [sorted](#) [p. 85] .

- **Name**

Shows the name of the thread. If the thread has not been named explicitly, the name is provided by the JVM. To make most use of this view, name your threads according to their function by invoking the `setName( )` method on all threads created by you.

- **Group**

Shows the name of the thread group associated with this thread.

- **Start time**

Shows the time when the thread has been started. This time is calculated relative to the start time of the first thread in the JVM.

- **End time**

This column is only visible when `show dead threads` is enabled in the [view settings dialog](#) [p. 133] . It shows the time when the thread has died and is empty if the thread is still alive. This time is calculated relative to the start time of the first thread in the JVM.

- **CPU time**


Shows the CPU time which has been consumed by the thread. This column may be empty if your system and JVM do not support thread specific CPU time reporting.

- **Status**

Shows the status of the thread which corresponds to the status reported in the [thread history view](#) [p. 130] .

You can decide which threads are shown in the thread monitor view by checking the desired filters in the [thread monitor view settings dialog](#) [p. 133] . If `Show dead threads` is not enabled, the `End time` column will not be visible.

#### B.6.13.3.2 Thread monitor view settings dialog

The invocation view settings dialog is accessed by bringing the [invocation tree view](#) [p. 119] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. The context menu also gives access to the view settings dialog. The following options are available:

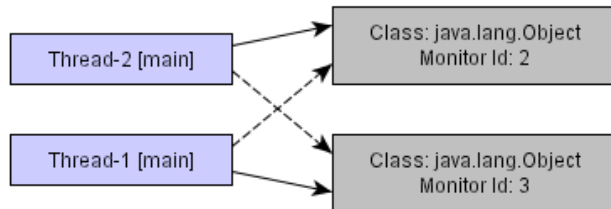
- Show runnable threads
- Show waiting threads
- Show blocking threads
- Show threads in net I/O
- Show dead threads

These options determine the filter for the [thread monitor view](#) [p. 133] . See the thread history [help page](#) [p. 130] for a detailed explanation of the different types of thread status.

### B.6.13.4 Deadlock detection graph

#### B.6.13.4.1 Deadlock detection graph

The deadlock detection graph shows a graph of threads and monitors that describe a deadlock detection. For the normal situation where no deadlocks are present a "No deadlocks detected" message is displayed in this view.



The deadlock detection graph has the following properties:

- Threads which participate in the deadlock are painted as violet rectangles. The rectangle includes information about
  - The thread name
  - The thread group (in brackets)
- Monitors which participate in the deadlock are painted as gray rectangles. The rectangle includes information about
  - The class of the monitor
  - The monitor id which can be used to get further information about the monitor in the [monitor contention views](#) [p. 135]
- The **ownership of monitors** which participate in the deadlock are painted as solid arrows. The arrowhead points from the thread to the monitor. To see details about where the monitor was entered, move the mouse over the arrow and see the information in the **tooltip window**.
- The **blocking causes** of threads which participate in the deadlock are painted as dashed arrows. The arrowhead points from the blocked thread to the monitor that the thread wants to enter. To see details about where the thread is blocking, move the mouse over the arrow and see the information in the **tooltip window**.

Multiple deadlocks are painted side by side.

## B.6.13.5 Monitor usage views

### B.6.13.5.1 Monitor contention views

Monitor contention views show a table where every row corresponds to a waiting or blocking event on a monitor. There are two monitor contention views:

- [current monitor usage view](#) [p. 135]

The current monitor usage view shows monitors that are currently involved in a waiting or blocking operation.

- [monitor usage history view](#) [p. 135]

The monitor usage history view shows the sequence of waiting and blocking operations on monitors above the [configured thresholds](#) [p. 66] .

Note that these views are unavailable if [monitor contention views are disabled](#) [p. 66] .

The monitor contention views share the following 6 columns: [sortable](#) [p. 85] .

- **Time**

The start time of the event.

- **Duration**

The duration of the event. The event may still be in progress.

- **Type**

The type of the event, one of "waiting" or "blocked".

- **Monitor ID**

The ID of the monitor for identifying multiple events on a particular monitor instance.

- **Monitor class**

The class of the monitor. If no Java object is associated with this monitor [`raw monitor`] is displayed.

- **Waiting thread**

The thread that is or was waiting during the event.

### B.6.13.5.2 Current monitor contentions view

The current monitor usage view shows monitors that are currently involved in a waiting or blocking operation. In addition to the common columns of the [monitor contention views](#) [p. 135] , an **Owning thread** column is displayed where the thread holding the monitor which is blocking the waiting thread is displayed. The owning thread is only relevant for the "blocked" event type.

In the lower part of the split pane, the stack traces of the waiting thread and - if applicable - of the owning thread are displayed.

