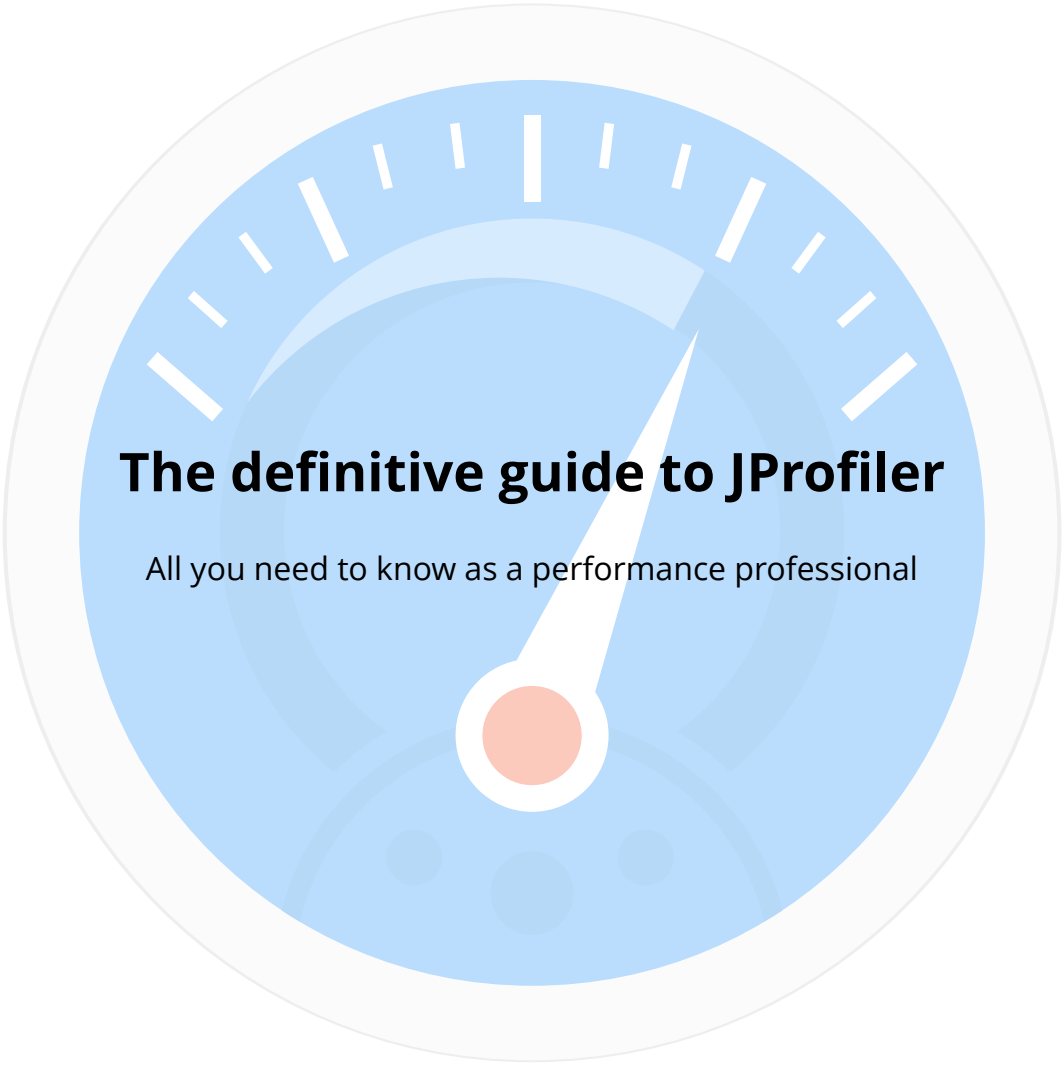
ej Technologies



The definitive guide to JProfiler

All you need to know as a performance professional

© 2021 ej-technologies GmbH. All rights reserved.

Index

Introduction 4

Architecture 5

Installing 7

Profiling a JVM 10

Recording data 23

Snapshots 36

Telemetries 43

CPU profiling 48

Method call recording 61

Memory profiling 66

The heap walker 75

Thread profiling 91

Probes 97

MBean browser 109

Offline profiling 113

Comparing snapshots 118

IDE integrations 124

A Custom probes 132

- A.1 Probe concepts 132
- A.2 Script probes 138
- A.3 Injected probes 142
- A.4 Embedded probes 147

B Call tree features in detail 151

- B.1 Auto-tuning for instrumentation 151
- B.2 Async and remote request tracking 154
- B.3 Viewing parts of the call tree 160
- B.4 Splitting the call tree 165
- B.5 Call tree analyses 169

C Advanced CPU analysis views 174

- C.1 Outlier detection 174
- C.2 Complexity analysis 178

C.3 Call tracer 180
C.4 Javascript XHR 182

D Heap walker features in detail 185
D.1 HPROF snapshots 185
D.2 Minimizing overhead 187
D.3 Filters and live interactions 189
D.4 Finding memory leaks 192

E Configuration in detail 199
E.1 Trouble shooting connection problems 199
E.2 Scripts 200

F Command line reference 203
F.1 Executables for profiling 203
F.2 Executables for snapshots 206
F.3 Gradle tasks 215
F.4 Ant tasks 219

Introduction To JProfiler

What is JProfiler?

JProfiler is a professional tool for analyzing what is going on inside a running JVM. You can use it in development, for quality assurance and for firefighting missions when your production system experiences problems.

There are four main topics that JProfiler deals with:

- **Method calls**

  This is commonly called "CPU profiling". Method calls can be measured and visualized in different ways. The analysis of method calls helps you to understand what your application is doing and find ways to improve its performance.

- **Allocations**

  Analyzing objects on the heap with respect to their allocations, reference chains and garbage collection falls into the category of "memory profiling". This functionality enables you to fix memory leaks, use less memory in general and allocate fewer temporary objects.

- **Threads and locks**

  Threads can hold locks, for example by synchronizing on an object. When multiple threads cooperate, deadlocks can occur and JProfiler can visualize them for you. Also, locks can be contended, meaning that threads have to wait before they can acquire them. JProfiler provides insight into threads and their various locking situations.

- **Higher level subsystems**

  Many performance problems occur on a higher semantic level. For example, with JDBC calls, you probably want to find out which SQL statement is the slowest. For subsystems like that, JProfiler offers "probes" that attach specific payloads to the call tree.

JProfiler's UI is delivered as a desktop application. You can interactively profile a live JVM or profile automatically without using the UI. Profiling data is persisted in snapshots that can be opened with the JProfiler UI. In addition, command line tools and build tool integrations help you with automating profiling sessions.

How do I continue?

This documentation is intended to be read in sequence, with later help topics building on the content of previous ones.

First, a technical overview over the architecture [p. 5] will help you to understand how profiling works.

The help topics on installing JProfiler [p. 7] and profiling JVMs [p. 10] will get you up and running.

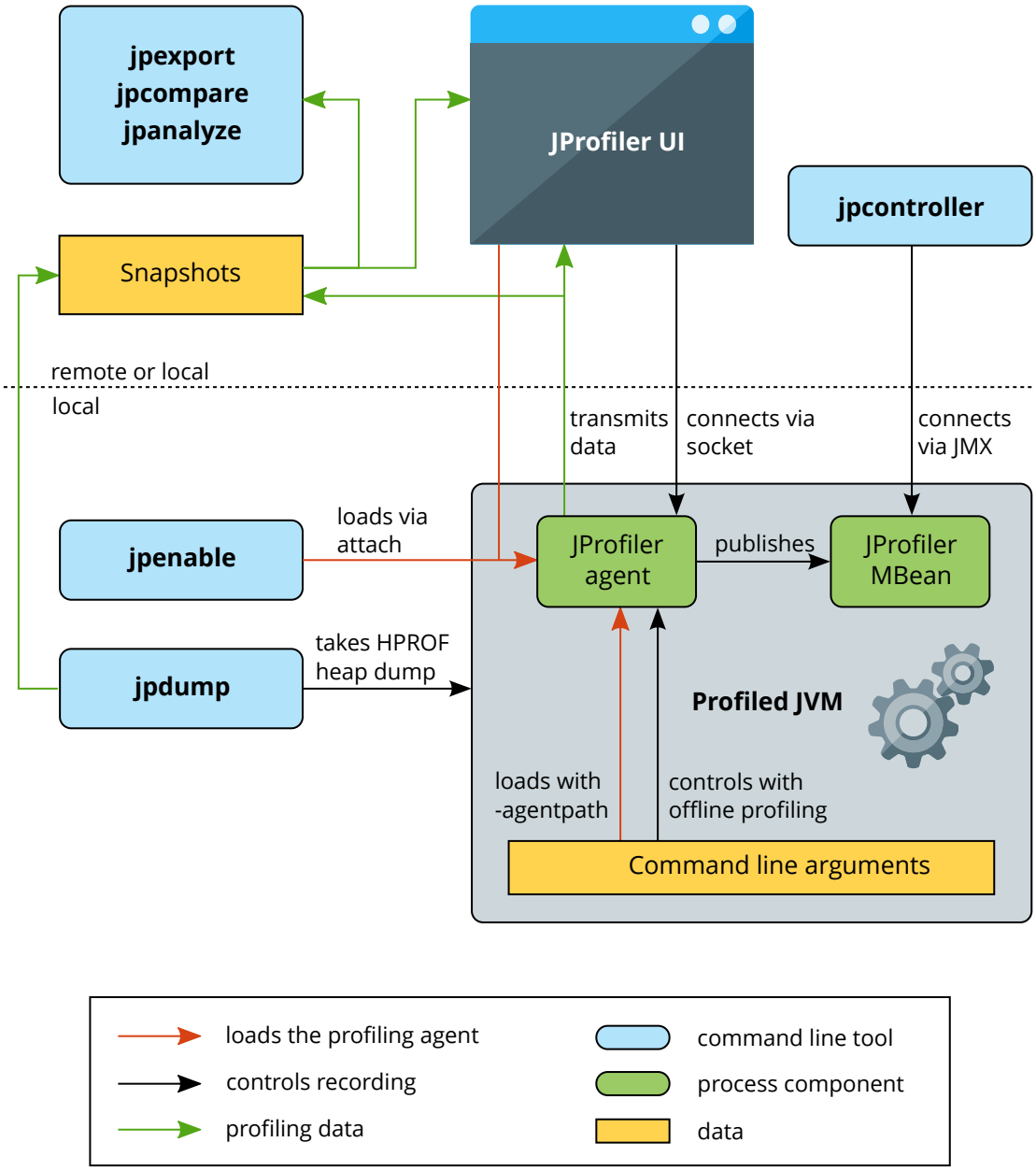
Following that, the discussion of data recording [p. 23] and snapshots [p. 36] take you to a level of understanding where you can explore JProfiler on your own.

Subsequent chapters build your expertise with respect to different functionality in JProfiler. The sections at the end are optional readings that should be consulted if you need certain features.

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.

JProfiler Architecture

The big picture of all important interactions involving the profiled application, the JProfiler UI and all command line utilities is given below.



The profiling agent

The "JVM tool interface" (JVMTI) is a native interface that a profiler uses to gain access to information and add hooks for inserting its own instrumentation. This means that at least part

of the profiling agent must be implemented as native code and so a JVM profiler is not platform-independent. JProfiler supports a range of platforms that are listed on the web site [1].

A JVM profiler is implemented as a native library that is loaded either at startup or at some point later on. To load it at startup, a VM parameter `-agentpath:<path to native library>` is added to the command line. You rarely have to add this parameter manually, because JProfiler will add it for you, for example in an IDE integration, an integration wizard or if it launches the JVM directly. However, it's important to know that this is what enables profiling.

If the JVM succeeds in loading the native library, it calls a special function in the library to give the profiling agent a chance to initialize itself. JProfiler will then print a couple of diagnostic messages prefixed with `JProfiler>` so you know that profiling is active. The bottom line is that if you pass the `-agentpath` VM parameter, the profiling agent is either loaded successfully or the JVM does not start.

Once loaded, the profiling agent asks the JVMTI to be notified of all kinds of events, such as thread creation or class loading. Some of these events directly deliver profiling data. Using the class loading event, the profiling agent instruments classes as they are loaded and inserts its own bytecode to perform its measurements.

JProfiler can load the agent into an already running JVM, either by using the JProfiler UI, or with the `bin/jpenable` command line tool. In that case, a substantial number of already loaded classes may have to be retransformed in order to apply the required instrumentation.

Recording data

The JProfiler agent only collects the profiling data. The JProfiler UI is started separately and connects to the profiling agent through a socket. This means that it is actually irrelevant if the profiled JVM is running on the local machine or on a remote machine - the communication mechanism between the profiling agent and the JProfiler UI is always the same.

From the JProfiler UI, you can instruct the agent to record data, display the profiling data in the UI and save snapshots to disk. As an alternative to the UI, the profiling agent can be controlled through its MBean [2]. A command line tool that uses this MBean is `bin/jpcontroller`.

Yet another way to control the profiling agent is with a predefined set of triggers and actions. In that way the profiling agent can operate in unattended mode. This is called "offline profiling" in JProfiler and is useful for automating profiling sessions.

Snapshots

While the JProfiler UI can show live profiling data, it is often necessary to save snapshots of all recorded profiling data. Snapshots are either saved manually in the JProfiler UI or automatically by trigger actions.

Snapshots can be opened and compared in the JProfiler UI. For automated processing, the command line tools `bin/jpexport` and `bin/jpcompare` can be used to extract data and create HTML reports from previously saved snapshots.

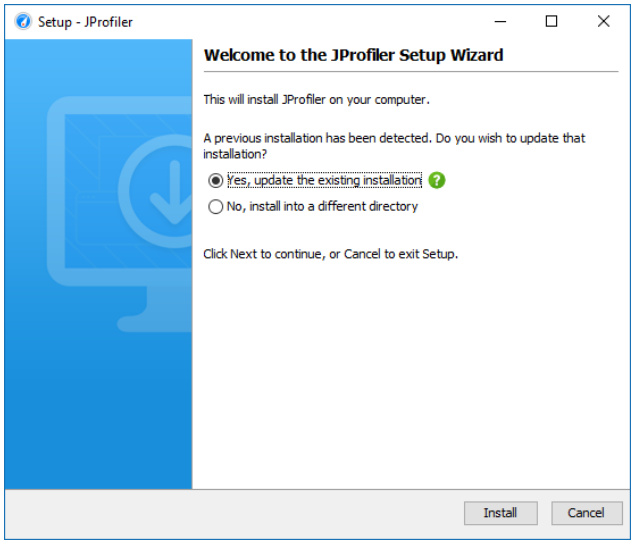
A low-overhead way of obtaining a heap snapshot from a running JVM is to use the `bin/jpdump` command line tool. It uses the built-in functionality of the JVM to save an HPROF snapshot that can be opened by JProfiler and does not require the profiling agent to be loaded.

[1] https://www.ej-technologies.com/products/jprofiler/featuresPlatforms.html

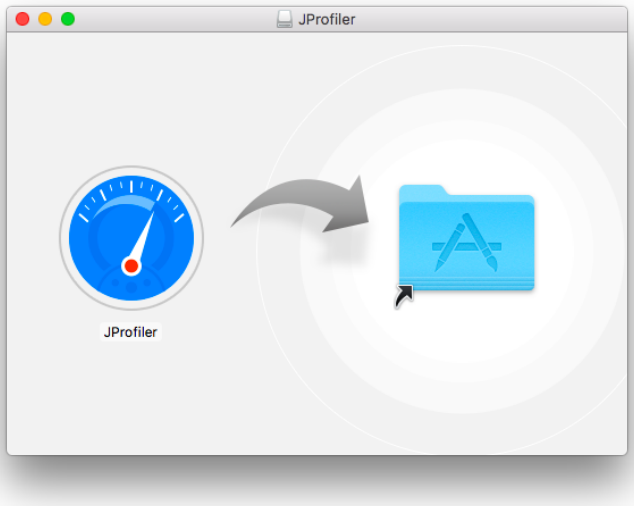
[2] https://en.wikipedia.org/wiki/Java_Management_Extensions

Installing JProfiler

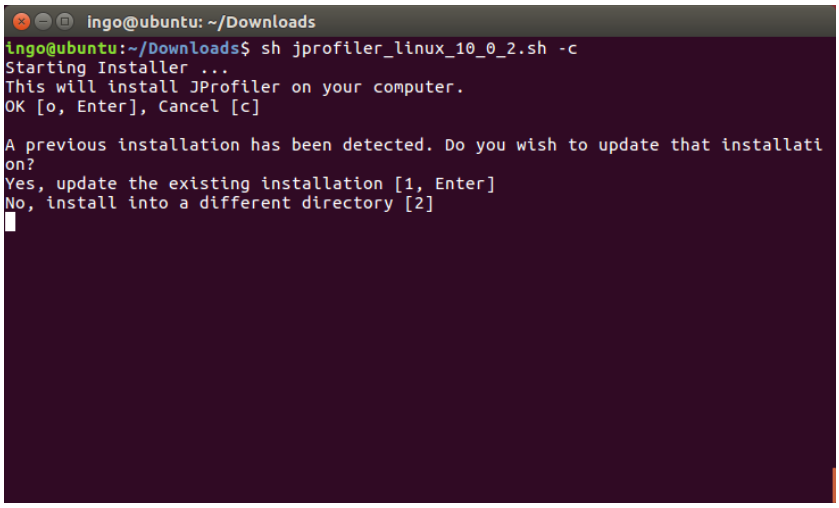
Executable installers are provided for Windows and Linux/Unix that lead you step-by step through the installation. If a previous installation is detected, the installation is streamlined.



On macOS, JProfiler uses the standard installation procedure for UI applications: a DMG archive is provided that you can mount in the Finder by double-clicking on it, then you can drag the JProfiler application bundle to the `/Applications` folder. That folder is visible as a symbolic link in the DMG itself.



On Linux/Unix, installers are not executable after download, so you have to prepend `sh` when executing them. The installer performs a command line installation if you pass the parameter `-c`. Completely unattended installations for Windows and Linux/Unix are performed with the parameter `-q`. In that case, you can pass the additional argument `-dir <directory>` in order to choose the installation directory.



After you run an installer, it will save a file `.install4j/response.varfile` that contains all user input. You can take that file and use it to automate unattended installations by passing the argument `-varfile <path to response.varfile>` on the command line.

To set licensing information for unattended installations, pass `-Vjprofiler.licenseKey=<license key> -Vjprofiler.licenseName=<user name>` and optionally `-Vjprofiler.licenseCompany=<company name>` as command line arguments. If you have a floating license, use `FLOAT:<server name or IP address>` instead of the license key.

Archives are also provided as ZIP files for Windows and as .tar.gz files for Linux. The command

```
tar xzvf filename.tar.gz
```

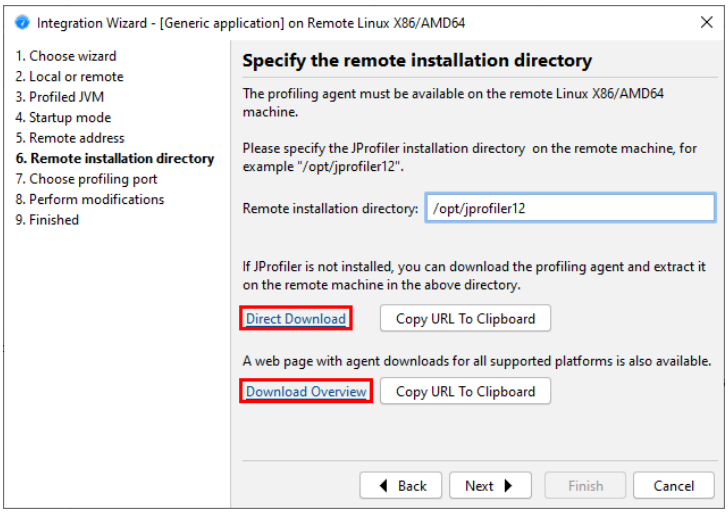
will extract a .tar.gz archive into a separate top-level directory. To start JProfiler, execute `bin/jprofiler` in the extracted directory. On Linux/Unix, the file `jprofiler.desktop` can be used to integrate the JProfiler executable into your window manager. For example, on Ubuntu you can drag the desktop file into the launcher side bar in order to create a permanent launcher item.

Distributing the profiling agent to remote machines

JProfiler has two parts: The desktop UI together with the command line utilities that operate on snapshots on the one hand, and the profiling agent together with the command line utilities that control the profiled JVM on the other hand. The installers and archives that you download from the web site contain both parts.

For remote profiling, however, you only need the profiling agent to be installed on the remote side. While you can simply extract an archive with the JProfiler distribution on the remote machine, you may want to limit the amount of required files, especially when automating a deployment. Also, the profiling agent is freely redistributable, so you can ship it with your application or install it on customer machines for trouble-shooting.

To get a minimal package with the profiling agent, the remote integration wizards shows you the download link for the appropriate agent archive as well as the download page with the agent archives for all supported platforms. In the JProfiler GUI, invoke *Session->Integration Wizards->New Server/Remote Integration*, select the "Remote" option and then proceed to the *Remote installation directory* step.



The URL for the HTML overview page for a particular JProfiler version is

```
https://www.ej-technologies.com/download/jprofiler/agent?version=12.0.4
```

The format of the download URLs for the single agent archives is

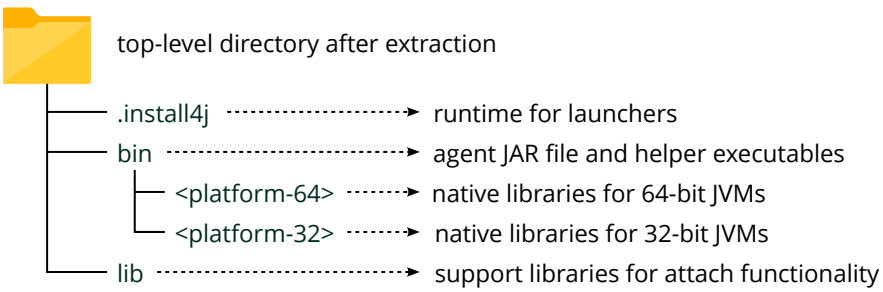
```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_<platform>_12_0_4.<extension>
```

where `platform` corresponds to the agent directory name in the `bin` directory and `extension` is `zip` on Windows, `.tgz` on macOS and `.tar.gz` for Linux/Unix. For Linux, x86 and x64 are grouped together, so for Linux x64 the URL is

```
https://download.ej-technologies.com/jprofiler/jprofiler_agent_linux-x86_12_0_4.tar.gz
```

The agent archive contains the required native agent libraries together with the `jpenable`, `jpdump` and`jpcontroller` executables. The executables in the archive only require Java 6 as a minimum version, while the profiling agent works with Java 5 or higher.

The sub-directories that you see after extracting the agent archive on the remote machine are described below. They are a subset of a full JProfiler installation on the respective target platform.



Profiling A JVM

To profile a JVM, JProfiler's profiling agent has to be loaded into the JVM. This can happen in two different ways: By specifying an `-agentpath` VM parameter in the start script or by using the attach API to load the agent into an already running JVM.

JProfiler supports both modes. Adding the VM parameter is the preferred way to profile and is used by the integration wizards, the IDE plugins and session configurations that launch a JVM from within JProfiler. Attaching works both locally as well as remotely over SSH.

-agentpath VM parameter

It is useful to understand how the VM parameter that loads the profiling agent is composed. `-agentpath` is a generic VM parameter provided by the JVM for loading any kind of native library that uses the JVMTI interface. Because the profiling interface JVMTI is a native interface, the profiling agent must be a native library. This means that you can only profile on the explicitly supported platforms [(1)]. 32-bit and 64-bit JVMs also need different native libraries. Java agents, on the other hand, are loaded with the `-javaagent` VM parameter and only have access to a limited set of capabilities.

After `-agentpath:`, the full path name to the native library is appended. There is an equivalent parameter `-agentlib:` where you only specify the platform-specific library name, but then you have to make sure that the library is contained in the library path. After the path to the library, you can add an equals sign and pass options to the agent, separated by commas. For example, on Linux, the whole parameter could look like this:

```
-agentpath:/opt/jprofiler10/bin/linux-x64/libjprofilerti.so=port=8849,nowait
```

The first equals sign separates the path name from the parameters, the second equals sign is part of the parameter `port=8849`. This common parameter defines the port on which the profiling agent is listening to connections from the JProfiler GUI. 8849 is actually the default port, so you can also omit that parameter. If you want to profile multiple JVMs on the same machine, you have to assign different ports, though. The IDE plugins and the locally launched sessions assign this port automatically, for integration wizards you have to choose the port explicitly.

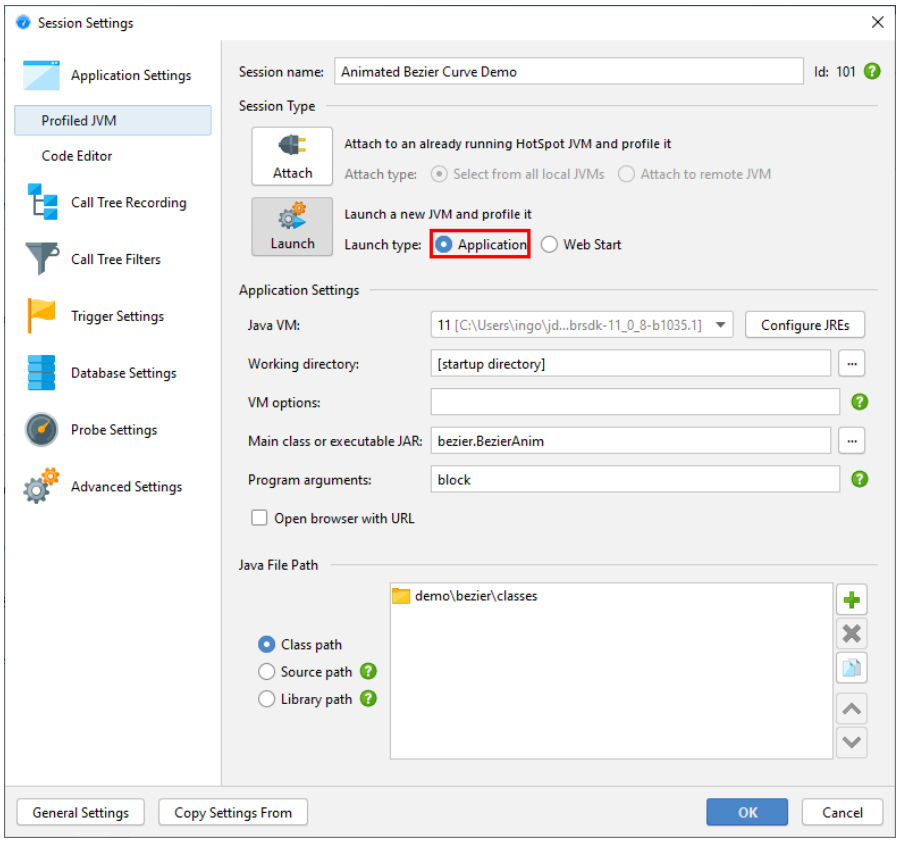
The second parameter `nowait` tells the profiling agent not to block the JVM at startup and wait for a JProfiler GUI to connect. Blocking at startup is the default, because the profiling agent does not receive its profiling settings as command line parameters but from the JProfiler GUI or alternatively from a config file. The command line parameters are only for bootstrapping the profiling agent, telling it how to get started and for passing debug flags.

By default, the JProfiler agent binds the communication socket to all available network interfaces. If this is not desirable for security reasons, you can add the option `address=[IP address]` in order to select a specific interface.

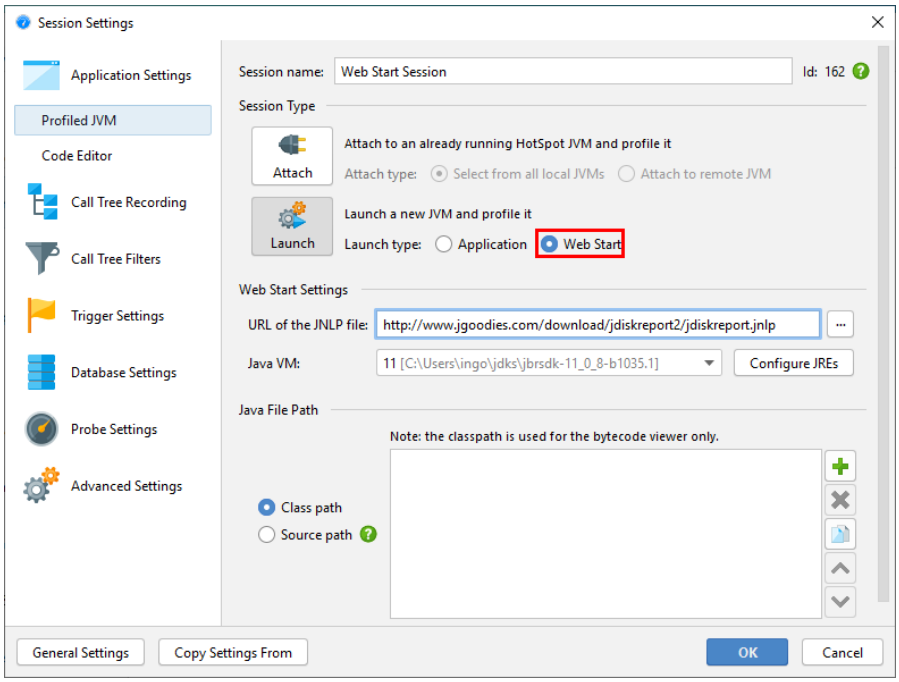
Locally launched sessions

Like "Run configurations" in an IDE, you can configure locally launched sessions directly in JProfiler. You specify the class path, the main class, working directory, VM parameters and arguments and JProfiler launches the session for you. All the demo sessions that ship with JProfiler are locally launched sessions.

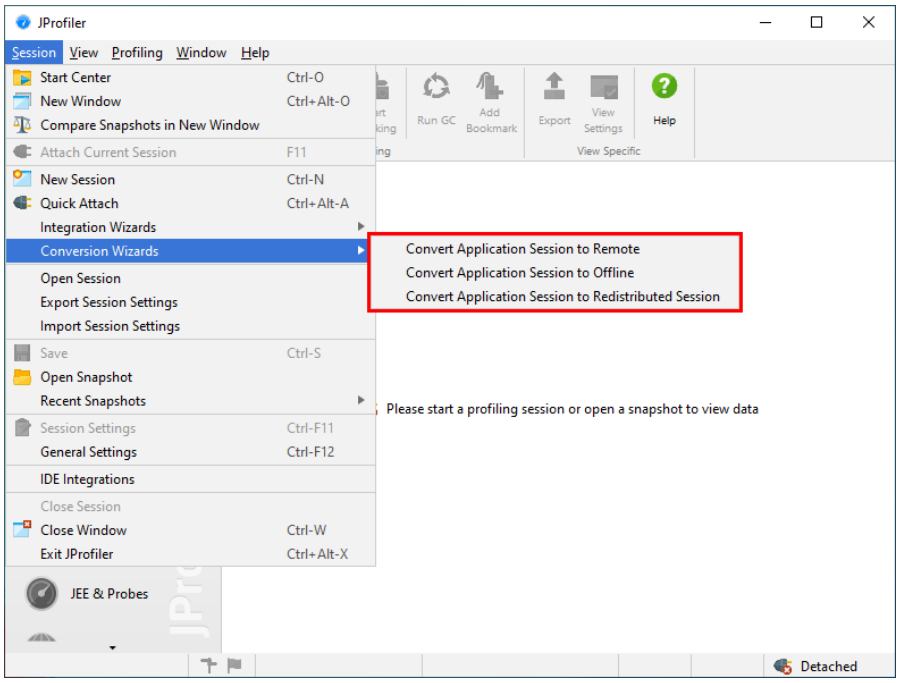
[(1)] https://www.ej-technologies.com/products/jprofiler/featuresPlatforms.html



A special launch mode is "Web Start" where you select the URL of the JNLP file and JProfiler will launch a JVM to profile it.



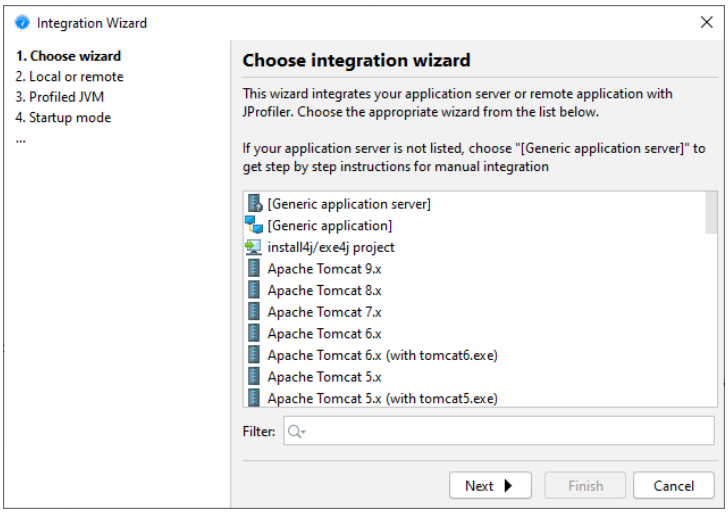
Locally launched sessions can be converted to standalone sessions with the conversion wizards by invoking *Session->Conversion Wizards* from the main menu. *Convert Application Session to Remote* simply creates a start script and inserts the `-agentpath` VM parameter into the Java call. *Convert Application Session to Offline* creates a start script for offline profiling [p. 113] which means that the config is loaded on startup and the JProfiler GUI is not required. *Convert Application Session to Redistributed Session* does the same thing, but creates a directory `jprofiler_redist` next to it that contains the profiling agent as well as the config file so you can ship it to a different machine where JProfiler is not installed.



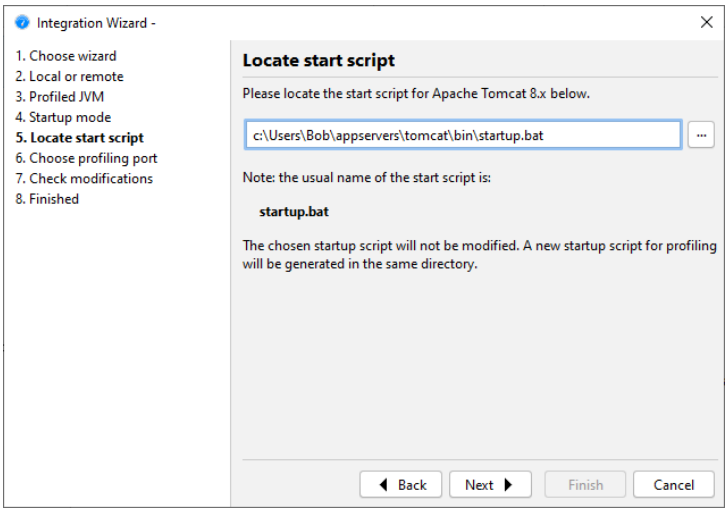
If you develop the profiled application yourself, consider using an IDE integration [p. 124] instead of a launched session. It will be more convenient and give you better source code navigation. If you do not develop the application yourself, but already have a start script, consider using the remote integration wizard. It will tell you the exact VM parameter that you have to add to the Java invocation.

Integration wizards

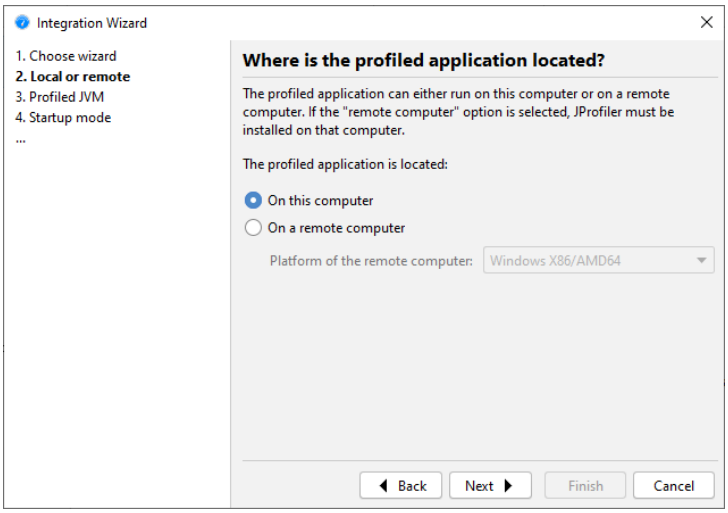
JProfiler's integration wizards handle many well-known third party containers with start scripts or config files that can be modified programatically to include additional VM parameters. For some products, start scripts can be generated where VM parameters are passed as arguments or via environment variables.



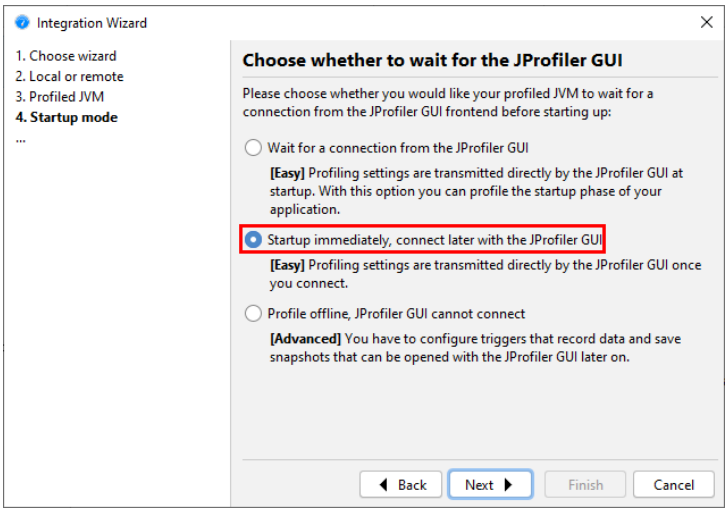
In all cases, you have to locate some specific file from the third-party product, so JProfiler has the necessary context to perform its modifications. Some generic wizards only give you instructions on what you have to do in order to enable profiling.



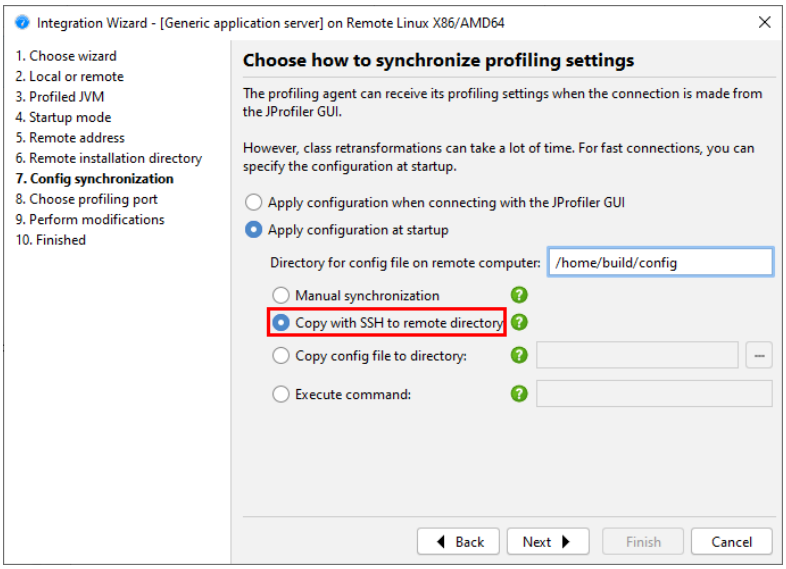
The first step in each integration wizard is the choice whether to profile on the local machine or on a remote machine. In the case of the local machine you have to provide less information, because JProfiler already knows the platform, where JProfiler is installed and where its config file is located.



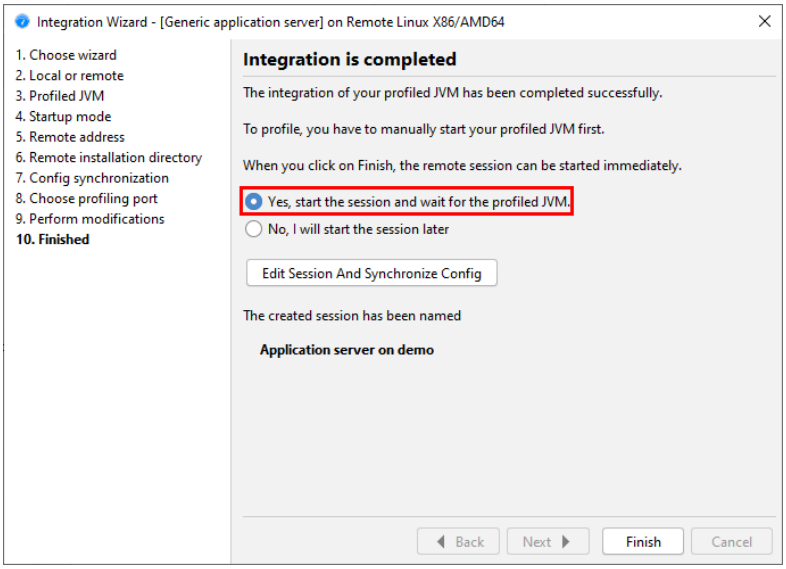
An important decision is the "startup mode" that was discussed above. By default, the profiling settings are transmitted from the JProfiler UI at startup, but you can also tell the profiling agent to let the JVM start immediately. In the latter case, the profiling settings can be applied once the JProfiler GUI connects.



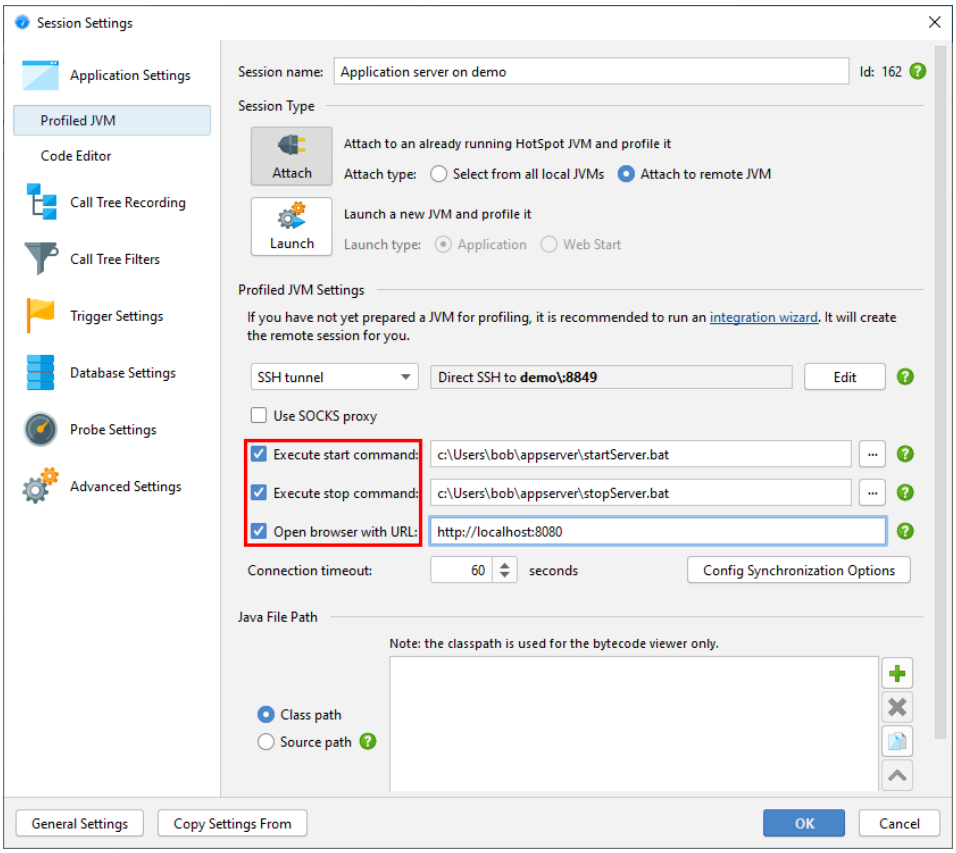
However, you can also specify a config file with the profiling settings which is much more efficient. This is done on the *Config synchronization* step. The main problem in this case is that you have to synchronize the config file with the remote side each time you edit the profiling settings locally. The most elegant way is to connect to the remote machine via SSH on the *Remote address* step, then the config file can be transferred automatically via SSH.



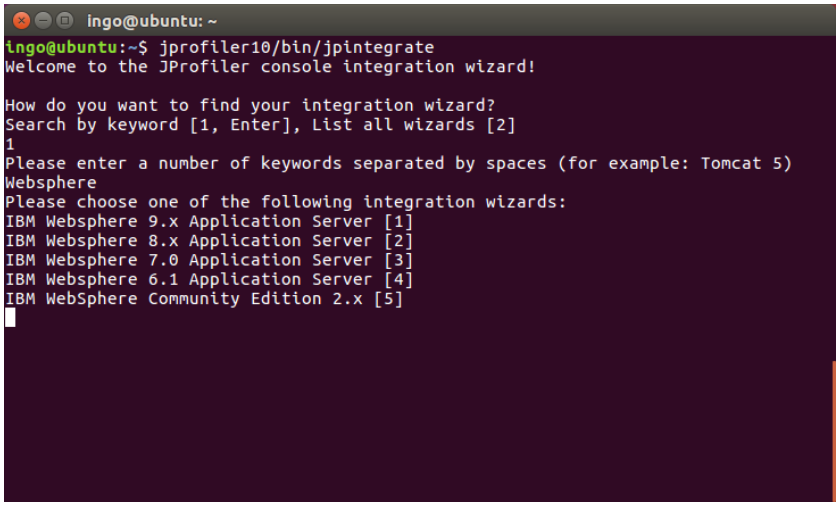
At the end of the integration wizard, a session will be created that starts profiling and - in the non-generic cases - also starts the third party product, such as an application server.



External start scripts are handled by the *Execute start script* and *Execute stop script* options on the *Application settings* tab of the session configuration dialog and URLs can be shown by selecting the *Open browser with URL* check box. This is also the place where you can change the address of the remote machine and the config synchronization options.



The integration wizards all handle cases where the profiled JVM is running on a remote machine. However, when a config file or start script has to be modified, you have to copy it to your local machine and transfer modified versions back to the remote machine. It may be more convenient to directly run the command line tool `jpintegrate` on the remote machine and let it perform its modifications in place. `jpintegrate` requires a full installation of JProfiler and has the same JRE requirements as the JProfiler GUI.

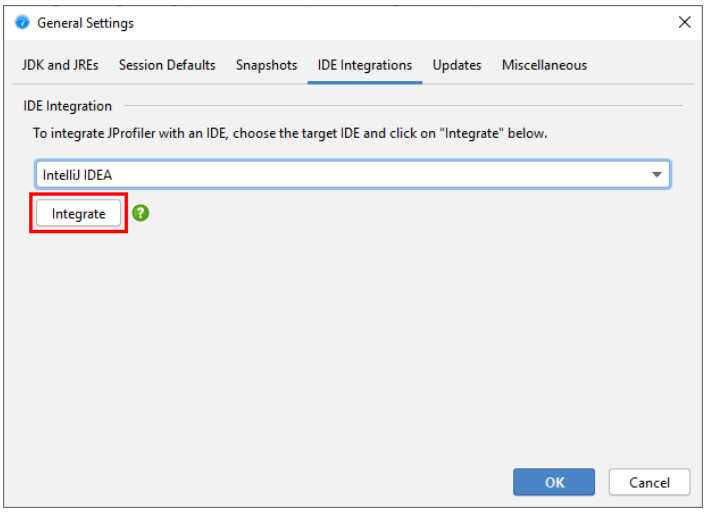


When an error occurs while starting a remote profiling session, see the trouble-shooting guide [p. 199] for a check list of steps that you can take to fix the problem.

IDE integrations

The most convenient way to profile an application is through an IDE integration. If you usually start your application from your IDE during development, the IDE already has all the required information and the JProfiler plugin can simply add the VM parameter for profiling, start JProfiler if necessary and connect the profiled JVM to a JProfiler main window.

All IDE integrations are contained in the `integrations` directory in the JProfiler installation. In principle, the archives in that directory can be installed manually with the plugin installation mechanisms in the respective IDEs. However, the preferred way to install IDE integrations is to invoke *Session->IDE integrations* from the main menu.



Profiling sessions from the IDE do not get their own session entry in JProfiler, because such a session could not be started from the JProfiler GUI. Profiling settings are persisted on a per-project or a per-run-configuration basis, depending on the settings in the IDE.

When connected to an IDE, JProfiler shows a window switcher in the tool bar that makes it easy to jump back to the associated window in the IDE. All the *Show Source* actions now show the source directly in the IDE instead of the built-in source viewer in JProfiler.

IDE integrations are discussed in detail in a later chapter [p. 124].

Attach mode

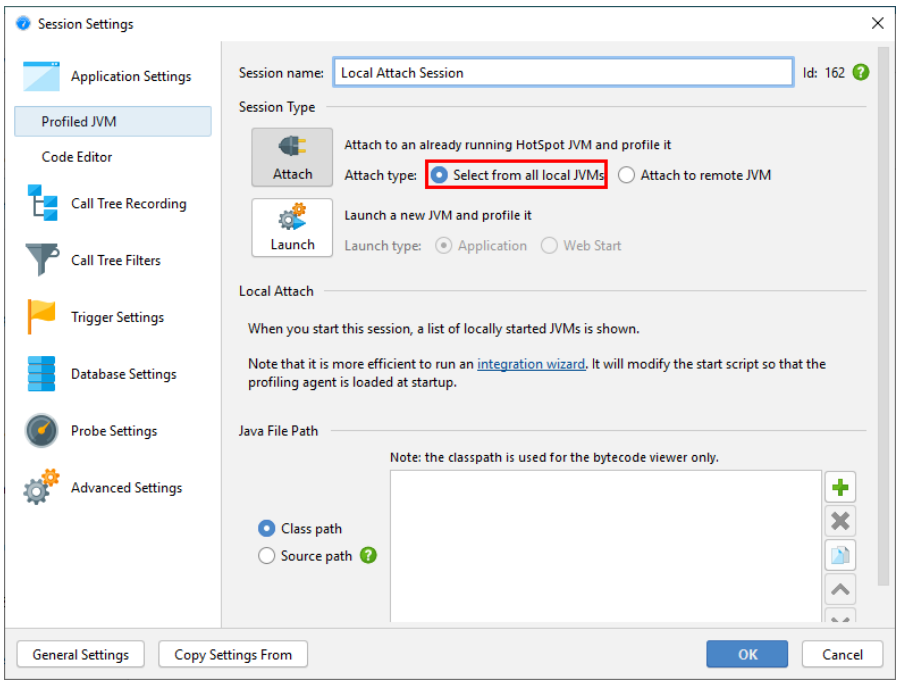
You do not necessarily have to decide beforehand that you intend to profile a JVM. With the attach functionality in JProfiler, you can select a running JVM and load the profiling agent on the fly. While attach mode it convenient, it has a couple of drawbacks that you should be aware of:

- You have to identify the JVM that you want to profile from a list of running JVMs. This can sometimes be tricky if a lot of JVMs are running on the same machine.
- There is additional overhead because potentially many classes have to be redefined to add instrumentation.
- Some features in JProfiler are not available in attach mode. This is mostly because some capabilities of the JVMTI can only be switched on when the JVM is being initialized and are not available in later phases of the JVM's lifecycle.
- Some features require instrumentation in a large fraction of all classes. Instrumenting while a class is being loaded is cheap, adding instrumentation later on when the class has already been loaded is not. Such features are disabled by default when you use attach mode.

- Attach functionality is only supported for Oracle JVMs with version 6 or higher.

The *Quick Attach* tab in JProfiler's start center lists all JVMs that can be profiled. The background color of the list entries indicates whether a profiling agent has already been loaded, whether a JProfiler GUI is currently connected or if offline profiling has been configured.

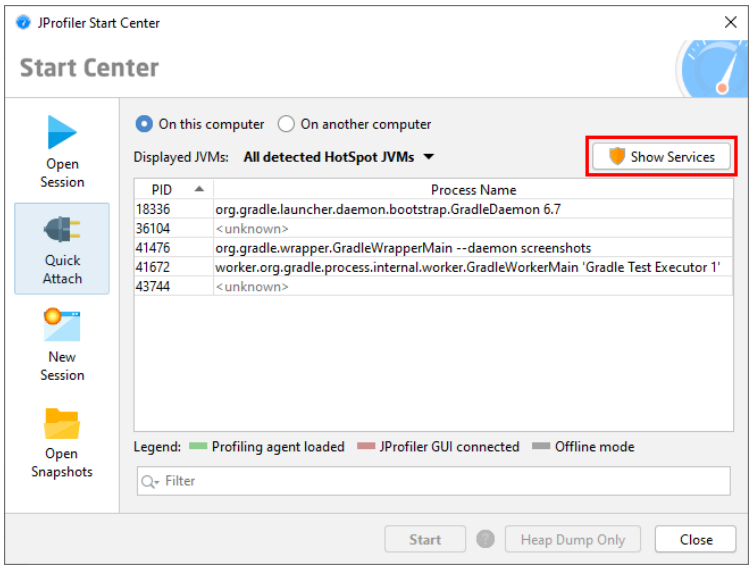
When you start a profiling session, you can configure profiling settings in the session settings dialog. When you repeatedly profile the same process, you do not want to re-enter the same configuration again and again, so a persistent session can be saved when you close a session that has been created with the quick attach feature. The next time you want to profile this process, start the saved session from the *Open Session* tab instead of the *Quick Attach* tab. You will still have to select a running JVM, but the profiling settings are the same ones that you have already configured before.



Attaching to local services

The attach API in the JVM requires that the invoking process runs as the same user as the process that you want to attach to, so the list of JVMs that are displayed by JProfiler is limited to the current user. Processes launched by different users are mostly services. The way to attach to services differs for Windows and Unix-based platforms.

On Windows, the attach dialog has a *Show Services* button that lists all locally running services. JProfiler launches bridge executables to be able to attach to those processes no matter what user they are running with.



On Unix-based platforms including macOS, you can execute the command line tool `jpenable` as a different user with `su` or `sudo`, depending on your Unix variant or Linux distribution. On macOS and Debian-based Linux distributions like Ubuntu, `sudo` is used.

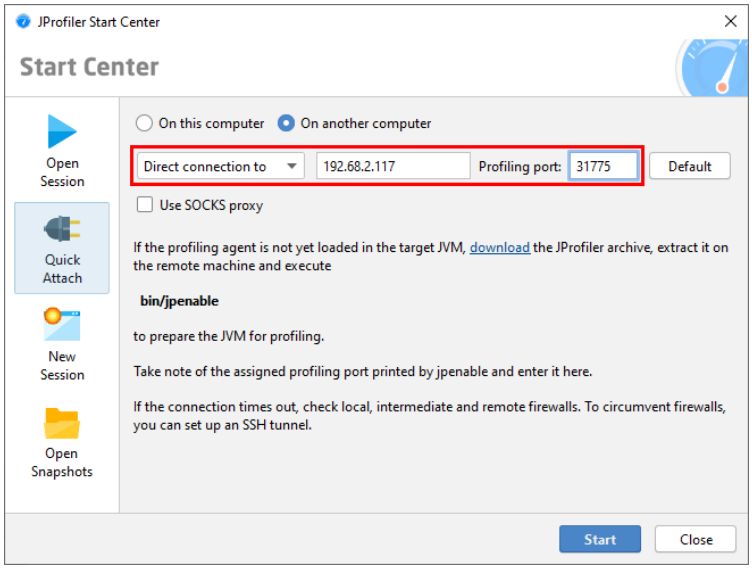
With sudo, call

```
sudo -u userName jpenable
```

with su, the required command line is

```
su userName -c jpenable
```

`jpenable` will let you select JVMs and tell you the port on which the profiling agent is listening. On the *Quick Attach* tab of the start center, you can then select the *On another computer* option and configure a direct connection to localhost and the given profiling port.