### B.6.13.5.3 Monitor contention history view


The monitor usage history view shows the sequence of waiting and blocking operations on monitors above the [configured thresholds](#) [p. 66] . See the [monitor usage overview](#) [p. 135] for additional information.


In the lower part of the split pane, the stack trace of the waiting thread is displayed.

## B.6.13.6 Monitor usage statistics

### B.6.13.6.1 Monitor usage statistics

The monitor usage statistics view shows statically calculated statistics for monitor usage. Note that this view is unavailable if [monitor contention views are disabled](#) [p. 66] .

To calculate a statistics, click  **Calculate statistics** in the tool bar or select *Edit->Calculate statistics* from JProfiler's main menu. If a statistics has been calculated, the context menu also provides access to this action.

Before a statistics is calculated, the [monitor usage statistics options dialog](#) [p. 136] is brought up. The resulting statistics table is static and can be re-calculated by executing  **Calculate statistics** again. The statistics options dialog remembers your last selection.

The package level statistics table displays five columns:

- **Monitors/Threads/Classes**

Displays the grouping criterion selected in the [statistics dialog](#) [p. 136] ,

- **Block count**

Shows how often a block operation has been performed on the monitors grouped in this row.

- **Block duration**

Shows the cumulative duration of all block operations performed on the monitors grouped in this row.

- **Wait count**

Shows how often a waiting operation has been performed on the monitors grouped in this row.

- **Wait duration**

Shows the cumulative duration of all waiting operations performed on the monitors grouped in this row.

### B.6.13.6.2 Monitor usage statistics options

The monitor usage statistics options dialog sets parameters for the output of the [monitor usage statistics view](#) [p. 136] . Select the criterion for which monitors will be cumulated, one of

- Monitors
- Threads
- Classes of monitors



## B.6.14 VM telemetry view section

### B.6.14.1 Telemetry view section

The telemetry view section shows a number of historic graphs which display cumulated information about the profiled JVM.

There are five views in this section:

- **Heap**

Shows the maximum heap size and the amount of used and free space in it. This view can be displayed as a line graph or area graph.

- **Objects**

Shows the total number of objects on the heap, divided into arrays and non-arrays. This view can be displayed as a line graph or area graph. Note that this view only displays [recorded objects](#) [p. 88] and is unavailable if no objects have been recorded so far. Objects that have been recorded are tracked even after recording has been stopped.

- **Garbage collector**

Shows the activity of the garbage collector which comprises one line for the objects freed and another line for the objects moved. The plotted values are time rates, so the total numbers in a time interval are given by the area under the respective lines. Note that this view only displays [recorded objects](#) [p. 88] and is unavailable if no objects have been recorded so far. Objects that have been recorded are tracked even after recording has been stopped.

- **Classes**

Shows the total number of classes loaded by the JVM, divided into [filtered and unfiltered](#) [p. 77] classes. This view can be displayed as a line graph or area graph.

- **Threads**

Shows the total number of alive threads in the JVM, divided into active and inactive threads. This view can be displayed as a line graph or area graph.

The graph type is a persistent view setting separate for each view and is thus also accessible through the [VM telemetry view settings dialog](#) [p. 138]. Where possible, switching between line and area graph is done by

- Choosing *Edit->Graph type->Line graph* or *Edit->Graph type->Area graph* from JProfiler's main menu.
- Choosing *Line graph* or *Area graph* from the context menu.
- Choosing *Line graph* or *Area graph* in the [VM telemetry view settings dialog](#) [p. 138].


When a view is shown as an area graph, the line which shows the total value is given by the upper bound of the filled area while the single contributions are shown as stacked area segments.

When you move the mouse across a telemetry view, the time at the position of the mouse cursor and the corresponding value on the vertical axis will be shown in JProfiler's status bar.



The telemetry views have two different display modes. The display mode is a persistent view setting separate for each view and is thus also accessible through the [VM telemetry view settings dialog](#) [p. 138].

- **fixed time scale**

If you currently are in the "scale to fit window" mode, you can switch to this mode by

- choosing the  scale mode selector button in the lower right corner of the view.
- choosing *Edit->Scale to fit window* from JProfiler's main menu.
- choosing *Scale to fit window* from the context menu.
- checking *Scale to fit window* in the [VM telemetry view settings dialog](#) [p. 138] .


In this mode, the time scale on the time axis does not change with time and the time axis can be scrolled with the scrollbar on the bottom which appears if the total time span does not fit into the current view size. If the current time is visible, the view is in **auto-follow mode** where the time axis is scrolled automatically when new data arrives to always show the current time. If you are not in auto-follow mode, because you scrolled back in time, just move the scrollbar to the right end of the time scale to re-enable auto-following.