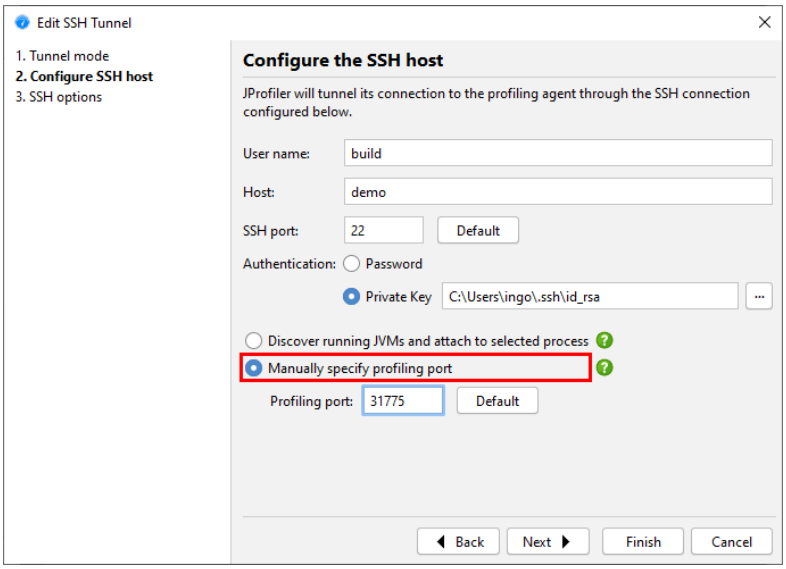


Attaching to JVMs on remote machines

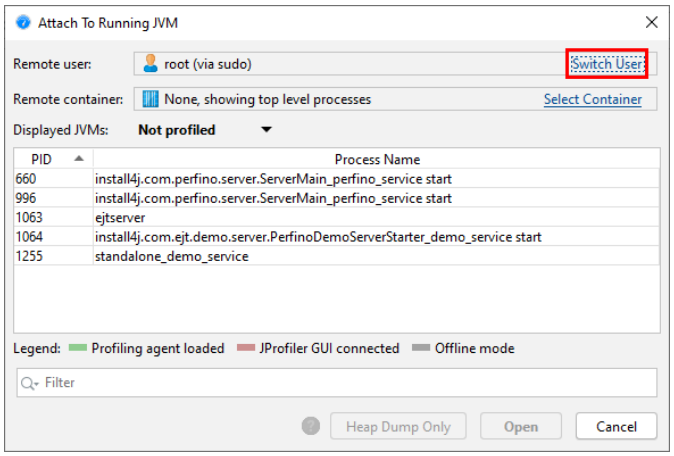
The most demanding setup for profiling is remote profiling - the JProfiler GUI runs on your local machine and the profiled JVM on another machine. For a setup where you pass the -agentpath VM parameter to the profiled JVM, you have to install JProfiler on the remote machine and set up a remote session on your local machine. With the remote attach functionality in JProfiler, no such modifications are required, you just need SSH credentials to log into the remote machine.

The SSH connection enables JProfiler to upload the agent package that was discussed in the "Installing JProfiler" [p. 7] help topic and execute the contained command line tools on the remote machine. You don't need SSH to be set up on your local machine, JProfiler ships with its own implementation. In the most straightforward setup you just define host, user name and authentication.

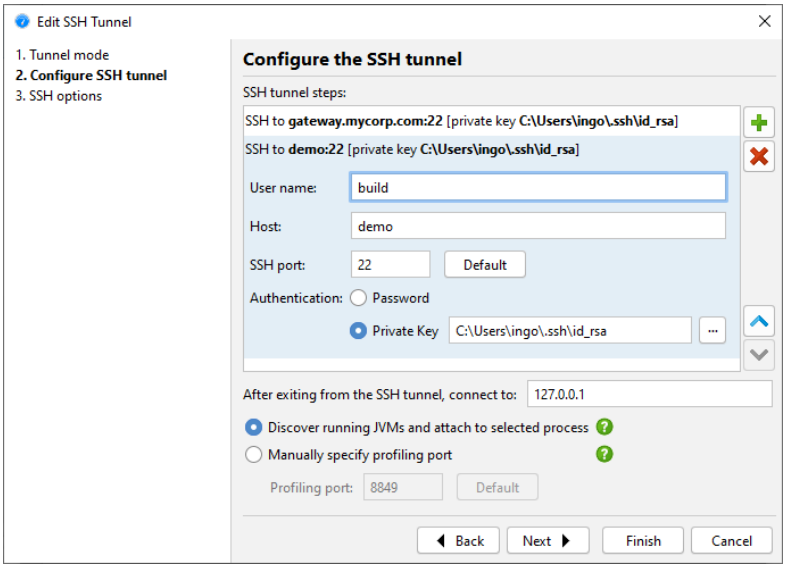
With an SSH connection, JProfiler can perform an automatic discovery of running JVMs or connect to a specific port where a profiling agent is already listening. For the latter case, you can use `jpenable` or `jpintegrate` on the remote machine as described above and prepare a special JVM for profiling. Then, the SSH remote attach can be configured to directly connect to the configured profiling port.



Automatic discovery will list all JVMs on the remote machine that have been started as the SSH login user. In most cases this will not be the user that has started the service that you would like to profile. Because users that start services usually are not allowed for SSH connections, JProfiler adds a *Switch User* hyperlink that lets you use `sudo` or `su` to switch to that user.



In complex network topologies, you sometimes cannot connect directly to the remote machine. In that case, you can tell JProfiler to connect with a multi-hop SSH tunnel in the GUI. At the end of the SSH tunnel you can make one direct network connection, usually to "127.0.0.1".

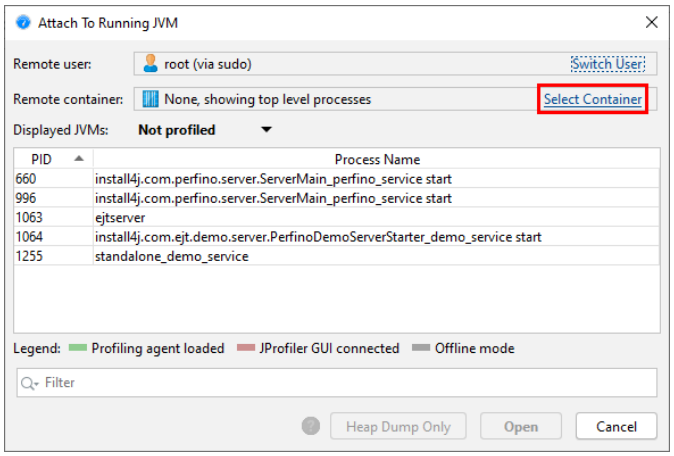


HPROF snapshots can only be taken for JVMs that were started with the SSH login user. This is because HPROF snapshots require an intermediate file that is written with the access rights of the user that has started the JVM. For security reasons, it is not possible to transfer file rights to the SSH login user for download. No such restriction exists for full profiling sessions.

Attaching to JVMs running in Docker containers

Docker containers usually do not have SSH servers installed and while you can use jpenable in a Docker container, the profiling port will not be accessible from the outside unless you have specified it in your Docker file.

In JProfiler, you can attach to a JVM running in a Docker container by using the SSH remote attach functionality and selecting a Docker container on the remote machine. If the login user is not in the docker group, you can first switch the user as described above.



With the *Switch container* hyperlink in the remote attach dialog you can choose a running Docker container and show all JVMs that are running in it. JProfiler will then use Docker commands to automatically install the profiling agent in the selected container, prepare the JVM for profiling and tunnel the profiling protocol to the outside.

Setting the display name of running JVMs

In the JVM selection table, the displayed process name is the main class of the profiled JVM together with its arguments. For launchers generated by exe4j or install4j, the executable name is displayed.

If you wish to set the displayed name yourself, for example because you have several processes with the same main class that would otherwise be undistinguishable, you can set the VM parameter `-Djprofiler.displayName=[name]`. If the name contains spaces, use single quotes: `-Djprofiler.displayName='My name with spaces'` and quote the entire VM parameter with double quotes if necessary. In addition to `-Djprofiler.displayName` JProfiler also recognizes `-Dvisualvm.display.name`.

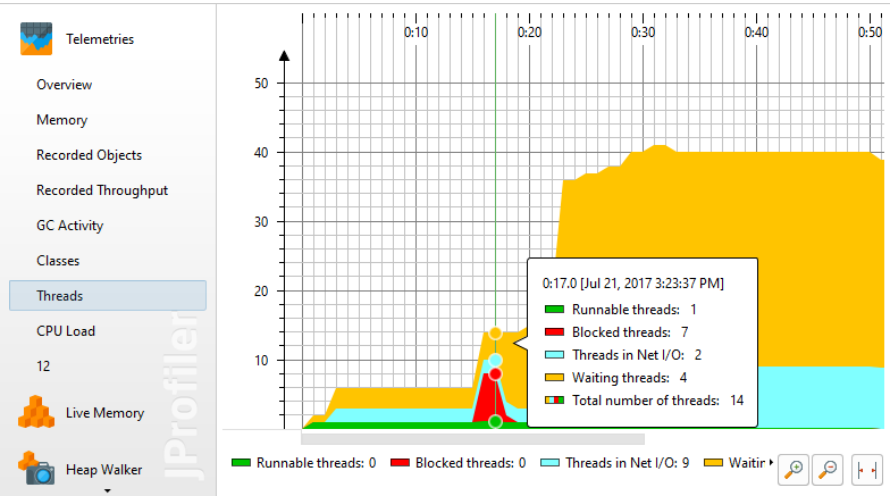
Recording Data

The main purpose of a profiler is to record runtime data from various sources that is useful for solving common problems. The principal problem with this task is that a running JVM generates such data at an enormous rate. If the profiler would always record all types of data, it would create an unacceptable overhead or quickly use up all available memory. Also, you often want to record data around a particular use case and not see any unrelated activity.

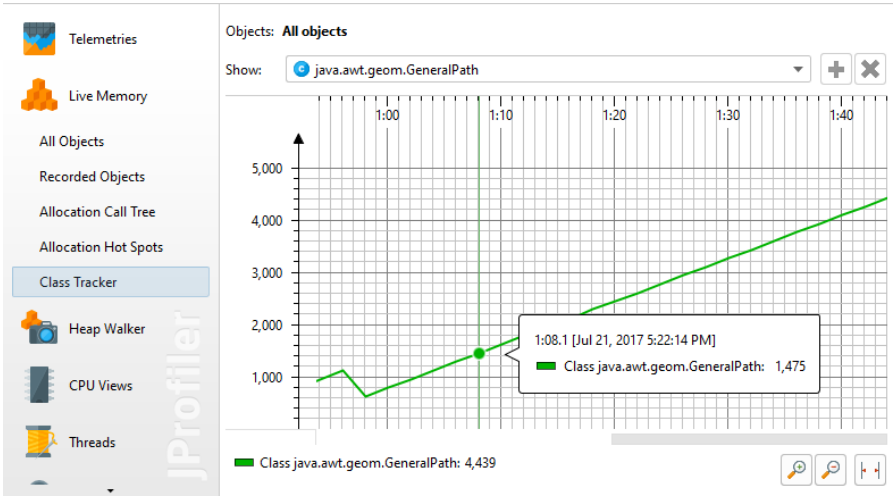
This is why JProfiler offers fine-grained mechanisms for controlling the recording of information that you are actually interested in.

Scalar values and telemetries

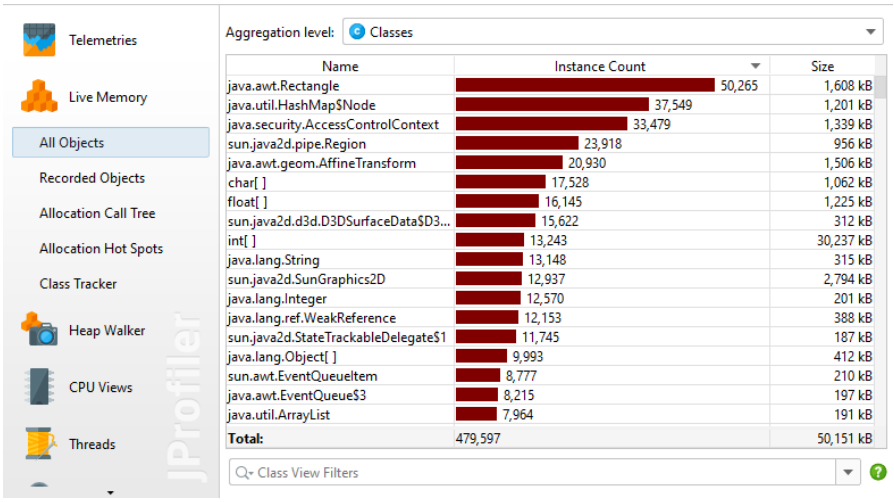
From a profiler's viewpoint, the least problematic form of data is scalar values, for example the number of active threads or the number of open JDBC connections. JProfiler can sample such values with a fixed macroscopic frequency - usually once per second - and show you the evolution over time. In JProfiler, views that show such data are called telemetries [p. 43]. Most telemetries are always recorded because the overhead of the measurement and the memory consumption are small. If data is recorded for a long time, older data points are consolidated so that memory consumption does not grow linearly with time.



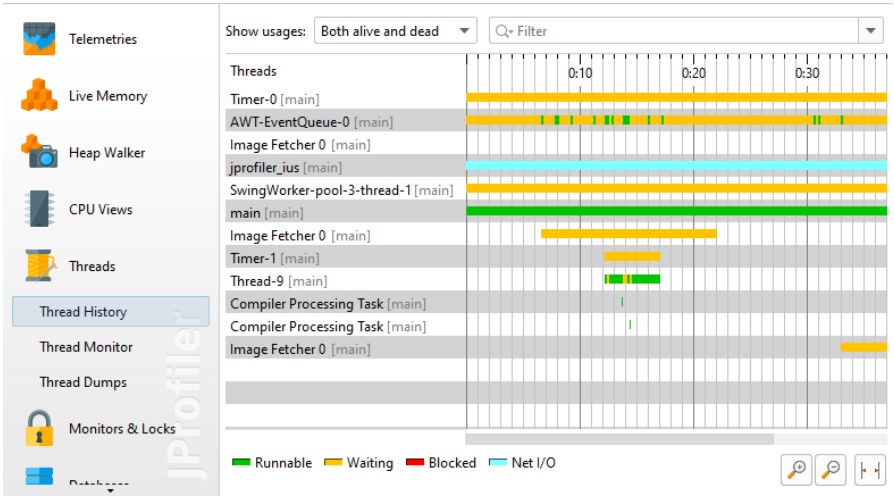
There are also parametrized telemetries, such as the number of instances for each class. The extra dimension makes a permanent chronological recording unsustainable. You can tell JProfiler to record telemetries of the instance counts of a number of selected classes, but not of each and every class.



To continue the previous example, JProfiler is able to show you the instance counts of all classes, but without the chronological information. This is the "All objects" view and it shows each class as a row in a table. The frequency for updating the view is lower than once per second and may be adjusted automatically depending on how much overhead the measurement causes. Determining the instance counts of all classes is relatively expensive and takes longer the more objects are on the heap. JProfiler limits the update frequency of the "All objects" view so that the overhead of the measurement never exceeds 10% over time in extreme cases. You can freeze the views to temporarily stop recording. Also, if the view is not active, data will not be recorded and there is no associated overhead.



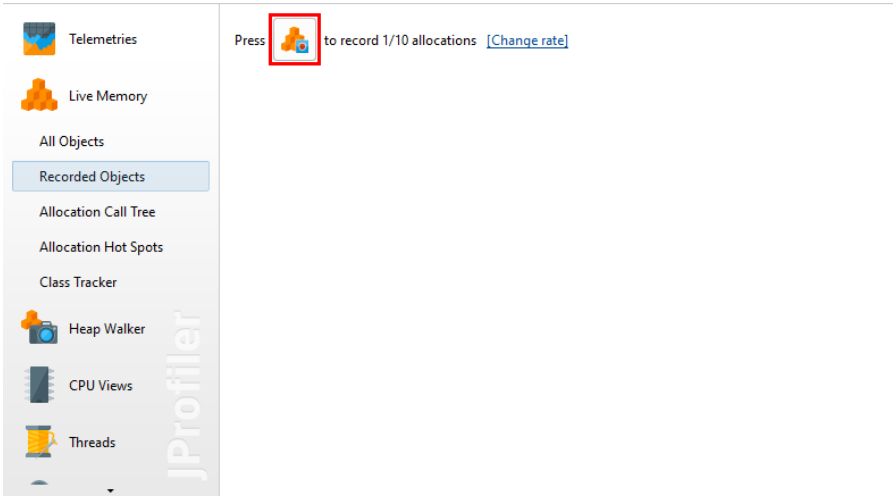
Some measurements capture enum-like values, such as the execution status a thread is currently in. This kind of measurement can be displayed as a colored time line and consumes a lot less memory than numerical telemetries. In the cases of thread statuses, the "Thread history" view shows the time lines for all threads in the JVM. Just like for the telemetries with numeric values, older values are consolidated and made more coarse-grained to reduce memory consumption.



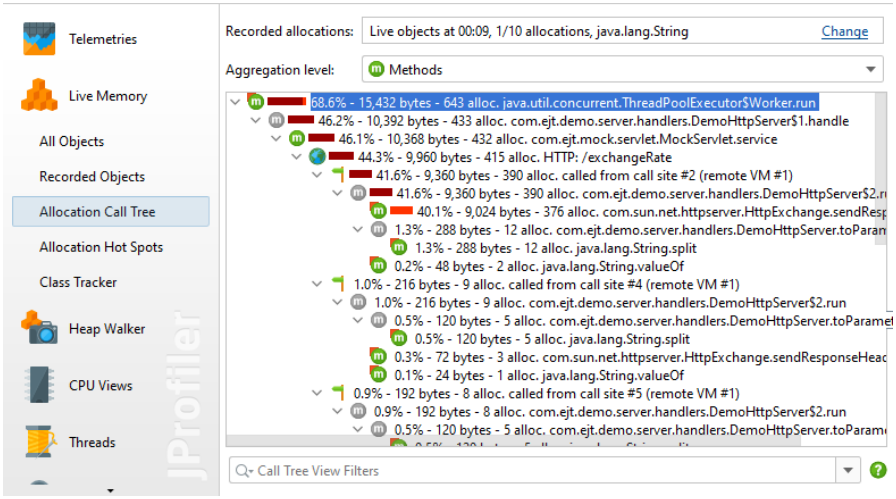
Allocation recording

If you are interested in instance counts that have been allocated during a certain time interval, JProfiler has to track all allocations. Contrary to the "All objects" view where JProfiler can iterate over all objects in the heap to get information on demand, tracking single allocations requires that additional code has to be executed for each object allocation. That makes it a very expensive measurement that can significantly change the runtime characteristics of the profiled application, such as the performance hot spots, especially if you allocate many objects. This is why allocation recording has to be started and stopped explicitly.

Views that have an associated recording initially show an empty page with a recording button. The same recording button is also found in the toolbar.



Allocation recording not only records the number of allocated instances, it also records the allocation stack traces. Keeping stack traces for each allocated recording in memory would create excessive overhead, so JProfiler cumulates recorded stack traces into a tree. This also has the advantage that you can interpret the data much more easily. However, the chronological aspect is lost and there is no way to extract certain time ranges from the data.

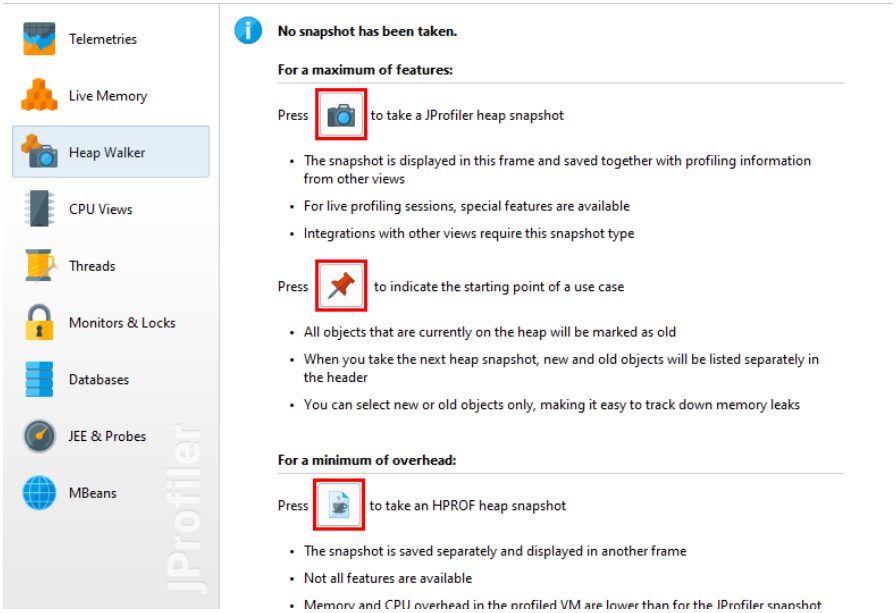


Memory analysis

Allocation recording can only measure where objects are allocated and has no information on the references between objects. Any memory analysis that requires references, such as solving a memory leak, is done in the heap walker. The heap walker takes a snapshot of the entire heap and analyzes it. This is an invasive operation that pauses the JVM - potentially for a long time - and requires a large amount of memory.

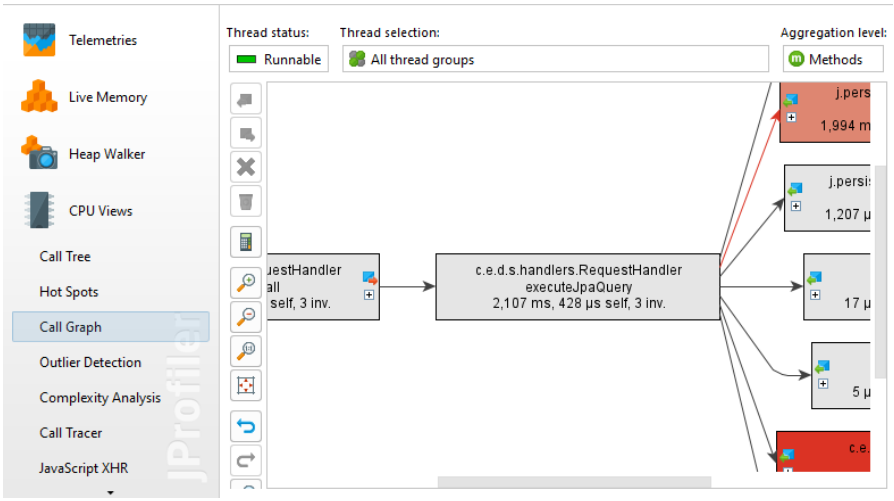
A more lightweight operation is marking all object on the heap before you start a use case, so that you can find all newly allocated objects when you take a heap snapshot later on.

The JVM has a special trigger for dumping the entire heap to a file that is named after the old HPROF profiling agent. This is not related to the profiling interface and does not operate under its constraints. For this reason, the HPROF heap dump is faster and uses less resources. The downside is that you will not have a live connection to the JVM when viewing the heap snapshot in the heap walker and that some features are not available.

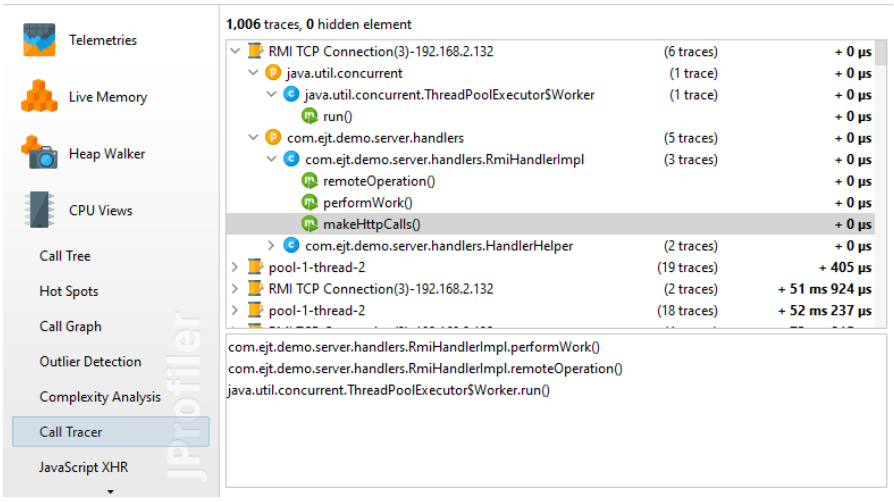


Method call recording

Measuring how long method calls take is an optional recording, just like allocation recording. Method calls are cumulated into a tree and there are various views that show the recorded data from different perspectives, such as a call graph. The recording for this type of data is called "CPU recording" in JProfiler.

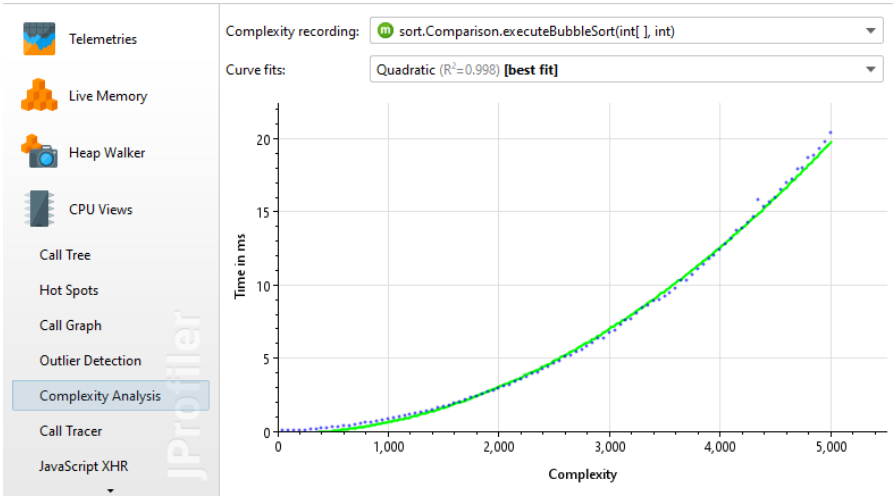


Under particular circumstances it may be useful to see the chronological sequence of method calls, especially if multiple threads are involved. For these special cases, JProfiler offers the "Call tracer" view. That view has a separate recording type that is not tied to the more general CPU recording. Note that the call tracer produces too much data to be useful for solving performance problems, it is only intended for a specialized form of debugging.



The call tracer depends on CPU recording and automatically switches it on if necessary.

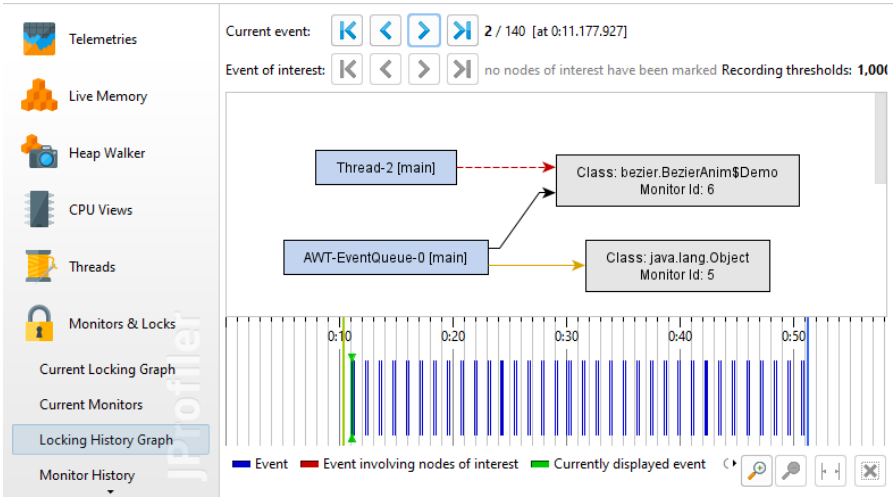
Another specialized view that has its own recording is the "Complexity analysis". It only measures the execution times of selected methods and does not require CPU recording to be enabled. Its additional data axis is a numeric value for the algorithmic complexity of a method call that you can calculate with a script. In this way, you can measure how the execution time of a method depends on its parameters.



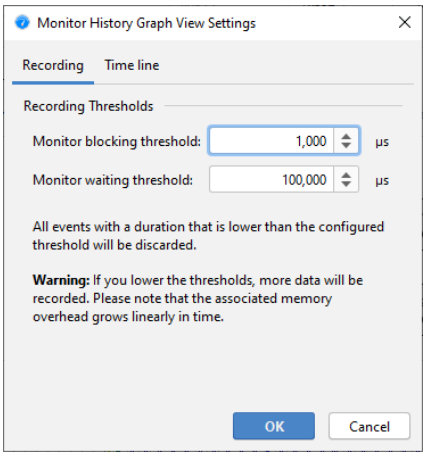
Monitor recording

To analyze why threads are waiting or blocking, the corresponding events have to be recorded. The rate of such events varies greatly. For a multi-threaded program where threads frequently coordinate tasks or share common resources, there can be an enormous amount of such events. This is why such chronological data is not recorded by default.

When you switch on monitor recording, the "Locking history graph" and the "Monitor history" view will start to show data.

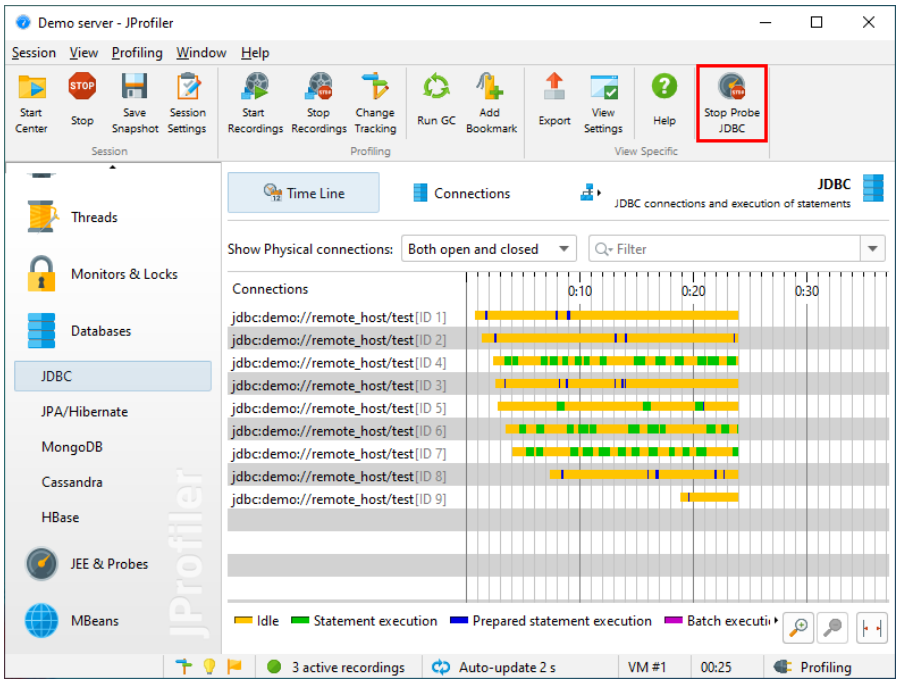


To eliminate noise and reduce memory consumption, very short events are not recorded. The view settings give you the possibility to adjust these thresholds.

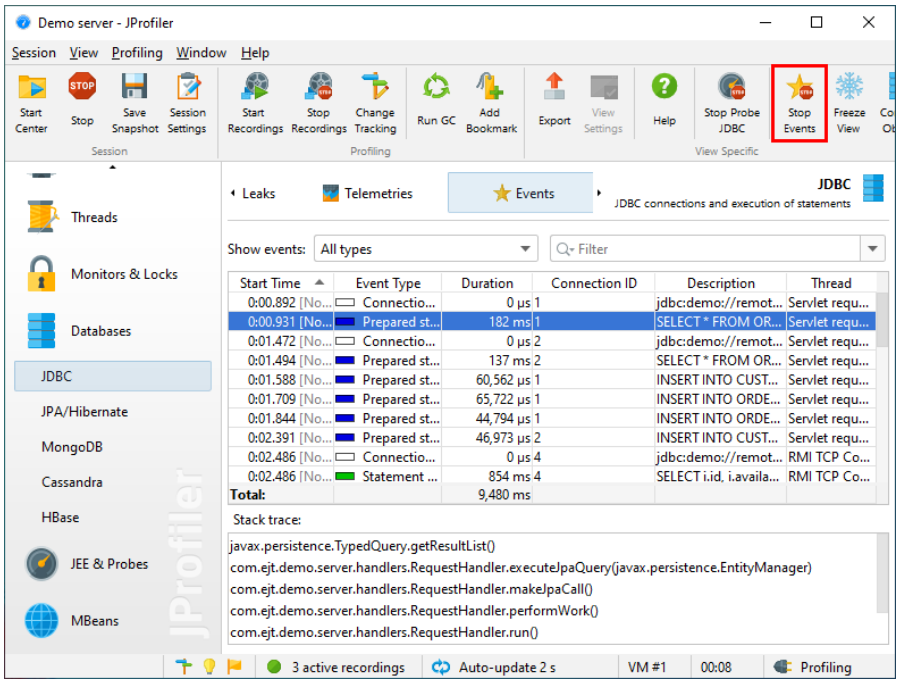


Probe recording

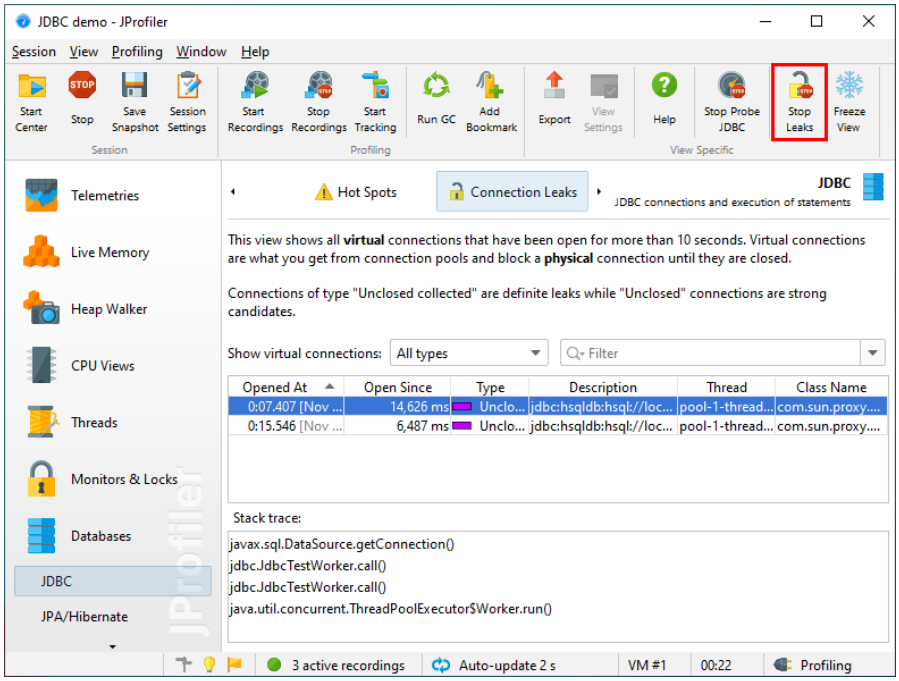
Probes show higher-level subsystems in the JVM, such as JDBC calls or file operations. By default, no probes are recorded and you can toggle recording separately for each probe. Some probes will add very little or no overhead and some will create a considerable amount of data, depending on what your application is doing and how the probes are configured.



Just like allocation recording and method call recording, probe data is cumulated and chronological information is discarded except for time lines and telemetries. However, most probes also have an "Events" view that allows you to inspect the single events. This adds a potentially large overhead and has a separate recording action. The status of that the recording action is persistent, so that when you toggle probe recording, the associated event recording is toggled as well if you have switched it on previously.



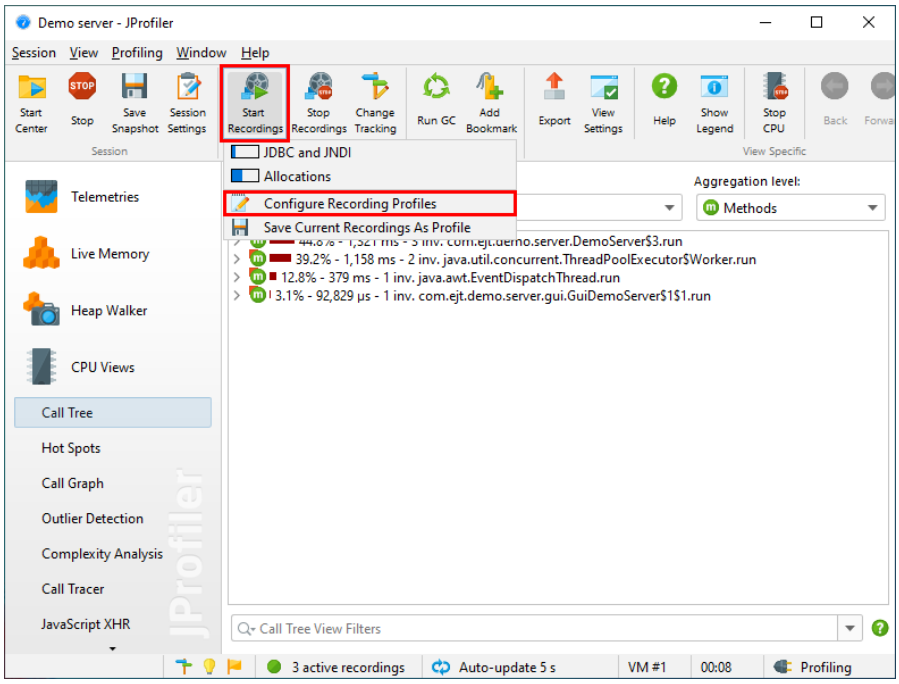
The JDBC probe has a third recording action for recording JDBC connection leaks. The associated overhead with looking for connection leaks is only incurred if you are actually trying to investigate such a problem. Just like the event recording action, the selection state of the leak recording action is persistent.



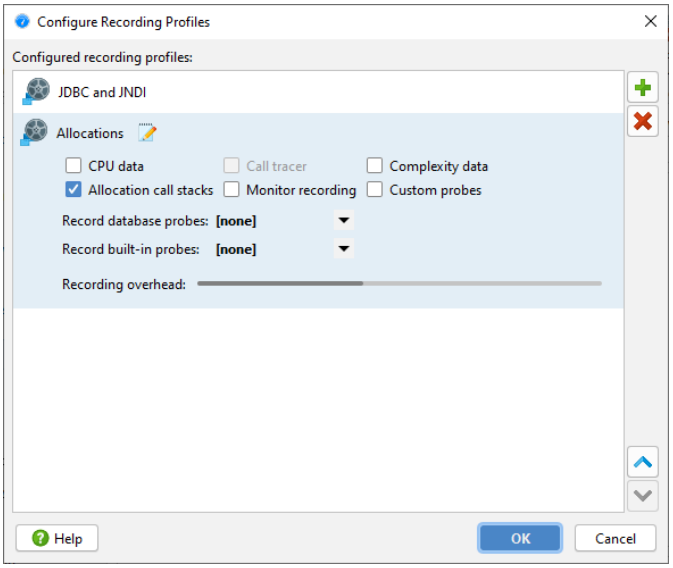
Recording profiles

In many situations, you want to start or stop various recordings together with a single click. It would be impractical to visit all the corresponding views and toggle the recording buttons one

by one. This is why JProfiler has recording profiles. Recording profiles can be created by clicking on the *Start Recordings* button in the tool bar.

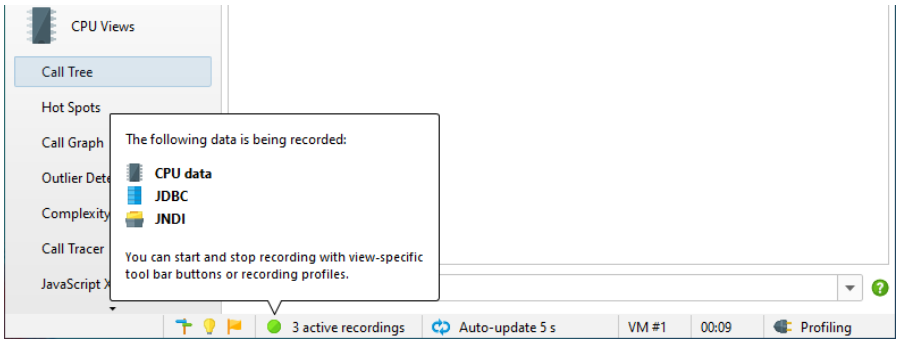


Recording profiles define one particular combination of recordings that can be activated atomically. JProfiler tries to give you a rough impression on the overhead that you create by the selected recordings and tries to discourage problematic combinations. In particular, allocation recording and CPU recording do not go well together because the timings of CPU data will be distorted significantly by allocation recording.

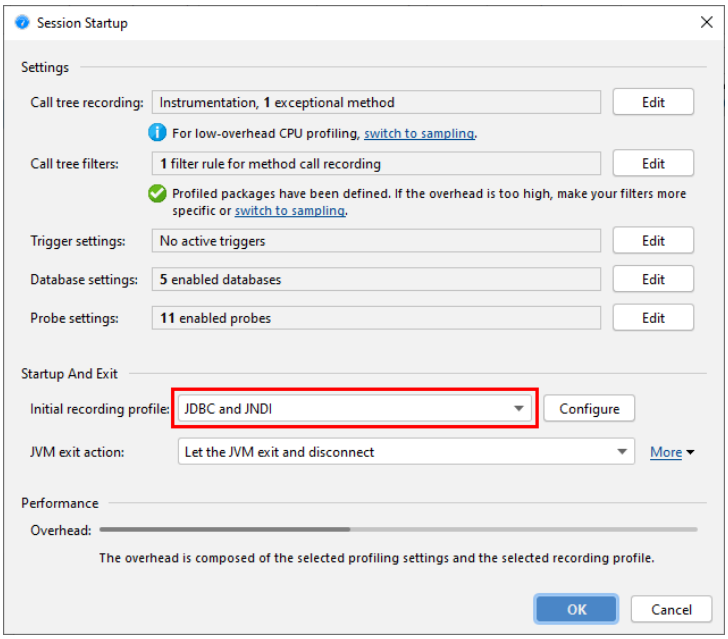


You can activate recording profiles at any time while a session is running. Recording profiles are not additive, they stop all recordings that are not included in the recording profile. With the *Stop Recordings* button you stop all recordings no matter how they have been activated. To check

what recordings are currently active, hover the mouse over the recordings label in the status bar.



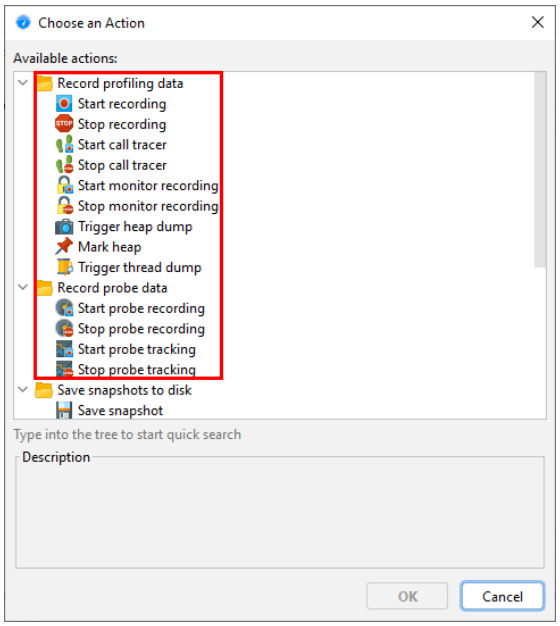
A recording profile can also be activated directly when you start profiling. The "Session startup" dialog has an *Initial recording profile* drop-down. By default, no recording profile is selected, but if you need data from the startup phase of the JVM, this is the place to configure the required recordings.



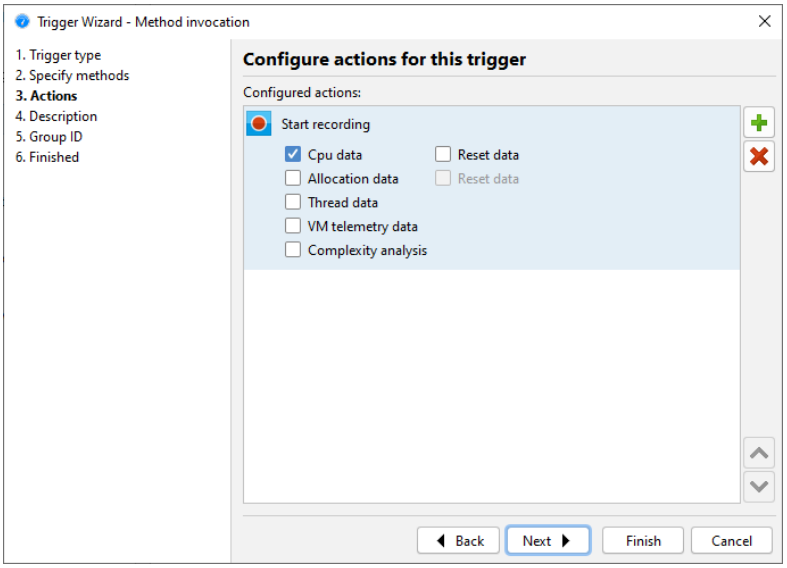
Recording with triggers

Sometimes you want to start a recording when a particular condition occurs. JProfiler has a system for defining triggers [p. 113] that execute a list of actions. The available trigger actions also include changes to the active recordings.

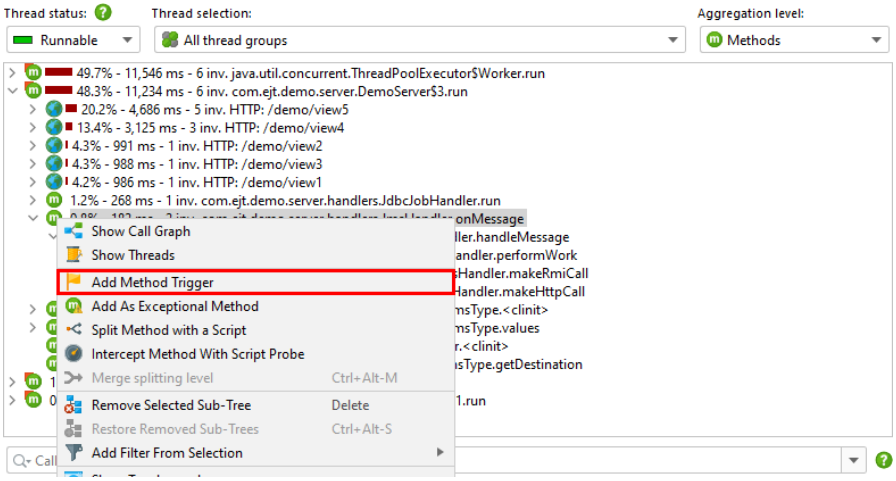
For example, you could want to start a recording only when a particular method is executed. In that case, you would go to the session settings dialog, activate the *Trigger Settings* tab and define a method trigger for that method. For the action configuration, you have a number of different recording actions available.



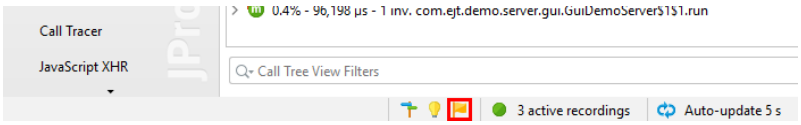
The "Start recording" action controls those recordings without any parameters. Usually, when you stop and re-start a recording, all previously recorded data is cleared. For the "CPU data" and "Allocation data" recordings, you also have the option to keep the previous data and continue cumulating across multiple intervals.



Method triggers can be added conveniently in the call tree by using the "Add method trigger" action in the context menu. If you already have a method trigger in the same session, you can choose to add a method interception to an existing trigger.



By default, triggers are active when the JVM is started for profiling. There are two ways to disable triggers at startup: You can disable them individually in the trigger configuration or deselect the *Enable triggers on startup* check box in the session startup dialog. During a live session, you can enable or disable all triggers by choosing *Profiling->(Enable|Disable) Triggers* from the menu or clicking on the trigger recording state icon in the status bar.



Sometimes, you need to toggle trigger activation for groups of triggers at the same time. This is possible by assigning the same group ID to the triggers of interest and invoking *Profiling->Enable Triggers Groups* from the menu.

Recording with jpcontroller

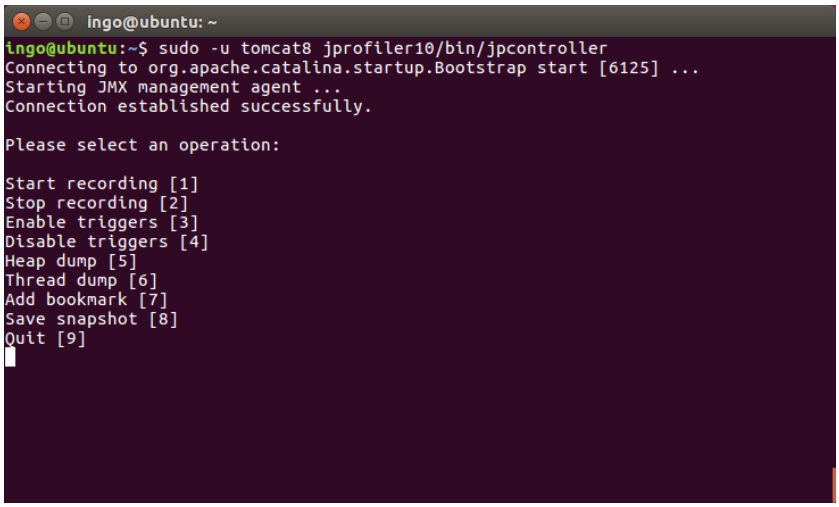
JProfiler has a command line executable for controlling the recordings in any JVM that is already being profiled. jpcontroller requires that the JProfiler MBean is published, otherwise it will not be able to connect to the profiled JVM. This is only the case if the profiling agent has already received profiling settings. Without profiling settings, the agent would not know what to record exactly.

One of the following conditions has to apply:

- You have already connected to the JVM with a JProfiler GUI
- The profiled JVM was started with an `-agentpath` VM parameter that included both the `nowait` and the `config` parameters. In the integration wizards, this corresponds to the *Startup immediately* mode and the *Apply configuration at startup* option in the *Config synchronization* step.
- The JVM was prepared for profiling with the `jpenable` executable and the `-offline` parameter was specified. See the output of `jpenable -help` for more information.

Specifically, `jpcontroller` will not work if the profiled JVM was started only with the `nowait` flag. In the integration wizards, the *Apply configuration when connecting with the JProfiler GUI* option on the *Config synchronization* step would configure such a parameter.

jpcontroller presents you with a looping multi-level menu for all recordings and their parameters. You can also save snapshots with it.



Programmatic way to start recordings

Yet another way to start recording is through the API. In the profiled VM, you can call the `com.jprofiler.api.controller.Controller` class to start and stop recordings programmatically. See the chapter on offline profiling [p. 113] for more information and for how to get the artifact that includes the controller class.

If you want to control recordings in a different JVM, you can access the same MBean in the profiled JVM that is also used by `jpcontroller`. Setting up programmatic usage of the MBean is somewhat involved and requires quite a bit of ceremony, so JProfiler ships with an example that you can reuse. Check the file `api/samples/mbean/src/MBeanProgrammaticAccessExample.java`. It records CPU data for 5 seconds in another profiled JVM and saves a snapshot to disk.

Snapshots

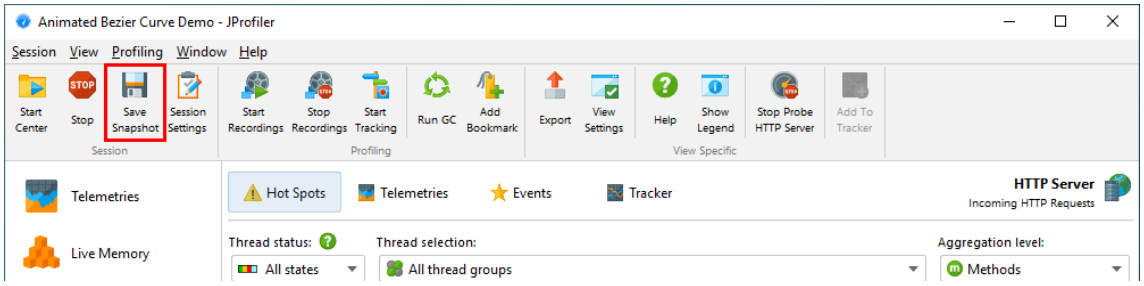
Until now, we have only looked at live sessions where the JProfiler GUI obtains the data from the profiling agent that is running inside the profiled JVM. JProfiler also supports snapshots where all profiling data is written to a file. This can be of advantage in several scenarios:

- You record profiling data automatically, for example as part of a test so that connecting with a JProfiler GUI is not possible.
- You want to compare profiling data from different profiling sessions or look at older recordings.
- You want to share profiling data with somebody else.

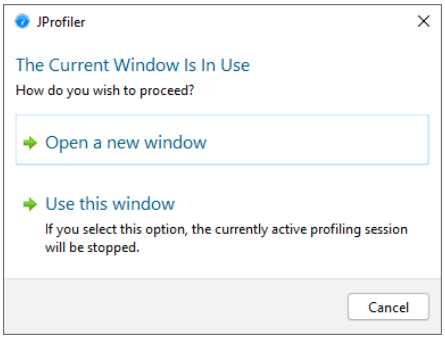
Snapshots include data from all recordings, including heap snapshots. To save disk space, snapshots are compressed, except for heap walker data which has to remain uncompressed to allow for direct memory mapping.

Saving and opening snapshots in the JProfiler GUI

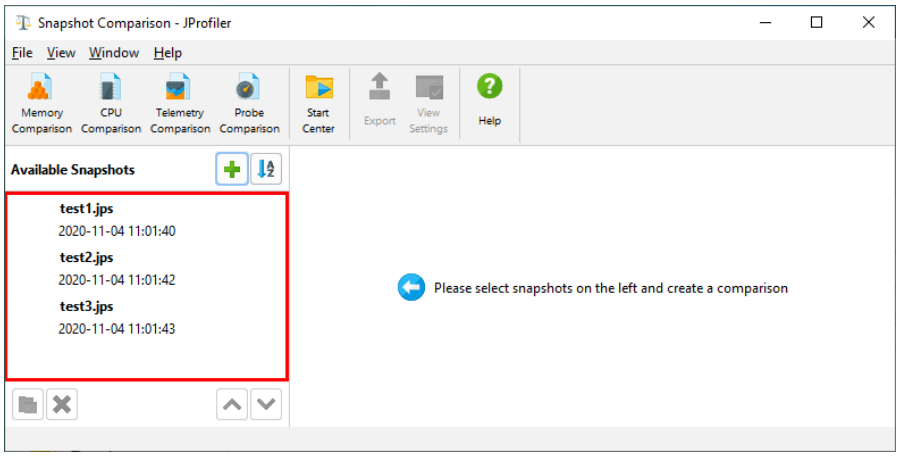
When you are profiling a live session, you can create snapshots with the *Save Snapshot* tool bar button. JProfiler pulls all profiling data from the remote agent and saves it to a local file with a ".jps" extension. You can save multiple such snapshots during the course of a live session. They are not opened automatically and you can continue to profile.



Saved snapshots are added automatically to the *File->Recent Snapshots* menu, so you can conveniently open a snapshot that you have just saved. When opening a snapshot while the live session is still running, you have a choice of terminating the live session or opening another JProfiler window.

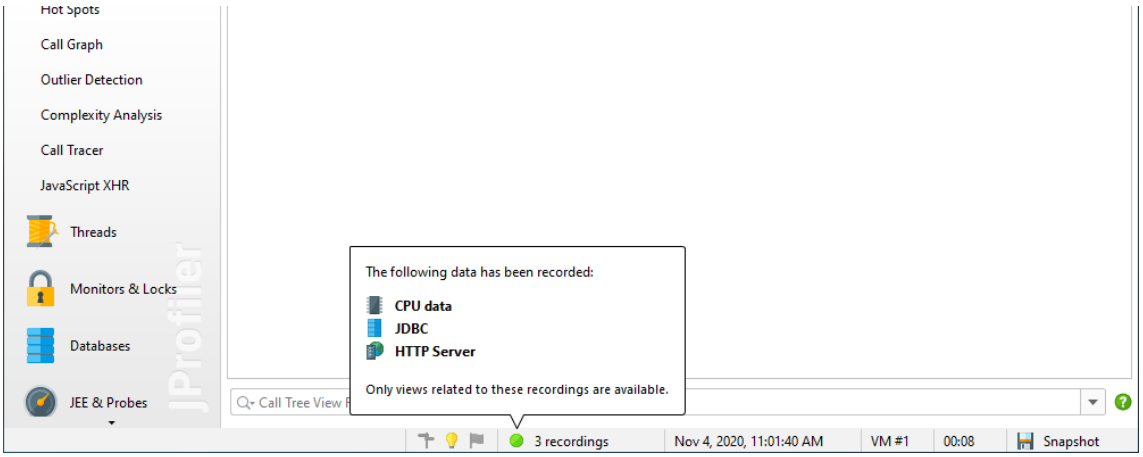


When you use the snapshot comparison feature in JProfiler, the list of snapshots is populated with all the the snapshots that you have saved for the current live session. This makes it easy to compare different use cases.



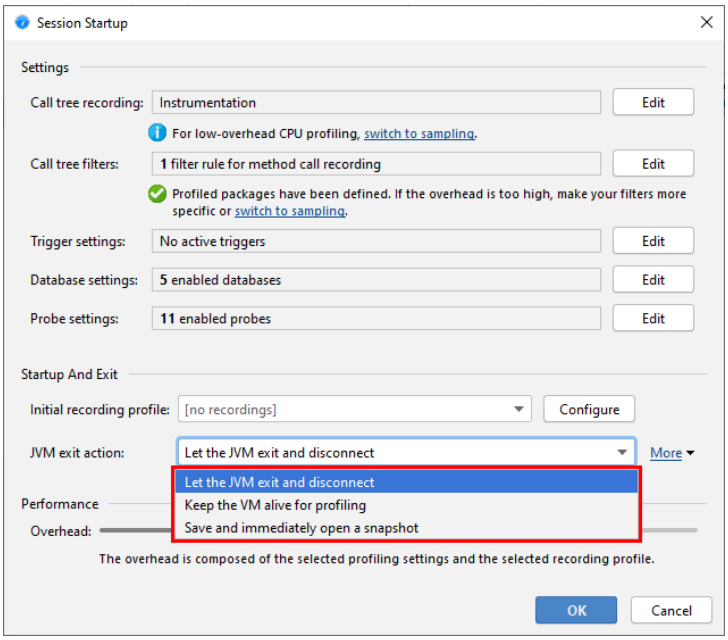
In general, you can open snapshots by invoking *Session->Open Snapshot* from the main menu or by double-clicking the snapshot file in the file manager. JProfiler's IDE integrations also support opening JProfiler snapshots through the generic *Open File* actions in the IDEs themselves. In that case, you get source code navigation into the IDE instead of the built-in source code viewer.

When you open a snapshot, all the recording actions are disabled and only views with recorded data are available. To discover what kind of data has been recorded, hover the mouse over the recording label in the status bar.



Profiling short-lived programs

For a live session, all profiling data resides in the process of the profiled JVM. So when the profiled JVM is terminated, the profiling session in JProfiler is closed as well. To continue profiling when a JVM exits, you have two options, both of which can be activated in the session startup dialog.



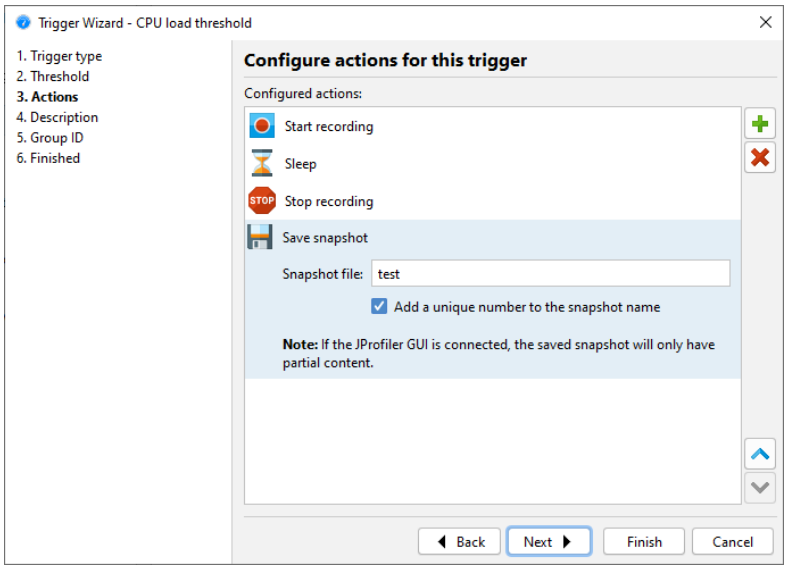
- You can prevent the JVM from actually exiting and keep it artificially alive as long as the JProfiler GUI is connected. This may be undesirable when you are profiling a test case from the IDE and want to see the status and total time in the test console of the IDE.
- You can ask JProfiler to save a snapshot when the JVM terminates and switch to it immediately. The snapshot is temporary and will be discarded when you close the session unless you use the *Save Snapshot* action first.

Saving snapshots with triggers

The final result of an automated profiling session is always a snapshot or a series of snapshots. In triggers, you can add a "Save a snapshot" action that saves the snapshot on the machine where the profiled JVM is running. When the trigger runs during a live session, the snapshot is also saved on the remote machine and may not include parts of the data that have already been transmitted to the JProfiler GUI.

There are two basic strategies for saving snapshots with triggers:

- For test cases, start recording in the "JVM startup" trigger and add a "JVM exit" trigger to save the snapshot when the JVM is terminated.
- For exceptional conditions like the "CPU load threshold" trigger or for periodic profiling with a "Timer trigger", save the snapshot after recording some data with a "Sleep" action in between.

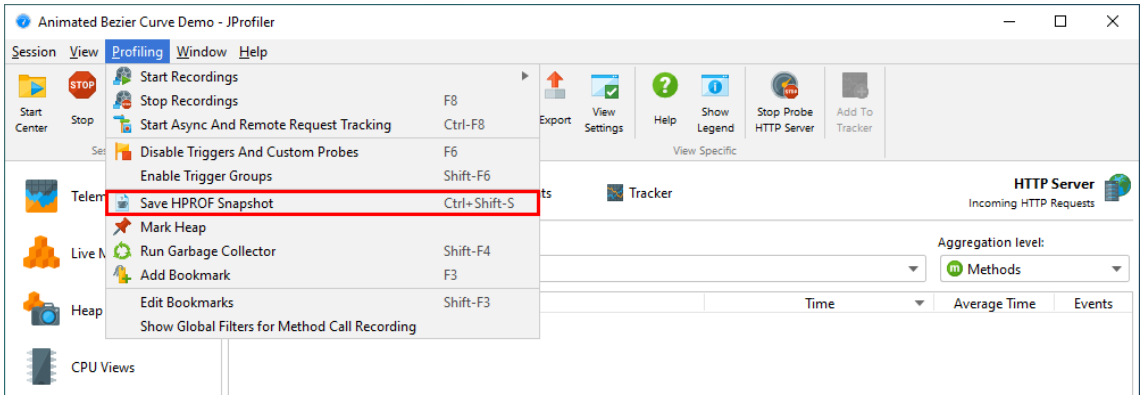


HPROF heap snapshots

In situations where taking a heap snapshot produces too much overhead or consumes too much memory, you can use the HPROF heap snapshots that the JVM offers as a built-in feature. Because the profiling agent is not required for this operation, this is interesting for analyzing memory problems in JVMs that are running in production.

With JProfiler, there are three ways to obtain such snapshots:

- For live sessions, the JProfiler GUI offers an action in the main menu to trigger an HPROF heap dump.

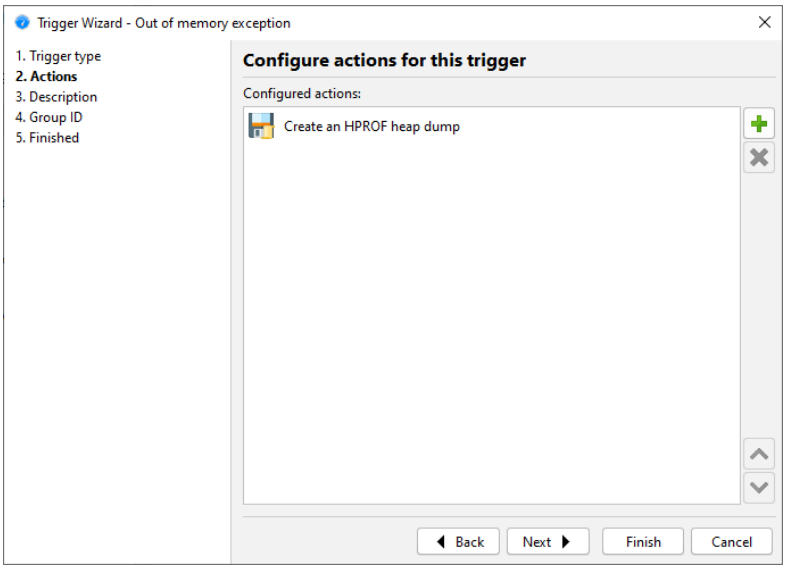


- JProfiler has a special "Out of memory exception" trigger to save an HPROF snapshot when an `OutOfMemoryError` is thrown. This corresponds to the VM parameter [1]

  ```
  -XX:+HeapDumpOnOutOfMemoryError
  ```

  that is supported by HotSpot JVMs.

[1] http://docs.oracle.com/javase/9/troubleshoot/command-line-options1.htm#JSTGD592



- The jmap executable in the JDK[2] can be used to extract an HPROF heap dump from a running JVM.

  JProfiler includes the command line tool `jpdump` that is more versatile than jmap. It lets you select a process, can connect to processes running as a service on Windows, has no problems with mixed 32-bit/64-bit JVMs and auto-numbers HPROF snapshot files. Execute it with the `-help` option for more information.

JDK Flight Recorder snapshots

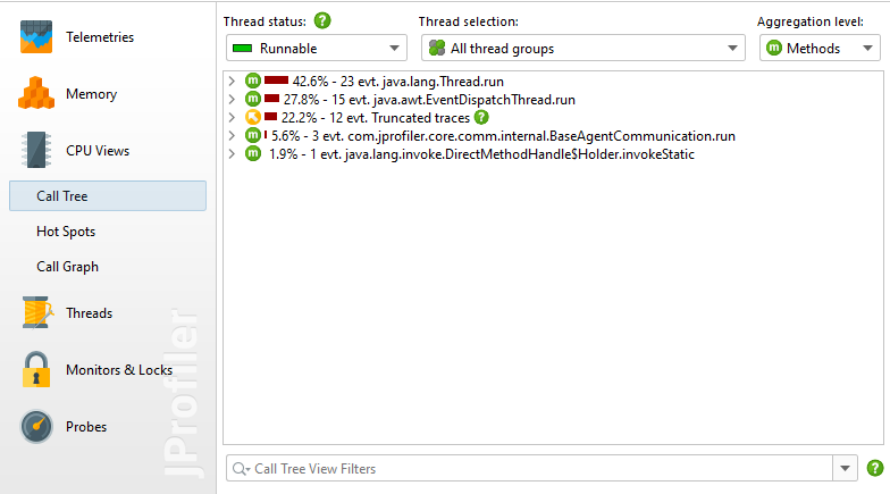
The JDK includes the JDK Flight Recorder (JFR) mechanism to capture events regarding method execution, object allocations, and other important subsystems of the JVM and save them to snapshots. Production environments may opt for a continuous use of this technology to minimize overhead and provide data for troubleshooting. JFR snapshots can be recorded and saved with the jcmd executable in the JDK[3] using the commands that start with "JFR.".

With JProfiler, you can open JDK Flight Recorder snapshots to analyze the recorded data. When opening JFR snapshots, only a fraction of JProfiler's views will be shown in the view selector corresponding to the types of data that are available from JFR recording.

First, the telemetry section is populated with a subset of JProfiler's telemetries, other telemetries are shown in the corresponding probe views. The "Memory" telemetry is only populated if heap statistics are recorded. There are some differences to the default JProfiler telemetries, for example, "Recorded throughput" shows sizes and not object counts.

Call tree and hot spot views are available, with the limitation that the "Runnable" thread state does not measure times in JFR, but shows sample counts. The other thread states like "Waiting" or "Blocking" still measure times. Because of this discrepancy, the "All thread states" mode is not available in the thread status selector. Also, the non-Runnable thread states are calculated from events which have a configurable minimum duration threshold that is shown in the thread status selector. The actual total time of these thread states may be significantly larger.

[2] https://docs.oracle.com/en/java/javase/11/tools/jmap.html#GUID-D2340719-82BA-4077-B0F3-2803269B7F41
[3] https://docs.oracle.com/en/java/javase/11/tools/jcmd.html#GUID-59153599-875E-447D-8D98-0078A5778F05



Because JFR works with fixed buffers, long call stacks are truncated. Truncated traces are not suitable for building an understandable call tree, so these traces are shown below a specially marked node. With the

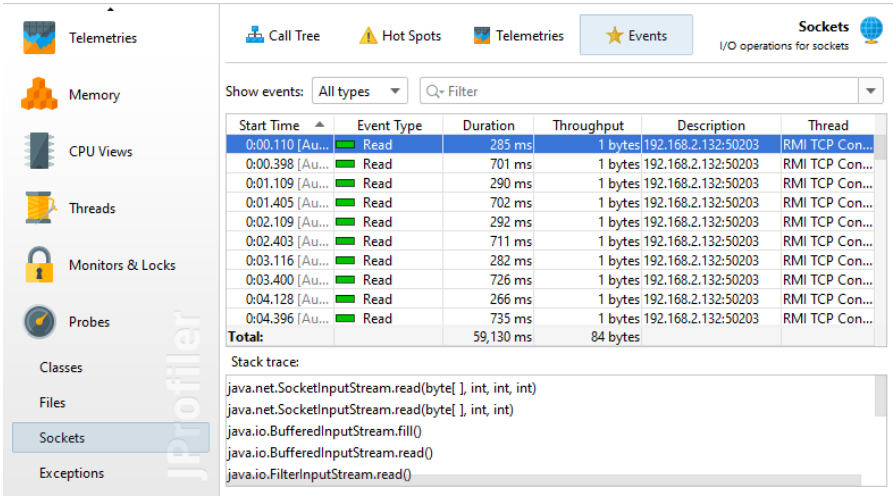
```
-XX:FlightRecorderOptions=stackdepth=>nnnn<
```

VM parameter, you can increase the size of the corresponding buffer in JFR and get rid of truncated traces for your application.

For memory recording, the recorded objects, allocation call tree and allocation hot spots views show the JFR allocation profiling data. The most important difference is that the recording uses allocation sampling, but the reported sizes are estimated total sizes. Only recorded objects are shown, statistics for all objects are not available. If heap statistics are recorded, the "Live objects" view will show class statistics for objects that are still on the heap after a full garbage collection. The "Difference" column shows how the numbers have changed between the first and the last heap statistics. If these times do not coincide with the start and end points of the snapshot recording, corresponding bookmarks are added in the telemetry views. Only classes with a total object size above a fixed threshold (usually 1% of the heap) are recorded. By default, this feature is disabled in JFR because it introduces a substantial overhead.

In the "Monitors & locks" section, only the monitor history and the monitor usage statistics views are present. In particular, the locking history graph is unavailable, so that the monitor tool tips in the thread history view link to the monitor usage history view.

Probes are shown for classes, sockets, files and exceptions. Data in these probes is only shown if the corresponding data recording has been configured for JFR. There is no thread status selector in the probe call tree and hot spots views and the probe tracker is unavailable.



When opening very large JFR snapshots, you can speed up snapshot processing and reduce memory usage by clicking on the "Customize analysis" check box in the file chooser and excluding the event types that are not required for your analysis. For example, if you are only interested in CPU data, you can exclude monitor events.

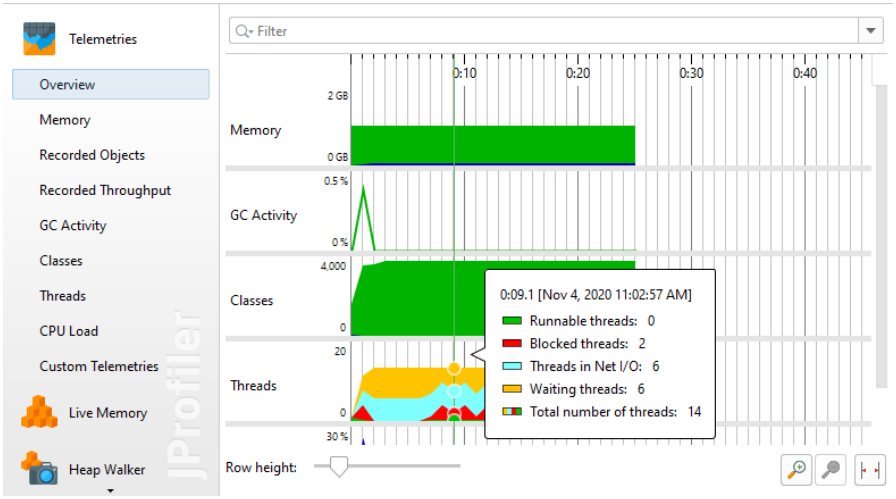
Telemetries

One aspect of profiling is monitoring scalar measurements over time, for example the used heap size. In JProfiler, such graphs are called telemetries. Observing telemetries gives you a better understanding of the profiled software, allows you to correlate important events over different measurements and may prompt you to perform a deeper analysis with other views in JProfiler if you notice unexpected behavior.

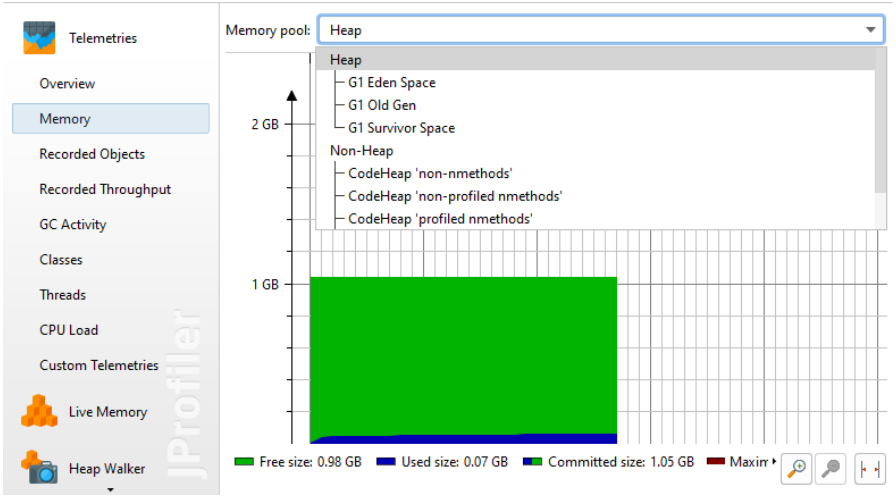
Standard telemetries

In the "VM Telemetries" section of the JProfiler UI, a number of telemetries are recorded by default. For interactive sessions they are always enabled and you do not have to start or stop their recording.

To compare multiple telemetries on the same time axis, the overview shows multiple small-scale telemetries on top of each other with a configurable row height. Clicking on the telemetry title activates the full telemetry view.

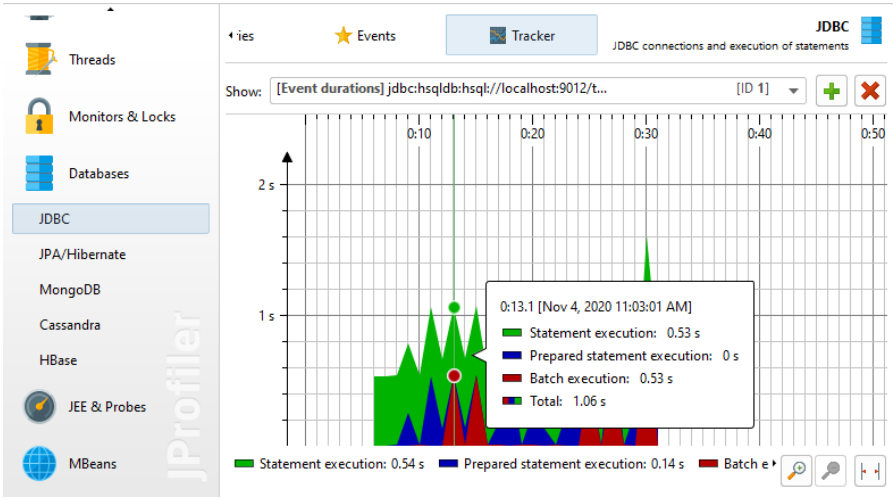


The full view shows a legend with current values and may have more options than what is visible in the overview. For example, the "Memory" telemetry allows you to select single memory pools.



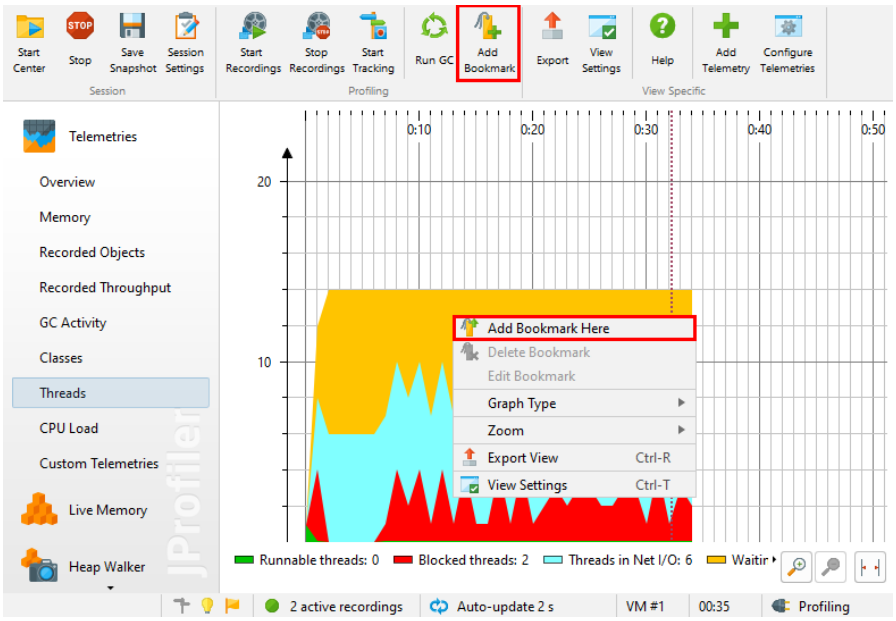
Probes also publish telemetries. These telemetries are not included in the "Telemetries" view section, but are part of the "Telemetries" tab of the corresponding probe. Recording of those telemetries is coupled to the recording of their parent probe.

Finally, there are "tracking" telemetries that monitor a scalar value that is selected in another view. For example, the class tracker view allows you to select a class and monitor its instance count over time. Also, each probe has a "Tracker" view where selected hot spots or control objects are monitored.



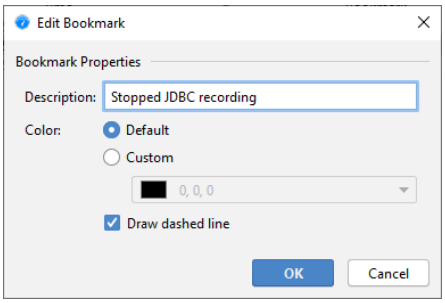
Bookmarks

JProfiler maintains a list of bookmarks that are shown in all telemetries. In an interactive session, you can add a bookmark at the current time by clicking on the *Add Bookmark* tool bar button, or by using the *Add Bookmark Here* feature in the context menu.

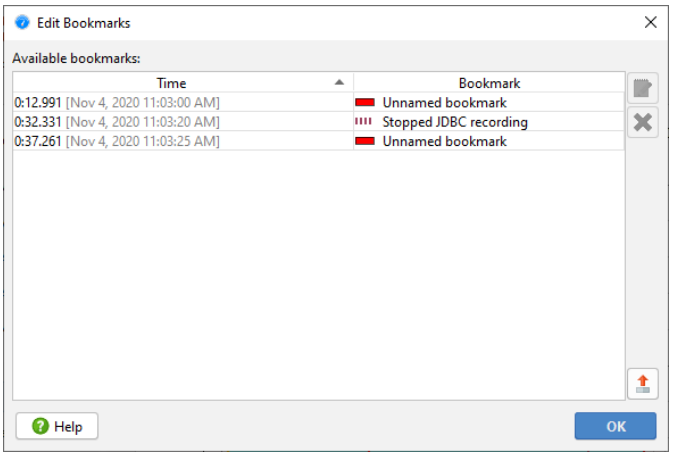


Bookmarks can not only be created manually, they are added automatically by the recording actions to indicate the beginning and the end of a particular recording. With trigger actions or with the controller API, you can add bookmarks programatically.

Bookmarks have color, a line style and also a name that shows up in the tool tip. You can edit existing bookmarks and change these properties.



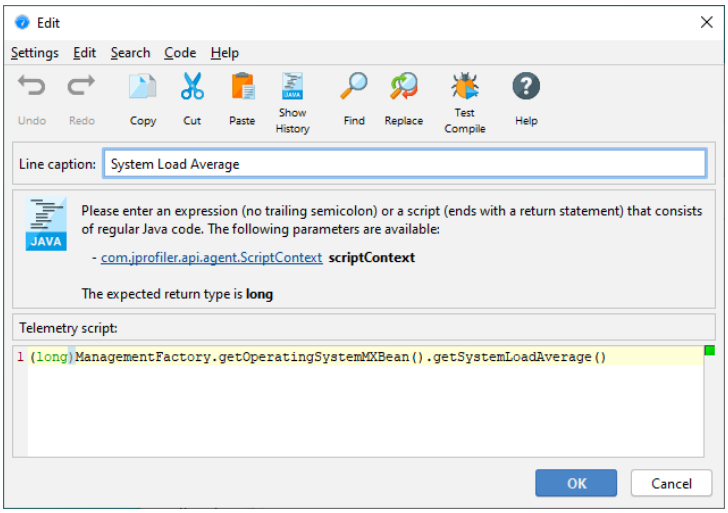
If right-clicking several bookmark in a telemetry is too inconvenient, you can use the *Profiling->Edit Bookmarks* action from the menu to get a list of bookmarks. This is also the place where you can export bookmarks to HTML or CSV.



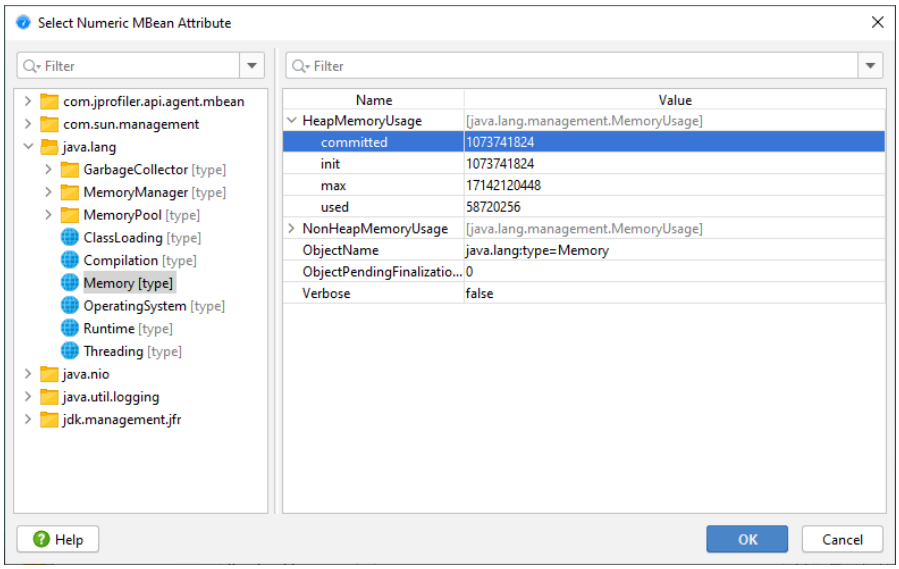
Custom telemetries

There are two ways to add your own telemetries: Either you write a script in the JProfiler UI to supply a numeric value or you select a numeric MBean attribute.

To add a custom telemetry, click on the *Configure Telemetries* tool bar button that is visible in the "Telemetries" section. In a script telemetry, you have access to all classes that are configured in the classpath of the current JProfiler session. If a value is not available directly, add a static method to your application that you can call in this script.

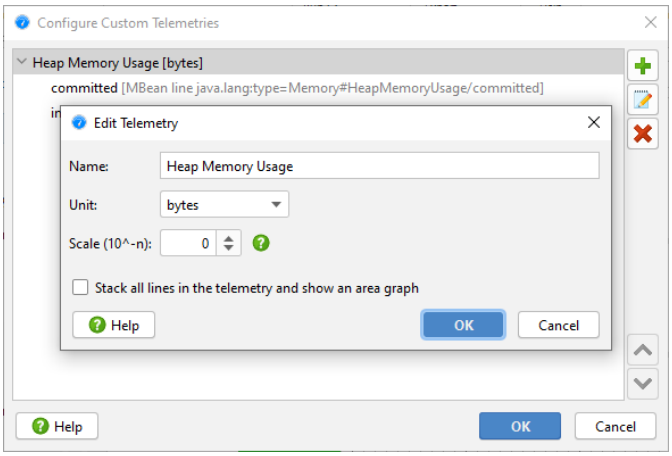


The above example shows a call to a platform MBean. Graphing scalar values of MBeans is more conveniently done with an MBean telemetry. Here, an MBean browser allows you to select a suitable attribute. The attribute value must be numeric.

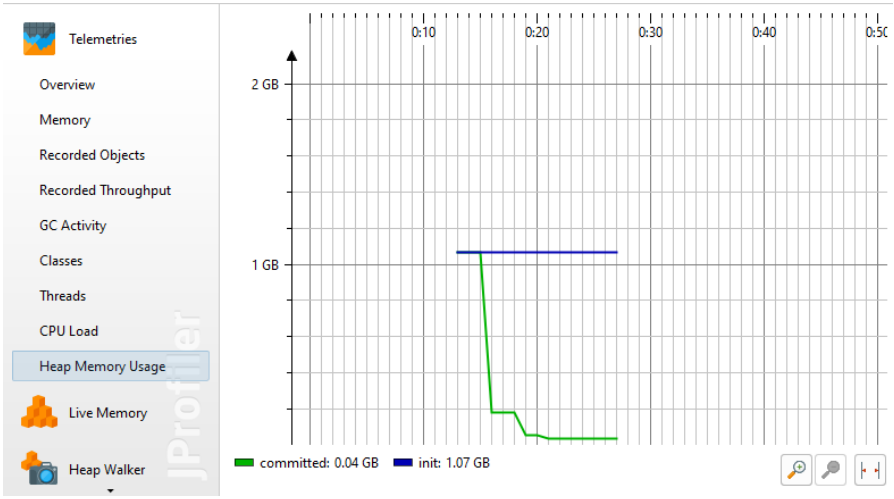


You can bundle several telemetry lines into a single telemetry. That's why the configuration is split into two parts: the telemetry itself and the telemetry line. In the telemetry line, you just edit the data source and the line caption, in the telemetry you can configure unit, scale and stacking which apply to all contained lines.

In a stacked telemetry, the single telemetry lines are additive and an area graph can be shown. The scale factor is useful to convert a value to a supported unit. For example, if the data source reports kB, the problem is that there is no matching "kB" unit in JProfiler. If you set the scale factor to -3, the values will be converted to bytes and by choosing "bytes" as the unit for the telemetry, JProfiler will automatically display the appropriate aggregate unit in the telemetry.



Custom telemetries are shown at the end of the "Telemetries" section in the order in which they are configured.



Overhead considerations

At first sight, it would seem that telemetries consume memory linearly with time. However, JProfiler consolidates older values and makes them progressively more coarse-grained in order to limit the total amount of memory consumed per telemetry.

The CPU overhead of telemetries is limited by the fact that their values are only polled once per second. For the standard telemetries, there is no additional overhead for this data collection. For custom telemetries, the overhead is governed by the underlying script or MBean.

CPU Profiling

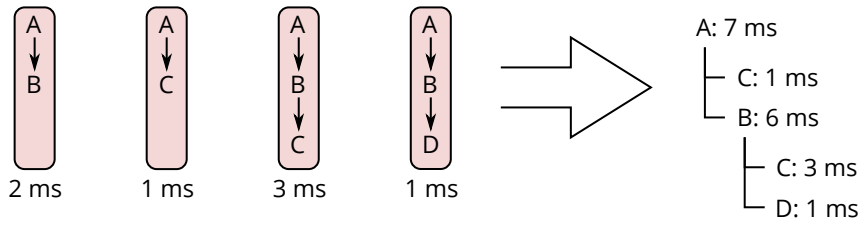
When JProfiler measures the execution times of method calls together with their call stacks, we call it "CPU profiling". This data is presented in a variety of ways. Depending on the problem you are trying to solve, one or the other presentation will be most helpful. CPU data is not recorded by default, you have to switch on CPU recording [p. 23] to capture interesting use cases.

Call tree

Keeping track of all method calls and their call stacks would consume a considerable amount of memory and could only be kept up for a short time until all memory is exhausted. Also, it is not easy to intuitively grasp the number of method calls in a busy JVM. Usually, that number is so great that locating and following traces is impossible.

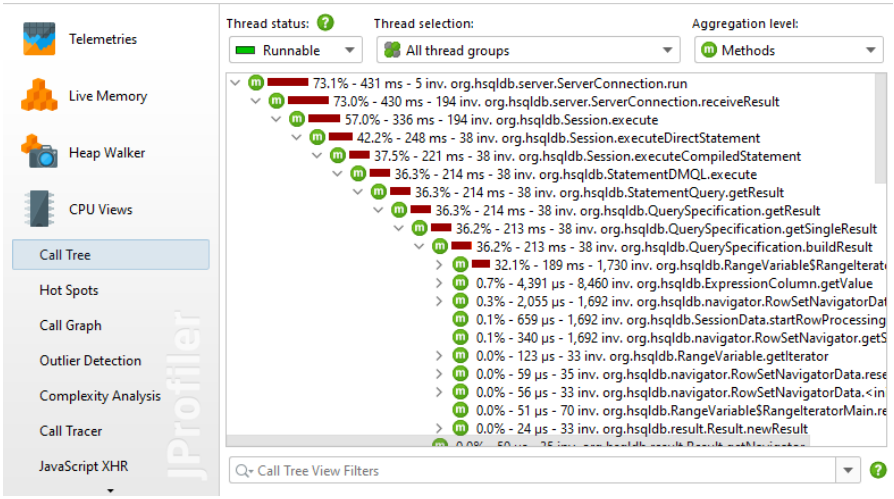
Another aspect is that many performance problems only become clear if the collected data is aggregated. In that way, you can tell how important method calls are with respect to the entire activity in a certain time period. With single traces, you have no notion of the relative importance of the data that you are looking at.

This is why JProfiler builds a cumulated tree of all observed call stacks, annotated with the observed timings and invocation counts. The chronological aspect is eliminated and only the total numbers are kept. Each node in the tree represents one call stack that was observed at least once. Nodes have children that represent all the outgoing calls that were seen at that call stack.

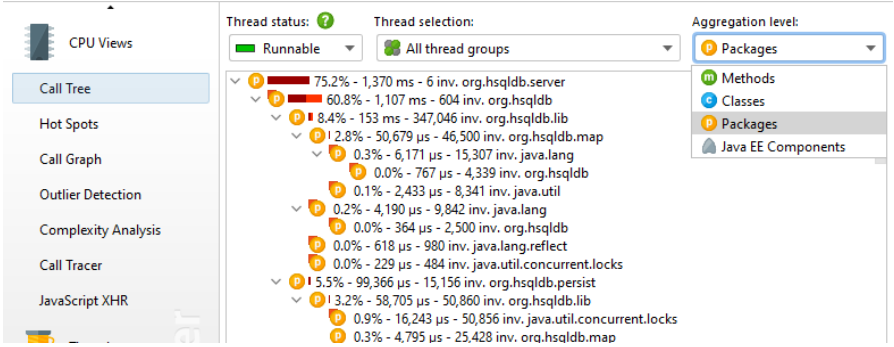


method invocations with call stacks **call tree**

The call tree is the first view in the "CPU views" section, and it's a good starting point when you start CPU profiling, because the top-down view that follows method calls from the starting points to the most granular end points is most easily understood. JProfiler sorts children by their total time, so you can open the tree depth-first to analyze the part of the tree that has the greatest performance impact.



While all measurements are performed for methods, JProfiler allows you to take a broader perspective by aggregating the call tree on the class or package level. The aggregation level selector also contains a "Java EE components" mode. If your application uses Java EE, you can use this mode to see only JEE components on a class level. Splitting nodes like URLs are retained in all aggregation levels.

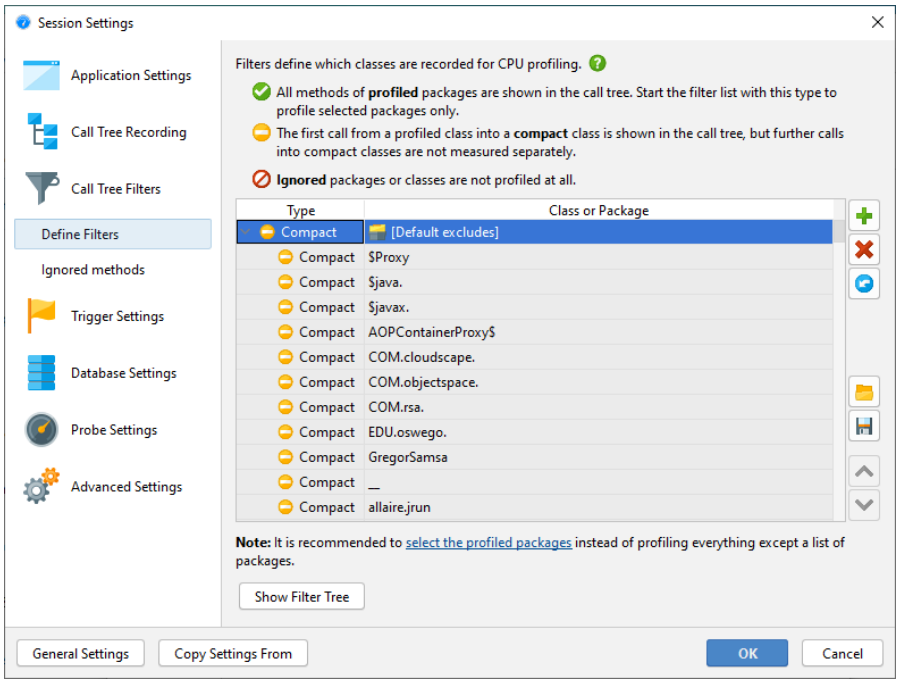


Call tree filters

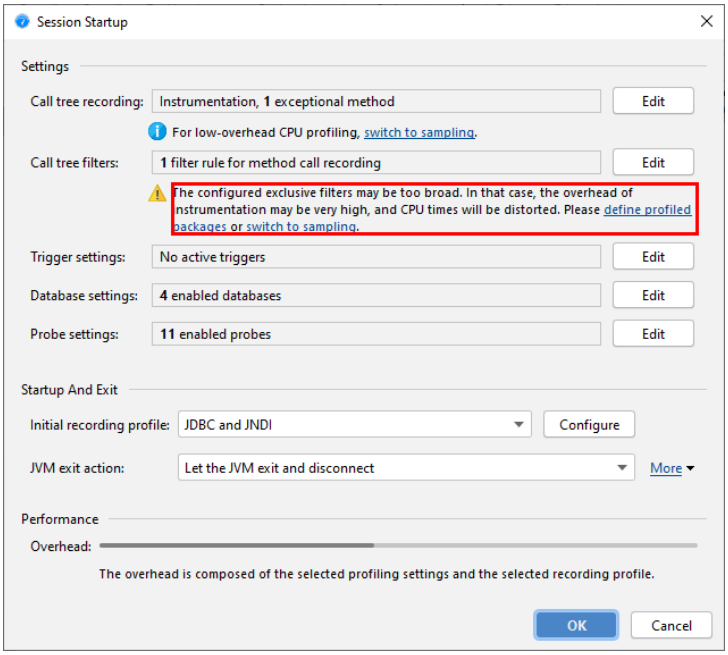
If methods from all classes are shown in the call tree, the tree is usually too deep to be manageable. If your application is called by a framework, the top of the call tree will consist of framework classes that you don't care about and your own classes will be deeply buried. Calls into libraries will show their internal structure, possibly with hundreds of levels of method calls that you are not familiar with and not in a position to influence.

The solution to this problem is to apply filters to the call tree, so that only some classes are recorded. As a positive side-effect, less data has to be collected and less classes have to be instrumented, so the overhead is reduced.

By default, profiling sessions are configured with a list of excluded packages from commonly used frameworks and libraries.



Of course this list is incomplete so it's much better that you delete it and define the packages of interest yourself. In fact, the combination of instrumentation [p. 61] and the default filters is so undesirable, that JProfiler suggests to change it in the session startup dialog.

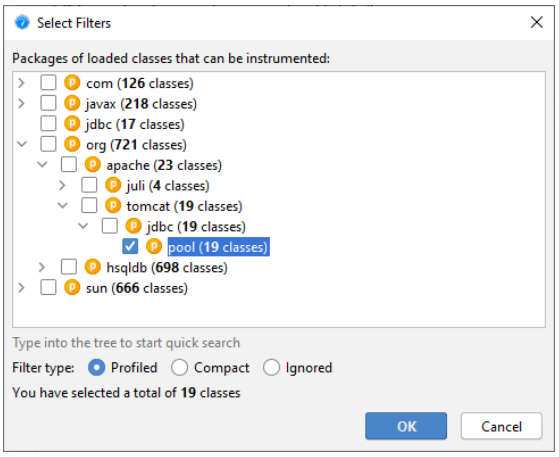


The filter expressions are compared against the fully qualified class name, so `com.mycorp.` matches classes in all nested packages, like `com.mycorp.myapp.Application`. There are three types of filters, called "profiled", "compact" and "ignored". All methods in "profiled" classes are measured. This is what you need for your own code.

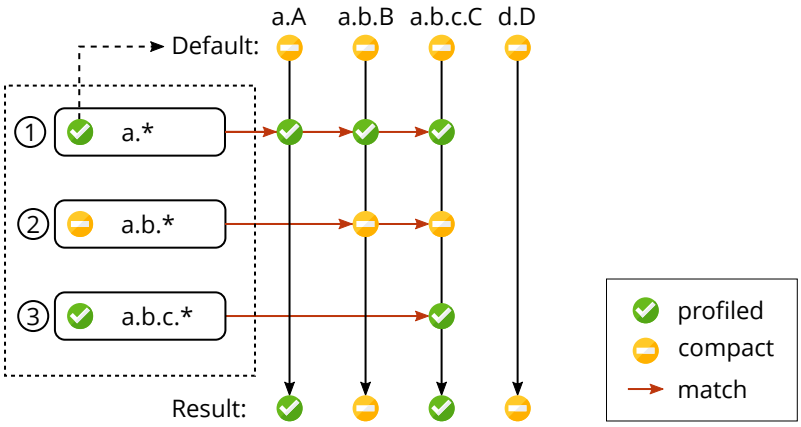
In a class that is contained by a "compact" filter, only the first call into that class is measured, but no further internal calls are shown. "Compact" is what you want for libraries, including the JRE. For example, when calling `hashMap.put(a, b)` you probably want to see `HashMap.put()` in the call tree, but not more than that - its inner workings should be treated as opaque unless you are the developer of the map implementation.

Finally, "ignored" methods are not profiled at all. They may be undesirable to instrument due to overhead considerations or they may simply be distracting in the call tree, such as internal Groovy methods that are inserted between dynamic calls.

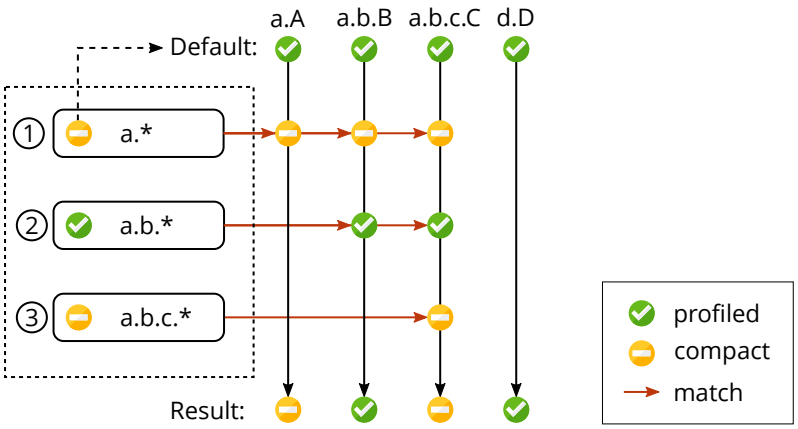
Entering packages manually is error prone, so you can use the package browser. Before you start the session, the package browser can only show you packages in the configured class path which often does not cover all the classes that are actually loaded. At runtime, the package browser will show you all loaded classes.



The configured list of filters is evaluated from top to bottom for each class. At each stage, the current filter type may change if there is a match. It's important what kind of filter starts off the list of filters. If you start with a "profiled" filter, the initial filter type of a class is "compact", meaning that only explicit matches are profiled.



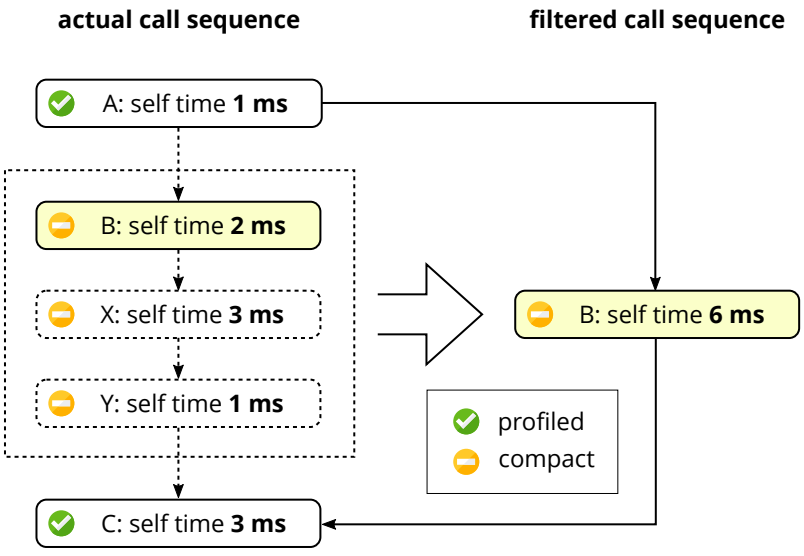
If you start it with a "compact" filter, the initial filter type of a class is "profiled". In this case, all classes are profiled except for explicitly excluded classes.



Call tree times

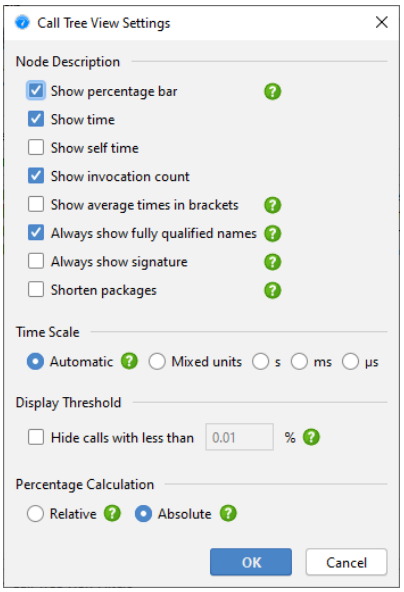
To interpret the call tree correctly, it's important to understand the numbers that are displayed on the call tree nodes. There are two times that are interesting for any node, the total time and the self time. The self time is the total time of the node minus the total time in the nested nodes.

Usually, the self time is small, except for compact-filtered classes. Most often, a compact-filtered class is a leaf node and the total time is equal to the self time because there are no child nodes. Sometimes, a compact-filtered class will invoke a profiled class, for example via a callback or because it's the entry point of the call tree, like the `run` method of the current thread. In that case, some unprofiled methods have consumed time, but are not shown in the call tree. That time bubbles up to the first available ancestor node in the call tree and contributes to the self time of the compact-filtered class.



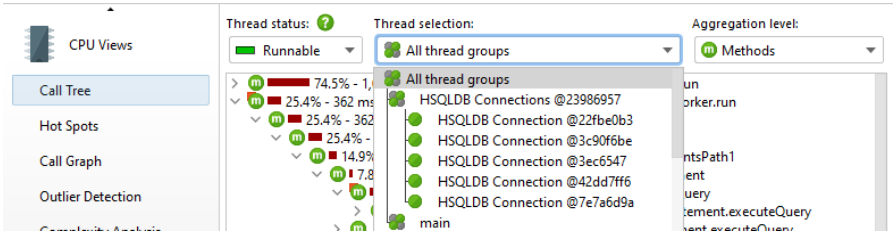
The percentage bar in the call tree shows the total time, but the self time portion is shown with a different color. Methods are shown without their signatures unless two methods on the same level are overloaded. There are various ways to customize the display of the call tree nodes in

the view settings dialog. For example, you may want to show self times or average times as text, always show method signatures or change the used time scale. Also, the percentage calculation can be based on the parent time instead of the time for the entire call tree.



Thread status

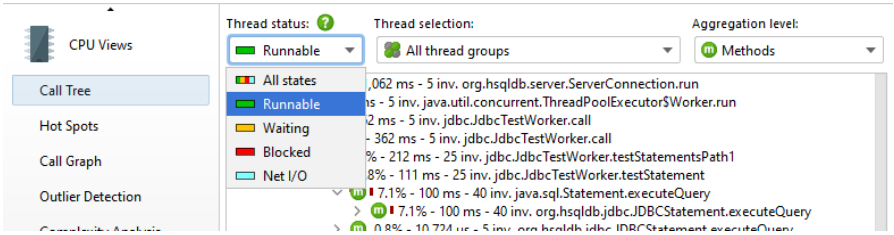
At the top of the call tree there are several view parameters that change the type and scope of the displayed profiling data. By default, all threads are cumulated. JProfiler maintains CPU data on a per-thread basis and you can show single threads or thread groups.



At all times, each thread has an associated thread status. If the thread is ready to process bytecode instructions or is currently executing them on a CPU core, the thread status is called "Runnable". That thread state is of interest when looking for performance bottlenecks, so it is selected by default.

Alternatively, a thread may by waiting on a monitor, for example by calling `Object.wait()` or `Thread.sleep()` in which case the thread state is called "Waiting". A thread that is blocked while trying to acquire a monitor, such as at the boundary of a `synchronized` code block is in the "Blocking" state.

Finally, JProfiler adds a synthetic "Net I/O" state that keeps track of the times when a thread is waiting for network data. This is important for analyzing servers and database drivers, because that time can be relevant for performance analysis, such as for investigating slow SQL queries.

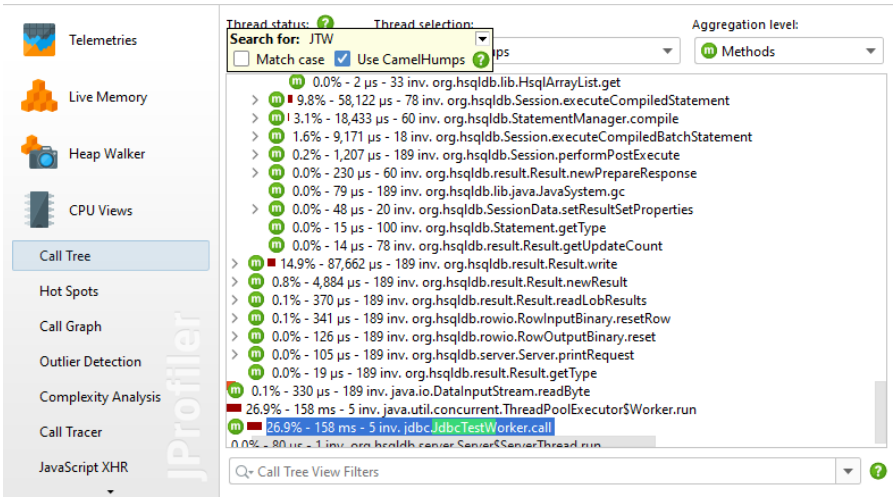


If you are interested in wall-clock times, you have to select the thread status "All states" and also select a single thread. Only then can you compare times with durations that you have calculated with calls to `System.currentTimeMillis()` in your code.

If you want to shift selected methods to a different thread status, you can do so with a method trigger and an "Override thread status" trigger action, or by using the `ThreadStatus` class in the embedded [p. 147] or injected [p. 142] probe APIs.

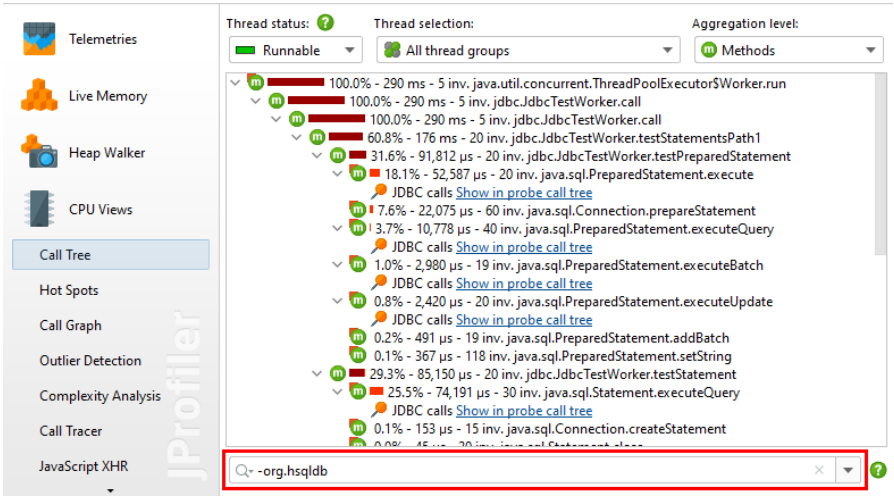
Finding nodes in the call tree

There are two ways to search for text in the call tree. First, there is the quicksearch option that is activated by invoking *View->Find* from the menu or by directly starting to type into the call tree. Matches will be highlighted and search options are available after pressing `PageDown`. With the `ArrowUp` and `ArrowDown` keys you can cycle through the different matches.