You can adjust the scale of the time axis by **zooming in or out**.  Zooming in increases the level of detail while  zooming out decreases it. You change the zoom level by

- using the zoom controls in the lower right corner of the view.
- choosing *Edit->Zoom in* and *Edit->Zoom out* from JProfiler's main menu.
- choosing *Zoom in* and *Zoom out* from the context menu.

- **scale to fit window**


If you currently are in the "fixed time scale" mode, you can switch to this mode by

- choosing the  scale mode selector button in the lower right corner of the view.
- choosing *Edit->Continue at fixed scale* from JProfiler's main menu.
- choosing *Continue at fixed scale* from the context menu.
- unchecking *Scale to fit window* in the [VM telemetry view settings dialog](#) [p. 138] .

The time scale on the time axis is adjusted continuously in order to show the total time span in the current size of the view. Zooming is not possible in this mode.

Horizontal and vertical grid lines of the VM telemetry views can be configured in the [VM telemetry view settings dialog](#) [p. 138] .

#### B.6.14.2 VM telemetry view settings dialog

The VM telemetry view settings dialog is accessed by bringing any [VM telemetry view](#) [p. 137] to front and choosing *Edit->View settings* from [JProfiler's main menu](#) [p. 81] or clicking on the corresponding  toolbar entry. View settings are saved separately for each telemetry view. The context menu also gives access to the view settings dialog. The following options are available:

- **Scale to fit window**

Determines whether the view operates in the "fixed time scale" or "scale to fit window" mode. These modes are described in the VM telemetry view [help page](#) [p. 137] .

- **Grid lines for time axis**

Controls on what ticks grid lines will be shown along the time axis.

- **Grid lines for vertical axis**

Controls on what ticks grid lines will be shown along the vertical axis.

- **Graph type**

This option is only visible for telemetry view which allow the [area graph display mode](#) [p. 137] .  
Choose between Line graph and Area graph.

## B.7 Offline profiling

### B.7.1 Offline profiling

JProfiler's offline profiling capability allows you to run profiling sessions from the command line without the need for starting JProfiler's GUI front end. Offline profiling makes sense if you want to

- perform profiling runs from a scripted environment (e.g. an [ant](#) build file)
- save snapshots on a regular basis for QA work
- profile server components on remote machines via slow network connections

Performing an offline profiling run for your application is analogous to [remote profiling](#) [p. 69] with special library parameters passed to the `-Xrunjprofiler` JVM argument:

- **offline switch**

Passing `offline` as a library parameter enables offline profiling. In this case, a connection with JProfiler's GUI is not possible.

- **session ID**

In order for JProfiler to set the correct profiling settings, a corresponding session has to be configured in JProfiler's GUI front end. The ID of that session has to be passed as a library parameter: `id=nnnn`. Your settings in the [profiling settings dialog](#) [p. 64] are used for offline profiling. The session ID can be seen in the top right corner of the [application settings dialog](#) [p. 59] .

- **config file location**

The config file that is read for extracting the session with the specified ID has to be passed via `config={path to config.xml}`. The config file is located in the `.jprofiler` directory in your user home directory (on Windows, the user home directory is typically `c:\Documents and Settings\%USER%`).

A summary of all library parameters is available in the [remote session invocation table](#) [p. 70] .

If you profile on a machine where JProfiler is not installed, you will need to transfer the contents of the `bin/{your platform}` directory as well as the JAR file `bin/agent.jar` and the config file `{User home directory}/.jprofiler3/config.xml`.

#### Example:

A typical invocation for offline profiling will look like this:

```
java -Xint "-Xrunjprofiler:offline,id=109,config=C:\Documents and
Settings\bob\.jprofiler3\config.xml"
"-Xbootclasspath/a:C:\Program Files\JProfiler\bin\agent.jar"
-classpath myapp.jar com.mycorp.MyApp
```

Please study the [remote session invocation table](#) [p. 70] to generate the correct invocation for your JVM. Also, please don't forget that the platform-specific native library path has to be modified, just like for [remote profiling](#) [p. 69] .

To control CPU profiling, triggering of heap dumps and saving of snapshots during an offline profiling session, please use JProfiler's [profiling API](#) [p. 141] .

### B.7.2 Profiling API

JProfiler provides a profiling API that allows you to control certain aspects of profiling at run time. The profiling API is contained in *bin/agent.jar* in your JProfiler installation. If the profiling API is used during a normal execution of your application, the API calls will just quietly do nothing.

For [offline profiling](#) [p. 140] , **you will need to save a snapshot at some point** in order to evaluate the data of the profiling run with JProfiler's GUI front end later on. The `saveSnapshot` method in JProfiler's profiling API does that job. For interactive use, this method call will do nothing.

In addition, you can optionally switch on CPU profiling at a suitable point and trigger heap dumps with the profiling API.