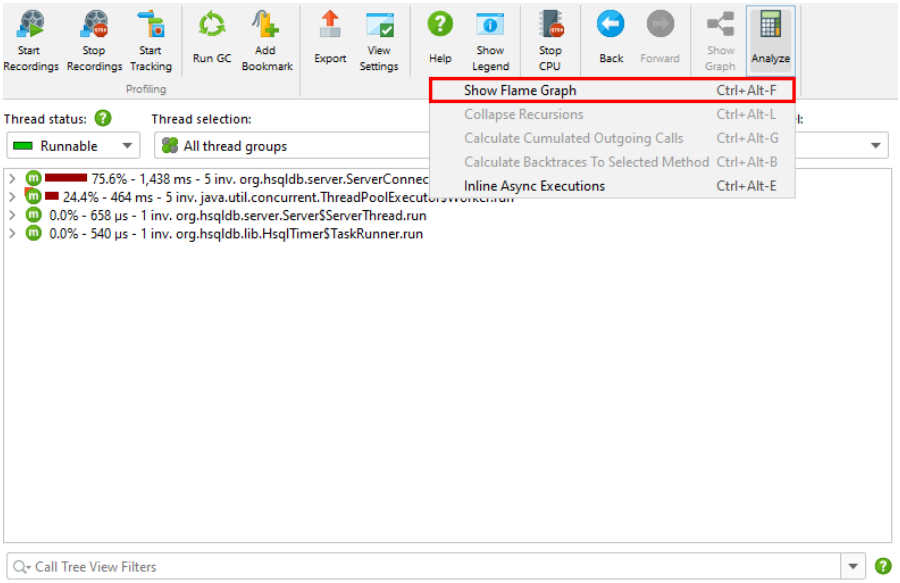
Another way to search for methods, classes or packages is to use the view filter at the bottom of the call tree. Here you can enter a comma-separated list of filter expressions. Filter expressions that start with a "-" are like ignored filters. Expression that start with a "!" are like compact filters. All other expressions are like profiled filters. Just like for the filter settings, the initial filter type determines if classes are included or excluded by default.

Clicking on the icon to the left of the view settings text field shows the view filter options. By default, the matching mode is "Contains", but "Starts with" may be more appropriate when searching for particular packages.

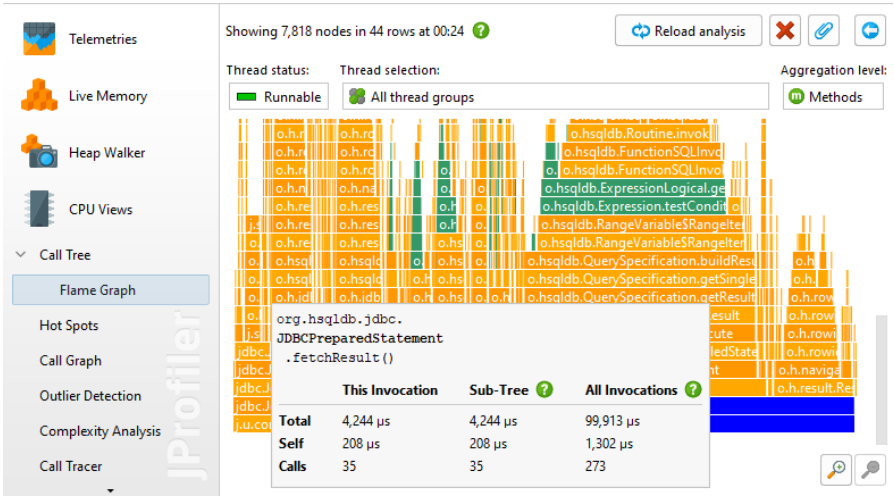


Flame graphs

Another way to view the call tree is as a flame graph. You can show the entire call tree or a portion of it as a flame graph by invoking the associated call tree analysis [p. 169].

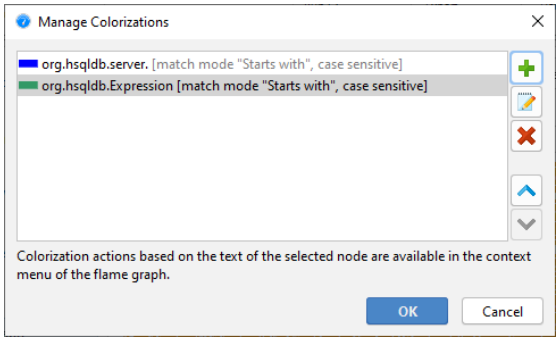


A flame graph shows the entire content of a call tree in one image. Calls originate at the bottom of the flame graph and propagate towards the top. The children of each node are arranged in the row directly above it. Child nodes are sorted alphabetically and are centered on their parent node. Due to the self-time that is spent in each node, the "flames" get progressively more narrow toward the top. More information about nodes is displayed in the tool tip where you can mark text to copy it to the clipboard.



Flame graphs have a very high information density, so it may be necessary to narrow the displayed content by focusing on selected nodes and their hierarchy of descendant nodes. While you can zoom in on areas of interest, you can also set a new root node by double-clicking on it or by using the context menu. When changing roots multiple times in a row, you can move back again in the history of roots.

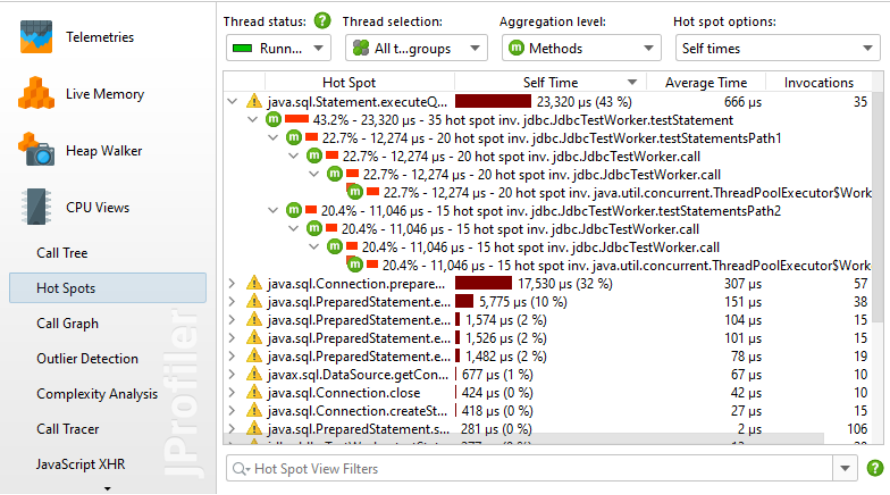
Another way to analyze flame graphs is to add colorizations based on class names, package names or arbitrary search terms. Colorizations can be added from the context menu and can be managed in the colorizations dialog. The first matching colorization is used for each node. In addition to colorizations, you can use the quick search functionality to find nodes of interest. With the cursor keys you can cycle through match results while the tooltip is being displayed for the currently highlighted match.



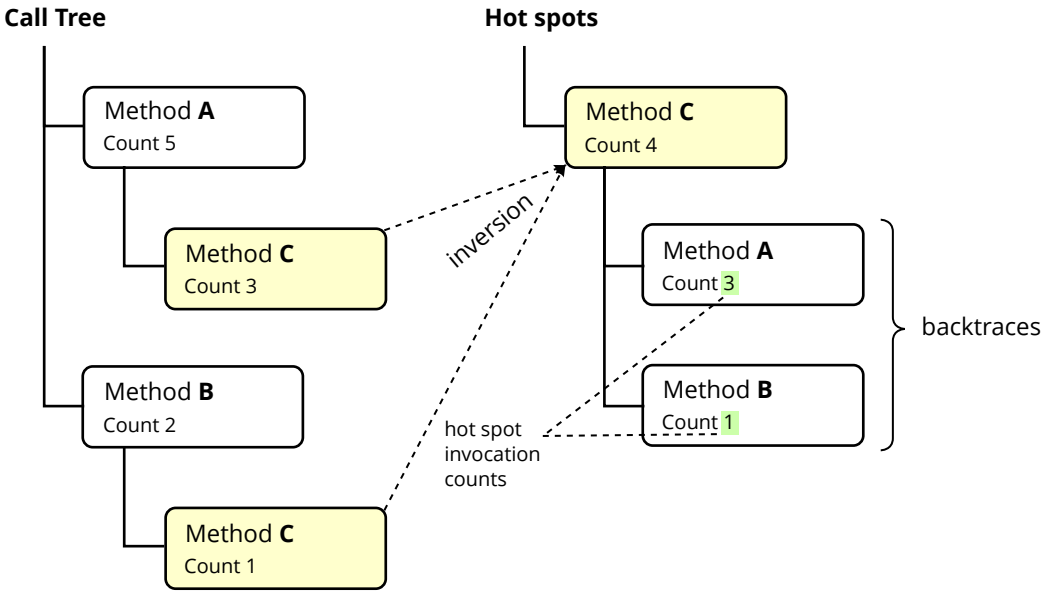
Hot spots

If your application is running too slowly, you want to find the methods that take most of the time. With the call tree, it is sometimes possible to find these methods directly, but often that does not work because the call tree can be broad with a huge number of leaf nodes.

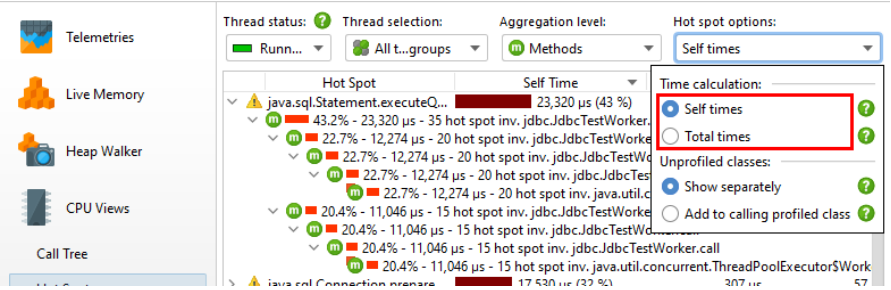
In that case, you need the inverse of the call tree: A list of all methods sorted by their total self time, cumulated from all different call stacks and with back traces that show how the methods were called. In a hot spot tree, the leafs are the entry points, like the `main` method of the application or the `run` method of a thread. From the deepest nodes in the hot spot tree, the call propagates upward to the top-level node.



The invocation counts and execution times in the backtraces do not refer to the method nodes, but rather to the number of times that the top-level hot spot node was called along this path. This is important to understand: At a cursory glance, you would expect the information on a node to quantify calls to that node. However, in a hot spot tree, that information shows the **contribution** of the node to the top-level node. So you have to read the numbers like this: Along this inverted call stack, the top-level hot spot was called n times with a total duration of t seconds.



By default, the hot spots are calculated from self time. You can also calculate them from total time. This is not very useful for analyzing performance bottlenecks, but can be interesting if you would like to see a list of all methods. The hot spot view only shows a maximum number of methods to reduce overhead, so a method you are looking for may not by displayed at all. In that case, use the view filters at the bottom to filter the package or the class. Contrary to the call tree, the hot spot view filters only filter the top-level nodes. The cutoff in the hot spot view is not applied globally, but with respect to the displayed classes, so new nodes may appear after applying a filter.

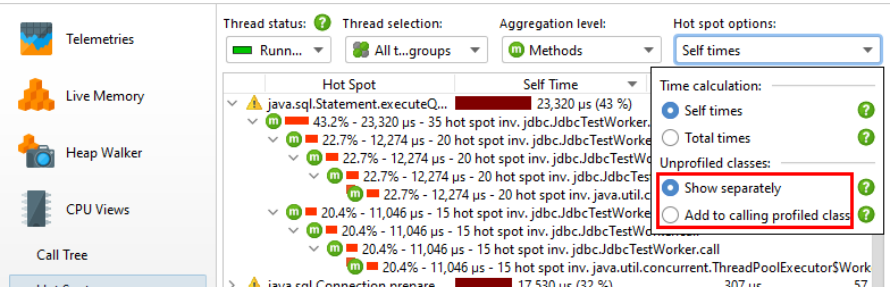


Hot spots and filters

The notion of a hot spot is not absolute but depends on the call tree filters. If you have no call tree filters at all, the biggest hot spots will most likely always be methods in the core classes of the JRE, like string manipulation, I/O routines or collection operations. Such hot spots would not be very useful, because you often don't directly control the invocations of these methods and also have no way of speeding them up.

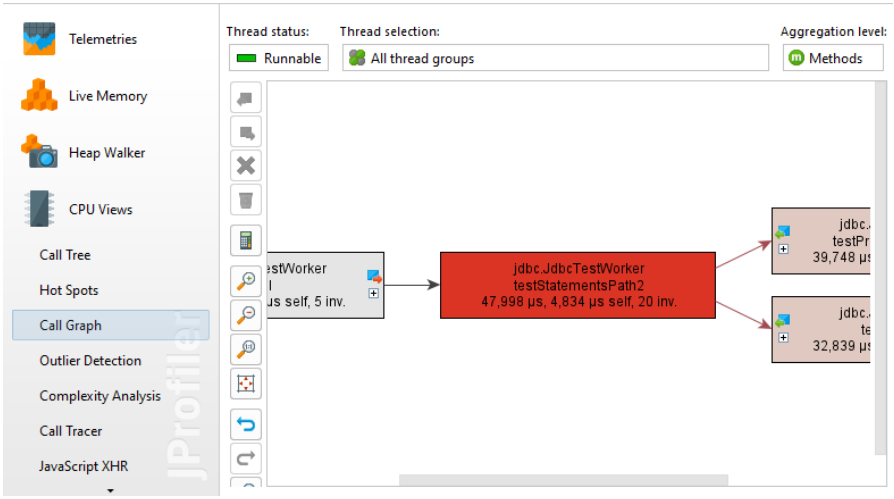
In order to be useful to you, a hot spot must either be a method in your own classes or a method in a library class that you call directly. In terms of the call tree filters, your own classes are in "profiled" filters and the library classes are in "compact" filters.

When solving performance problems, you may want to eliminate the library layer and only look at your own classes. You can quickly switch to that perspective in the call tree by selecting the *Add to calling profiled class* radio button in the hot spot options popup.

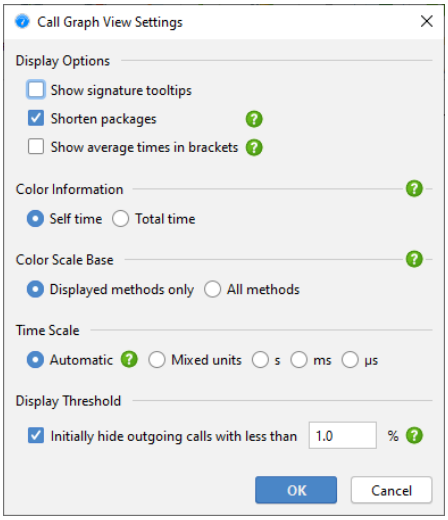


Call graph

Both in the call tree as well in the hot spots view each node can occur multiple times, especially when calls are made recursively. In some situations you are interested in a method-centric statistics where each method only occurs once and all incoming and outgoing calls are visible. Such a view is best displayed as a graph and in JProfiler, it is called the call graph.

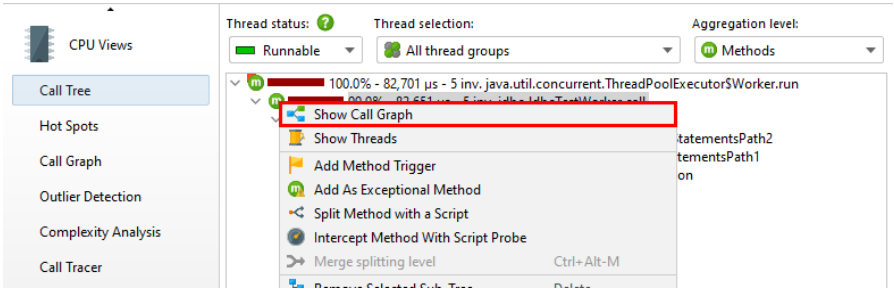


One drawback of graphs is that their visual density is lower than that of trees. This is why JProfiler abbreviates package names by default and hides outgoing calls with less than 1% of the total time by default. As long as the node has an outgoing expansion icon, you can click on it again to show all calls. In the view settings, you can configure this threshold and turn off package abbreviation.



When expanding the call graph, it can get messy very quickly, especially if you backtrack multiple times. Use the undo functionality to restore previous states of the graph. Just like the call tree, the call graph offers quick search. By typing into the graph, you can start the search.

The graph and the tree views each have their advantages and disadvantages, so you may sometimes wish to switch from one view type to another. In interactive sessions the call tree and hot spots views show live data and are updated periodically. The call graph however, is calculated on request and does not change when you expand nodes. The *Show in Call Graph* action in the call tree calculates a new call graph and shows the selected method.



Switching from the graph to the call tree is not possible because the data is usually not comparable anymore at a later time. However, the call graph offers call tree analyses with its *View->Analyze* actions that can show you trees of cumulated outgoing calls and backtraces for each selected node.

Beyond the basics

The ensemble of call tree, hot spots view and call graph has many advanced features that are explained in detail in a different chapter [p. 151]. Also, there are other advanced CPU views that are presented separately [p. 174].

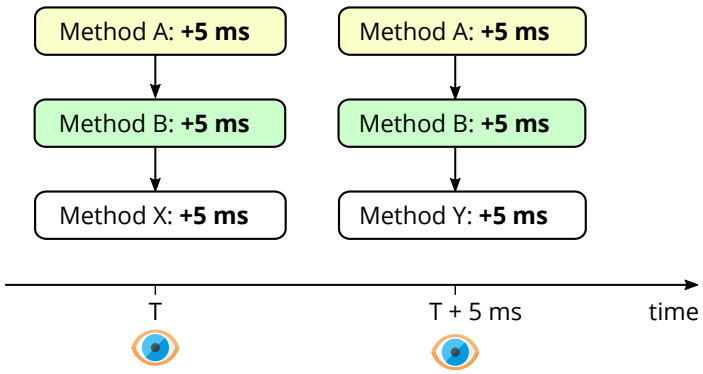
Method Call Recording

Recording method calls is is one of the most difficult tasks for a profiler, because it operates under conflicting constraints: Results should to be accurate, complete and produce such a small overhead that the conclusions you draw from the measured data do not become incorrect. Unfortunately, there is no single type of measurement that fulfills all these requirements for all types of applications. This is why JProfiler requires you to make a decision on which method to use.

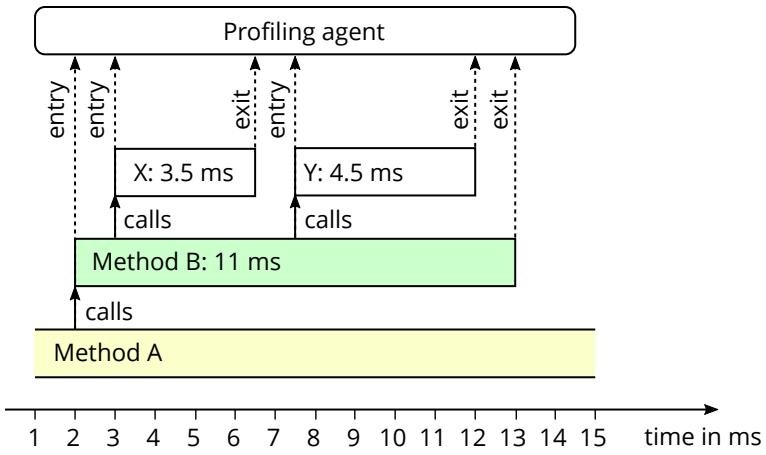
Sampling versus instrumentation

Measuring method calls can be done with two fundamentally different techniques called "sampling" and "instrumentation", each of which has advantages and drawbacks: With sampling, the current call stacks of threads are inspected periodically. With instrumentation, the bytecode of selected classes is modified to trace method entry and exit. Instrumentation measures all invocations and can produce invocation counts for all methods.

When processing sampling data, the full sampling period (typically 5 ms) is attributed to the sampled call stack. With a large number of samples, a statistically correct picture emerges. The advantage of sampling is that it has a very low overhead because it happens infrequently. No bytecode has to be modified and the sampling period is much larger than the typical duration of a method call. The downside is that you cannot determine any method invocation counts. Additionally, short running methods that are called only a few times might not show up at all. This does not matter if you are looking for performance bottlenecks, but can be inconvenient if you are trying to understand the detailed runtime characteristics of your code.



Instrumentation, on the other hand, can introduce a large overhead if many short-running methods are instrumented. This instrumentation distorts the relative importance of performance hot spots because of the inherent overhead of the time measurement but also because many methods that would otherwise be inlined by the hot spot compiler must now remain separate method calls. For method calls that take a longer amount of time, the overhead is insignificant. If you can find a good set of classes that mainly perform high-level operations, instrumentation will add a very low overhead and can be preferable to sampling. JProfiler's overhead hotspot detection can also improve the situation after some runs. Additionally, the invocation count is often important information that makes it much easier to see what is going on.

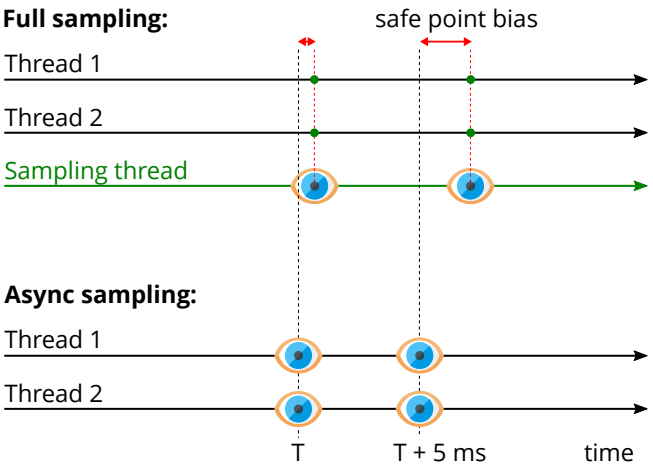


Full sampling versus async sampling

JProfiler offers two different technical solutions for sampling: "Full sampling" is done with a separate thread that pauses all threads in the JVM periodically and inspects their stack traces. However, the JVM only pauses threads at certain "safe points" thereby introducing a bias. If you have highly multi-threaded CPU bound code, the profiled distribution of hotspots may be skewed significantly. On the other hand, if code also performs significant I/O, this bias will generally not be a problem.

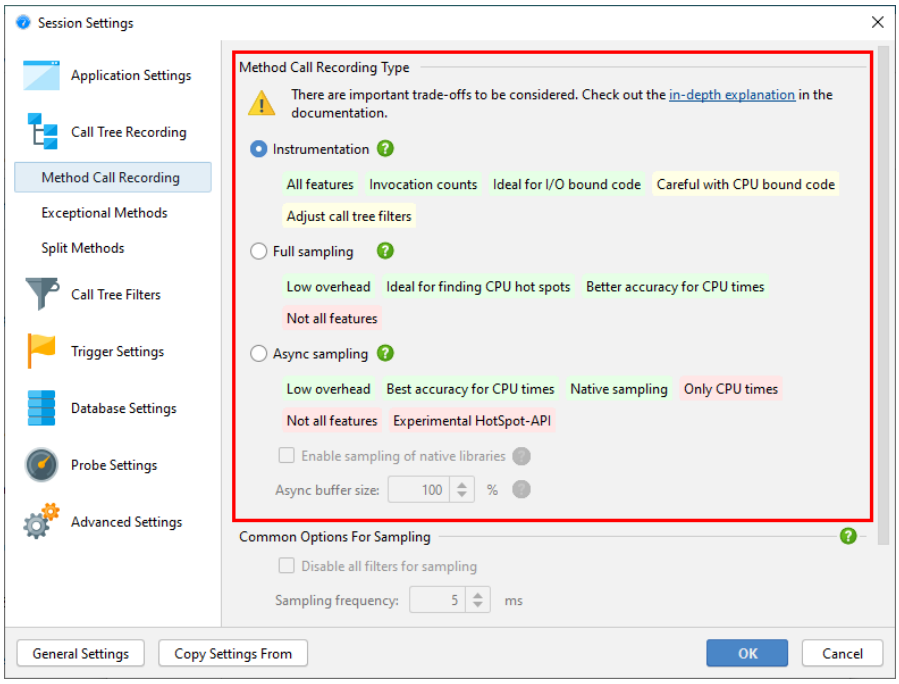
To help with getting accurate numbers for highly CPU bound code, JProfiler also offers async sampling. With async sampling, a profiling signal handler is called on the running threads themselves. The profiling agent then inspects the native stack and extracts the Java stack frames. The main benefit is that there is no safe-point bias with this sampling method and the overhead for highly multi-threaded CPU bound applications is lower. However, only the "Running" thread state can be observed for the CPU views while "Waiting", "Blocking" or "Net I/O" thread states cannot be measured in this way. Probe data is always collected with bytecode instrumentation, so you will still get all thread states for JDBC and similar data.

Async sampling is only supported on Linux and macOS. Windows is not supported, because the operating system does not offer POSIX-style signal handlers.



Choosing a method call recording type

Which method call recording type to use for profiling is an important decision and there no right choice for all circumstances, so you need to make an informed decision. When you create a new session, the session startup dialog will ask you which method call recording type you want to use. At any later point in time you can change the method call recording type in the session settings dialog.



As a simple guide, consider the following questions that test whether your application falls into one of two clear categories on opposite sides of the spectrum:

- **Is the profiled application I/O bound?**

  This is the case for many web applications that wait on REST service and JDBC database calls most of the time. In that case, instrumentation will be the best option under the condition that you carefully select your call tree filters to only include your own code.

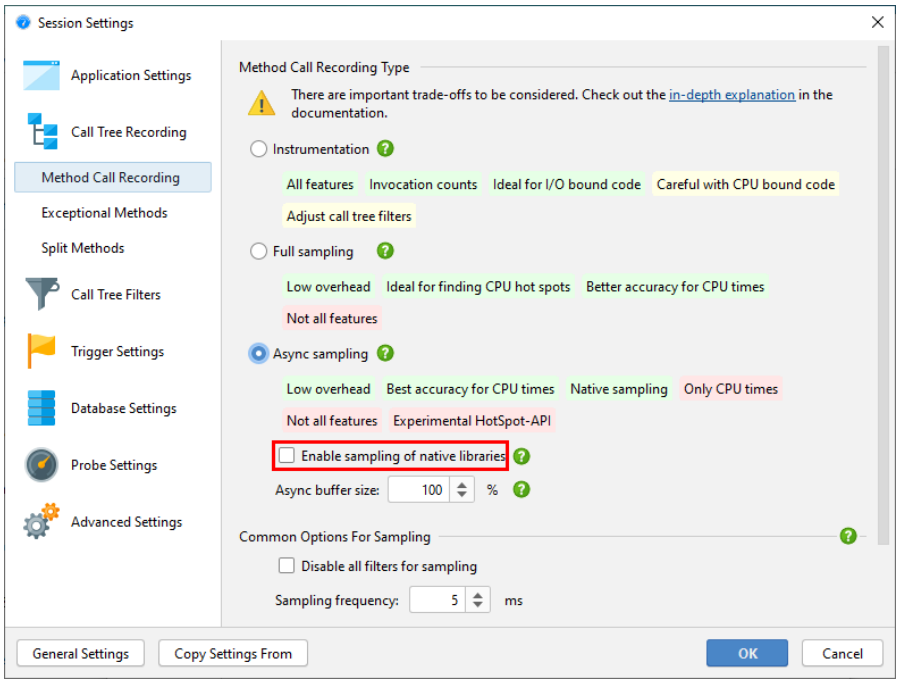
- **Is the profiled application heavily multi-threaded and CPU bound?**

  For example, this could be the case for a compiler, image processing application or a web server that is running a load test. If you are profiling on Linux or macOS, you should choose async sampling to get the most accurate CPU times in this case.

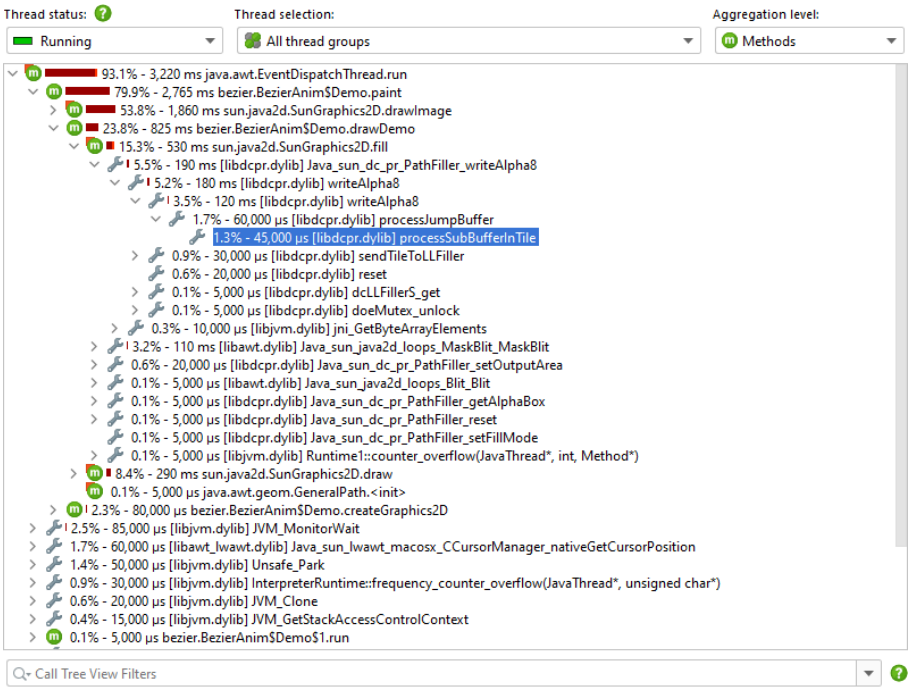
Otherwise, "Full sampling" is generally the most suitable option and is suggested as the default for new sessions.

Native sampling

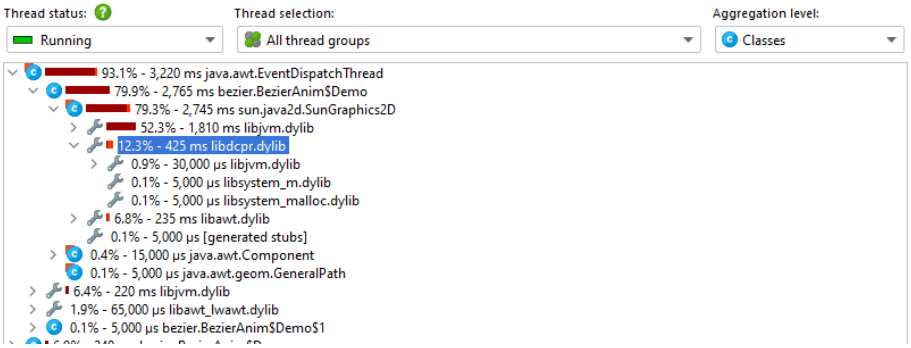
Because async sampling has access to the native stack, it can also perform native sampling. By default, native sampling is not enabled, because it introduces a lot of nodes into call trees and shifts the focus of hot spot calculation to native code. If you do have a performance problem in native code, you can choose async sampling and enable native sampling in the session settings.



JProfiler resolves the path of the library that belongs to each native stack frame. On native method nodes in the call tree, JProfiler shows the file name of the native library in square brackets at the beginning.



With respect to the aggregation level, native libraries act like classes, so in the "classes" aggregation level all subsequent calls within the same native library will be aggregated into a single node. The "packages" aggregation level aggregates all subsequent native method calls into a single node regardless of the native library.



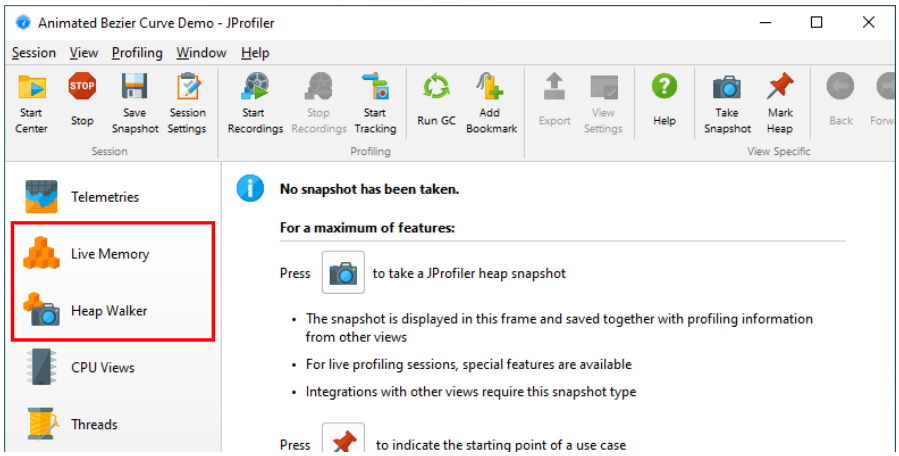
To eliminate selected native libraries, you can remove a node [p. 160] from that native library and choose to remove the entire class.

Memory Profiling

There are two ways of getting information about objects on the heap. On the one hand, a profiling agent can track the allocation and the garbage collection of each object. In JProfiler, this is called "allocation recording". It tells you where objects have been allocated and can also be used to create statistics about temporary objects. On the other hand, the profiling interface of the JVM allows the profiling agent to take a "heap snapshot" in order to inspect all live objects together with their references. This information is required to understand why objects cannot be garbage collected.

Both allocation recording and heap snapshots are expensive operations. Allocation recording has a large impact on the runtime characteristics, because the `java.lang.Object` constructor has to be instrumented and the garbage collector continuously has to report to the profiling interface. This is why allocations are not recorded by default and you have to start and stop recording [p. 23] explicitly. Taking a heap snapshot is a one-time operation. However, it can pause the JVM for several seconds and the analysis of the acquired data may take a relatively long time, scaling with the size of the heap.

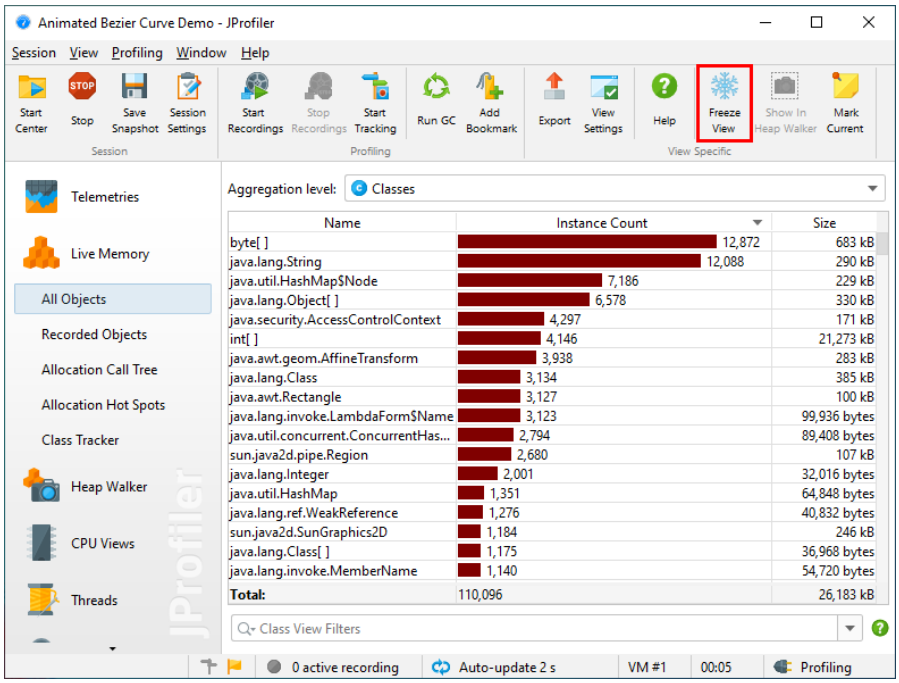
JProfiler splits its memory analysis into two view sections: The "Live memory" section presents data that can be updated periodically whereas the "Heap walker" section shows a static heap snapshot. Allocation recording is controlled in the "Live memory" section but the recorded data is also displayed by the heap walker.



The three most common problems that can be solved with memory profiling are: Finding a memory leak [p. 192], reducing memory consumption and reducing the creation of temporary objects. For the first two problems, you will mainly use the heap walker, mostly by looking at who is holding on to the biggest objects in the JVM and where they were created. For the last problem you can only rely on the live views that show recorded allocations, because it involves objects that have already been garbage collected.

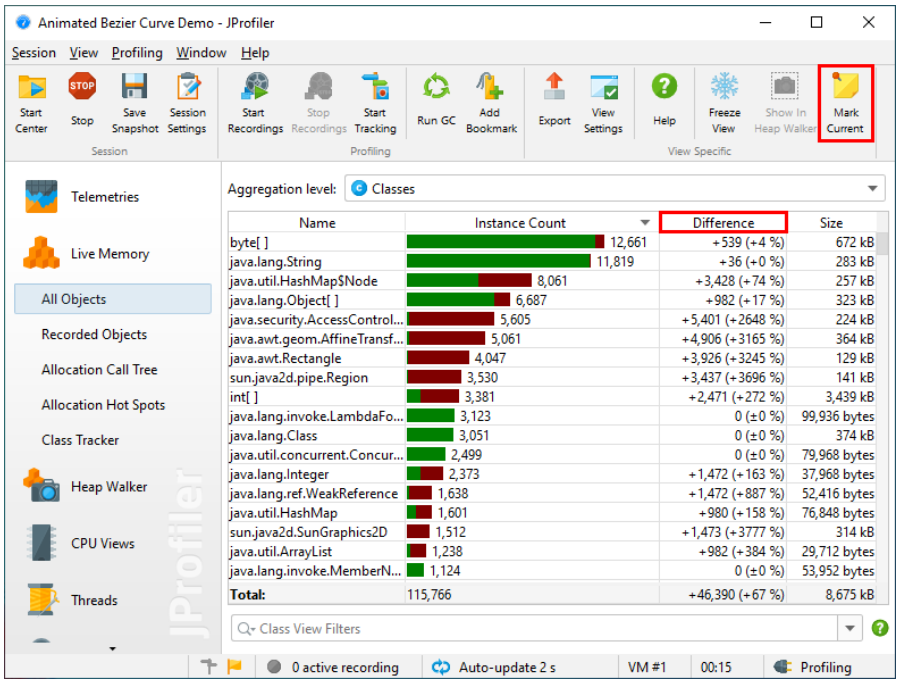
Tracking instance counts

To get an overview of what objects are on the heap, the "All objects" view shows you a histogram of all classes and their instance counts. The data that is shown in this view is not collected with allocation recording but by performing a mini heap snapshot that only calculates the instance counts. The larger the heap, the longer it takes to perform this operation, so the update frequency of the view is automatically lowered according to the measured overhead. When the view is not active, no data is collected and the view does not generate any overhead. As with most views that are updated dynamically, a *Freeze* tool bar button is available to stop updating the displayed data.

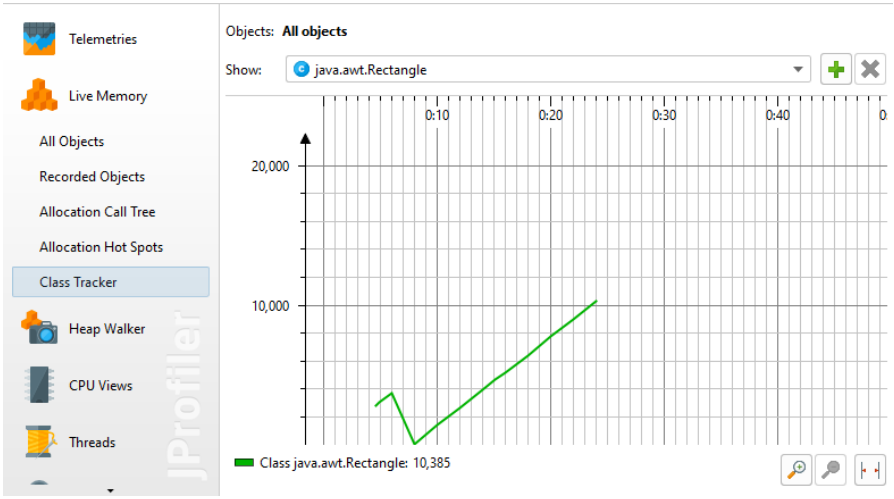


The "Recorded objects" view, on the other hand, only shows the instance counts for objects that have been allocated after you have started allocation recording. When you stop allocation recording, no new allocations are added, but garbage collection continues to be tracked. In this way you can see what objects remain on the heap for a certain use case. Note that objects may not be garbage collected for a long time. With the *Run GC* tool bar button you can speed up this process.

When looking for a memory leak, you often want to compare instance counts over time. To do that for all classes, you can use the differencing functionality of the view. With the *Mark Current* toolbar button, a *Difference* column is inserted and the histogram of the instance counts shows the baseline values at the time of the marking in green color.



For selected classes, you can also show a time-resolved graph with the *Add Selection to Class Tracker* action from the context menu.



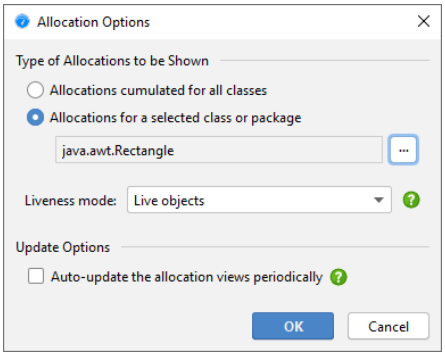
Allocation spots

When allocation recording is active, JProfiler takes note of the call stack each time an object is allocated. It does not use the exact call stack, for example from the stack-walking API, because that would be prohibitively expensive. Instead, the same mechanism is used that is configured for CPU profiling. This means that the call stack is filtered according to the call tree filters [p. 48] and that the actual allocation spot can be in a method that is not present in the call stack, because it is from an ignored or compact-filtered class. However, these changes are intuitively easy to understand: A compact-filtered method is responsible for all allocations that are made in further calls to compact-filtered classes.

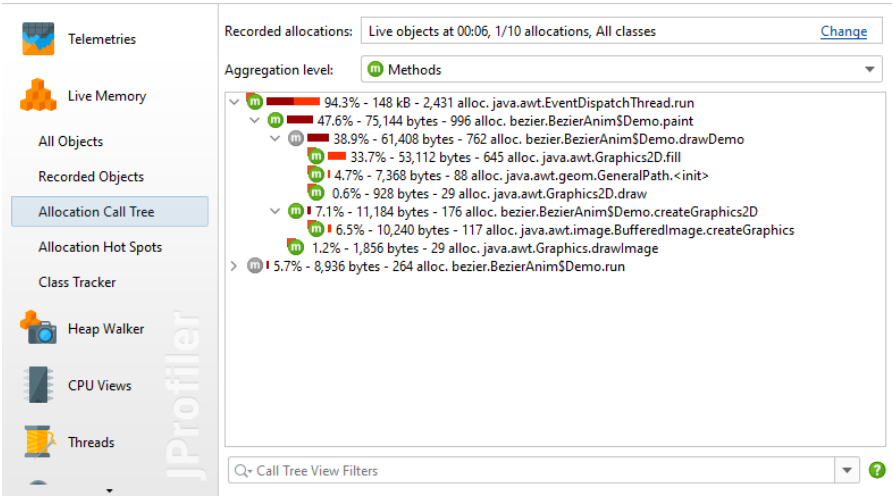
If you use sampling, the allocation spots become approximate and may be confusing. Unlike for time measurements, you often have a clear idea of where certain classes can be allocated and

where not. Because sampling paints a statistical rather than an exact picture, you may see allocation spots that are seemingly impossible, such as `java.util.HashMap.get` allocating one of your own classes. For any kind of analysis where exact numbers and call stacks are important, it is recommended to use allocation recording together with instrumentation.

Just like for CPU profiling, the allocation call stacks are presented as a call tree, only with allocation counts and allocated memory rather than invocation counts and time. Unlike for the CPU call tree, the allocation call tree is not displayed and updated automatically, because the calculation of the tree is more expensive. JProfiler can show you the allocation tree not only for all objects, but also for a selected class or package. Together with other options, this is configured in the options dialog that is shown after you ask JProfiler to calculate an allocation tree from the current data.

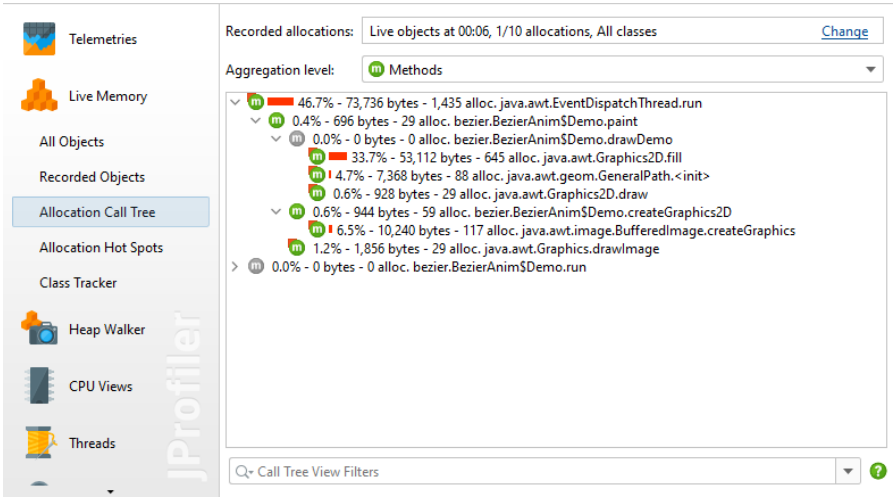


A useful property of the CPU call tree is that you can follow the cumulated time from top to bottom because each node contains the time that is spent in the child nodes. By default the allocation tree behaves in the same way, meaning that each node contains the allocations that are made by the child nodes. Even if allocations are only performed by leaf nodes deep down in the call tree, the numbers propagate up to the top. In this way, you can always see which path is worth investigating when opening branches of the allocation call tree. "Self-allocations" are those that are actually performed by a node and not by its descendants. Like in the CPU call tree, the percentage bar shows them with a different color.

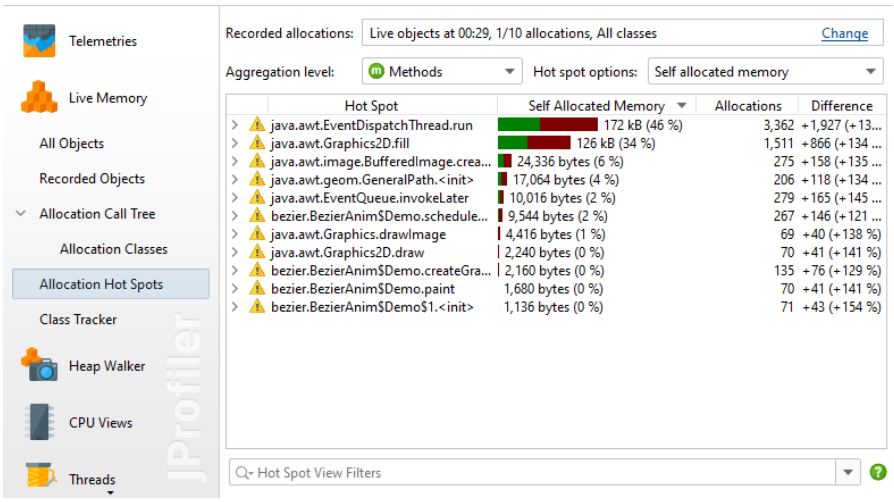


In the allocation call tree, there are often a lot of nodes where no allocations are performed at all, especially if you show allocations for a selected class. These nodes are only there to show you the call stack leading to the node where the actual allocation has taken place. Such nodes are called "bridge" nodes in JProfiler and are shown with a gray icon as you can see in the above

screen shot. In some cases, the cumulation of allocations can get in the way and you only want to see the actual allocation spots. The view settings of the allocation tree offers an option to show uncumulated numbers for that purpose. If activated, bridge nodes will always show zero allocations and have no percentage bar.



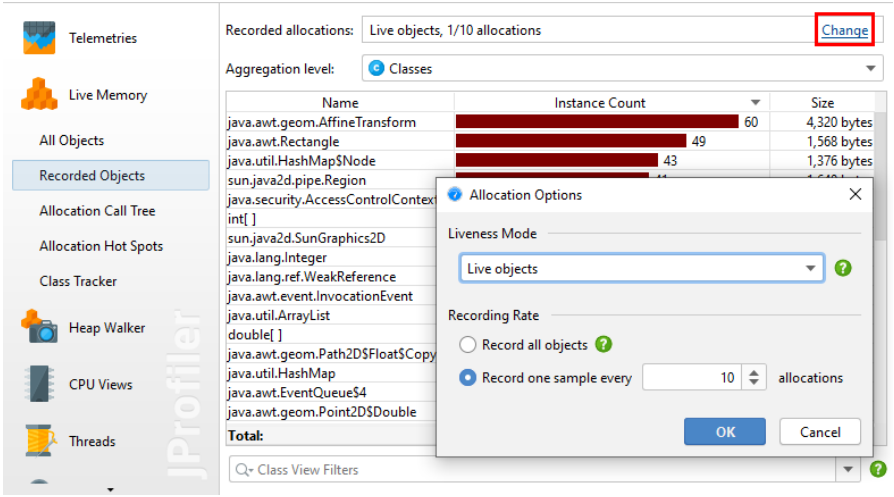
The allocation hot spots view is populated together with the allocation call tree and allows you to directly focus on the methods that are responsible for creating the selected classes. Like the recorded objects view, the allocation hot spots view supports marking the current state and observing the differences over time. A difference column is added to the view that shows how much the hot spots have changed since the time when the *Mark Current Values* action was invoked. Because the allocation views are not updated periodically by default, you have to click on the *Calculate* tool bar button to get a new data set that is then compared to the baseline values. Auto-update is available in the options dialog but not recommended for large heap sizes.



Allocation recording rate

Recording each and every allocation adds a significant overhead. In many cases, the total numbers for allocations are not important and relative numbers are sufficient to solve problems. That is why JProfiler only records every 10th allocation by default. This reduces the overhead to roughly 1/10 compared to recording all allocations. If you would like to record all allocations, or if even

less allocations are sufficient for your purpose, you can change the recording rate in the recorded objects view as well as the parameter dialog of the allocation call tree and hot spot views.

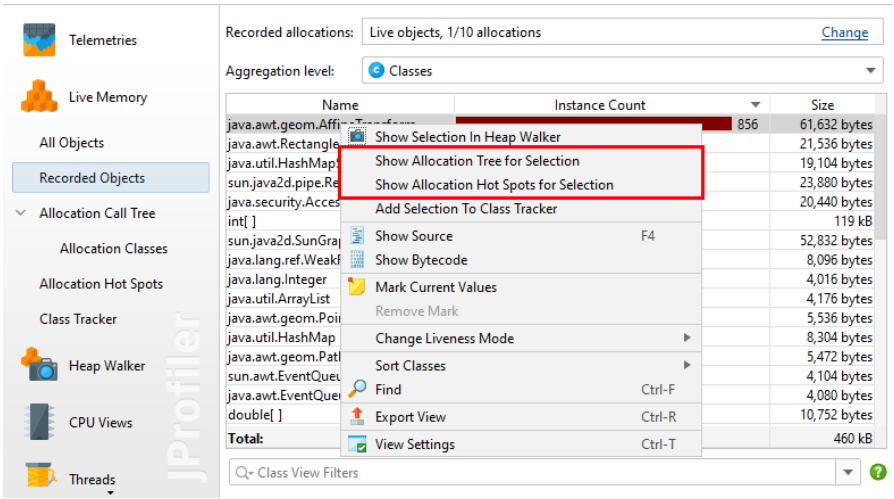


The setting can also be found on the "Advanced Settings->Memory profiling" step of the session settings dialog where it can be adjusted for offline profiling sessions.

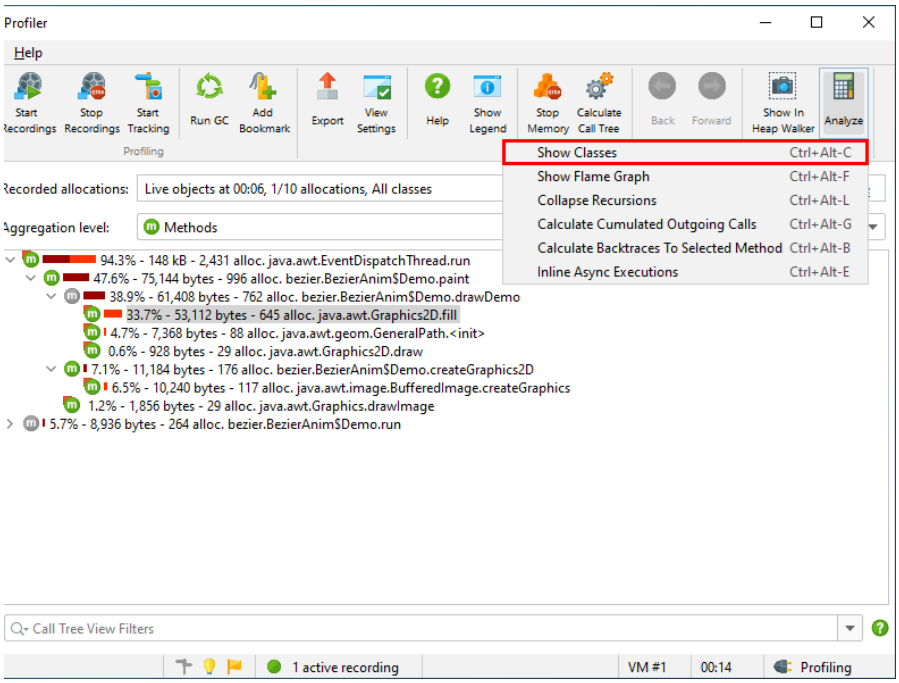
The allocation recording rate influences the VM telemetries for "Recorded objects" and "Recorded throughput" whose values will be measured at the configured fraction. When comparing snapshots [p. 118], the allocation rate of the first snapshot will be reported and other snapshots will be scaled accordingly, if necessary.

Analyzing allocated classes

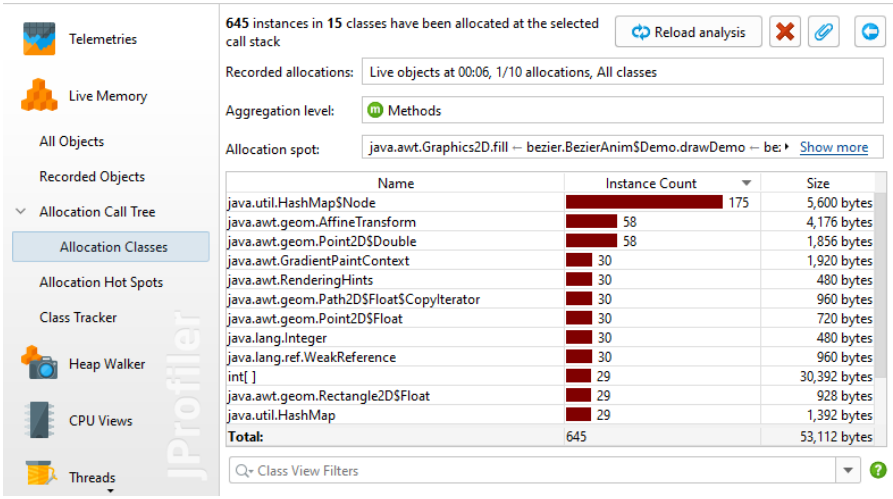
When calculating the allocation tree and allocation hot spot views, you have to specify the class or package whose allocations you want to see up-front. This works well if you already focused on particular classes, but is inconvenient when trying to find allocation hot spots without any pre-conceptions. One way is to start to look at the "Recorded objects" view and use the actions in the context menu for switching to the allocation tree or allocation hot spot views for the selected class or package.



Another way is to start with the allocation tree or allocation hot spots for all classes and use the *Show classes* action to show the classes for a selected allocation spot or allocation hot spot.



The histogram of the allocated classes is shown as a call tree analysis [p. 169]. This action also works from other call tree analyses.

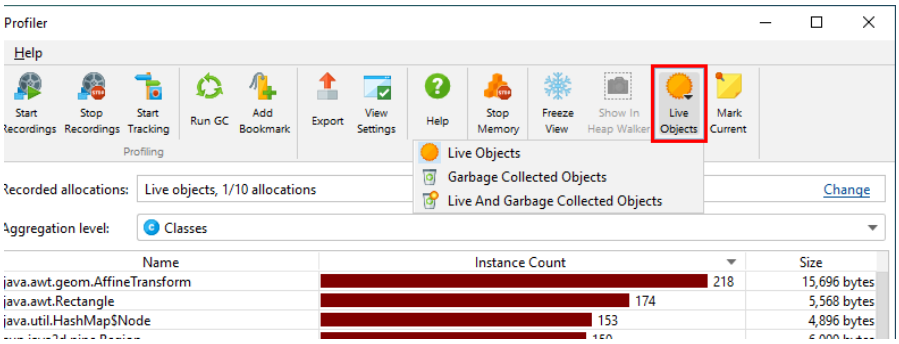


The classes analysis view is static and is not updated when the allocation tree and hot spot views are recalculated. The *Reload Analysis* action will first update the allocation tree and then recalculate the current analysis view from the new data.

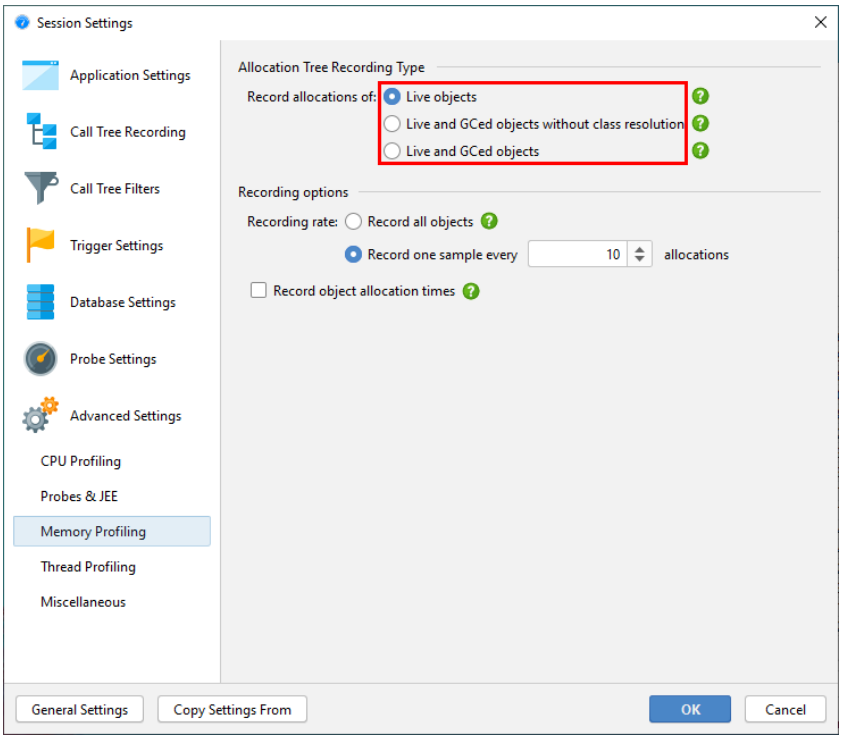
Analyzing garbage collected objects

Allocation recording cannot only show the live objects, but also keeps information on garbage collected objects. This is useful when investigating temporary allocations. Allocating a lot of temporary objects can produce significant overhead, so reducing the allocation rate often improves performance considerably.

To show garbage collected objects in the recorded objects view, change the liveness selector to either *Garbage collected objects* or *Live and garbage collected objects*. The options dialog of the allocation call tree and allocation hot spot views has an equivalent drop-down.



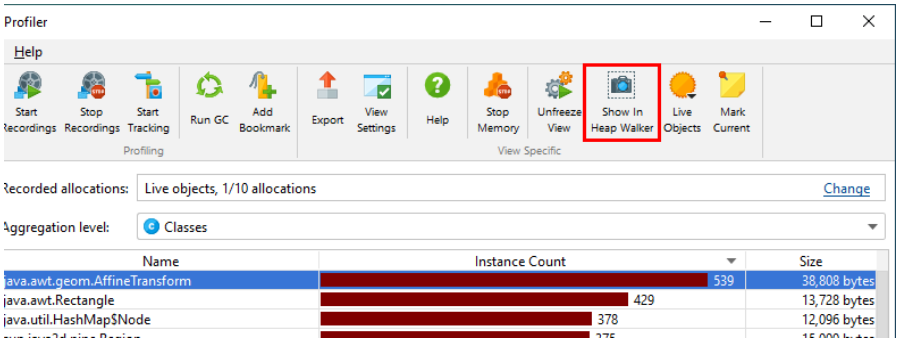
However, JProfiler does not collect allocation tree information for garbage collected objects by default, because the data for live objects only can be maintained with far less overhead. When switching the liveness selector in the "Allocation Call Tree" or "Allocation Hotspots" view to a mode that includes garbage collected objects, JProfiler suggests to change the recording type. This is a change in the profiling settings, so all previously recorded data will be cleared if you choose to apply the change immediately. If you would like to change this setting in advance, you can do so in "Advanced Settings" -> "Memory Profiling" in the session settings dialog.



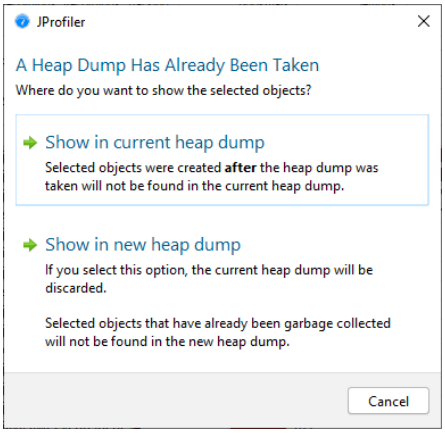
Next stop: heap walker

Any more advanced type of question will involve references between objects. For example, the sizes that are displayed in the recorded objects, allocation tree and allocation hot spot views are **shallow sizes**. They just include the memory layout of the class, but not any referenced classes. To see how heavy objects of a class really are, you often want to know the **retained size**, meaning the amount of memory that would be freed if those objects were removed from the heap.

This kind of information is not available in the live memory views, because it requires enumerating all objects on the heap and performing expensive calculations. That job is handled by the heap walker. To jump from a point of interest in the live memory views into the heap walker, the *Show in Heap Walker* tool bar button can be used. It will take you to the equivalent view in the heap walker.



If no heap snapshot is available, a new heap snapshot is created, otherwise JProfiler will ask you whether to use the existing heap snapshot.



In any case, it is important to understand that the numbers in the live memory views and in the heap walker will often be very different. Apart from the fact that the heap walker shows a snapshot at a different point in time than the live memory views, it also eliminates all unreferenced objects. Depending on the state of the garbage collector, unreferenced objects can occupy a significant portion of the heap.

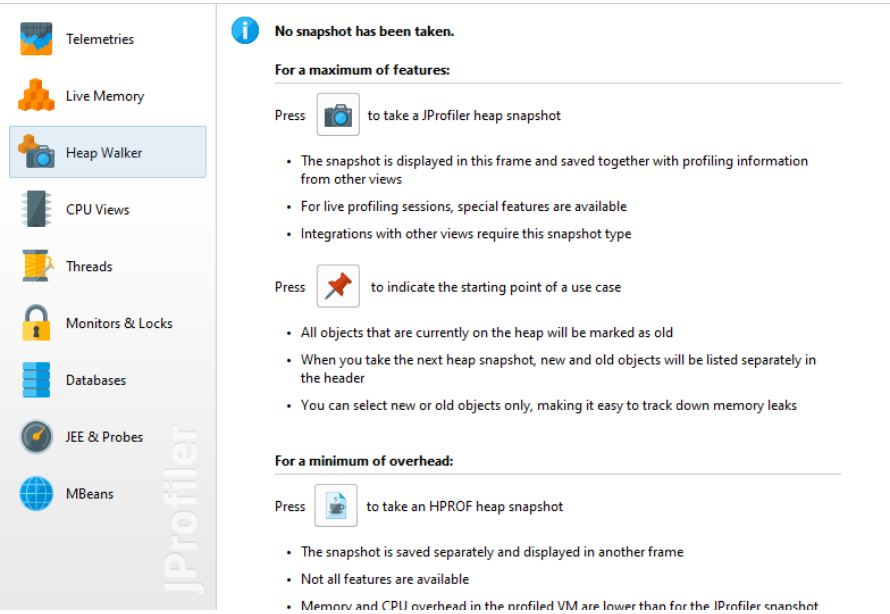
The Heap Walker

Heap snapshots

Any heap analysis that involves references between objects requires a heap snapshot, because it is not possible to ask the JVM what the incoming references to an object are - you have to iterate over the entire heap to answer that question. From that heap snapshot, JProfiler creates an internal database that is optimized for producing the data required for serving the views in the heap walker.

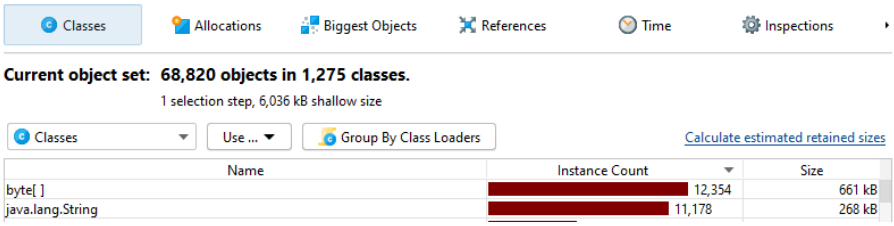
There are two sources of heap snapshots: JProfiler heap snapshots and HPROF/PHD heap snapshots. JProfiler heap snapshots support all available features in the heap walker. The profiling agent uses the profiling interface JVMTI to iterate over all references. If the profiled JVM is running on a different machine, all information is transferred to the local machine and further calculations are performed there. HPROF/PHD snapshots are created with a built-in mechanism in the JVM and are written to disk in a standard format that JProfiler can read. HPROF snapshots are provided by HotSpot JVMs and PHD snapshots are produced by Eclipse OpenJ9 JVMs.

On the overview page of the heap walker, you can choose if a JProfiler heap snapshot or an HPROF/PHD heap snapshot should be created. By default, the JProfiler heap snapshot is recommended. The HPROF/PHD heap snapshot is useful in special situations that are discussed in another chapter [p. 185].



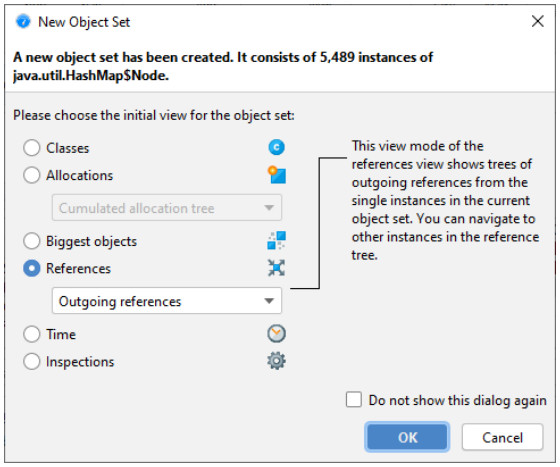
Selection steps

The heap walker consists of several views that show different aspects of a selected set of objects. Right after you take the heap snapshot, you are looking at all objects on the heap. Each view has navigation actions for turning some selected objects into the **current object set**. The header area of the heap walker shows information on how many objects are contained in the current object set.

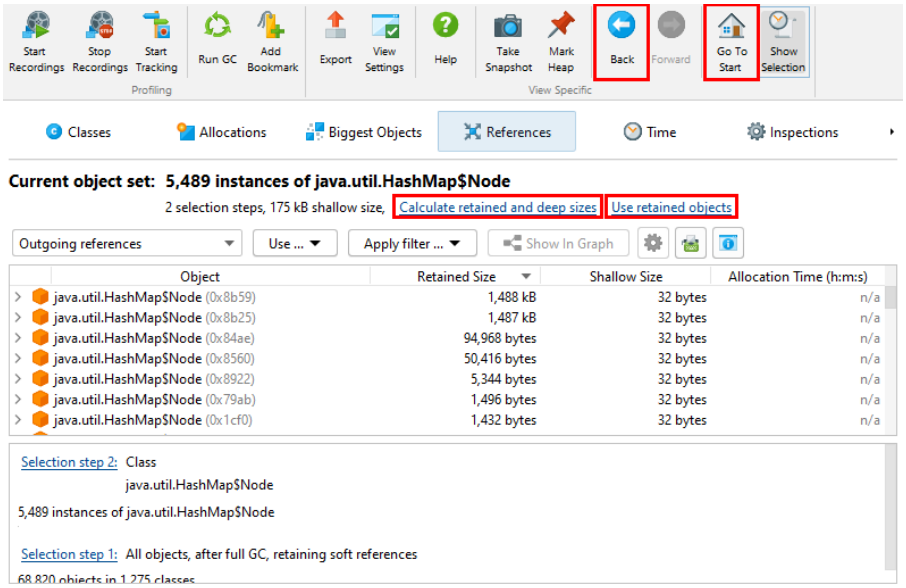


Initially, you are looking at the "Classes" view which is similar to the "All objects" view in the live memory section [p. 66]. By selecting a class and invoking *Use->Selected Instances*, you create a new object set that only contains instances of that class. In the heap walker, "using" always means creating a new object set.

For the new object set, showing the classes view of the heap walker would not be interesting, because it would effectively just filter the table to the previously selected class. Instead, JProfiler suggests another view with the "New object set" dialog. You can cancel this dialog to discard the new object set and return to the previous view. The outgoing references view is suggested, but you could also choose another view. This is just for the initially displayed view, you can switch views in the view selector of the heap walker afterwards.



The header area now tells you that there are two selection steps and includes links for calculating the retained and deep sizes or for using all objects that are retained by the current object set. The latter would add another selection step and suggest the classes view because there would likely be multiple classes in that object set.

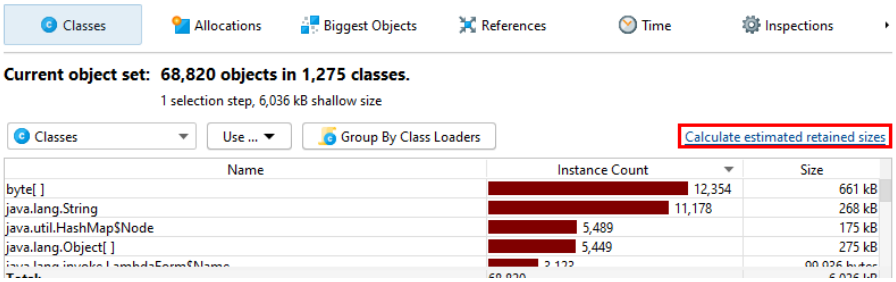


In the lower part of the heap walker, the selection steps up to this point are listed. Clicking on the hyperlinks will take you back to any selection step. The first data set can also be reached with the *Go To Start* button in the tool bar. The back and forward buttons in the tool bar are useful if you need to backtrack in your analysis.

Classes view

The view selector at the top of the heap walker contains five views that show different information for the current object set. The first one of those is the "Classes" view.

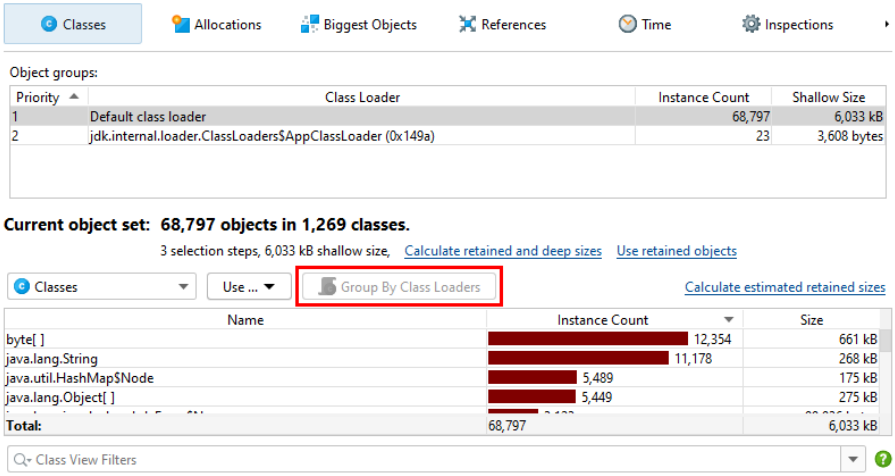
The classes view is similar to the "All objects" view in the live memory section and has an aggregation level chooser that can group classes into packages. In addition, it can show estimated retained sizes for classes. This is the amount of memory that would be freed if all instances of a class were removed from the heap. If you click on the *Calculate estimated retained sizes* hyperlink, a new *Retained Size* column is added. The displayed retained sizes are estimated lower bounds, calculating the exact numbers would be too slow. If you really need an exact number, select the class or package of interest and use the *Calculate retained and deep sizes* hyperlink in the header of the new object set.



Based on your selection of one or more classes or packages, you can select the instances themselves, the associated `java.lang.Class` objects, or all retained objects. Double-clicking is the quickest selection mode and uses the selected instances. If multiple selection modes are available, as in this case, a *Use* drop-down menu is shown above the view.

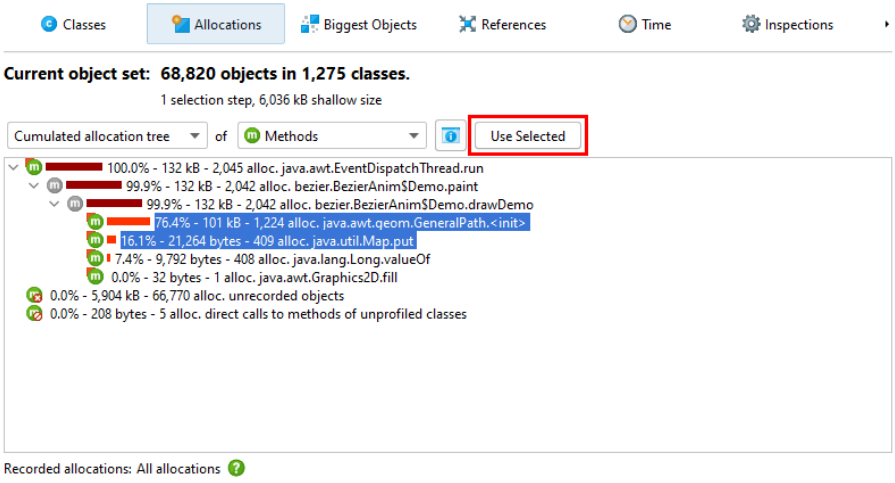
When solving class loader-related problems, you often have to group instances by their class loader. The *Inspections* tab offers a "Group by class loaders" inspection that is made available

on the classes view, because it is especially important in that context. If you execute that analysis, a grouping table at the top shows all class loaders. Selecting a class loader filters the data accordingly in the view below. The grouping table remains in place when you switch to the other views of the heap walker until you perform another selection step. Then, the class loader selection becomes part of that selection step.



Allocation recording views

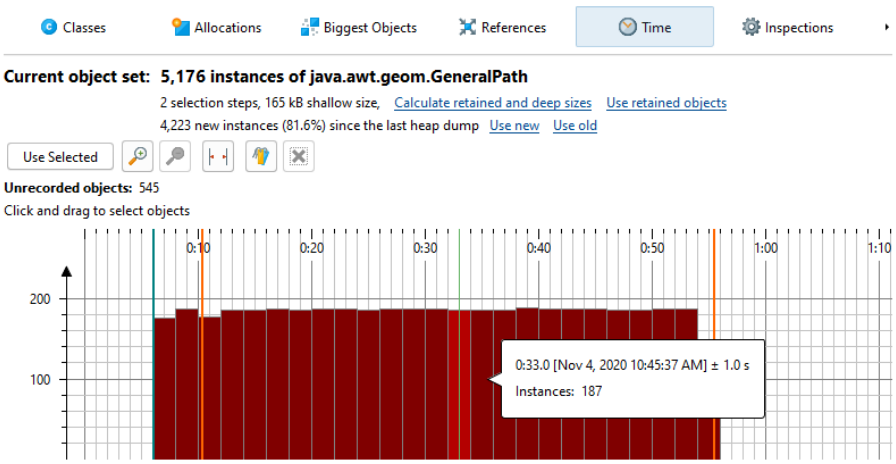
The information where objects have been allocated can be important when narrowing down suspects for a memory leak or when trying to reduce memory consumption. For JProfiler heap snapshots, the "Allocations" view shows the allocation call tree and the allocation hot spots for those objects where allocations have been recorded. Other objects are grouped in the "unrecorded objects" node in the allocation call tree. For HPROF/PHD snapshots, this view is not available.



Like in the classes view, you can select multiple nodes and use the *Use Selected* button at the top to create a new selection step. In the "Allocation hot spots" view mode, you can also select nodes in the back traces. This will only select objects in the associated top-level hot spot that have been allocated on a call stack that ends with the selected back trace.

Another piece of information that JProfiler can save when recording allocations is the time when an object was allocated. The "Time" view in the heap walker shows a histogram of the allocation

times for all recorded instances in the current object set. You can click and drag to select one or multiple intervals and then create a new object set with the *Use Selected* button.



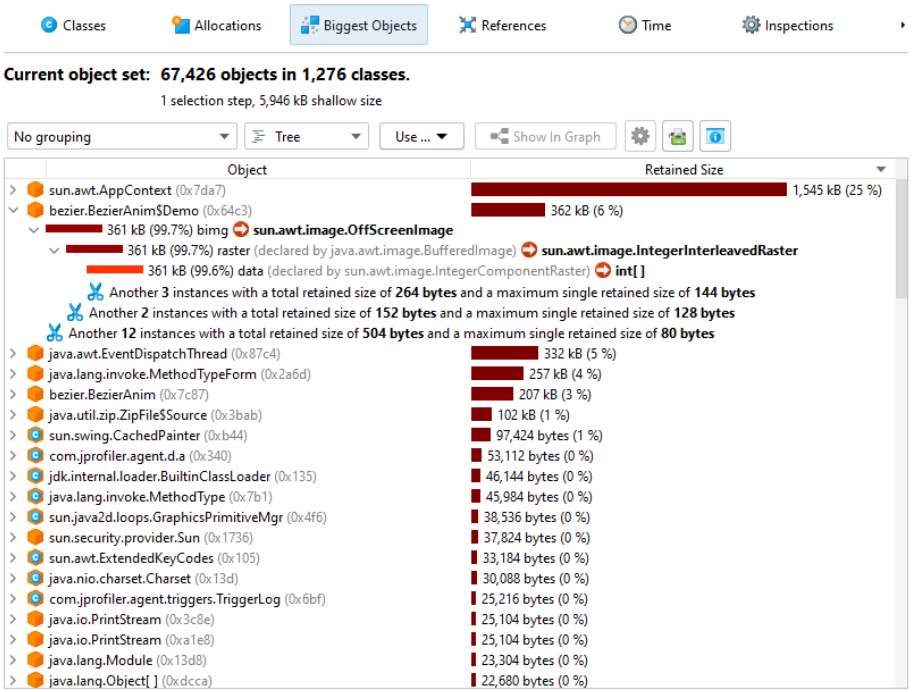
For a more precise selection of a time interval, you can specify a range of bookmarks [p. 43]. All objects between the first and last selected bookmark will then be marked.

In addition to the time view, allocation times are displayed as a separate column in the reference views. However, allocation time recording is not enabled by default. You can switch it on directly in the time view or edit the setting in *Advanced Settings -> Memory Profiling* in the session settings dialog.

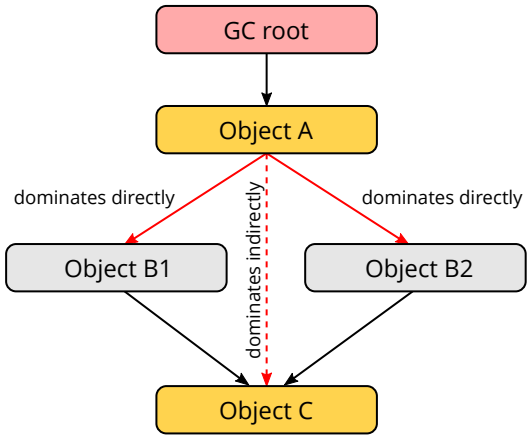
Biggest objects view

The biggest objects view shows a list of the most important objects in in the current object set. "Biggest" in this context means the objects that would free most memory if they were removed from the heap. That size is called the **retained size**. In contrast, the **deep size** is the total size of all objects that are reachable through strong references.

Each object can be expanded to show outgoing references to other objects that are retained by this object. In this way, you can recursively expand the tree of retained objects that would be garbage collected if one of the ancestors were to be removed. This kind of tree is called a "dominator tree". The information displayed for each object in this tree is similar to the outgoing reference view except that only dominating references are displayed.



Not all dominated objects are directly referenced by their dominators. For example, consider the references in the following figure:

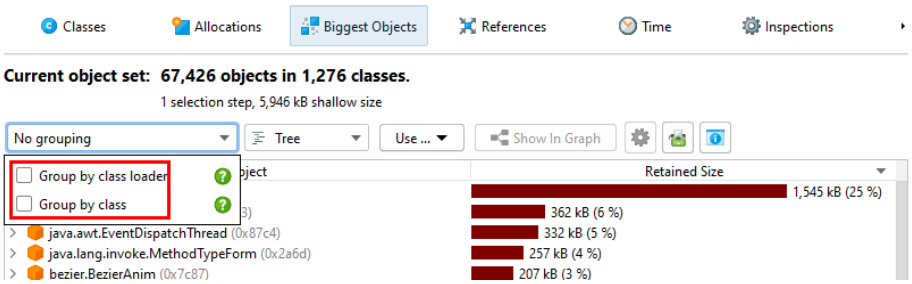


Object A dominates objects B1 and B2 and it does not have a direct reference to object C. Both B1 and B2 reference C. Neither B1 nor B2 dominates C, but A does. In this case, B1, B2 and C are listed as direct children of A in the dominator tree, and C will not be listed a child of B1 and B2. For B1 and B2, the field names in A by which they are held are displayed. For C, "[transitive reference]" is displayed on the reference node.

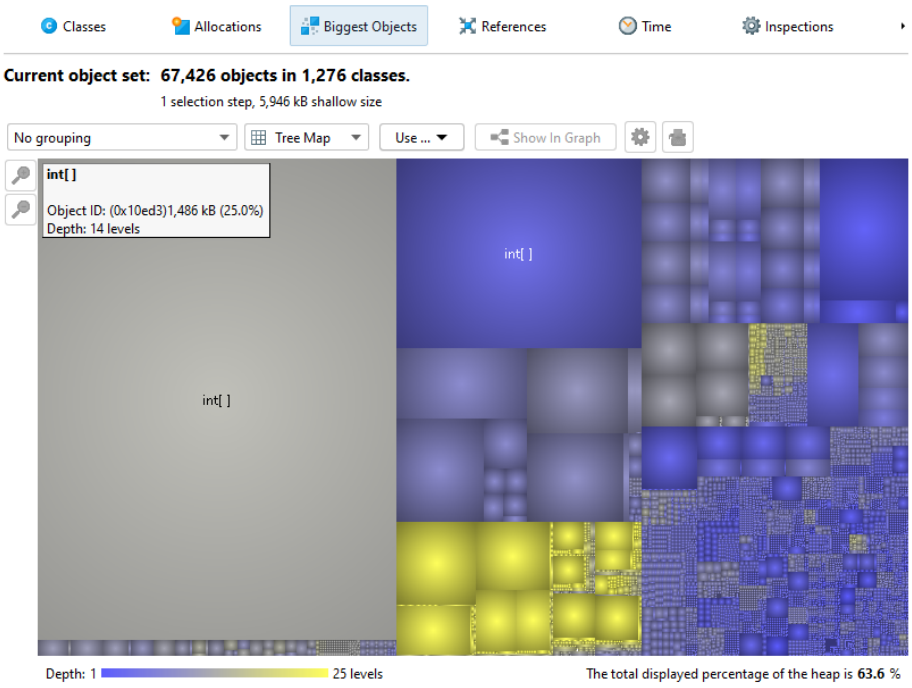
At the left side of each reference node in the dominator tree, a size bar shows what percentage of the retained size of the top-level object is still retained by the target object. The numbers will decrease as you drill down further into the tree. In the view settings, you can change the percentage base to the total heap size.

The dominator tree has a built-in cutoff that eliminates all objects that have a retained size that is lower than 0.5% of the retained size of the parent object. This is to avoid excessively long lists of small dominated objects that distract from the important objects. If such a cutoff occurs, a special "cutoff" child node will be shown that notifies you about the number of objects that are not shown on this level, their total retained size and the maximum retained size of the single objects.

Instead of showing single objects, the dominator tree can also group biggest objects into classes. The grouping drop-down at the top of the view contains a check box that activates this display mode. In addition, you can add a class loader grouping at the top level. The class loader grouping is applied after the biggest objects are calculated and shows who loaded the classes of the biggest objects. If you want to analyze the biggest objects for one particular class loader instead, you can use the "Group by class loader" inspection first.



The view mode selector above the biggest objects view allows you to switch to an alternate visualization: A tree map that shows all dominated objects as a set of nested rectangles.



In the tree map, each rectangle represents a dominated object with an area proportional to its retained size. In contrast to the tree, the tree map gives you a flattened perspective of all leafs in the dominator tree. If you are mostly interested in big arrays, you can use the tree map in order to find them quickly without having to dig into the branches of the tree. Also, the tree map

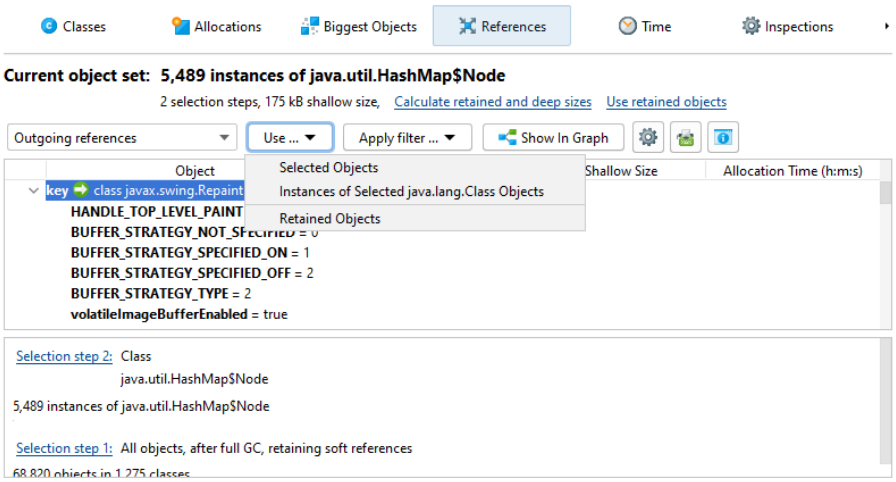
gives you an overall impression of the relative importance of dominated objects and the object size distribution on the heap.

At the bottom right of the tree map you can see the total percentage of the entire heap that is represented by the tree map. If you have not zoomed in, the remaining part of the heap is dominated by objects that have not made it into the list of biggest objects due to the internal threshold for retained sizes.

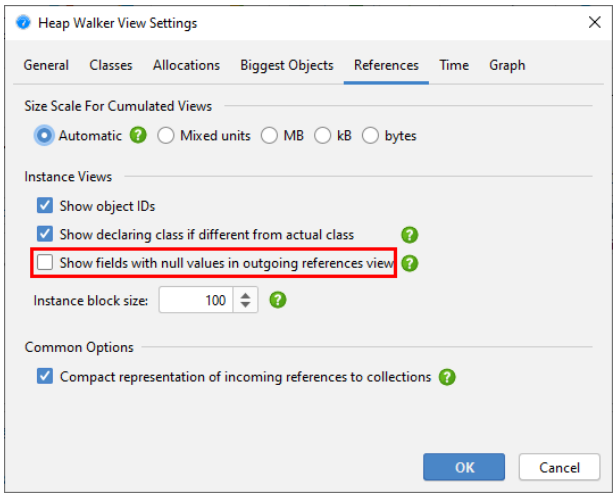
Reference views

Unlike the previous views, the reference views are only available if you have performed at least one selection step. For the initial object set these views are not useful, because the incoming and outgoing reference views show all individual objects and the merged reference views can only be interpreted for a focused set of objects.

The outgoing references view is similar to the view that a debugger would show in an IDE. When opening an object, you can see the primitive data and references to other objects. Any reference type can be selected as a new object set and you can select multiple objects at once. Like in the classes view, you can select retained objects or associated `java.lang.Class` objects. If the selected object is a standard collection, you can also select all contained elements with a single action. For class loader objects, there is an option to select all loaded instances.

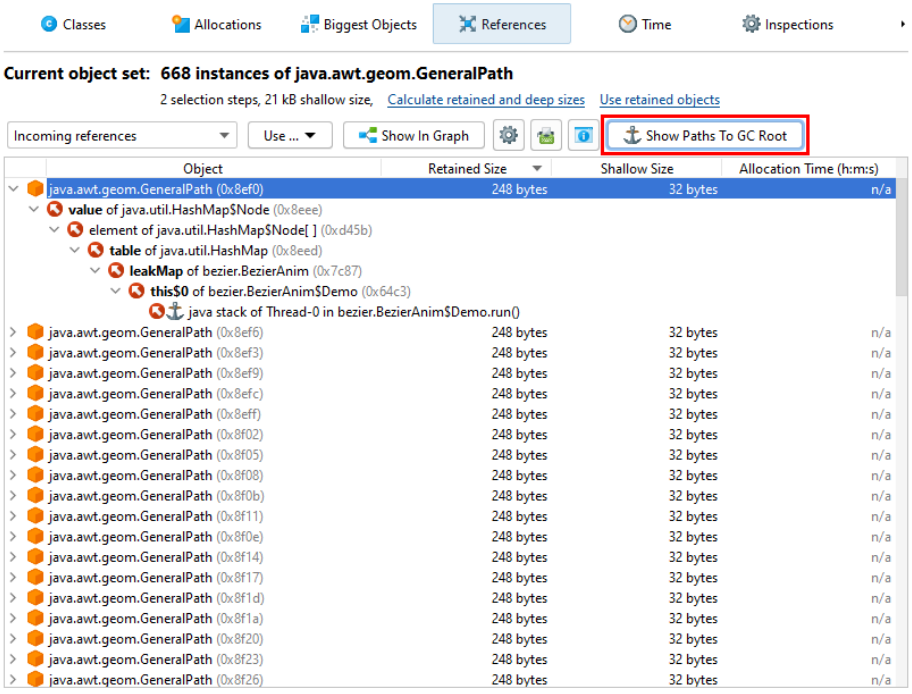


Fields with null references are not shown by default because that information may be distracting for a memory analysis. If you want to see all fields for debugging purposes, you can change this behavior in the view settings.



Beside the simple selection of displayed instances, the outgoing references view has powerful filtering capabilities [p. 189]. For live sessions, both outgoing and incoming reference views have advanced manipulation and display functionality that is discussed in the same chapter.

The incoming references view is the main tool for solving memory leaks. To find out why an object is not garbage collected, the *Show Paths To GC Root* button will find reference chains to garbage collector roots. The chapter on memory leaks [p. 192] has detailed information on this important topic.



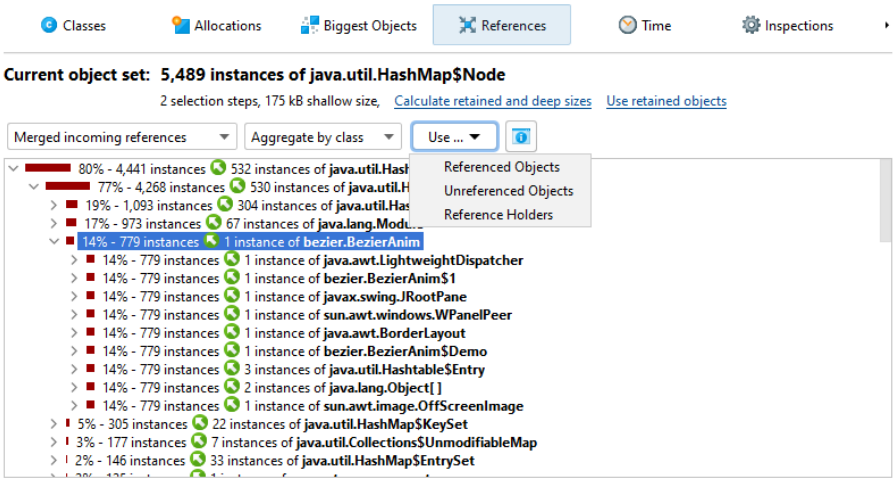
Merged references

Checking references for a lot of different objects can be tedious, so JProfiler can show you the merged outgoing and incoming references of all objects in the current object set. By default, the references are aggregated by classes. If instances of a class are referenced by other instances of the same class, a special node is inserted that shows the original instances plus the instances

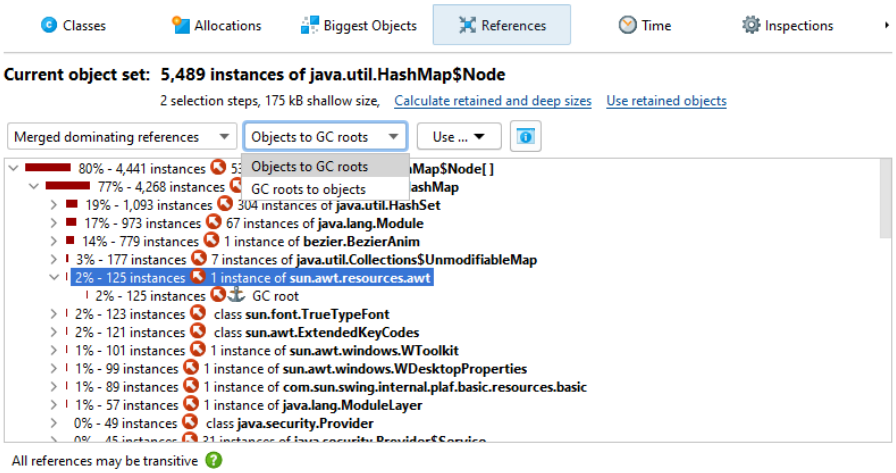
from these class-recursive references. This mechanism automatically collapses internal reference chains in common data structures, such as in a linked list.

You can also choose to show the merged references grouped by field. In that case, each node is a reference type, such as a particular field of a class or the content of an array. For standard collections, internal reference chains that would break cumulation are compacted, so you see reference types like "map value of java.lang.HashMap". Unlike for class aggregation, this mechanism only works for explicitly supported collections from the standard library of the JRE.

In the "Merged outgoing references" view, the instance counts refer to the referenced objects. In the "Merged incoming references" view, you see two instance counts on each row. The first instance count shows how many instances in the current object set are referenced along this path. The bar icon at the left side of the node visualizes this fraction. The second instance count after the arrow icon refers to the objects that hold the references to the parent node. When performing a selection step, you can choose whether you want to select objects from the current object set that are referenced in the selected way or if you are interested in the objects with the selected reference - the reference holders.



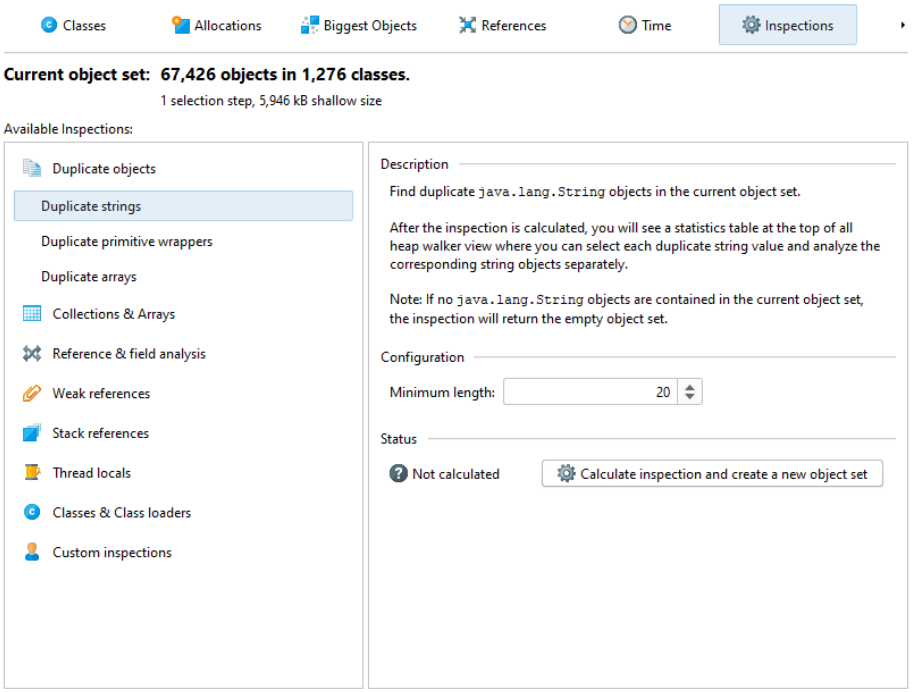
With the "Merged dominating references" view you can find out which references must be removed so that some or all of the objects in the current object set can be garbage collected. The dominating reference tree can be interpreted as the merged inverse of the dominator tree in the biggest objects view, aggregated for classes. The reference arrows may not express a direct reference between the two classes, but there may be other classes in between that hold non-dominating references. In the case of multiple garbage collector roots, no dominating references may exist for some or all objects in the current object set.



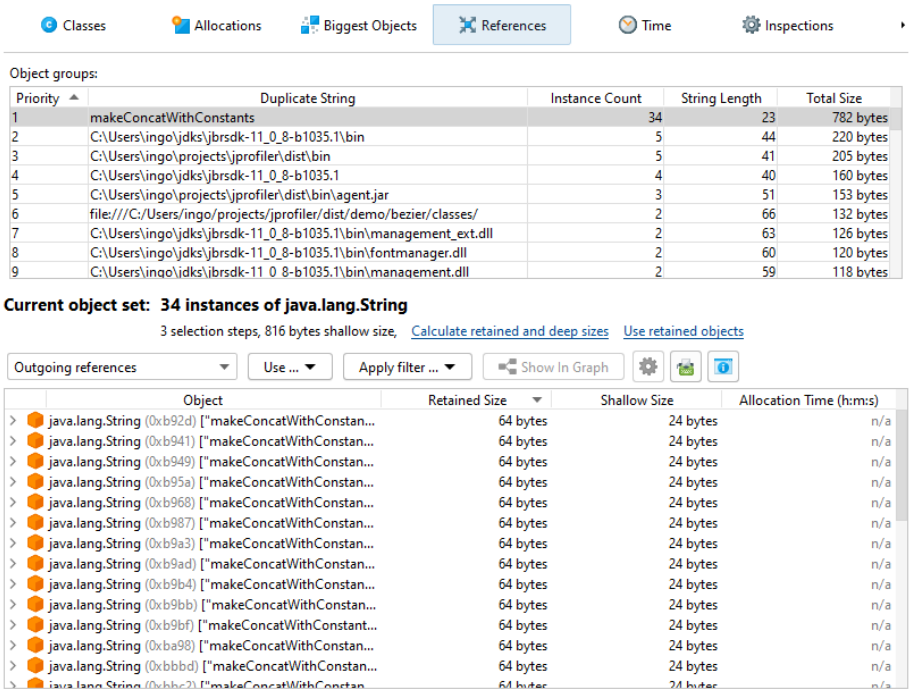
By default, the "Merged dominating references" view shows incoming dominating references and by opening the tree, you can reach the objects that are held by the GC roots. Sometimes, the reference tree may lead to the same root objects along many different paths. By choosing the "GC roots to objects" view mode in the drop down at the top of the view, you can see the reverse perspective where the roots are at the top level and the objects in the current object set are in the leaf nodes. In that case, the references go from the top level towards the leaf nodes. Which perspective is better depends on whether the references you want to eliminate are close to the current object set or close to the GC roots.

Inspections

The "Inspections" view does not show data by itself. It presents a number of heap analyses that create new object sets according to rules that are not available in the other views. For example, you may want to see all objects that are retained by a thread local. This would be impossible to do in the reference views. Inspections are grouped into several categories and explained in their descriptions.



An inspection can partition the calculated object set into groups. Groups are shown in a table at the top of the heap walker. For example, the "Duplicate strings" inspection shows the duplicate string values as groups. If you are in the reference view, you can then see the `java.lang.String` instances with the selected string value below. Initially, the first row in the group table is selected. By changing the selection, you change the current object set. The *Instance Count* and *Size* columns of the group table tell you how large the current object set will be when you select a row.



The group selection is not a separate selection step in the heap walker, but it becomes part of the selection step made by the inspection. You can see the group selection in the selection step pane at the bottom. When you change the group selection, the selection step pane is updated immediately.

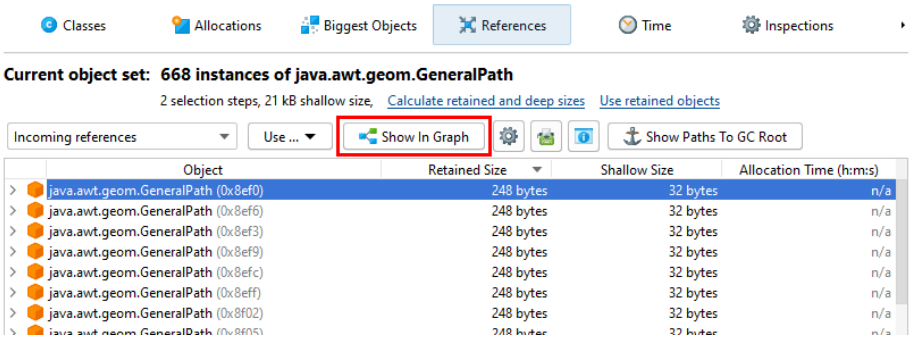
Each inspection that creates groups decides which groups are most important in the context of the inspection. Because this does not always correspond to the natural sort order of one of the other columns, the *Priority* column in the group table contains a numeric value that enforces the sort order for the inspection.

Inspections can be expensive to calculate for large heaps, so the results are cached. In this way, you can go back in the history and look at the results of previously calculated inspections without waiting.

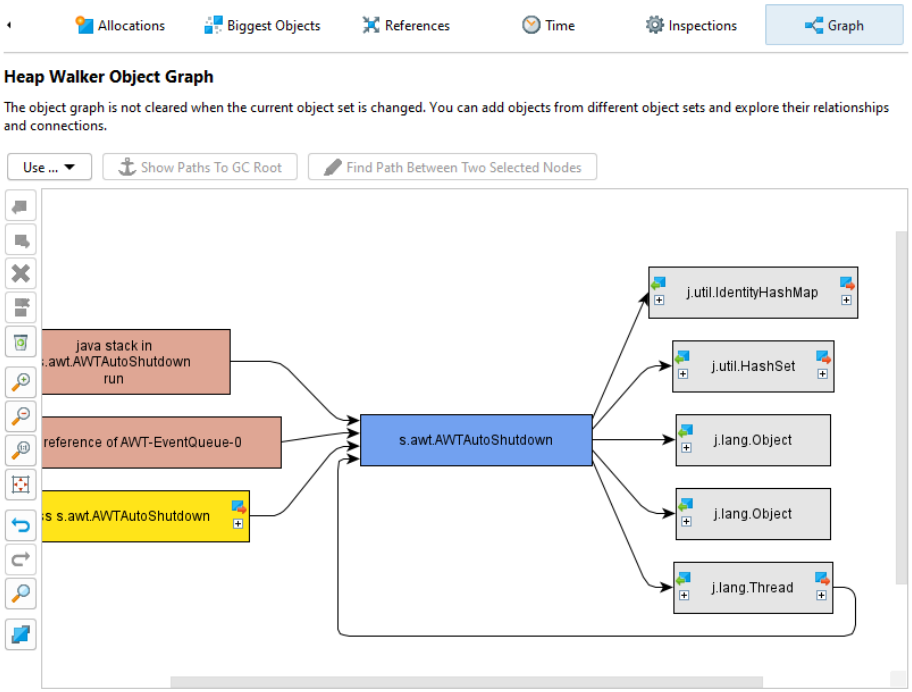
Heap walker graph

The most realistic representation of instances together with their references is a graph. While the graph has a low visual density and is impractical for some types of analyses, it still is the best way to visualize relationships between objects. For example, circular references are difficult to interpret in a tree, but immediately evident in a graph. Also, it may be beneficial to see incoming and outgoing references together, which is impossible in a tree structure where you can see either one or the other.

The heap walker graph does not automatically show any objects from the current object set, nor is it cleared when you change the current object set. You manually add selected objects to the graph from the outgoing references view, the incoming references view or the biggest objects view by selecting one or more instances and using the *Show In Graph* action.

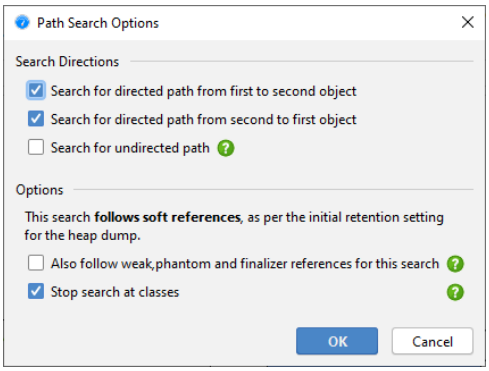


Package names in the graph are shortened by default. Like in the CPU call graph, you can enable the full display in the view settings. References are painted as arrows. If you move the mouse over the reference, a tooltip window will be displayed that shows details for the particular reference. Instances that were manually added from the reference views have a blue background. The more recently an instance has been added, the darker the background color. Garbage collector roots have a red background and classes have a yellow background.



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 direct outgoing references for that object, depending on the direction you're moving in. With the expansion controls on the left and right sides of an instance you can selectively open incoming and outgoing references. If you need to backtrack, use the undo functionality to restore previous states of the graph, so you don't get distracted by too many nodes. To trim the graph, there are actions for removing all unconnected nodes or even for removing all objects.

Like in the incoming references view, the graph has a *Show Path To GC Root* button that will expand one or more reference chains to a garbage collector root [p. 192] if available. In addition, there is a *Find Path Between Two Selected Nodes* action that is active if two instances are selected. It can search for directed and undirected paths and optionally also along weak references. If a suitable path is found, it is shown in red.

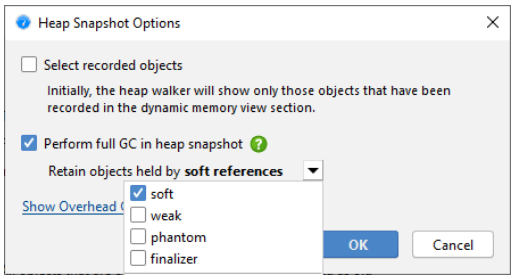


Initial object set

When you take a heap snapshot, you can specify options that control the initial object set. If you have recorded allocations, the *Select recorded objects* check box restricts the initially displayed

objects to those that have been recorded. The numbers will usually differ from those in the live memory views, because unreferenced objects are removed by the heap walker. Unrecorded objects are still present in the heap snapshot, they are just not displayed in the initial object set. With further selection steps you can reach unrecorded objects.

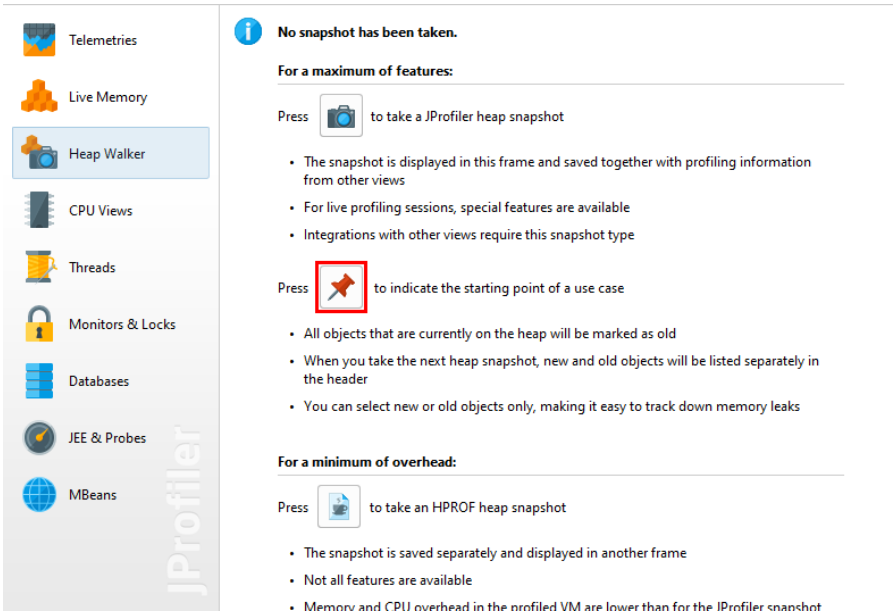
In addition, the heap walker performs a garbage collection and removes weakly referenced objects, except for soft references. This is usually desirable because weakly referenced objects are distracting when looking for memory leaks where only strongly referenced objects are relevant. However, in those cases where you are interested in weakly referenced objects you can tell the heap walker to retain them. The four weak reference types in the JVM are "soft", "weak", "phantom" and "finalizer" and you can chose which of them should be sufficient for retaining an object in the heap snapshot.



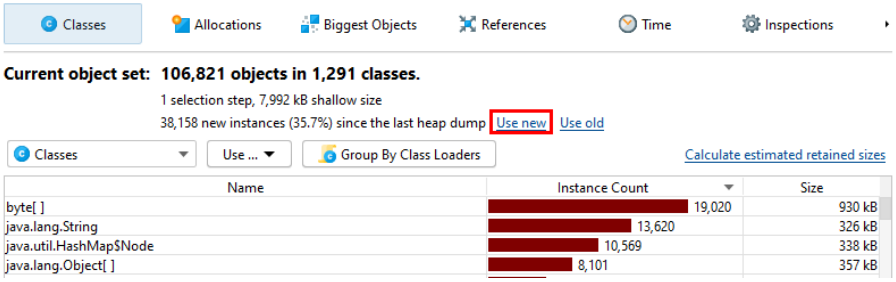
If present, weakly referenced objects can be selected or removed from the current object set by using the "Weak reference" inspections in the heap walker.

Marking the heap

Often you want to look at the objects that have been allocated for a particular use case. While you could do this by starting and stopping allocation recording around that use case, there is a much better way that has a lot less overhead and preserves the allocation recording feature for other purposes: The *Mark Heap* action that is advertised on the heap walker overview and that is also available in the *Profiling* menu or as a trigger action marks all objects on the heap as "old". When you take the next heap snapshot, it is now clear what the "new" objects should be.



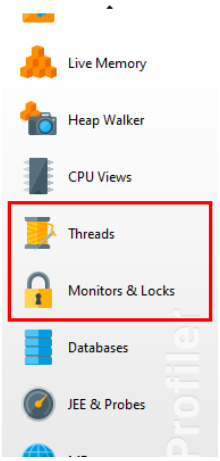
If there was a previous heap snapshot or a mark heap invocation, the title area of the heap walker shows the new instance count and two links titled *Use new* and *Use old* that allow you to select either the instances that have been allocated since that point in time, or the surviving instances that were allocated before. This information is available for each object set, so you can drill down first and select new or old instances later on.



Thread Profiling

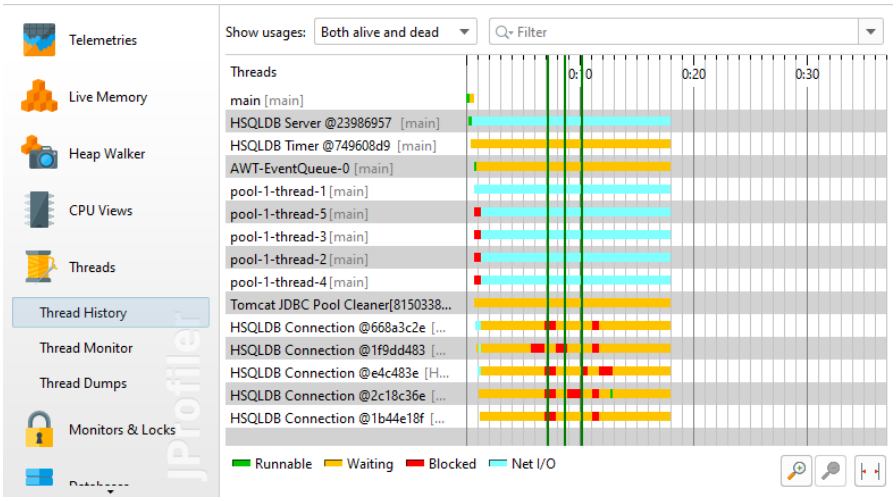
Using threads incorrectly can create many different kinds of problems. Too many active threads can result in thread starvation, threads can block each other and impact the liveness of your application or acquiring locks in the wrong order can lead to deadlocks. In addition, information about threads is important for debugging purposes.

In JProfiler, thread profiling is split into two view sections, the "Threads" section deals with the life-cycle of threads and with capturing thread dumps. The "Monitors & locks" section offers functionality for analyzing the interaction of multiple threads.



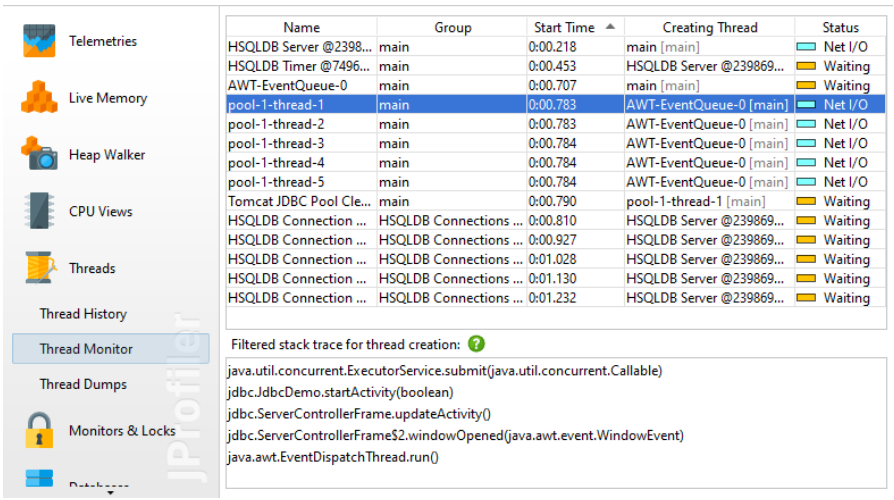
Inspecting threads

The thread history view shows each thread as a colored row in a time-line where the color indicates the recorded thread status. Threads are sorted by their creation time and can be filtered by name. When monitor events have been recorded, you can hover over parts of a thread where it was in the "Waiting" or "Blocked" state and see the associated stack trace with a link into the monitor history view.

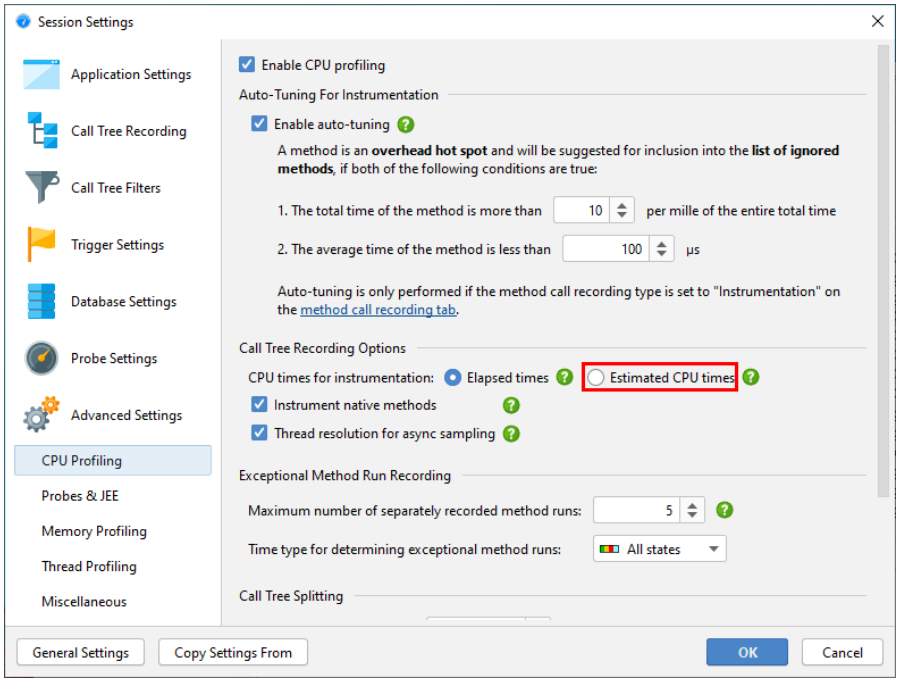


A tabular view of all threads is available in the thread monitor view. If CPU recording is active while a thread is being created, JProfiler saves the name of the creating thread and displays it in the table. At the bottom, the stack trace of the creating thread is shown. For performance reasons, no actual stack trace is requested from the JVM, but the current information from CPU

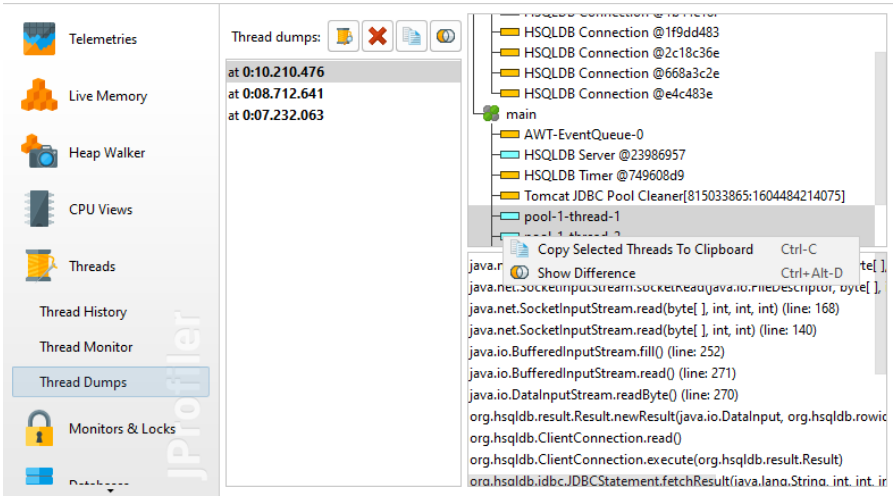
recording is used. This means that the stack traces will only show those classes that satisfy the filter settings for call tree collection.



If you enable the recording of estimated CPU times in the profiling settings, a *CPU Time* column is added to the table. CPU time is only measured when you record CPU data.



Like most debuggers, JProfiler can also take thread dumps. The stack traces of thread dumps are the full stack traces provided by the JVM and do not depend on CPU recording. Different thread dumps can be compared in a diff viewer when you select two thread dumps and click the *Show Difference* button. It is also possible to compare two threads from a single thread dump by selecting them and choosing *Show Difference* from the context menu.

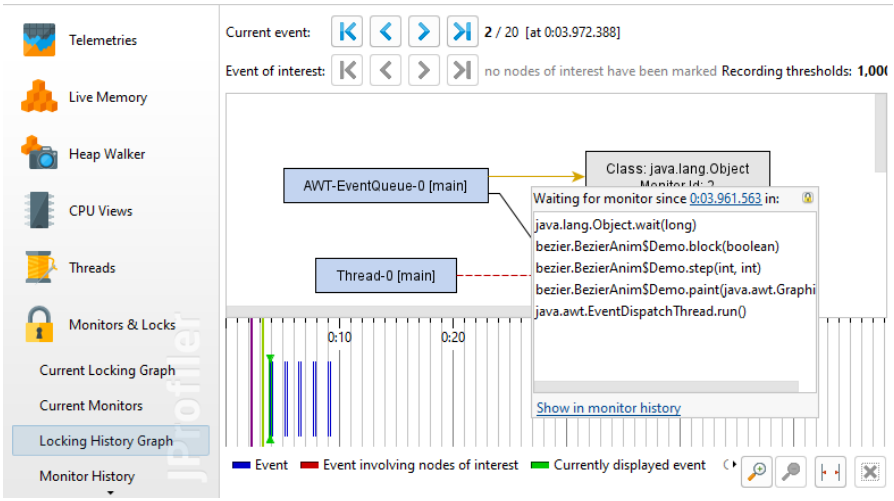


Thread dumps can also be taken with the "Trigger thread dump" trigger action or via the API.

Analyzing locking situations

Every Java object has an associated monitor that can by used for two synchronization operations: A thread can wait on a monitor until another thread issues a notification on it, or it can acquire a lock on a monitor, possibly blocking until another thread has given up the ownership of the lock. In addition, Java offers classes in the `java.util.concurrent.locks` package for implementing more advanced locking strategies. Locks in that package do not use monitors of objects but a different native implementation.

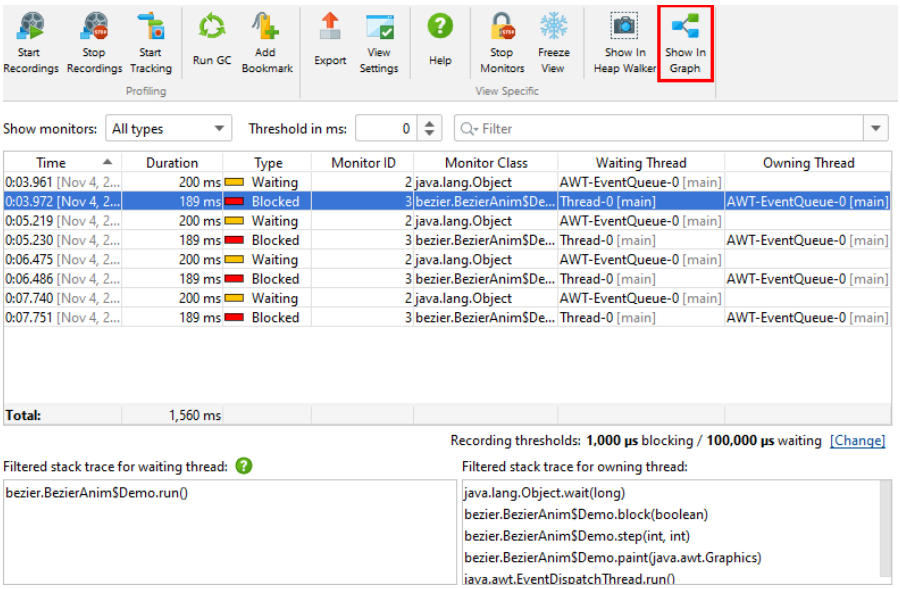
JProfiler can record locking situations for both of the above mechanisms. In a locking situation, there are one or multiple threads, a monitor or an instance of `java.util.concurrent.locks.Lock` as well as a waiting or blocking operation that takes a certain amount of time. These locking situations are presented in a tabular fashion in the monitor history view, and visually in the locking history graph.



The locking history graph focuses on the entire set of relationships of all involved monitors and threads rather than the duration of isolated monitor events. Threads and monitors participating in a locking situation are painted as blue and gray rectangles, if they are part of a deadlock, they are painted in red. Black arrows indicate ownership of a monitor, yellow arrows extend from waiting threads to the associated monitors, while a dashed red arrow indicates that a thread

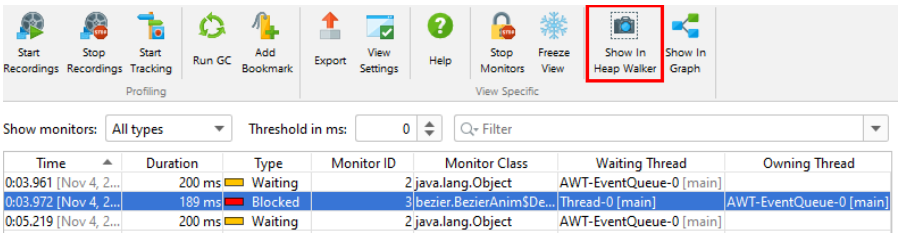
wants to acquire a monitor and is currently blocking. Stack traces are available when hovering over blocking or waiting arrows if CPU data has been recorded. Those tool tips contain hyperlinks that take you to the corresponding row in the monitor history view.

The tabular monitor history view shows monitor events. They have a duration that is displayed as a column so you can find the most important events by sorting the table. For any selected row in the tabular view, you can jump to the graph with the *Show in Graph* action.



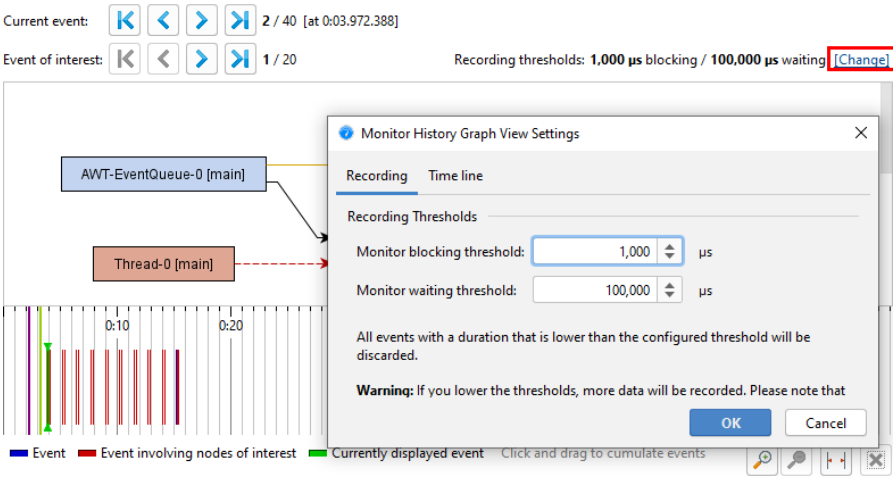
Each monitor event has an associated monitor. The *Monitor Class* column shows the class name of the instance whose monitor is used, or "[raw monitor]" if no Java object is associated with the monitor. In any case, monitors have a unique ID that is displayed in a separate column, so you can correlate the usage of the same monitor over multiple events. Each monitor event has a waiting thread that is performing the operation and optionally an owning thread that is blocking the operation. If available, their stack traces are shown in the lower part of the view.

If you have further questions about a monitor instance, the *Show in Heap Walker* action in both monitor history view and locking history graph provides a link into the heap walker and selects the monitor instance as a new object set.



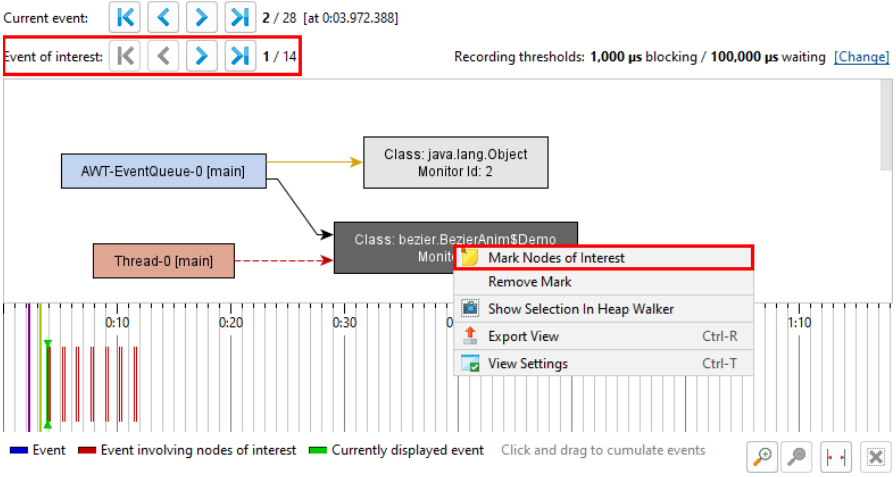
Limiting the events of interest

One fundamental problem with analyzing monitor events is that applications may generate monitor events at an extraordinary rate. That is why JProfiler has default thresholds for waiting and blocking events below which events are immediately discarded. These thresholds are defined in the view settings and can be increased in order to focus on longer events.



To the recorded events, you can further apply filters. The monitor history view offers a threshold, an event type and a text filter at the top of the view. The locking history graph allows you to select a thread or a monitor of interest and only show locking situations that involve the marked entities. Events of interest are shown with a different color in the time line and there is a secondary navigation bar to step through those events. If the current event is not an event of interest, you can see how many events are between the current event and the next event of interest in either direction.

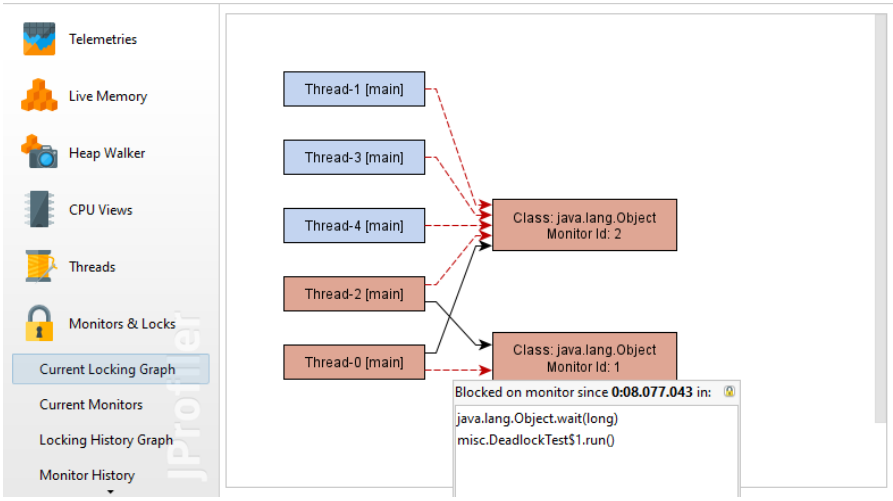
In addition to locking situations where the selected thread or monitor are present, the locking situations where it is removed from the graph are shown as well. This is because each monitor event is defined by two such locking situations, one where an operation is started and one where it has ended. This also means that a completely empty graph is a valid locking situation that indicates that there are no more locks in the JVM.



Another strategy to reduce the number of events that need your attention is to cumulate locking situations. In the locking history graph, there is a time line at the bottom that shows all recorded events. Clicking and dragging in it selects a time range and data from all contained events is shown in the locking graph above. In a cumulated graph, each arrow can contain multiple events of the same type. In that case, the tool tip window shows the number of events as well as the total time of all contained events. A drop-down list in the tool tip window shows the time stamps and lets you switch between the different events.

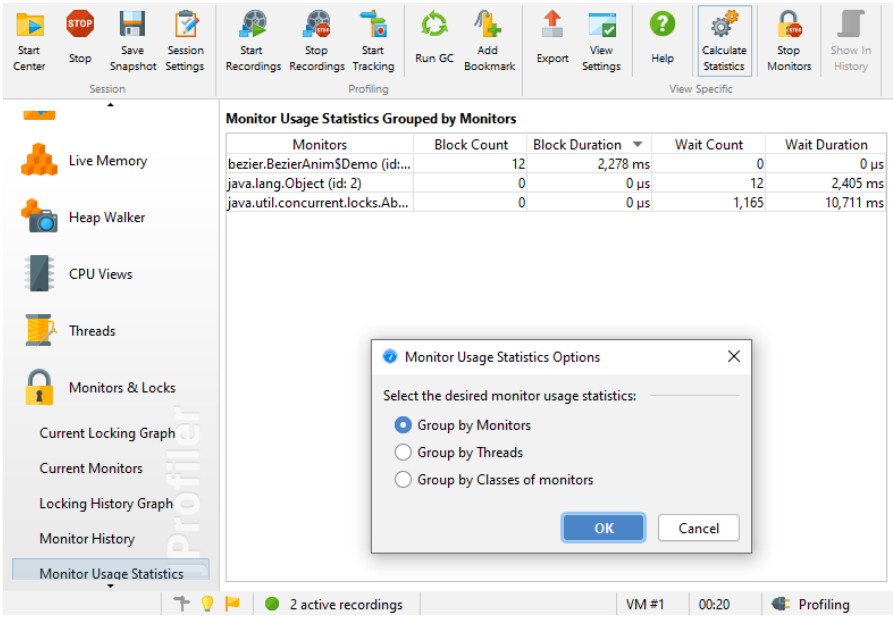
Deadlock detection

Data in the current locking graph and current monitors views is always shown, regardless of whether monitor events are recorded. These views show the current locking situations and the monitor events that are in progress. Blocking operations are usually short-lived, but in the event of a deadlock, both views will display a permanent view of the issue. In addition, the current locking graph shows the threads and monitors that produce a deadlock in red, so you can spot such a problem immediately.



Monitor usage statistics

To investigate blocking and waiting operations from a more elevated perspective, the monitor statistics view calculates reports from the monitor recording data. You can group monitor events by monitors, thread names, or classes of monitors and analyze cumulated counts and durations for each row.

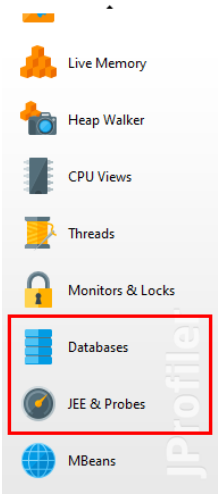


Probes

CPU and memory profiling are primarily concerned with objects and method calls, the basic building blocks of an application on the JVM. For some technologies, a more high-level approach is required that extracts semantic data from the running application and displays it in the profiler.

The most prominent example for this is profiling calls to a database with JDBC. The call tree shows when you use the JDBC API and how long those calls take. However, different SQL statements may be executed for each call and you have no idea which of those calls are responsible for a performance bottleneck. Also, JDBC calls often originate from many different places in your application and it is important to have a single view that shows all database calls instead of having to search for them in the generic call tree.

To solve this problem, JProfiler offers a number of probes for important subsystems in the JRE. Probes add instrumentation into specific classes to collect their data and display them in dedicated views in the "Databases" and "JEE & Probes" view sections. In addition, probes can annotate data into the call tree so you can see both generic CPU profiling as well as high-level data at the same time.

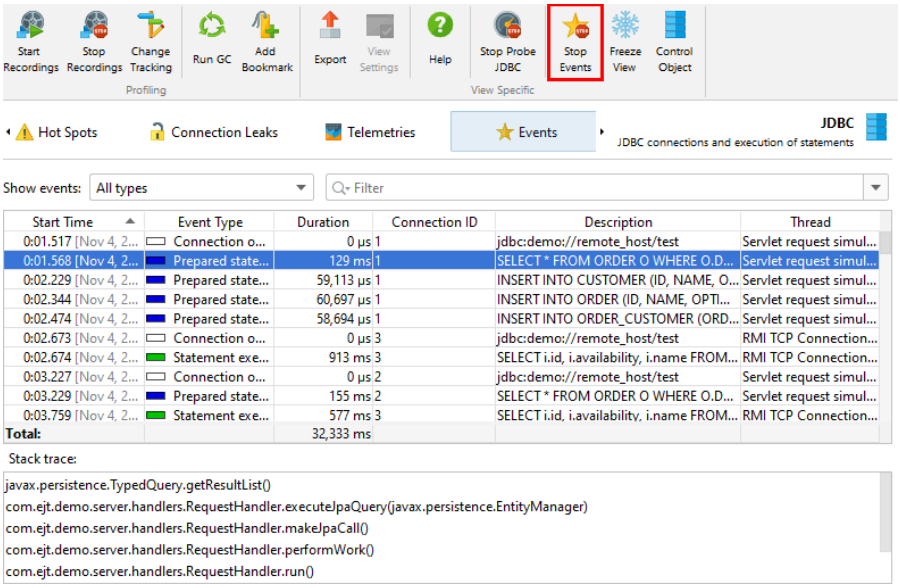


If you are interested in getting more information about a technology that is not directly supported by JProfiler, you can write your own probe [p. 142] for it. Some libraries, containers or database drivers may ship with their own embedded probe [p. 147] that becomes visible in JProfiler when they are used by your application.

Probe events

Because probes add overhead, they are not recorded by default, but you have to start recording [p. 23] separately for each probe, either manually or automatically.

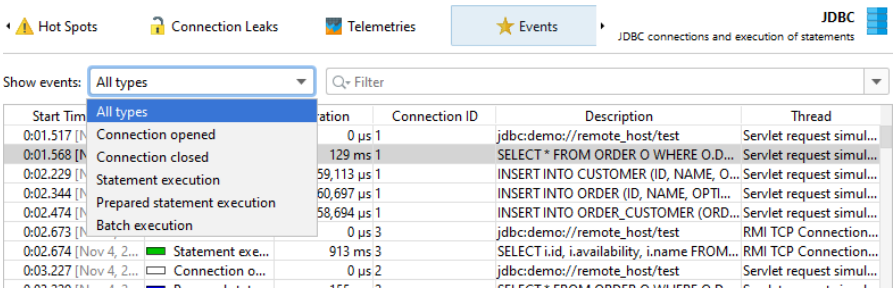
Depending on the capabilities of the probe, probe data is displayed in a number of views. At the lowest level are probe events. Other views show data that cumulates probe events. By default, probe events are not retained even when a probe is being recorded. When single events become important, you can record them in the probe events view. For some probes, like the file probe, this is generally not advisable because they usually generate events at a high rate. Other probes, like the "HTTP server" probe or the JDBC probe may generate events at a much lower rate and so recording single events may be appropriate.



Probe events capture a probe string from a variety of sources, including method parameters, return values, the instrumented object and thrown exceptions. Probes may collect data from multiple method calls, for example like the JDBC probe that has to intercept all setter calls for prepared statements in order to construct the actual SQL string. The probe string is the basic information about the higher-level subsystem that is measured by the probe. In addition, an event contains a start time, an optional duration, the associated thread and a stack trace.

At the bottom the of the table, there is a special total row that sums all numeric columns in the table. For the default columns, this only includes the *Duration* column, Together with the filter selector above the table, you can analyze the collected data for selected subsets of events. By default, the text filter works on all available columns. In order to be more specific, you can select a particular column from the filter options popup menu.

Probes can record different kinds of activities and associate an event type with their probe events. For example, the JDBC probe shows statements, prepared statements and batch executions as event types with different colors.

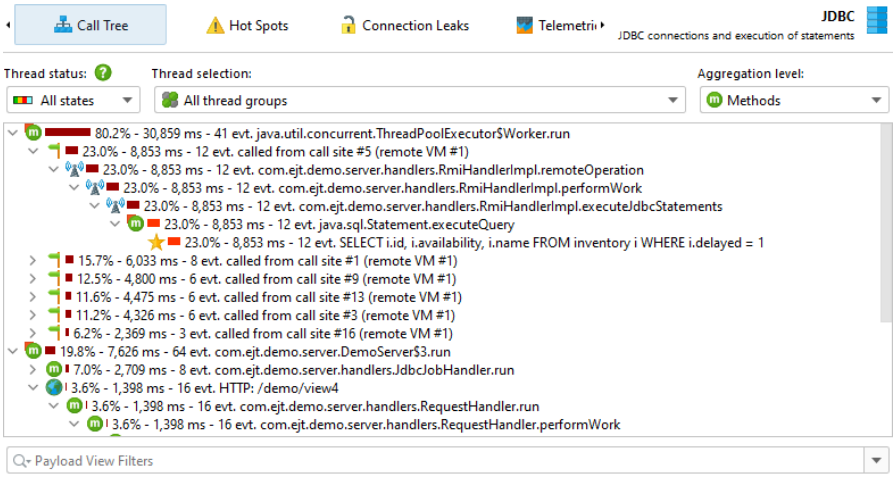


To prevent excessive memory usage when single events are recorded, JProfiler consolidates events. The event cap is configured in the profiling settings and applies to all probes. Only the most recent events are retained, older events are discarded. This consolidation does not affect the higher-level views.

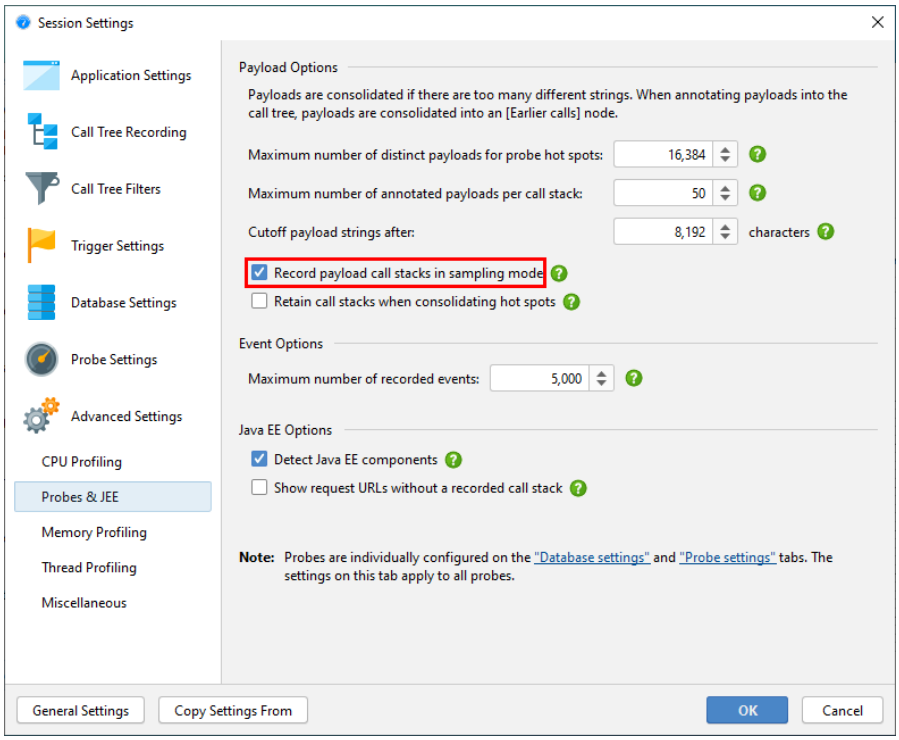
Probe call tree and hot spots

Probe recording works closely together with CPU recording. Probe events are aggregated into a probe call tree where the probe strings are the leaf nodes, called "payloads". Only call stacks

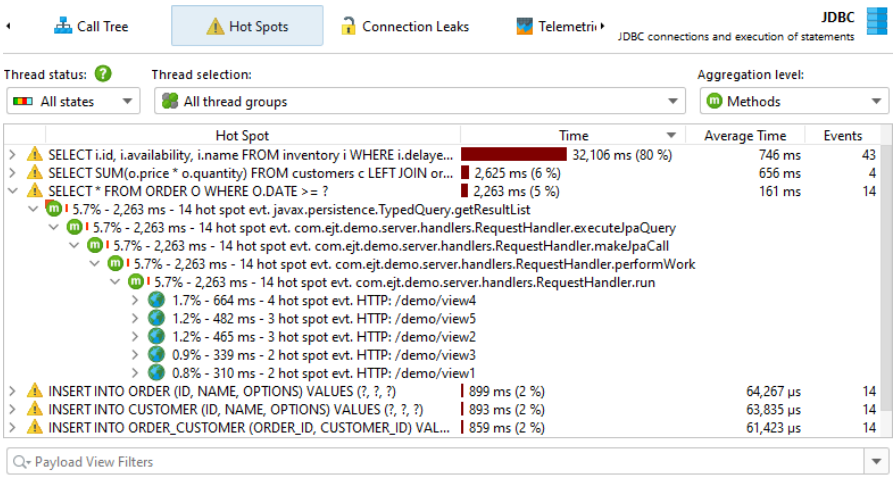
where a probe event has been created are included in that tree. The information on the method nodes refers to the recorded payload names. For example, if an SQL statement was executed 42 times at a particular call stack with a total time of 9000 ms, this adds an event count of 42 and a time of 9000 ms to all ancestor call tree nodes. The cumulation of all recorded payloads forms the call tree that shows you which call paths consume most of the probe-specific time. The focus of the probe tree is the payloads, so the view filter searches for payloads by default, although its context menu also offers a mode to filter classes.



If CPU recording is switched off, the back traces will only contain a "No CPU data was recorded" node. If CPU data was only partially recorded, there may be a mixture of these nodes with actual back traces. Even if sampling is enabled, JProfiler records the exact call traces for probe payloads by default. If you want to avoid this overhead, you can switch it off in the profiling settings. There are several other tuning options for probe recording that can be adjusted to increase data collection or reduce overhead.



Hot spots can be calculated from the probe call tree. The hot spot nodes are now payloads and not method calls like in the CPU view section [p. 48]. This is often the most immediately useful view of a probe. If CPU recording is active, you can open the top-level hot spots and analyze the method backtraces, just like in the regular CPU hot spots view. The numbers on the back trace nodes indicate how many probe events with what total duration were measured along the call stack extending from the deepest node to the node just below the hot spot.



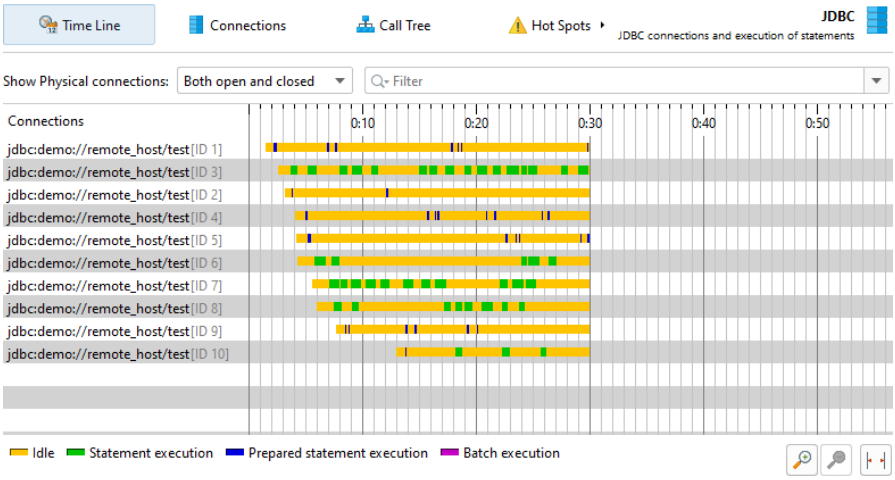
Both probe call tree as well as probe hot spots view allow you to select a thread or thread group, the thread status and an aggregation level for method nodes, just like in the corresponding CPU views. When you come from the CPU views to compare data, it is important to keep in mind that the default thread status in the probe views is "All states" and not "Runnable" like in the CPU views. This is because a probe event often involves external systems like database calls, socket operations or process executions where it is important to look at the total time and not only on the time that the current JVM has spent working on it.

Control objects

Many libraries that provide access to external resources give you a connection object that you can use for interacting with the resource. For example, when starting a process, the `java.lang.Process` object lets you read from the output streams and write to the input stream. When working with JDBC, you need a `java.sql.Connection` object to perform SQL queries. The generic term that is used in JProfiler for this kind of object is "control object".

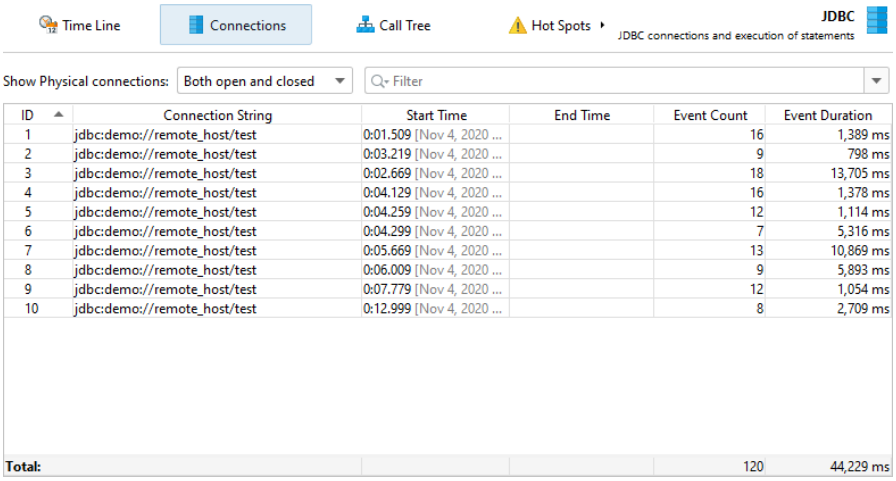
Grouping the probe events with their control objects and showing their life cycle can help you to better understand where a problem comes from. Also, creating control objects is often expensive, so you want to make sure that your application does not create too many and closes them properly. For this purpose, probes that support control objects have a "Time line" and a "Control objects" view, where the latter may be named more specifically, for example "Connections" for the JDBC probe. When a control object is opened or closed, the probe creates special probe events that are shown in the events view, so you can inspect the associated stack traces.

In the time line view, each control object is shown as a bar whose coloring shows when the control object was active. Probes can record different event types and the time line is colored accordingly. This status information is not taken from the list of events, which may be consolidated or not even available, but is sampled every 100 ms from the last status. Control objects have a name that allows you to identify them. For example, the file probe creates control objects with the file name while the JDBC probe shows the connection string as the name of the control object.



The control objects view shows all control objects in tabular form. Both open and closed control objects are present by default. You can use the controls at the top to restrict the display to open or closed control objects only or to filter the contents of a particular column. In addition to the basic life cycle data for control objects, the table shows data for the cumulated activity of each control object, for example the event count and the average event duration.

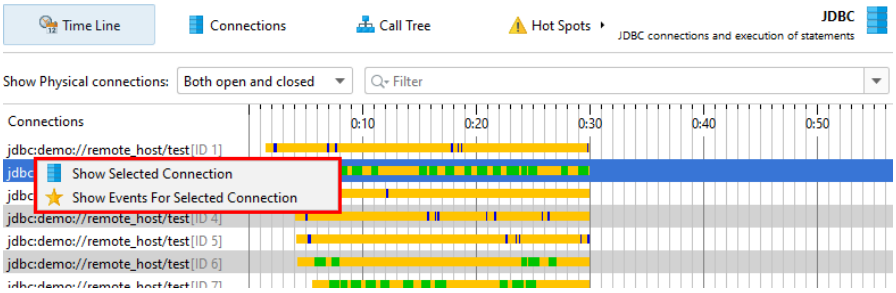
Different probes show different columns here, the process probe for example shows separate sets of columns for read and write events. This information is also available if single event recording is disabled. Just like for the events view, the total row at the bottom can be used together with filtering to get cumulated data on partial sets of control objects.



ID	Connection String	Start Time	End Time	Event Count	Event Duration
1	jdbc:demo://remote_host/test	0:01.509 [Nov 4, 2020 ...		16	1,389 ms
2	jdbc:demo://remote_host/test	0:03.219 [Nov 4, 2020 ...		9	798 ms
3	jdbc:demo://remote_host/test	0:02.669 [Nov 4, 2020 ...		18	13,705 ms
4	jdbc:demo://remote_host/test	0:04.129 [Nov 4, 2020 ...		16	1,378 ms
5	jdbc:demo://remote_host/test	0:04.259 [Nov 4, 2020 ...		12	1,114 ms
6	jdbc:demo://remote_host/test	0:04.299 [Nov 4, 2020 ...		7	5,316 ms
7	jdbc:demo://remote_host/test	0:05.669 [Nov 4, 2020 ...		13	10,869 ms
8	jdbc:demo://remote_host/test	0:06.009 [Nov 4, 2020 ...		9	5,893 ms
9	jdbc:demo://remote_host/test	0:07.779 [Nov 4, 2020 ...		12	1,054 ms
10	jdbc:demo://remote_host/test	0:12.999 [Nov 4, 2020 ...		8	2,709 ms
Total:				120	44,229 ms

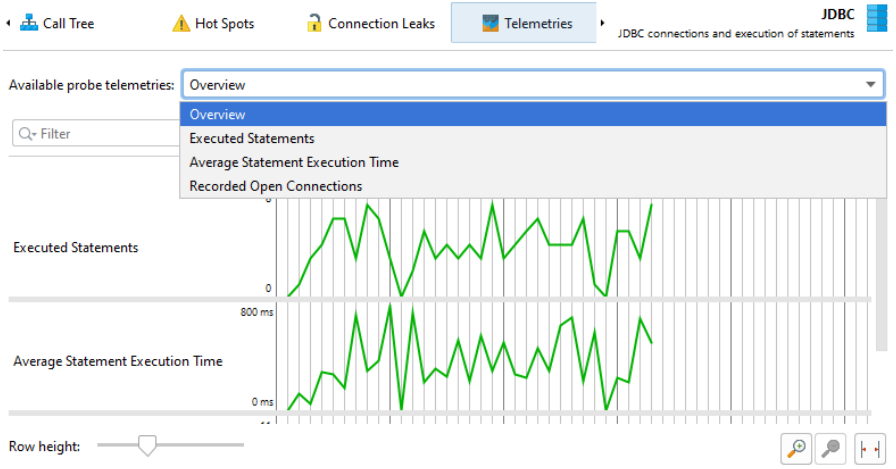
A probe can publish certain properties in a nested table. This is done to reduce the information overload in the main table and give more space to table columns. If a nested table is present, such as for the file and process probes, each row has an expansion handle at the left side that opens a property-value table in place.

The time line, control objects view and the events view are connected with navigation actions. For example, in the time line view, you can right-click a row and jump to each of the other views so that only the data from the selected control object is displayed. This is achieved by filtering the control object ID to the selected value.

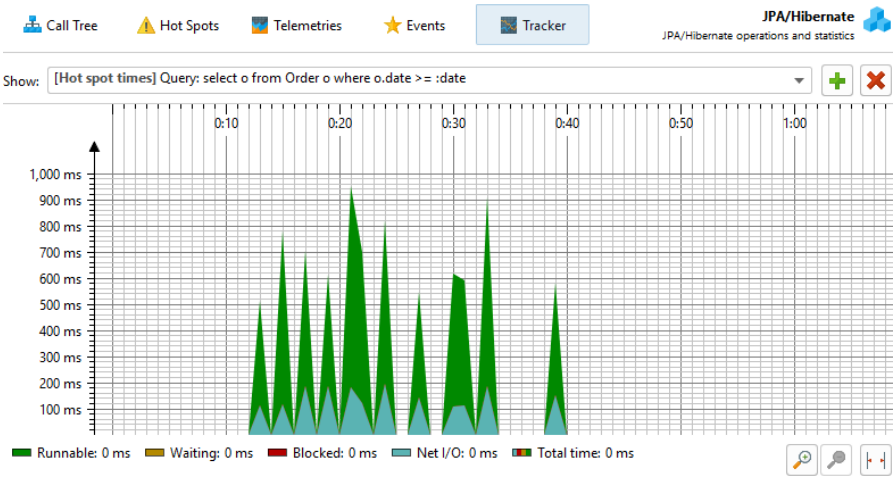


Telemetries and tracker

From the cumulated data that is collected by a probe, several telemetries are recorded. For any probe, the number of probe events per second and some average measure for probe events like the average duration or the throughput of an I/O operation are available. For probes with control objects, the number of open control objects is also a canonical telemetry. Each probe can add additional telemetries, for example the JPA probe shows separate telemetries for query counts and entity operation counts.

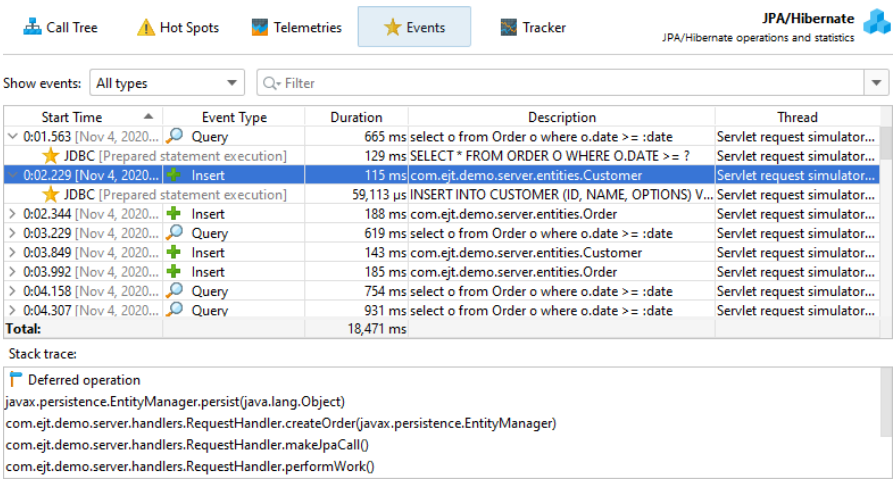


The hot spots view and the control objects view show cumulated data that can be interesting to track over time. These special telemetries are recorded with the probe tracker. The easiest way to set up tracking is to add new telemetries with the *Add Selection to Tracker* action from the hot spots or control object views. In both cases, you have to choose if you want to track times or counts. When tracking control objects, the telemetry is a stacked area graph for all different probe event types. For tracked hot spots, the tracked times are split into the different thread states.

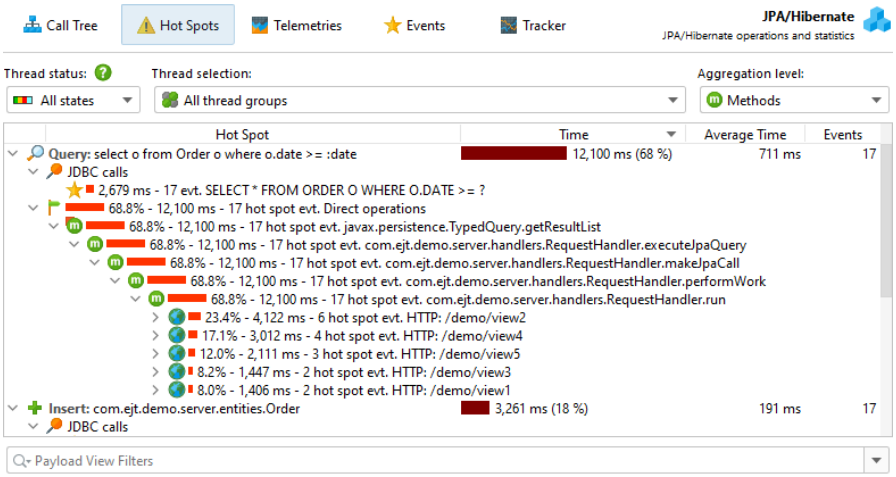


JDBC and JPA

The JDBC and JPA probes work hand in hand. In the events view of the JPA probe, you can expand single events to see the associated JDBC events if the JDBC probe was recorded along with the JPA probe.

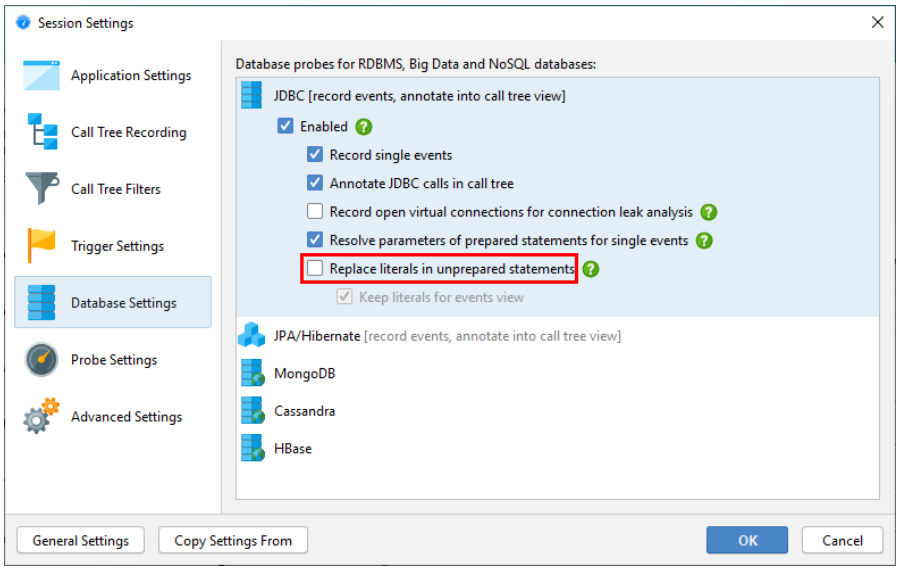


Similarly, the hot spots view adds a special "JDBC calls" node to all hot spots that contains the JDBC calls that were triggered by the JPA operation. Some JPA operations are asynchronous and are not executed immediately, but at some arbitrary later point in time when the session is flushed. When looking for performance problems, the stack trace of that flush is not helpful, so JProfiler remembers the stack traces of where existing entities have been acquired or where new entities have been persisted and ties them to the probe events. In that case, the back traces of the hot spot are contained inside a node that is labelled "Deferred operations", otherwise a "Direct operations" node is inserted.



Other probes like the MongoDB probe support both direct and asynchronous operations. Asynchronous operations are not executed on the current thread but somewhere else, either on one or multiple other threads in the same JVM or in another process. For such probes, the back traces in the hot spots are sorted into "Direct operations" and "Async operation" container nodes.

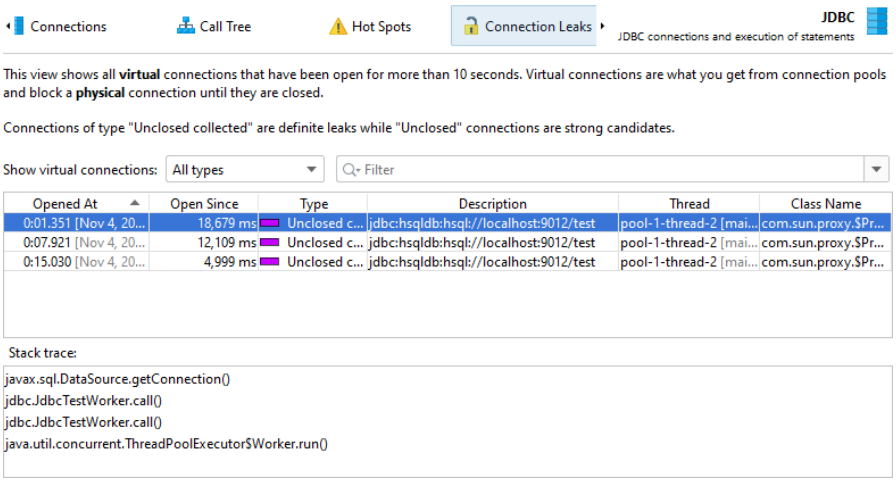
A special problem in the JDBC probe is that you can only get good hot spots if literal data like IDs is not included in the SQL strings. This is automatically the case if prepared statements are used, but not if regular statements are executed. In the latter case, you will likely get a list of hot spots, where most queries are executed just once. As a remedy, JProfiler offers a non-default option in the JDBC probe configuration for replacing literals in unprepared statements. For debugging purposes, you may still want to see the literals in the events view. Deactivating that option reduces memory overhead, because JProfiler will not have to cache so many different strings.



On the other hand, JProfiler collects the parameters for prepared statements and shows a complete SQL string without placeholders in the events view. Again, this is useful when debugging, but if you do not need it, you can switch it off in the probe settings in order to conserve memory.

JDBC connection leaks

The JDBC probe has a "Connection leaks" view that shows open virtual database connections that have not been returned to their database pool. This only affects virtual connections that are created by a pooled database source. Virtual connections block a physical connection until they are closed.

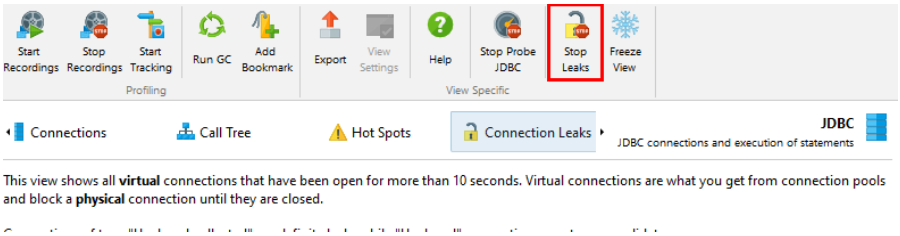


There are two types of leak candidates, "unclosed" connections and "unclosed collected" connections. Both types are virtual connections where the connection objects that have been handed out by the database pool are still on the heap, but `close()` has not been called on them. "Unclosed collected" connections have been garbage collected and are definite connection leaks.

"Unclosed" connection objects are still on the heap. The greater the *Open Since* duration, the more likely such a virtual connection is a leak candidate. A virtual connection is considered as a potential leak when it has been open for more than 10 seconds. However, `close()` may still be called on it, and then the entry in the "Connection leaks" view would be removed.

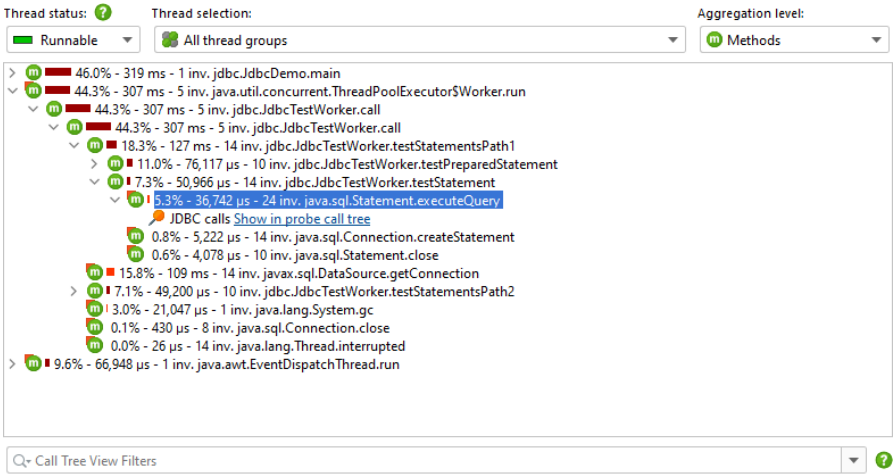
The connection leaks table includes a *Class Name* column that shows the name of the connection class. This will tell you which type of pool has created the connection. JProfiler explicitly supports a large number of database drivers and connection pools and knows which classes are virtual and physical connections. For unknown pools or database drivers, JProfiler may mistake a physical connection for a virtual one. Since physical connections are often long-lived, it would then show up in the "Connection leaks" view. In this case, the class name of the connection object will help you to identify it as a false positive.

By default, when you start probe recording, the connection leak analysis is not enabled. There is a separate recording button in the connection leaks view whose state corresponds to the *Record open virtual connections for connection leak analysis* check box in the JDBC probe settings. Just like for event recording, the state of the button is persistent, so if you start the analysis once, it will automatically be started for the next probe recording session.

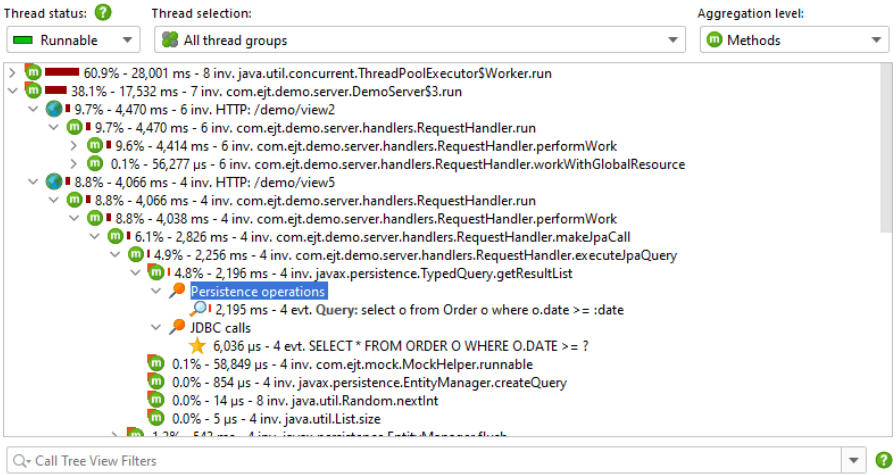


Payload data in the call tree

When looking at the CPU call tree, it is interesting to see where probes have recorded payload data. That data may help you to interpret the measured CPU times. That is why many probes add cross-links into the CPU call tree. For example, the class loader probe can show you where class loading has been triggered. This is otherwise not visible in the call tree and can add unexpected overhead. A database call that is otherwise opaque in the call tree view can be further analyzed in the corresponding probe with a single click. This even works for call tree analyses where the analysis is automatically repeated in the context of the probe call tree view when you click on the probe link.



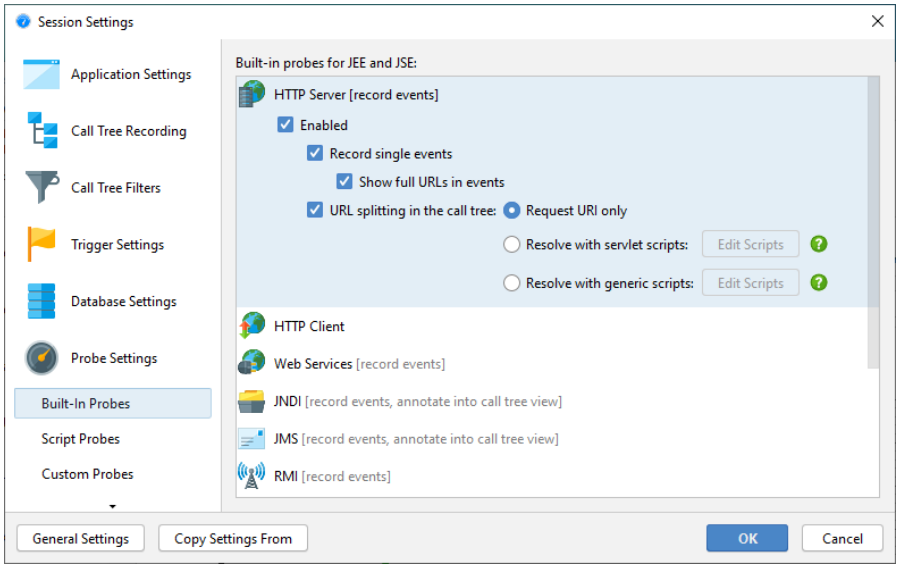
Another possibility is to show the payload information inline directly in the CPU call tree. All relevant probes have an *Annotate in call tree* option in their configuration for that purpose. In that case, no links into the probe call tree are available. Each probe has its own payload container node. Events with the same payload names are aggregated and the number of invocations and total times are displayed. Payload names are consolidated on a per-call stack basis, with the oldest entries being aggregated into an "[earlier calls]" node. The maximum number of recorded payload names per call stack is configurable in the profiling settings.



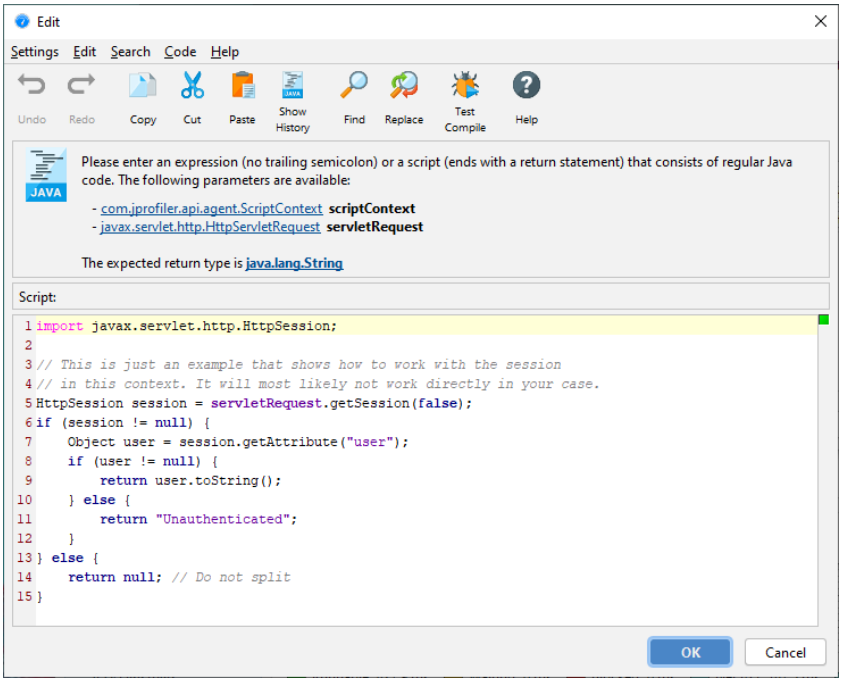
Call tree splitting

Some probes do not use their probe strings to annotate payload data into the call tree. Rather, they split the call tree for each different probe string. This is especially useful for server-type

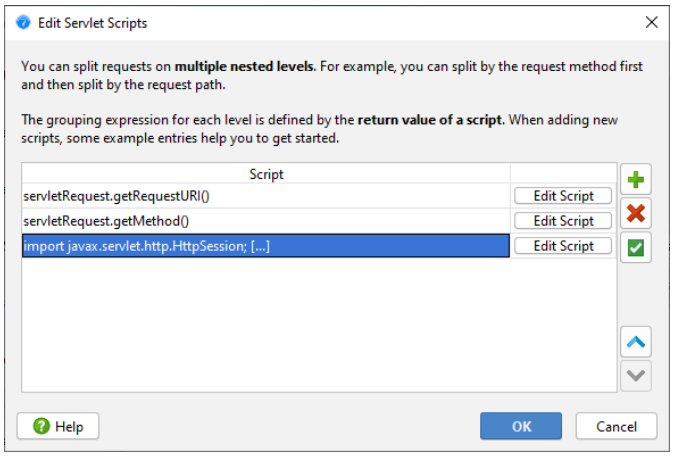
probes, where you want to see the call tree separately for each different type of incoming request. The "HTTP server" probe intercepts URLs and gives you fine grained control over what parts of the URL should be used for splitting the call tree. By default, it only uses the request URI without any parameters.



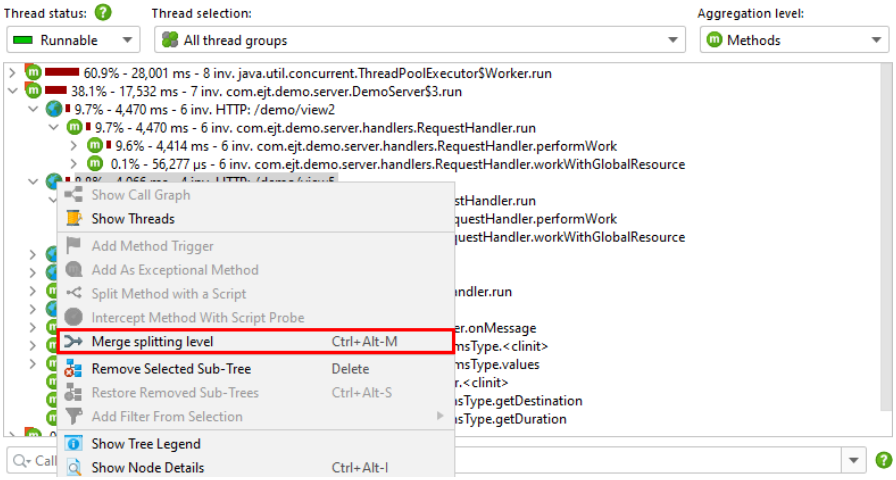
For more flexibility, you can define a script that determines the split string. In the script, you get the current `javax.servlet.http.HttpServletRequest` as a parameter and return the desired string.



What's more, you are not limited to a single splitting level, but can define multiple nested splittings. For example, you can split by the request URI first and then by the user name that is extracted from the HTTP session object. Or, you can group requests by their request method before splitting by the request URI.



By using nested splittings, you can see separate data for each level in the call tree. When looking at the call tree, a level might get in the way and you would find yourself in need of eliminating it from the "HTTP server" probe configuration. More conveniently and without loss of recorded data, you can temporarily merge and unmerge splitting levels in the call tree on the fly by using the context menu on the corresponding splitting nodes.



Splitting the call tree can cause considerable memory overhead, so it should be used carefully. To avoid memory overload, JProfiler caps the maximum number of splits. If the splitting cap for a particular split level has been reached, a special "[capped nodes]" splitting node is added with a hyperlink to reset the cap counter. If the default cap is too low for your purposes, you can increase it in the profiling settings.

MBean Browser

Many application servers and frameworks such as Apache Camel[1] use JMX to expose a number of MBeans for configuration and monitoring purposes. The JVM itself also publishes a number of platform MXBeans[2] that present interesting information around the low-level operations in the JVM.

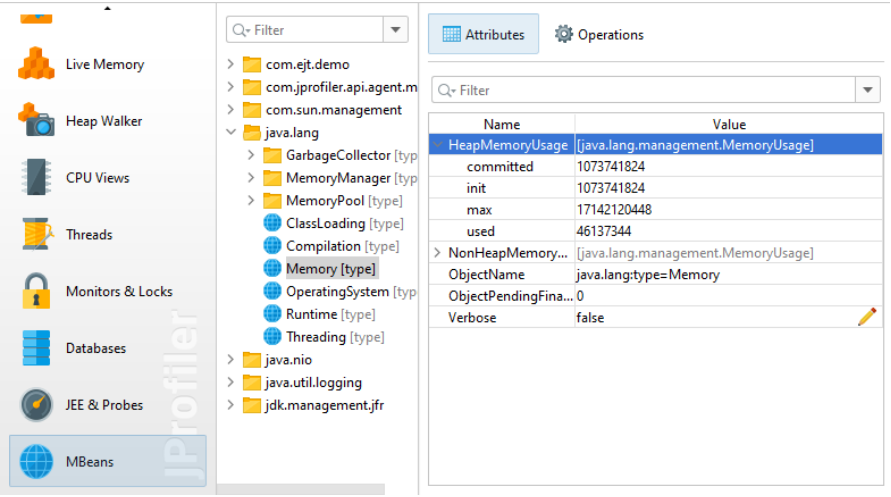
JProfiler includes an MBean browser that shows all registered MBeans in the profiled VM. The remote management level of JMX for accessing MBean servers is not required, because the JProfiler agent is already running in-process and has access to all registered MBean servers.

JProfiler supports the type system of **Open MBeans**. Besides defining a number of simple types, Open MBeans can define complex data types that do not involve custom classes. Also, arrays and tables are available as data structures. With **MXBeans**, JMX offers an easy way to create Open MBeans automatically from Java classes. For example, the MBeans provided by the JVM are MXBeans.

While MBeans have no hierarchy, JProfiler organizes them into a tree by taking the object domain name up to the first colon as the first tree level and using all properties as recursively nested levels. The property value is shown first with the property key in brackets at the end. The `type` property is prioritized to appear right below the top-level node.

Attributes

At the top level of the tree table showing the MBean content, you see the MBean attributes.



The following data structures are shown as nested rows:

- **Arrays**

  Elements of primitive arrays and object arrays are shown in nested rows with the index as the key name.

- **Composite data**

  All items in a composite data type are shown as nested rows. Each item can be an arbitrary type, so nesting can continue to an arbitrary depth.

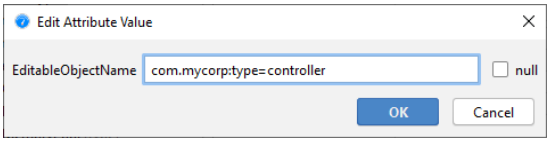
[1] https://camel.apache.org/camel-jmx.html
[2] https://docs.oracle.com/javase/7/docs/technotes/guides/management/mxbeans.html

- **Tabular data**

  Most frequently you will encounter tabular data in MXBeans where instances of `java.util.Map` are mapped to a tabular data type with one key column and one value column. If the type of the key is a simple type, the map is shown "inline", and each key-value pair is shown as a nested row. If the key has a complex type, a level of "map entry" elements with nested key and value entries is inserted. This is also the case for the general tabular type with composite keys and multiple values.

Optionally, MBean attributes can be editable in which case an edit icon will be displayed next to their value and the *Edit Value* action becomes active. Composite and tabular types cannot be edited in the MBean browser, but arrays or simple types are editable.

If a value is nullable, such as an array, the editor has a check box to choose the null state.



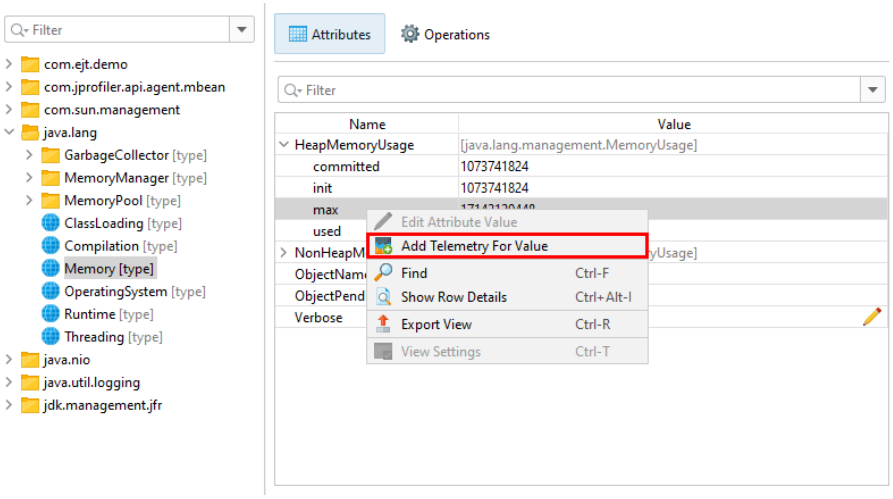
Array elements are separated by semicolons. One trailing semicolon can be ignored, so `1` and `1;` are equivalent. A missing value before a semicolon will be treated as a null value for object arrays. For string arrays, you can create empty elements with double quotes ("") and elements that contain semicolons by quoting the entire element. Double quotes in string elements must be doubled. For example, entering a string array value of

```
"Test";"";;"embedded "" quote";"A;B";;
```

creates the string array

```
new String[] {"Test", "", null, "embedded \" quote", "A;B", null}
```

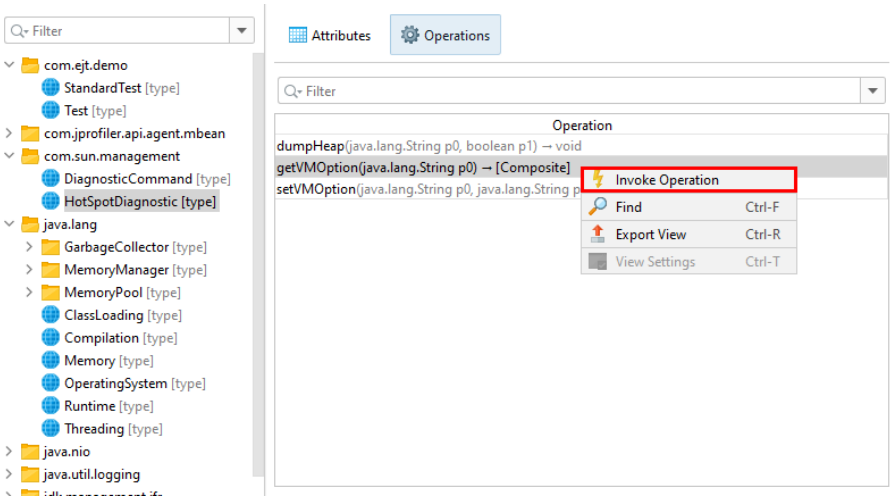
JProfiler can create custom telemetries from numeric MBean attribute values. When you define an MBean telemetry line [p. 43] for a custom telemetry, an MBean attribute browser will be shown that lets you choose an attribute that provides the telemetry data. When you are already working in the MBean Browser, the *Add Telemetry For Value* action in the context menu provides a convenient way to create a new custom telemetry.



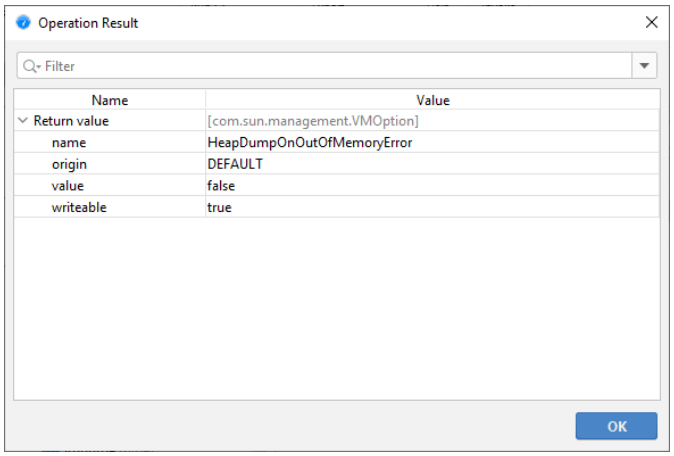
A telemetry can also track nested values in composite data or tabular data with simple keys and single values. When you chose the nested row, a value path is built where path components are separated by forward slashes.

Operations

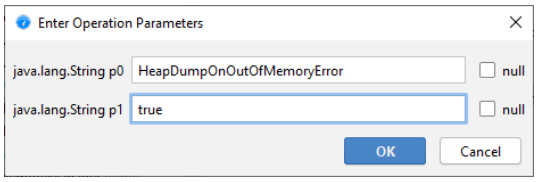
In addition to inspecting and modifying MBean attributes, you can invoke MBean operations and check their return values. MBean operations are methods on the MBean interface that are not setters or getters.



The return value of an operation may have a composite, tabular or array type, so a new window with a content similar to the MBean attribute tree table is shown. For a simple return type, there is only one row named "Return value". For other types, the "Return value" is the root element into which the result is added.



MBean operations can have one or more arguments. When you enter them, the same rules and restrictions apply as when editing an MBean attribute.



Offline Profiling

There are two fundamentally different ways to profile an application with JProfiler: By default, you profile with the JProfiler GUI attached. The JProfiler GUI provides you with buttons to start and stop recording and shows you all recorded profiling data.

There are situations where you would like to profile without the JProfiler GUI and analyze the results later on. For this scenario, JProfiler offers offline profiling. Offline profiling allows you to start the profiled application with the profiling agent but without the need to connect with a JProfiler GUI.

However, offline profiling still requires some actions to be performed. At least one snapshot has to be saved, otherwise no profiling data will be available for analysis later on. Also, to see CPU or allocation data, you have to start recording at some point. Similarly, if you wish to be able to use the heap walker in the saved snapshot, you have to trigger a heap dump.

Profiling API

The first solution to this problem is the controller API. With the API, you can programmatically invoke all profiling actions in your code. In the `api/samples/offline` directory, there is a runnable example that shows you how to use the controller API in practice. Execute `../gradlew` in that directory to compile and run it and study the Gradle build file `build.gradle` to understand how the test program is invoked.

The Controller API is the main interface for managing profiling actions at run time. It is contained in `bin/agent.jar` in your JProfiler installation or as a Maven dependency with the coordinates

```
group: com.jprofiler
artifact: jprofiler-probe-injected
version: <JProfiler version>
```

and the repository

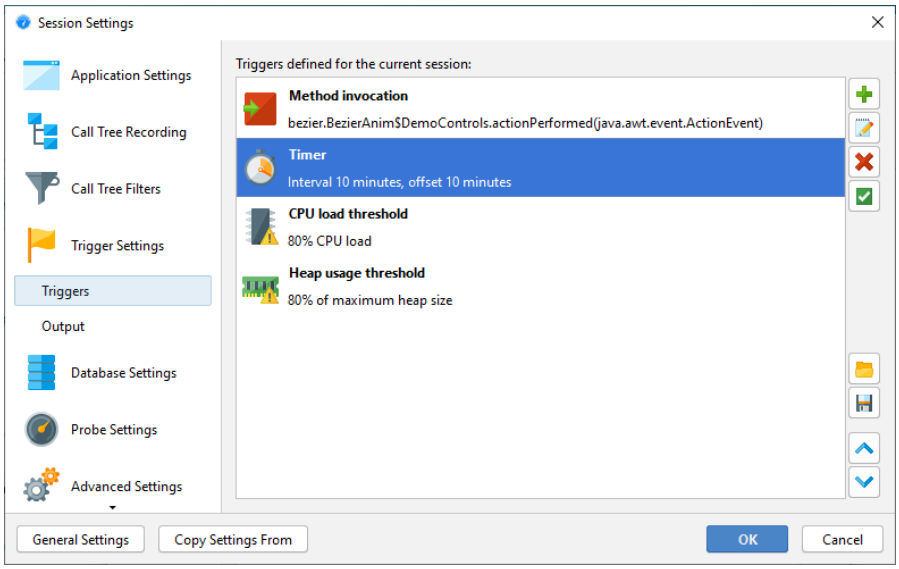
```
https://maven.ej-technologies.com/repository
```

If the profiling API is used during a normal execution of your application, the API calls will just quietly do nothing.

The drawback of this approach is that you have to add the JProfiler agent library to the class path of your application during development, add profiling instructions to your source code and recompile your code each time you make a change to the programmatic profiling actions.

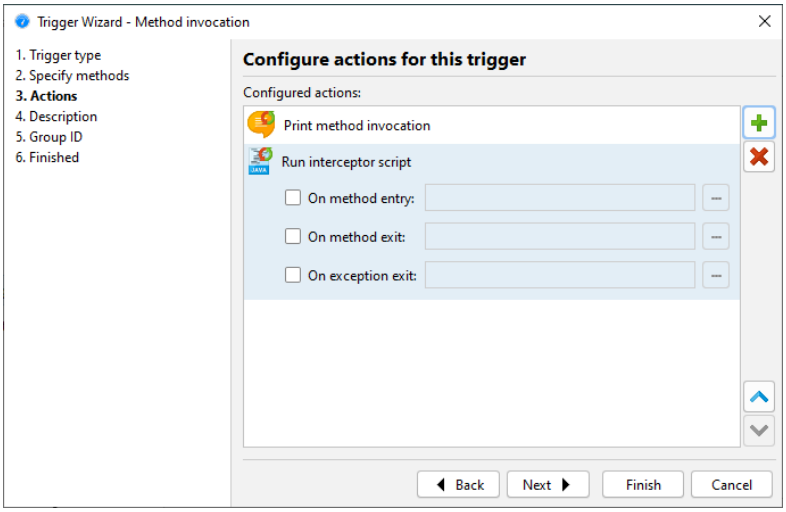
Triggers

With triggers [p. 23], you can specify all profiling actions in the JProfiler GUI without modifying your source code. Triggers are saved in the JProfiler config file. The config file and the session ID are passed to the profiling agent on the command line when you start with offline profiling enabled, so the profiling agent can read those trigger definitions.



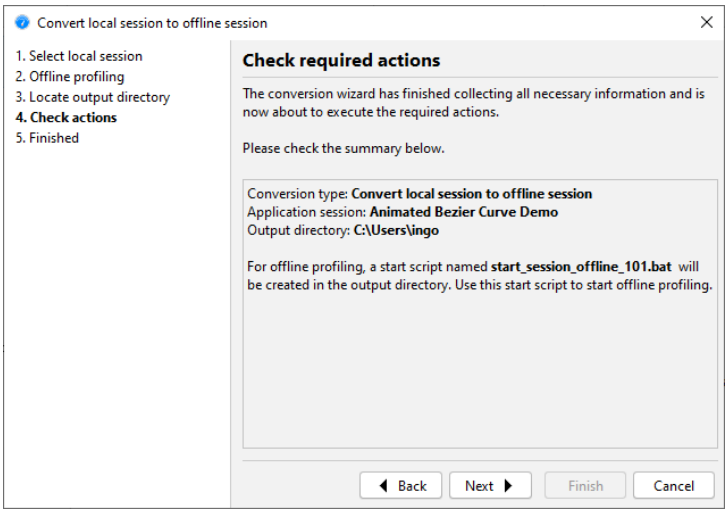
In contrast to the profiling API, where you add API calls to your source code, triggers are activated when a certain event occurs in the JVM. For example, instead of adding an API call for a certain profiling action at the beginning or at the end of a method, you can use a method invocation trigger. As another use case, instead of creating your own timer thread to periodically save a snapshot, you can use a timer trigger.

Each trigger has a list of actions that are performed when the associated event occurs. Some of these actions correspond to profiling actions in the controller API. In addition, there are other actions that go beyond the controller functionality such as the action to print method calls with parameters and return values or the action to invoke interceptor scripts for a method.

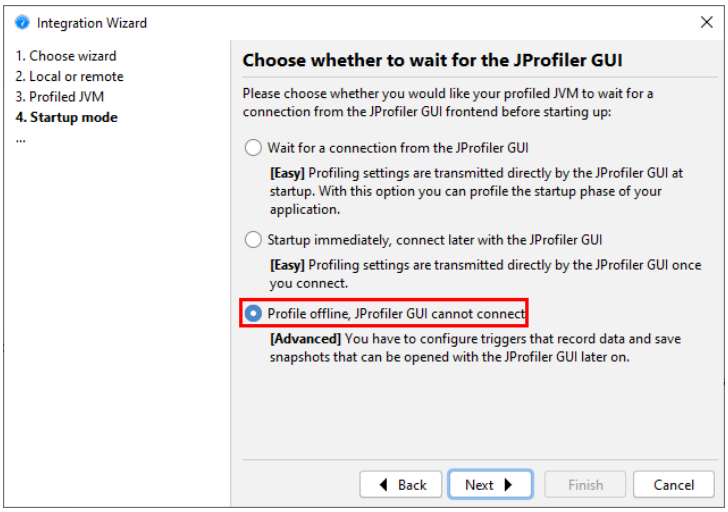


Configuring offline profiling

If you have configured a launched session in JProfiler, you can convert it to an offline session by invoking *Session->Conversion Wizards->Convert Application Session To Offline* from the main menu. This will create a start script with the appropriate VM parameters and take the profiling settings from the same session that you use in the JProfiler UI. If you want to move the invocation to another computer, you have to use *Session->Export Session Settings* to export the session to a config file and make sure that the VM parameter in the start script references that file.



When profiling an application server with the integration wizards, there is always a start script or config file that is being modified so that the VM parameters for profiling are inserted into the Java invocation. All integration wizards have a "Profile offline" option on the "Startup" step in order to configure the application server for offline profiling instead of interactive profiling.



You may want to pass the VM parameter yourself to a Java call, for example if you have a start script that is not handled by the integration wizards. That VM parameter has the format

```
-agentpath:<path to jprofilerti library>=offline,id=<ID>[,config=<path>]
```

and is available from the `[Generic application]` wizard.

Passing `offline` as a library parameter enables offline profiling. In this case, a connection with the JProfiler GUI is not possible. The `session` parameter determines which session from the config file should be used for the profiling settings. The ID of a session can be seen in the top right corner of the *Application settings* tab in the session settings dialog. The optional `config` parameter points to the config file. This is a file that you can export by invoking *Session->Export Session Settings*. If you omit the parameter, the standard config file will be used. That file is located in the `.jprofiler12` directory in your user home directory.

Offline profiling with Gradle and Ant

When you start offline profiling from Gradle or Ant, you can use the corresponding JProfiler plugins to make your work easier. A typical usage of the Gradle task for profiling tests is shown below:

```
plugins {
    id 'com.jprofiler' version 'X.Y.Z'
    id 'java'
}

jprofiler {
    installDir = file('/opt/jprofiler')
}

task run(type: com.jprofiler.gradle.TestProfile) {
    offline = true
    configFile = file("path/to/jprofiler_config.xml")
    sessionId = 1234
}
```

The `com.jprofiler.gradle.JavaProfile` task profiles any Java class in the same way that you execute it with the standard `JavaExec` task. If you use some other method of launching your JVM that is not directly supported by JProfiler, the `com.jprofiler.gradle.SetAgentPathProperty` task can write the required VM parameter to a property. It is added by default when applying the JProfiler plugin, so you can simply write:

```
setAgentPathProperty {
    propertyName = 'agentPathProperty'
    offline = true
    configFile = file("path/to/jprofiler_config.xml")
    sessionId = 1234
}
```

and then use `agentPathProperty` as a project property reference elsewhere after the task has been executed. The features of all Gradle tasks and the corresponding Ant tasks are documented in detail in separate chapters [p. 203].

Enabling offline profiling for running JVMs

With the command line utility `bin/jpenable`, you can start offline profiling in any running JVM with a version of 1.6 or higher. Just like for the VM parameter, you have to specify an `offline` switch, a session ID and an optional config file:

```
jpenable --offline --id=12344 --config=/path/to/jprofiler_config.xml
```

With an invocation like this, you have to select a process from a list of running JVMs. With the additional arguments `--pid=<PID> --noinput` other you can automate the process so that it requires no user input at all.

On the other hand, when enabling offline profiling on the fly, it may be necessary to manually start some recordings or to save a snapshot. This is possible with the `bin/jpcontroller` command line tool.

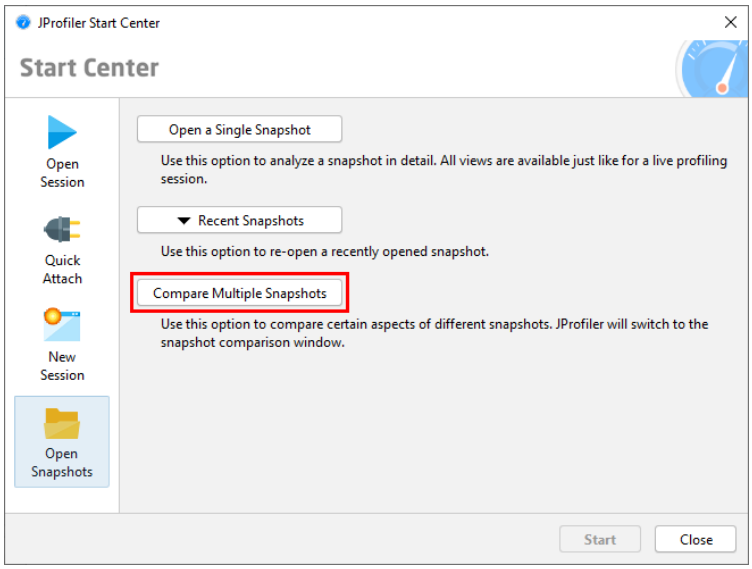
If the profiling agent is only loaded, but no profiling settings have been applied, no recording actions can be switched on and so `jpcontroller` will not be able to connect. This includes the

case where you enable profiling with `jpenable`, but without the `offline` parameter. If you enable offline mode, the profiling settings are specified and `jpcontroller` can be used.

More information on the `jpenable` and `jpcontroller` executables is available in the command line reference [p. 203].

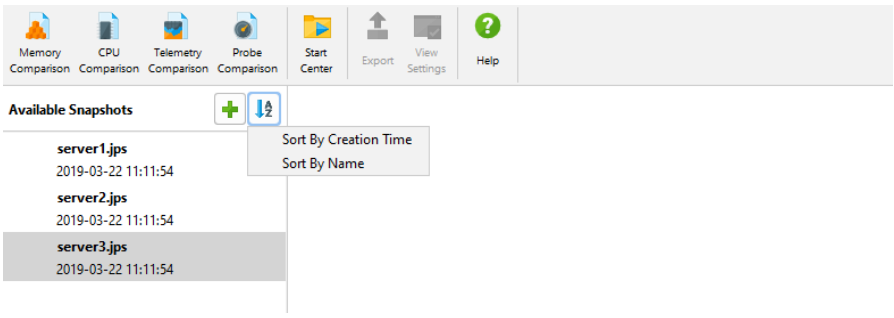
Comparing Snapshots

Comparing the runtime characteristics of your current application against a previous version is a common quality assurance technique for preventing performance regressions. It also can be helpful for solving performance problems within the scope a single profiling session, where you may want to compare two different use cases and find out why one is slower than the other. In both cases, you save snapshots with the recorded data of interest and use the snapshot comparison functionality in JProfiler by invoking *Session->Compare Snapshots in New Window* from the menu or clicking the *Compare Multiple Snapshots* button on the *Open Snapshots* tab of the start center.



Selecting snapshots

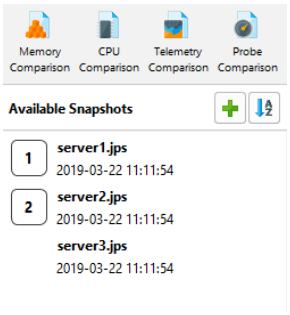
Comparisons are created and viewed in a separate top-level window. First, you add a number of snapshots in the snapshot selector. Then you can create comparisons from two or more of the listed snapshots by selecting the snapshots of interest and clicking on a comparison tool bar button. The order of the snapshot files in the list is significant because all comparisons will assume that snapshots further down in the list have been recorded at a later time. Apart from arranging snapshots manually, you can sort them by name or creation time.



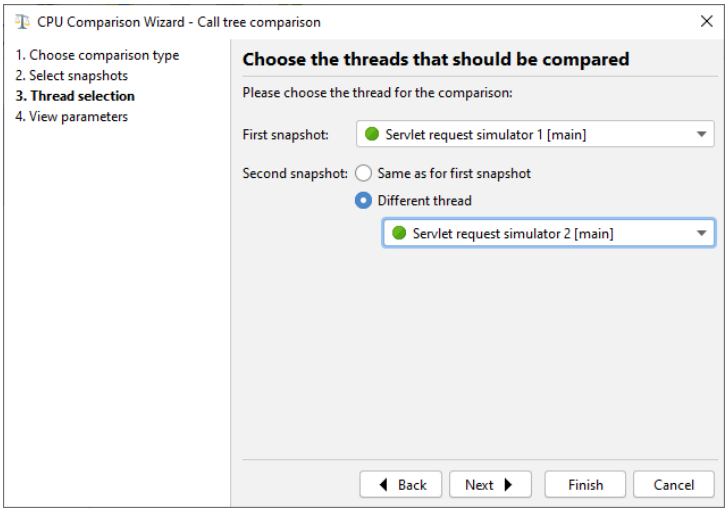
Unlike for the views in JProfiler's main window, the comparison views have fixed view parameters that are shown at the top instead of drop-down lists that let you adjust the parameters on the fly. All comparisons show wizards for collecting the parameters for the comparison, and you can perform the same comparison multiple times with the same parameters. The wizards remember their parameters from previous invocations so you don't have to repeat the configuration if you

compare several sets of snapshots. At any point, you can shortcut the wizard with the *Finish* button or jump to another step by clicking on the step in the index.

When a comparison is active, the snapshots that were analyzed are shown with number prefixes. For comparisons that work with two snapshot, the displayed differences are the measurements from snapshot 2 minus the measurements from snapshot 1.

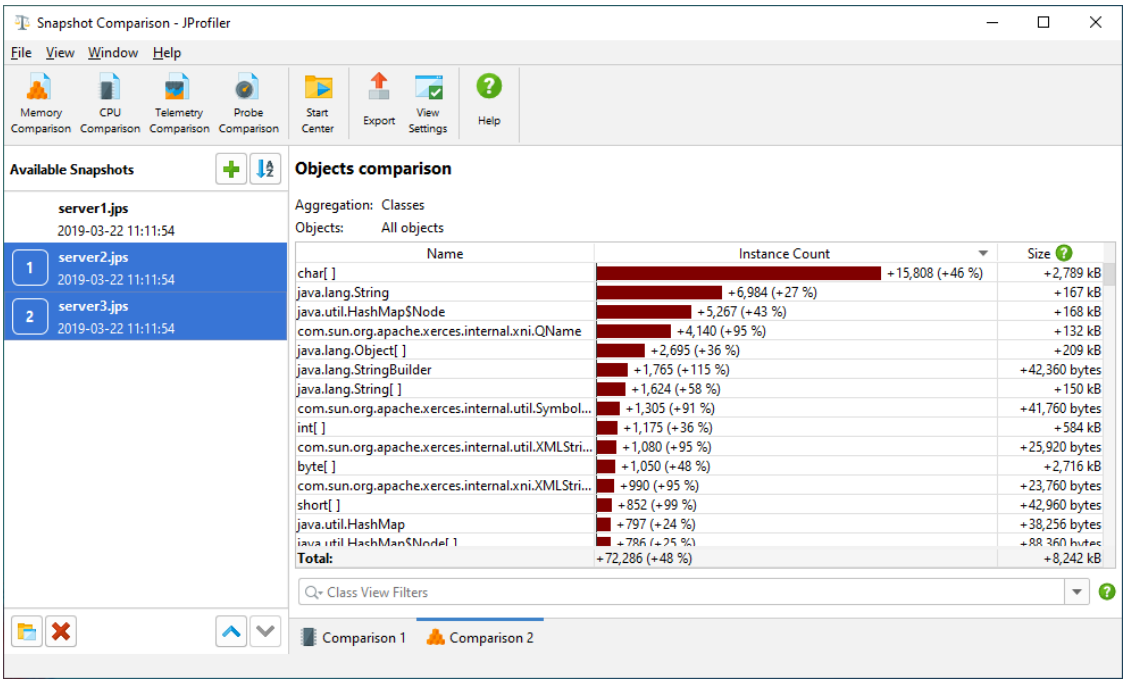


For the CPU comparisons, you can use the same snapshot as the first and second snapshot and select different threads or thread groups in the wizard.

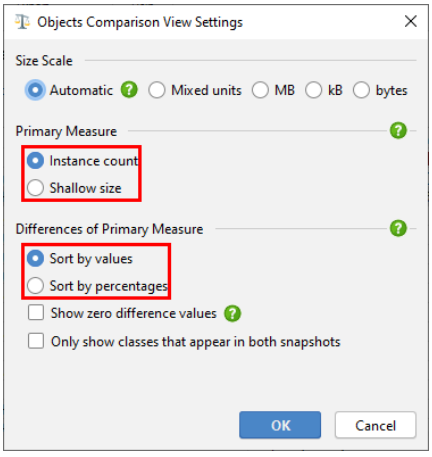


Comparisons with tables

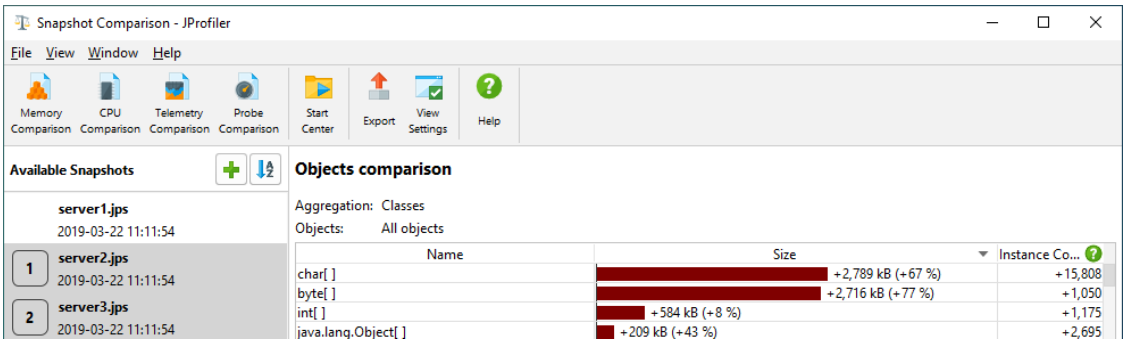
The simplest comparison is the "Objects" memory comparison. It can compare data from the "All objects", "Recorded objects" or the "Classes" view of the heap walker. The columns in the comparison show differences for instance counts and size, but only the *Instances Count* column shows the bidirectional bar chart where increases are painted in red and to the right, while decreases are painted in green and to the left.



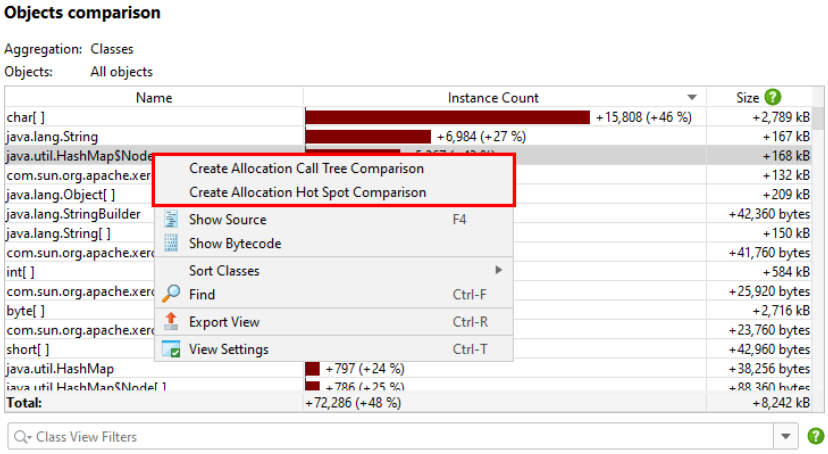
In the view settings dialog you can choose whether you want this bar chart to display absolute changes or percentages. The other value is displayed in parentheses. This setting also determines how the column is sorted.



The measurement in the first data column is called the primary measure and you can switch it from the default instance counts to shallow sizes in the view settings.



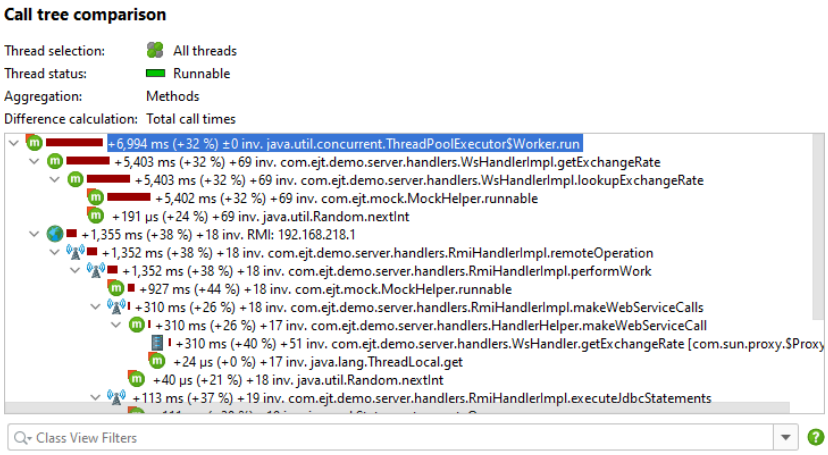
The context menu of the table gives you a shortcut into the other memory comparisons with the same comparison parameters and for the selected class.



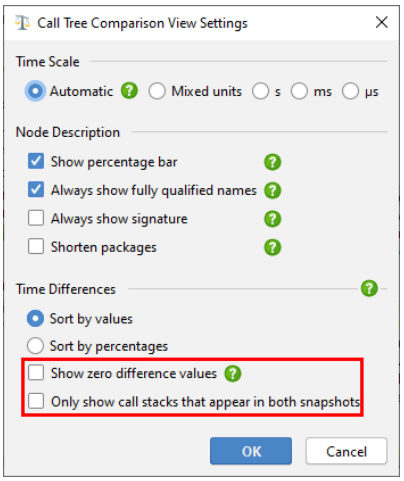
Like the objects comparison, CPU hot spot, probe hot spot and allocation hot spot comparisons are shown in a similar table.

Comparisons with trees

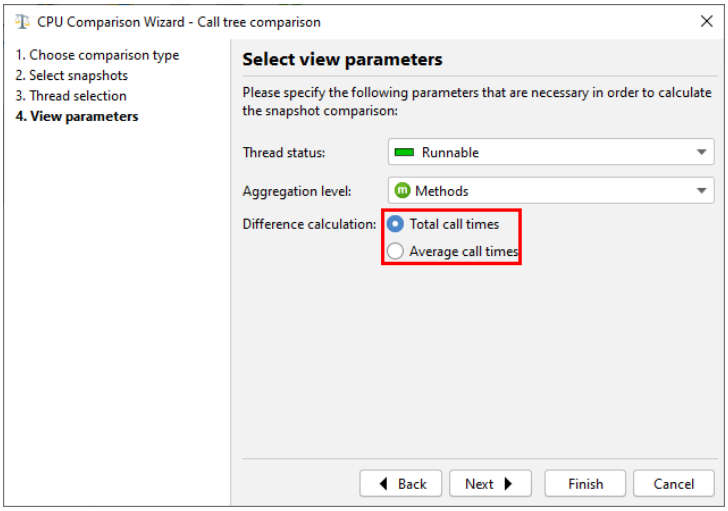
For each of the CPU call tree, the allocation call tree and the probe call tree you can calculate another tree that shows the differences between the selected snapshots. In contrast to the regular call tree views, the inline bar diagram now displays the change, either in red for increases or in green for decreases.



Depending on the task at hand, it may make it easier for you if you only see call stacks that are present in both snapshot files and that have changed from one snapshot file to the other. You can change this behavior in the view settings dialog.

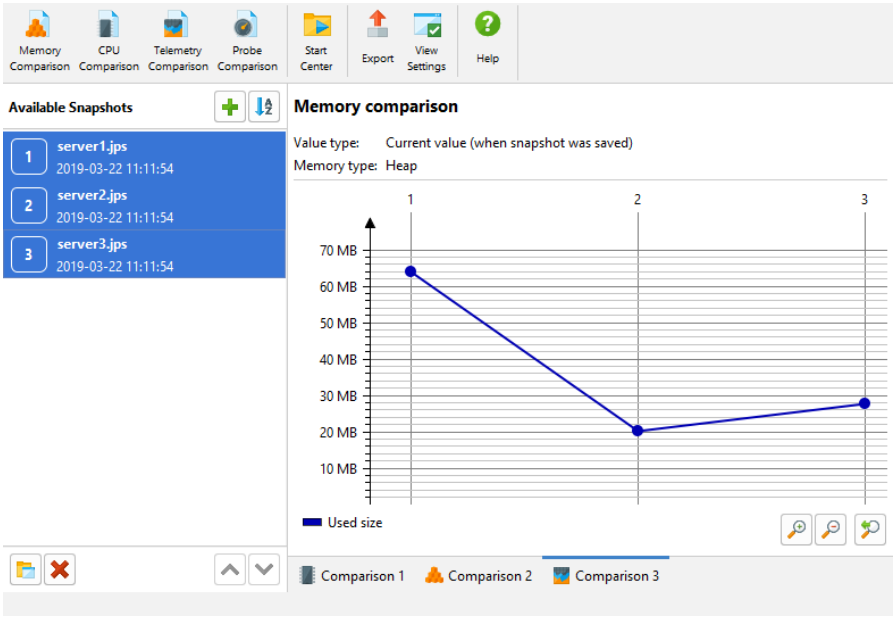


For the CPU and probe call tree comparisons it may be interesting to compare the average times instead of the total times. This is an option on the "View parameters" step of the wizard.

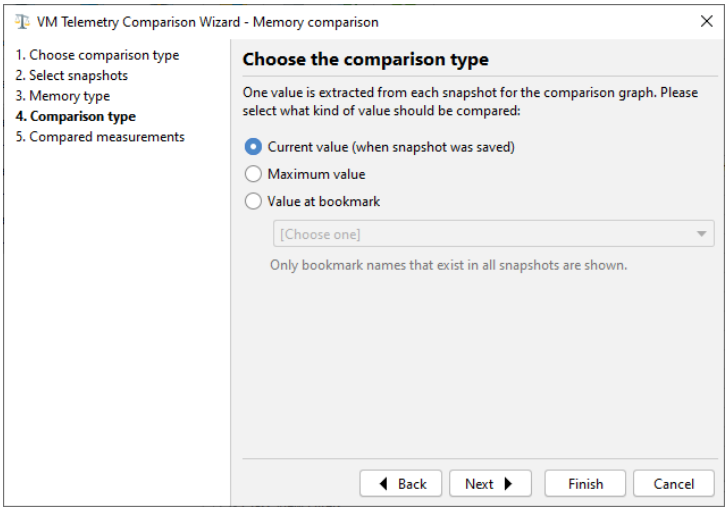


Telemetry comparisons

For telemetry comparisons you can compare more than 2 snapshots at the same time. If you don't select any snapshots in the snapshot selector, the wizard will assume that you want to compare all of them. Telemetry comparisons do not have a time axis, but show the numbered selected snapshots as an ordinal x-axis instead. The tool tips contain the full name of the snapshot.



The comparison extracts one number from each snapshot. Because telemetry data is time-resolved, there are multiple ways to do so. The "comparison type" step of the wizard gives you the option to use the value when the snapshot was saved, calculate the maximum value or find the value at a selected bookmark.

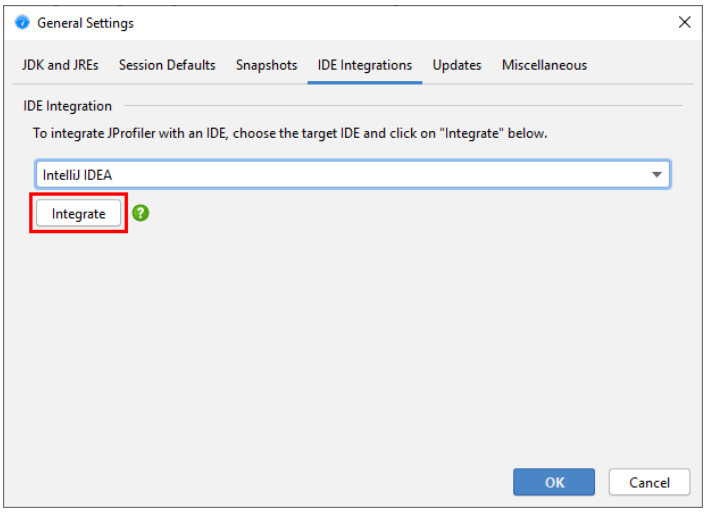


IDE Integrations

When you profile your application, the methods and classes that come up in JProfiler's views often lead to questions that can only be answered by looking their source code. While JProfiler provides a built-in source code viewer for that purpose, it has limited functionality. Also, when a problem is found, the next move is usually to edit the offending code. Ideally, there should be a direct path from the profiling views in JProfiler to the IDE, so you can inspect and improve code without any manual lookups.

Installing IDE integrations

JProfiler offers IDE integrations for IntelliJ IDEA, eclipse and NetBeans. To install an IDE plugin, invoke *Session->IDE Integrations* from the main menu. The plugin installation for IntelliJ IDEA is performed with the plugin management in the IDE, for other IDEs the plugin is installed directly be JProfiler. The installer also offers this action to make it easy to update the IDE plugin along with the JProfiler installation. The integration wizard connects the plugin with the current installation directory of JProfiler. In the IDE plugin settings, you can change the used version of JProfiler at any time. The protocol between the plugin and the JProfiler GUI is backwards compatible and can work with older versions of JProfiler as well.

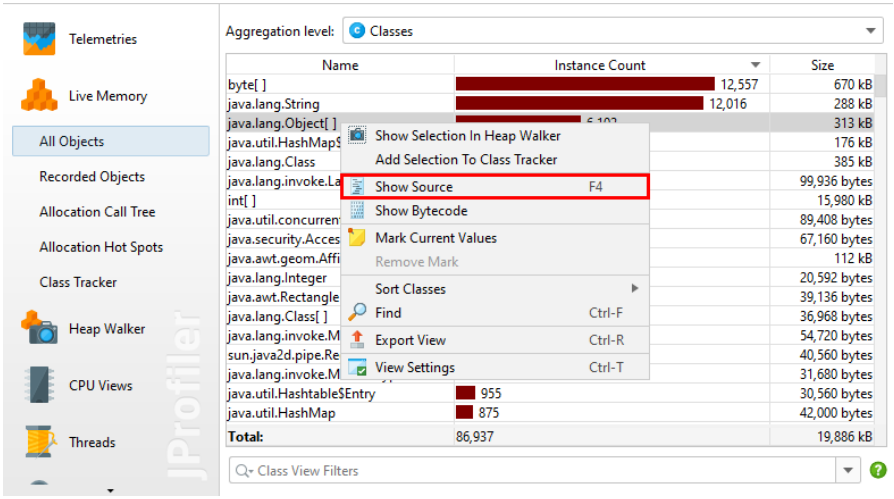


The IntelliJ IDEA integration can also be installed from the plugin manager. In that case, the plugin will ask you for the location of the JProfiler executable when you profile for the first time.

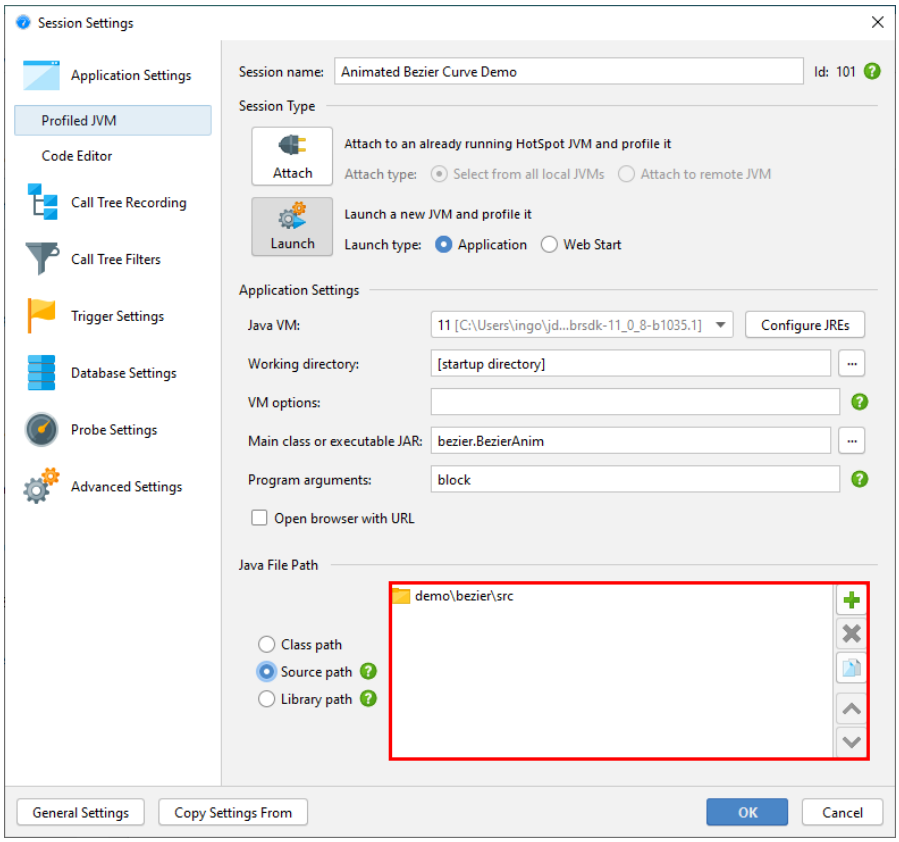
On different platforms, the JProfiler executable is located in different directories. On Windows, it's `bin\jprofiler.exe`, on Linux or Unix `bin/jprofiler` and on macOS there is a special helper shell script `Contents/Resources/app/bin/macos/jprofiler.sh` in the JProfiler application bundle for the IDE integrations.

Source code navigation

Everywhere a class name or a method name is shown in JProfiler, the context menu contains a *Show Source* action.

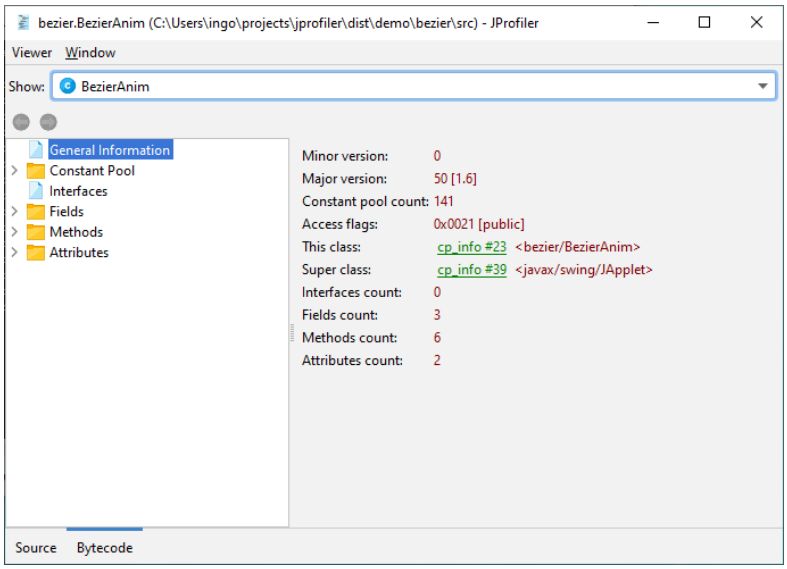


If the session was not started from the IDE, the built-in source code viewer is shown that utilizes line number tables in the compiled class files to find methods. A source file can only be found if its root directory or a containing ZIP file is configured in the application settings



Together with the source code display, a bytecode viewer based on the jclasslib bytecode viewer [(1)] shows the structure of the compiled class file.

[(1)] https://github.com/ingokegel/jclasslib



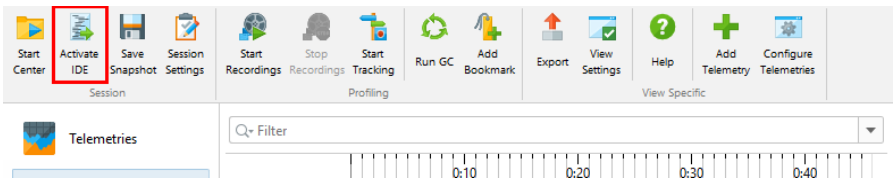
If the session is launched from the IDE, the integrated source code viewer is not used and the *Show Source* action defers to the IDE plugin. The IDE integrations support launched profiling sessions, opening saved snapshots as well as attaching to running JVMs.

For live profiling sessions, you start the profiled application for the IDE similarly to running or debugging it. The JProfiler plugin will then insert the VM parameter for profiling and connect a JProfiler window to it. JProfiler is running as a separate process and is started by the plugin if required. Source code navigation requests from JProfiler are sent to the associated project in the IDE. JProfiler and the IDE plugin cooperate to make window switching seamless without blinking task bar entries, just as if you were dealing with a single process.

When starting the session, the "Session startup" dialog lets you configure all profiling settings. The configured profiling settings that are used for a launched session are remembered by JProfiler on a per-project or on a per-run-configuration basis, depending on the IDE integrations. When a session is profiled for the first time, the IDE plugin automatically determines a list of profiled packages based on the topmost classes in the package hierarchy of your source files. At any later point, you can go to the filter settings step in the session settings dialog and use the reset button to perform this calculation again.

For snapshots, the IDE integration is set up by opening a snapshot file from within the IDE with the *File->Open* action or by double-clicking on it in the project window. Source code navigation from JProfiler will then be directed into the current project. Finally, the IDE plugin adds an *Attach to JVM* action to the IDE that lets you select a running JVM and get source code navigation into the IDE, similar to the mechanism for snapshots.

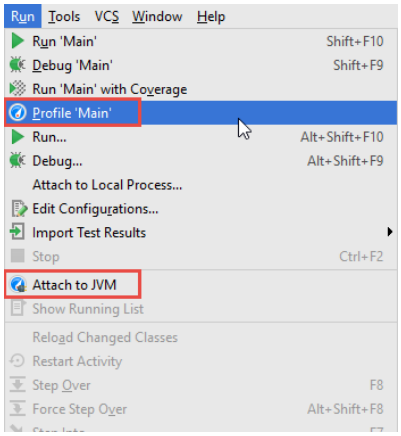
Sometimes you may want to switch to the IDE without a particular class or method in mind. For that purpose, the tool bar in the JProfiler window has an *Activate IDE* button that is shown for profiling sessions that are opened by an IDE integration. The action is bound to the F11 key, just like the JProfiler activation action in the IDE, so you can switch back and forth between the IDE and JProfiler with the same key binding.

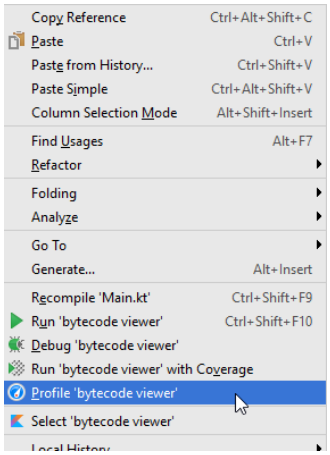


IntelliJ IDEA integration

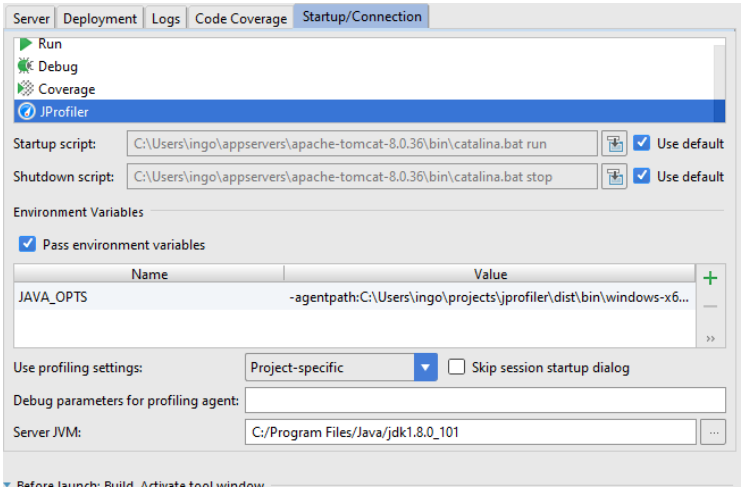
To profile your application from IntelliJ IDEA, choose one of the profiling commands in the *Run* menu, the context menu in the editor, or click on the corresponding toolbar button.





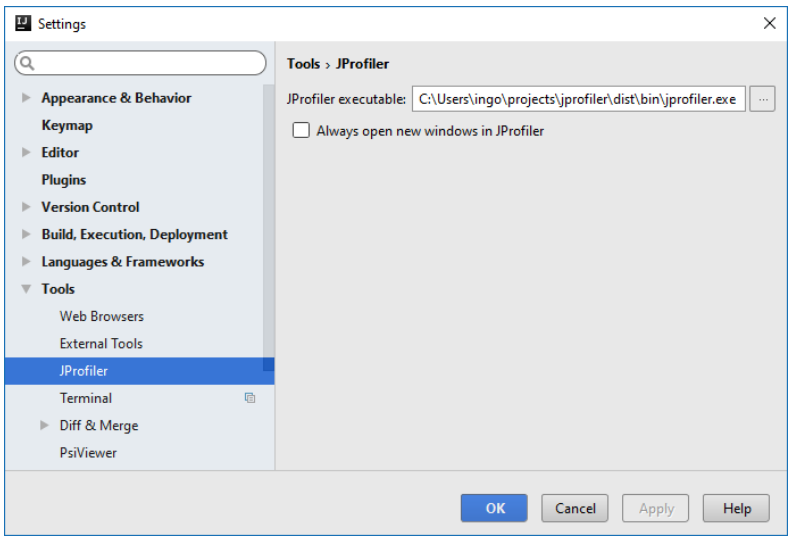


JProfiler can profile most run configuration types from IDEA, including applications servers. To configure further settings, edit the run configuration, choose the *Startup/Connection* tab, and select the *JProfiler* entry. The screen shot below shows the startup settings for a local server configuration.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

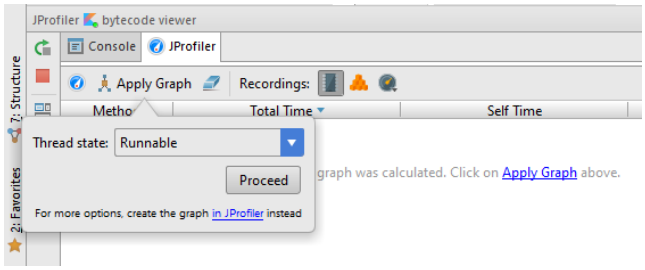


The profiled application is then started just as with the usual "Run" commands. Precise source code navigation is implemented for Java and Kotlin.

On the *JProfiler* tab of the IDE settings, you can adjust the used JProfiler executable and whether you always want to open a new window in JProfiler for new profiling sessions.



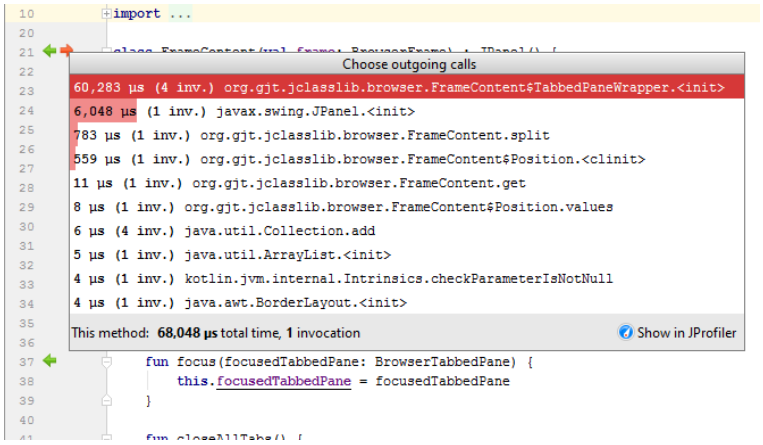
The JProfiler tool window in IDEA is shown when you profile a run configuration from IDEA, when you open a JProfiler snapshot or when you attach to a running JVM.



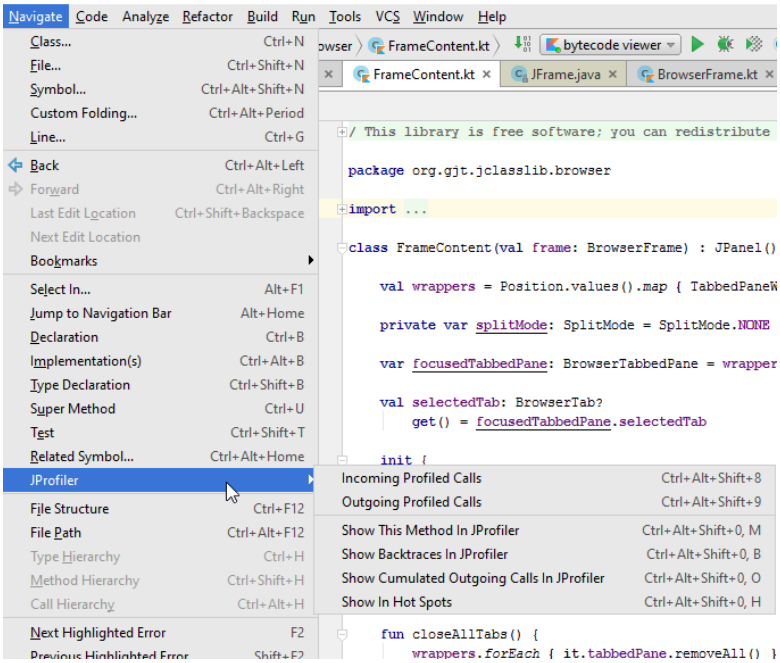
The action in the tool bar with the JProfiler icon activates the JProfiler window. On the right side of the tool bar, several toggle buttons give access to important recording actions in JProfiler. If a recording is active, the corresponding toggle button is selected.

Of particular relevance is the *CPU recording* action, because CPU graph data can be shown directly in the IDE. The only parameter for graph calculation that is offered in the IDE is the thread status. To configure advanced parameters like thread selection or to use the call tree root, call tree removal and call tree view filter settings from the call tree view, you can generate the graph in the JProfiler window, it will then be shown in the IDE as well.

When you calculate a graph, the list of hot spots will be populated and the source code will be annotated with gutter icons for incoming and outgoing calls. The popup on the gutter icons shows an inline graph, clicking on a method will navigate to it. The list of hot spots shows you interesting entry points for analyzing the graph. When double-clicking on a table row, the source code is shown.



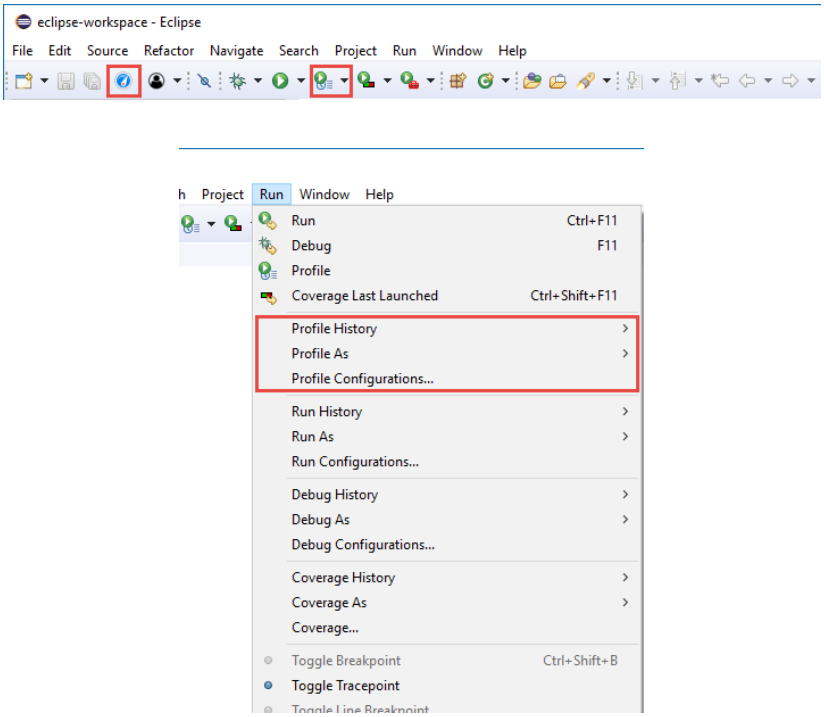
The *Show in JProfiler* button contains actions that activate the JProfiler window, either the selected node in the method graph or the corresponding call tree analysis in the method graph. For outgoing calls, the "Cumulated outgoing calls" analysis is offered, for the incoming calls, the "Backtraces" analysis is shown. All these actions are also available in the context menu of the hot spot list or as keyboard actions.



eclipse integration

The eclipse plugin can profile most common launch configuration types including test run configurations and WTP run configurations. The eclipse plugin only works with the full eclipse SDKs and not with partial installations of the eclipse framework.

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, except for the *Run->Attach JProfiler to JVM* menu item which is added by the JProfiler plugin.



If the menu item *Run->Profile ...* does not exist in the Java perspective, enable the "Profile" actions for this perspective under *Window->Perspective->Customize Perspective* by bringing the *Action Set Availability* tab to front and selecting the *Profile* checkbox.

Several JProfiler-related settings including the location of the JProfiler executable can be adjusted in eclipse under *Window->Preferences->JProfiler*.

NetBeans integration

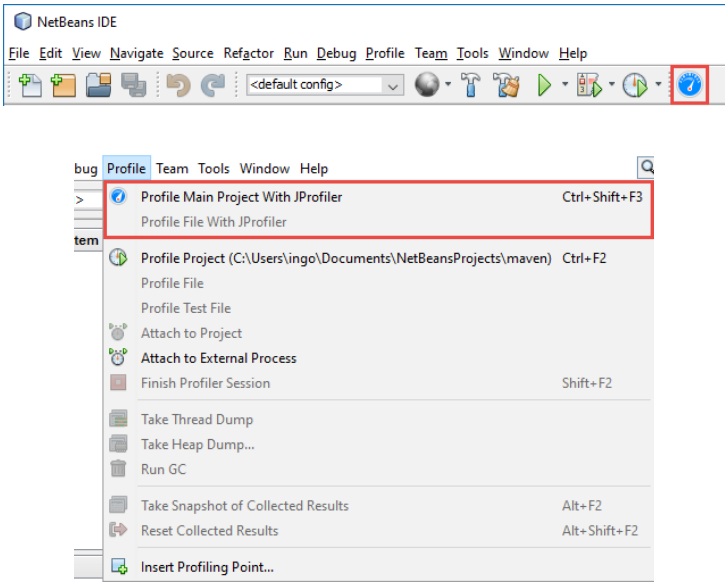
In NetBeans, you can profile standard, free form and Maven projects that use the exec Maven plugin. To profile your application from NetBeans, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button. For Maven projects that start an application in another way and for Gradle projects, start the project normally and use the *Profile->Attach JProfiler To A Running JVM* action in the menu.

For free form projects, you have to debug your application once before trying to profile it, because the required file`nbproject/ide-targets.xml` is set up by the debug action. JProfiler will add a target named "profile-jprofiler" to it with the same contents as the debug target and will try to modify the VM parameters as needed. If you have problems profiling a free form project, check the implementation of this target.

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 main project with JProfiler* starts the Tomcat server with profiling enabled.

If you use NetBeans with the bundled GlassFish Server and your main project is set up to use a GlassFish Server, selecting *Profile main project with JProfiler* starts the application server with profiling enabled.

The location of the JProfiler executable and the policy for opening new JProfiler windows can be adjusted under *Miscellaneous->JProfiler* in the options dialog.



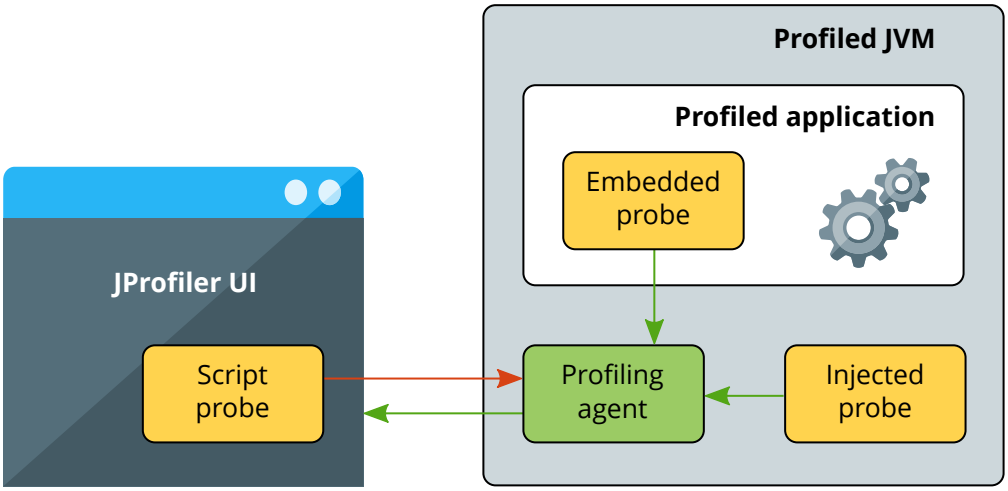
A Custom Probes

A.1 Probe Concepts

To develop a custom probe for JProfiler, you should be aware of some basic concepts and terminology. The common basis of all of JProfiler's probes is that they intercept specific methods and use the intercepted method parameters and other data sources to build a string with interesting information that you would want to see in the JProfiler UI.

The initial problem when defining a probe is how to specify the intercepted methods and get an environment where you can use the method parameters and other relevant objects for building the string. In JProfiler, there are 3 different ways to do that:

- A **script probe** [p. 138] is completely defined in the JProfiler UI. You can right-click a method in the call tree, choose the script probe action and enter an expression for the string in a built-in code editor. This is great for experimenting with probes, but only exposes a very limited segment of the capabilities of custom probes.
- The **embedded probe** [p. 147] API can be called from your own code. If you write a library, a database driver or a server, you can ship probes with your product. Anybody who profiles your product with JProfiler, will get your probes added automatically to the JProfiler UI.
- With the **injected probe** [p. 142] API, you can write probes for 3rd party software in your IDE using the full capability of JProfiler's probe system. The API makes use of annotations to define the interceptions and to inject method parameters and other useful objects.

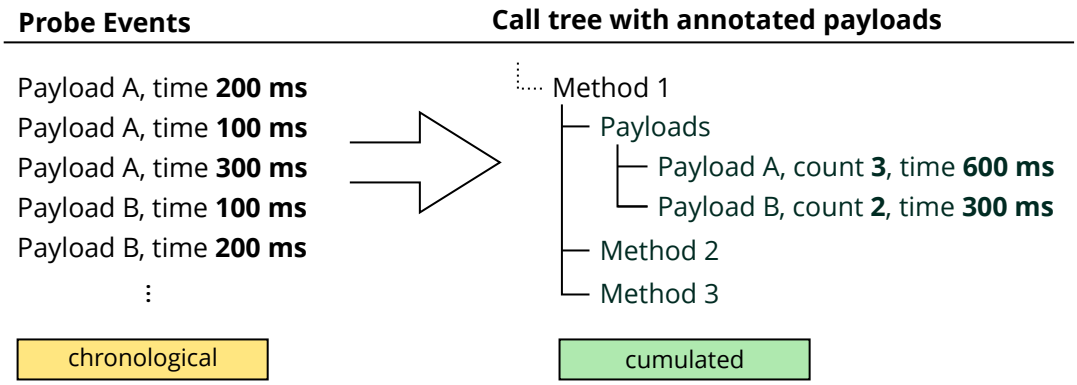


The next question is: what should JProfiler do with the string that you have created? There are two different strategies available: payload creation or call tree splitting.

Payload creation

The string that is built by a probe can be used to create a **probe event**. The event has a description that is set to that string, a duration that is equal to the invocation time of the intercepted method, as well as an associated call stack. At their corresponding call stacks, probe descriptions and

timings are cumulated and saved as **payloads** into the call tree. While events are consolidated after a certain maximum number, the cumulated payloads in the call tree show the total numbers for the entire recording period. If both CPU data and your probe are being recorded, the probe call tree view will show the merged call stacks with the payload strings as leaf nodes and the CPU call tree view will contain **annotated links** into the probe call tree view.



Just like for CPU data, payloads can be shown in a call tree or in a hot spots view. The hot spots show which payloads are responsible for most of the expended time and the back traces show you which parts of your code are responsible for creating these payloads. In order to get a good list of hot spots, the payload strings should not contain any unique IDs or timestamps, because if every payload string is different, there will be no cumulation and no clear distribution of hot spots. For example, in the case of a prepared JDBC statement, the parameters should not be included in the payload string.

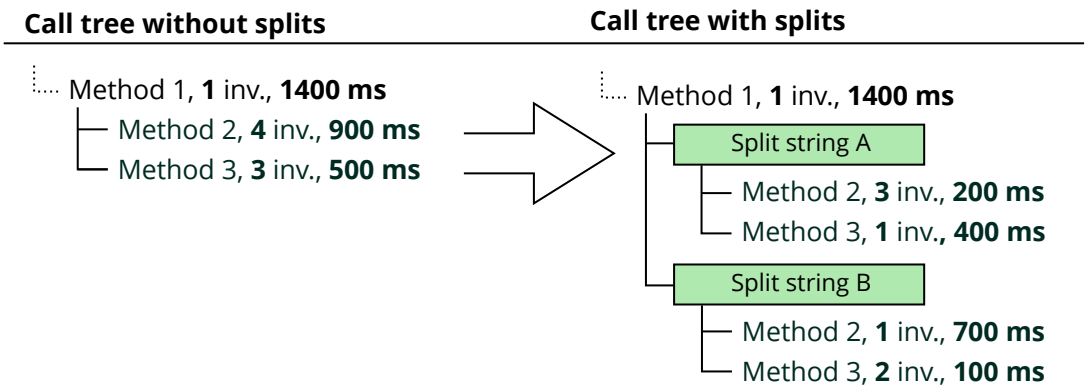
Script probes create payloads automatically from the return value of the configured script. Injected probes are similar, they return the payload description from an interception handler method annotated with `PayloadInterception` either as a string or as a `Payload` object for advanced functionality. Embedded probes, on the other hand, create payloads by calling `Payload.exit` with the payload description as an argument, where the time between `Payload.enter` and `Payload.exit` is recorded as the probe event duration.

Payload creation is most useful if you're recording calls to services that happen at different call sites. A typical example is a database driver where the payload string is some form of query string or command. The probe takes the perspective of the call site, where the work that is measured is performed by another software component.

Call tree splitting

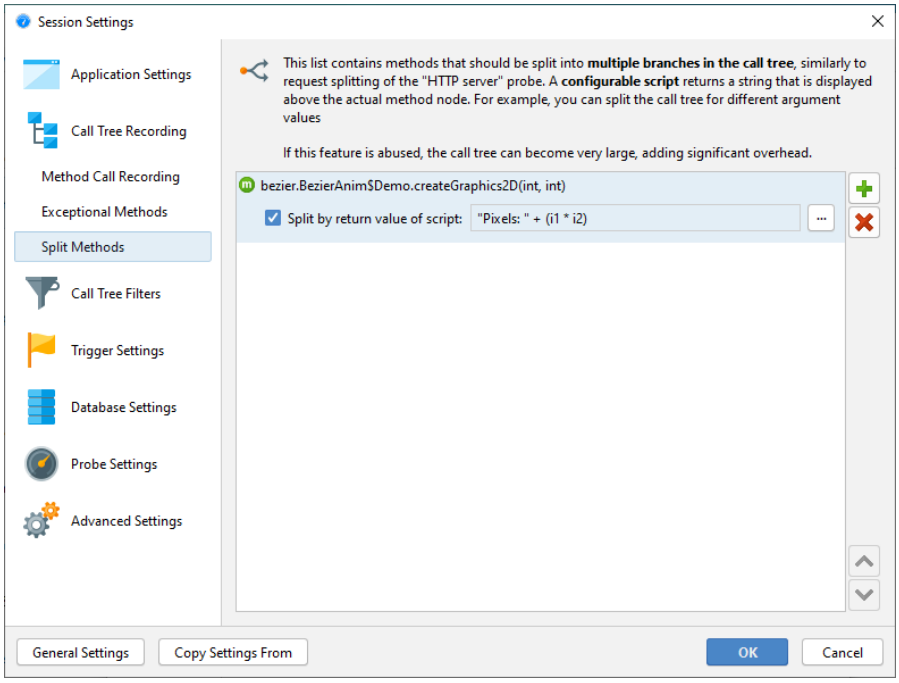
The probe can also take the perspective of the execution site. In that case, it is not important how the intercepted method is called, but rather what method calls are executed after it. A typical example is a probe for a servlet container where the extracted string is a URL.

More important than creating payloads is now the ability to split the call tree for each distinct string that is built by the probe. For each such string, a splitting node will be inserted into the call tree that contains the cumulated call tree of all corresponding invocations. Where otherwise there would be just one cumulated call tree, now there is a set of splitting nodes segmenting the call tree into different parts that can be analyzed separately.



Multiple probes can produce nested splits, a single probe by default produces only one split level, unless is has been configured as **reentrant** which is not supported for script probes.

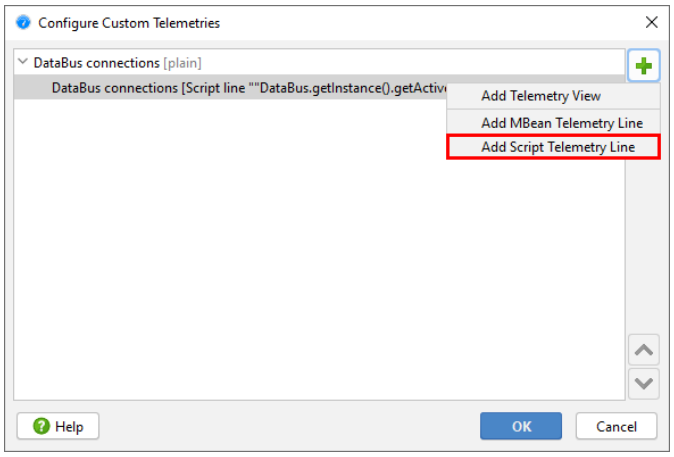
In the JProfiler UI, call tree splitting is not bundled with the script probe feature, but is a separate feature [p. 165]called "Split methods". They just split the call tree without creating payloads, so no probe view with name and description is required. Injected probes return the split string from an interception handler method annotated with `SplitInterception`, while embedded probes call `Split.enter` with the split string.



Telemetries

Custom probes have two default telemetries: The event frequency and the average event duration. Injected and embedded probes support additional telemetries that are created with annotated methods in the probe configuration classes. In the JProfiler UI, script telemetries are independent

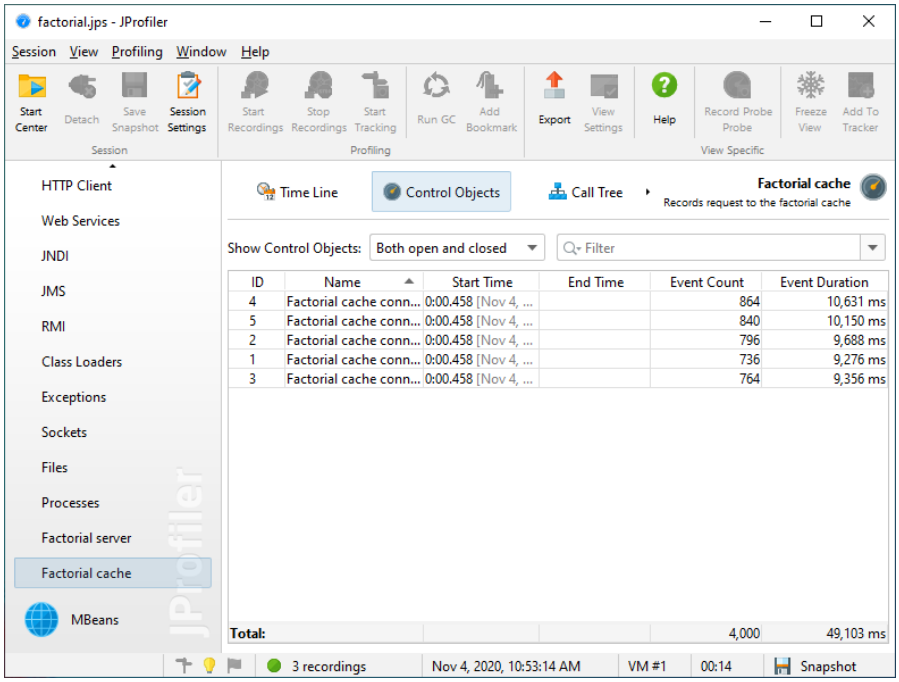
from the script probe feature and are found in the "Telemetries" section, under the *Configure Telemetries* button in the tool bar.



Telemetry methods are polled once per second. In the `Telemetry` annotation, you can configure the unit and a scale factor. With the `line` attribute, multiple telemetries can be combined into a single telemetry view. With the `stacked` attribute of the `TelemetryFormat` you can make the lines additive and show them as a stacked line graph. The telemetry-related APIs in the embedded and injected probes are equivalent but only applicable for their respective probe types.

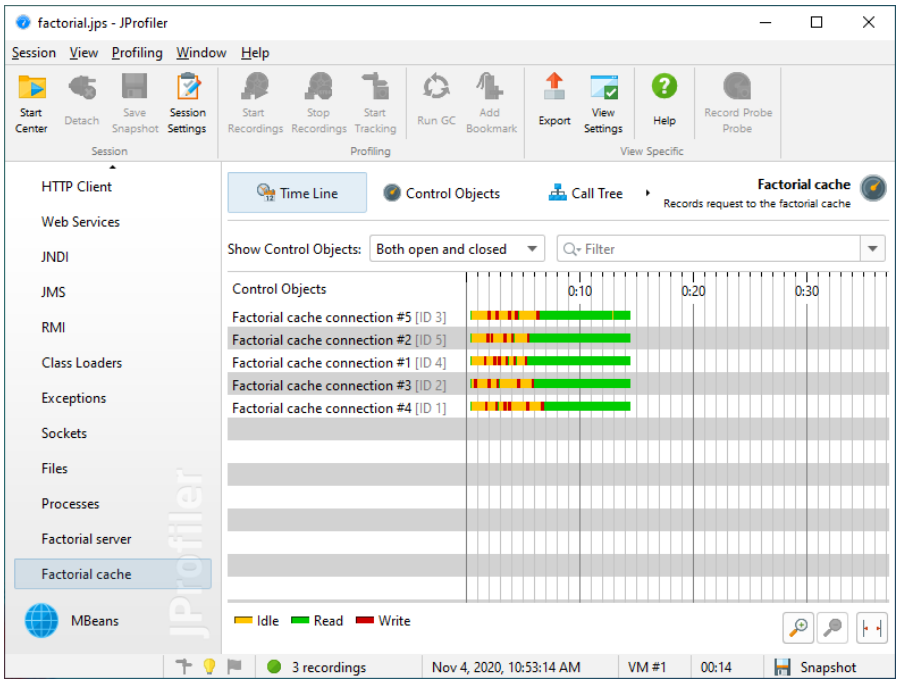
Control objects

Sometimes it is interesting to tie probe events to associated long-lived objects that are called "control objects" in JProfiler. For example, in a database probe, that role is taken by the physical connection that executes a query. Such control objects can be opened and closed with the embedded and the injected probe APIs which generates corresponding events in the probe events view. When a probe event is created, the control object can be specified, so that the probe event contributes to the statistics that is shown in the "Control objects" view of the probe.



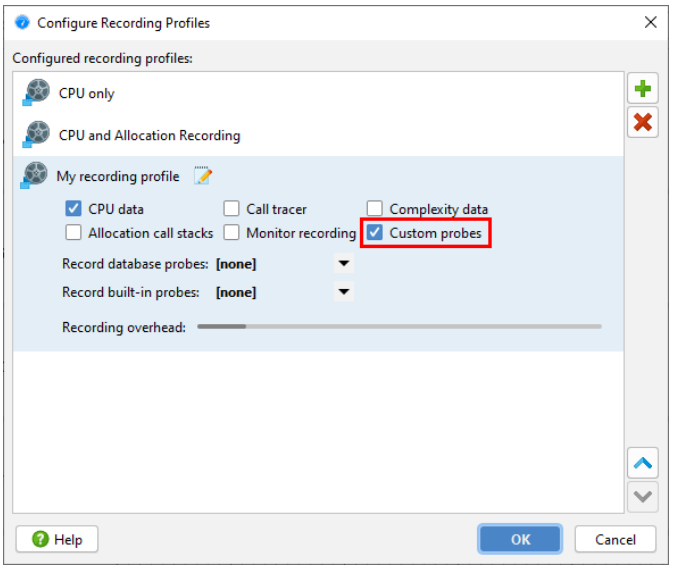
Control objects have display names that have to be specified when they are opened. If a new control object is used when creating a probe event, the probe has to provide a name resolver in its configuration.

In addition, probes can define custom event types via an enum class. When the probe event is created, one of those types can be specified and shows up in the events view where you can can filter for single event types. More importantly, the time line view of the probe that shows control objects as lines on a time axis is colored according to the event type. For a probe without custom event types, the coloring shows the idle state where no events are recorded and the default event state for the duration of probe events. With custom types, you can differentiate states, for example "read" and "write".

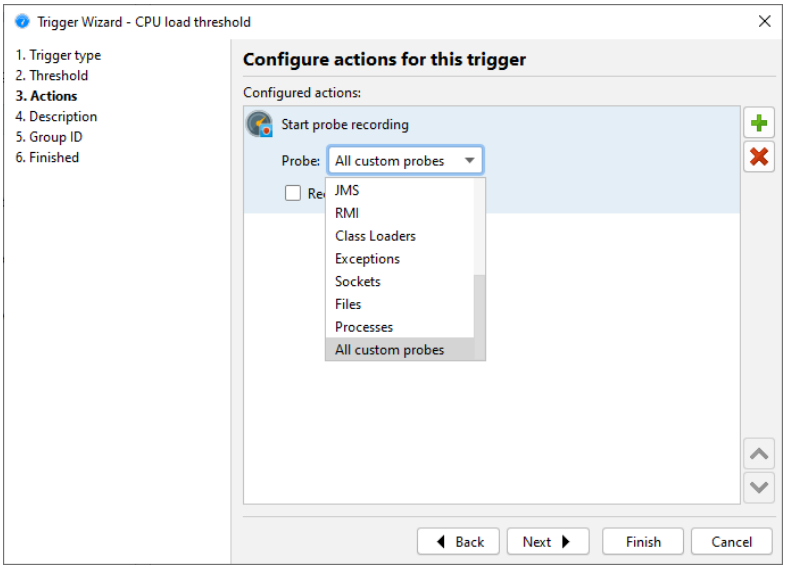


Recording

Like for all probes, custom probes do not record data by default, but you have to enable and disable recording as necessary. While you can use the manual start/stop action in the probe view, it is often necessary to switch on probe recording at the beginning. Because JProfiler does not know about custom probes in advance, the recording profiles have a *Custom probes* check box that applies to all custom probes.



Similarly, you can choose *All custom probes* for the trigger actions that start and stop probe recording.



For programmatic recording, you can call `Controller.startProbeRecording(Controller.PROBE_NAME_ALL_CUSTOM, ProbeRecordingOptions.EVENTS)` to record all custom probes, or pass the class name of the probe in order to be more specific.

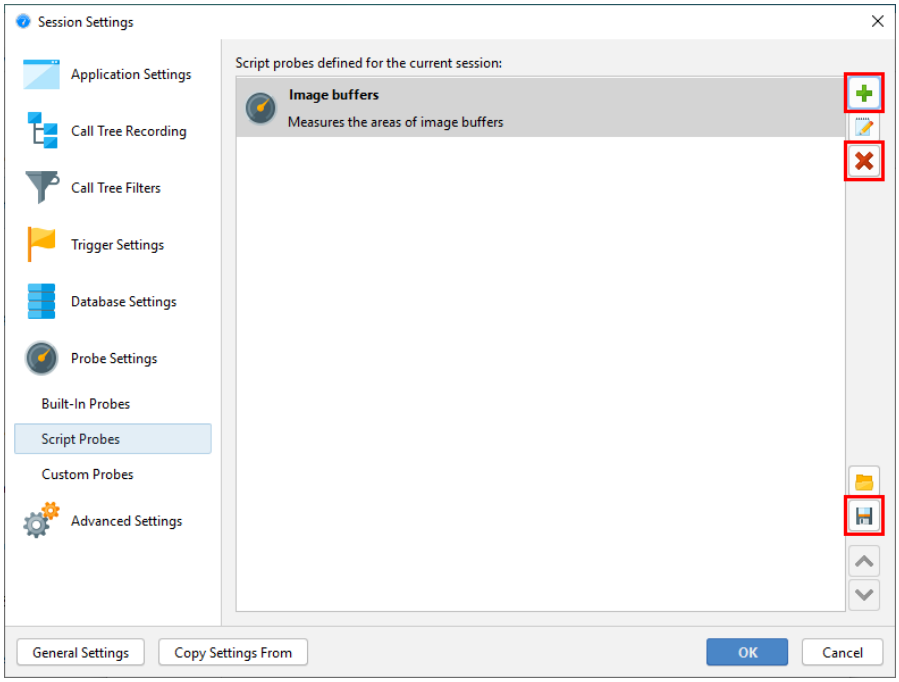
A.2 Script Probes

Developing a custom probe in your IDE requires a clear understanding of the interception point and the benefits that the probe will provide. With script probes, on the other hand, you can quickly define simple probes directly in the JProfiler GUI and experiment without having to learn any API. Unlike embedded or injected custom probes, script probes can be redefined during a running profiling session, leading to a fast edit-compile-test loop.

Defining script probes

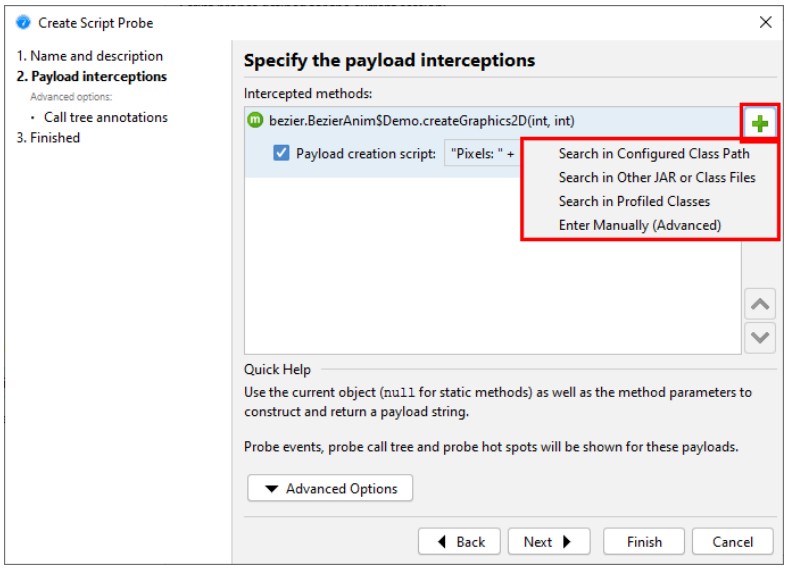
A script probe is defined by selecting an intercepted method and entering a script that returns the payload string for the probe. Multiple such method-script pairs can be bundled in a single probe.

The script probe configuration is accessed in the session settings. This is the place to create and delete script probes as well as for saving your script probes to a set that can be imported by other profiling sessions.

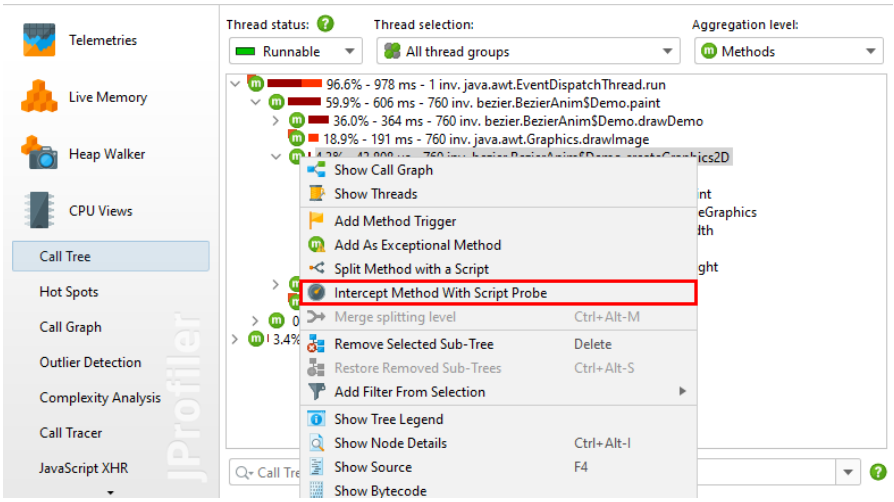


Each script probe needs a name and optionally a description. The name is used to add a probe view to JProfiler's view selector in the "JEE & Probes" section. The description is shown in the header of the probe view and should be a short explanation of its purpose.

For selecting a method you have multiple options, including selecting a class from the configured classpath or selecting a class from the profiled classes if the profiling session is already running. In the second step, you can then select a method from the selected class.



A much easier way to select the intercepted method is from the call tree view. In the context menu, the *Intercept Method With Script Probe* action will ask you if you want to create a new probe or add an interception to an existing probe.

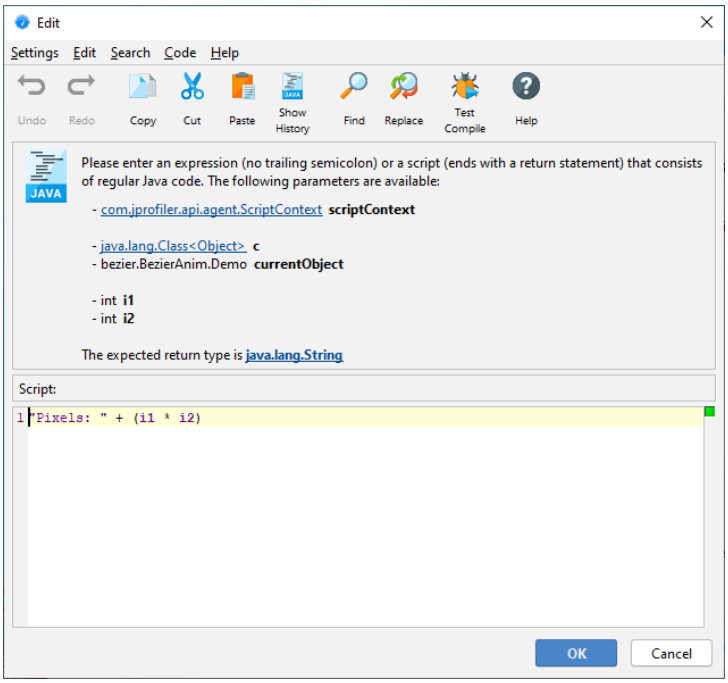


Probe scripts

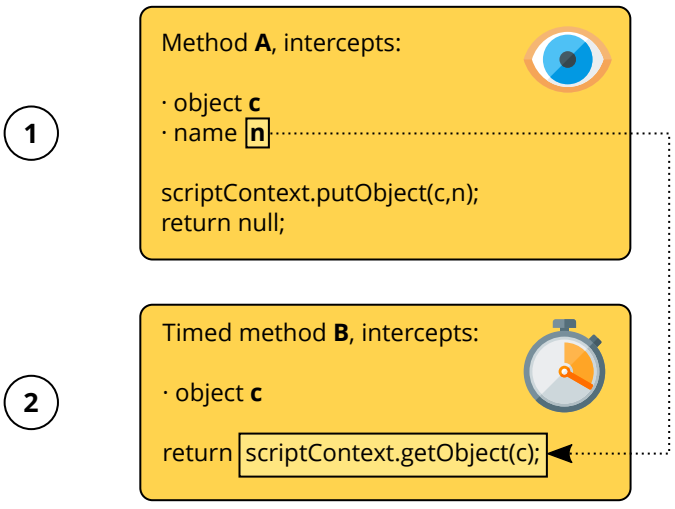
In the script editor, you have access to all parameters of the intercepted method as well as the object on which the method was called. If you need access to the return value of the intercepted method or any thrown exceptions, you have to write an embedded or injected probe.

In this environment, your script can construct the payload string, either as an expression or as a sequence of statements with a return statement. The simplest version of such a script simply returns `parameter.toString()` for one parameter or `String.valueOf(parameter)` for a parameter with a primitive type. If it returns `null`, no payload will be created.

If you record CPU and probe data at the same time, the call tree view in the CPU section will show links into the probe view at the appropriate call stacks. Alternatively, you can select to show the payload strings inline in the CPU call tree view. The "Payload interceptions->Call tree annotations" step of the probe wizard contains this option.



One more parameter that is available to the script is the script context, an object of type `com.jprofiler.api.agent.ScriptContext` that allows you to store data between invocations of any script that is defined for the current probe. For example, let's suppose that the intercepted method A only sees objects that have no good text representation, but the association between object and display name could be obtained by intercepting method B. Then you could intercept method B in the same probe and save the object-to-text association directly to the script context. In method A you would then get that display text from the script context and use it to build the payload string.



If these kinds of concerns get too complex, you should consider switching to the embedded or injected probe APIs.

Missing capabilities

Script probes are designed to facilitate an easy entry to custom probe development, but they are missing a couple of capabilities from the full probe system that you should be aware of:

- Script probes cannot do call tree splitting. In the JProfiler UI this is a separate feature as explained in the custom probes concepts [p. 132]. Embedded and injected probes offer call tree splitting functionality directly.
- Script probes cannot create control objects or create custom probe event types. This is only possible with embedded or injected probes.
- Script probes cannot access return values or thrown exceptions, unlike embedded and injected probes.
- Script probes cannot handle reentrant interceptions. If a method is called recursively, only the first call into it is intercepted. Embedded and injected probes offer you fine-grained control over reentrant behavior.
- It is not possible to bundle telemetries other than default telemetries into the probe view. Instead, you can use the script telemetry feature as shown in the custom probes concepts. [p. 132]

A.3 Injected Probes

Similarly to script probes, injected probes define interception handlers for selected methods. However, injected probes are developed outside of the JProfiler GUI in your IDE and rely on the injected probe API that is provided by JProfiler. The API is licensed under the permissive Apache License, version 2.0, making it easy distribute the associated artifacts.

The best way to get started with injected probes is to study the example in the `api/samples/simple-injected-probe` directory of your JProfiler installation. Execute `../gradlew` in that directory to compile and run it. The gradle build file `build.gradle` contains further information about the sample. The example in `api/samples/advanced-injected-probe` shows more features of the probe system, including control objects.

Development environment

The probe API that you need for developing an injected probe is contained in the single artifact with maven coordinates

```
group: com.jprofiler
artifact: jprofiler-probe-injected
version: <JProfiler version>
```

where the JProfiler version corresponding to this manual is 12.0.4.

Jar, source and javadoc artifacts are published to the repository at

```
https://maven.ej-technologies.com/repository
```

You can either add the probe API to your development class path with a build tool like Gradle or Maven, or use the JAR file

```
api/jprofiler-probe-injected.jar
```

in the JProfiler installation.

To browse the Javadoc, go to

```
api/javadoc/index.html
```

That javadoc combines the javadoc for all APIs that are published by JProfiler. The overview for the `com.jprofiler.api.probe.injected` package is a good starting point for exploring the API.

Probe structure

An injected probe is a class annotated with `com.jprofiler.api.probe.injected.Probe`. The attributes of that annotation expose configuration options for the entire probe. For example, if you create a lot of probe events that are not interesting for individual inspection, the `events` attribute allows you to disable the probe events view and reduce overhead.

```
@Probe(name = "Foo", description = "Shows foo server requests", events = "false")
public class FooProbe {
    ...
}
```

To the probe class, you add specially annotated static methods in order to define interception handlers. The `PayloadInterception` annotation creates payloads while the `SplitInterception` annotation splits the call tree. The return value of the handler is used as the payload or the split string, depending on the annotation. Like for script probes, if you return `null`, the interception has no effect. Timing information is automatically calculated for the intercepted method.

```
@Probe(name = "FooBar")
public class FooProbe {
    @PayloadInterception(
        invokeOn = InvocationType.ENTER,
        method = @MethodSpec(className = "com.bar.Database",
                             methodName = "processQuery",
                             parameterTypes = {"com.bar.Query"},
                             returnType = "void"))
    public static String fooRequest(@Parameter(0) Query query) {
        return query.getVerbose();
    }

    @SplitInterception(
        method = @MethodSpec(className = "com.foo.Server",
                             methodName = "handleRequest",
                             parameterTypes = {"com.foo.Request"},
                             returnType = "void"))
    public static String barQuery(@Parameter(0) Request request) {
        return request.getPath();
    }
}
```

As you can see in the above example, both annotations have a `method` attribute for defining the intercepted methods with a `MethodSpec`. In contrast to script probes, the `MethodSpec` can have an empty class name, so all methods with a particular signature are intercepted, regardless of the class name. Alternatively, you can use the `subtypes` attribute of the `MethodSpec` to intercept entire class hierarchies.

Unlike for script probes where all parameters are available automatically, the handler methods declare parameters to request values of interest. Each parameter is annotated with an annotation from the `com.jprofiler.api.probe.injected.parameter` package, so the profiling agent knows which object or primitive value has to be passed to the method. For example, annotating a parameter of the handler method with `@Parameter(0)` injects the first parameter of the intercepted method.

Method parameters of the intercepted method are available for all interception types. Payload interceptions can access the return value with `@ReturnValue` or a thrown exception with `@ExceptionValue` if you tell the profiling agent to intercept the exit rather than the entry of the method. This is done with the `invokeOn` attribute of the `PayloadInterception` annotation.

In contrast to script probes, injected probes handlers can be called for recursive invocations of the intercepted method if you set the `reentrant` attribute of the interception annotation to `true`. With a parameter of type `ProbeContext` in your handler method, you can control the probes's behavior for nested invocations by calling `ProbeContext.getOuterPayload()` or `ProbeContext.restartTiming()`.

Advanced interceptions

Sometimes a single interception is not sufficient to collect all necessary information for building the probe string. For that purpose, your probe can contain an arbitrary number of interception handlers annotated with `Interception` that do not create payloads or splits. Information can be stored in static fields of your probe class. For thread safety in a multi-threaded environment,

you should use `ThreadLocal` instances for storing reference types and the atomic numeric types from the `java.util.concurrent.atomic` package for maintaining counters.

Under some circumstances, you need interceptions for both method entry and method exit. A common case is if you maintain state variables like `inMethodCall` that modify the behavior of another interception. You can set `inMethodCall` to `true` in the entry interception, which is the default interception type. Now you define another static method directly below that interception and annotate it with `@AdditionalInterception(invokeOn = InvocationType.EXIT)`. The intercepted method is taken from the interception handler above, so you do not have to specify it again. In the method body, you can set your `inMethodCall` variable to `false`.

```
...

private static final ThreadLocal<Boolean> inMethodCall =
    ThreadLocal.withInitial(() -> Boolean.FALSE);

@Interception(
    invokeOn = InvocationType.ENTER,
    method = @MethodSpec(className = "com.foo.Server",
                         methodName = "internalCall",
                         parameterTypes = {"com.foo.Request"},
                         returnType = "void"))
public static void guardEnter() {
    inMethodCall.set(Boolean.TRUE);
}

@AdditionalInterception(InvocationType.EXIT)
public static void guardExit() {
    inMethodCall.set(Boolean.FALSE);
}

@SplitInterception(
      method = @MethodSpec(className = "com.foo.Server",
                           methodName = "handleRequest",
                           parameterTypes = {"com.foo.Request"},
                           returnType = "void"),
      reentrant = true)
public static String splitRequest(@Parameter(0) Request request) {
    if (!inMethodCall.get()) {
        return request.getPath();
    } else {
        return null;
    }
}

...
```

You can see a working example of this use case in `api/samples/advanced-injected-probe/src/main/java/AdvancedAwtEventProbe.java`.

Control objects

The control objects view is not visible unless the `controlObjects` attribute of the `Probe` annotation is set to `true`. For working with control objects you have to obtain a `ProbeContext` by declaring a parameter of that type in your handler method. The sample code below shows how to open a control object and associate it with a probe event.

```
@Probe(name = "Foo", controlObjects = true, customTypes = MyEventTypes.class)
public class FooProbe {
    @Interception(
            invokeOn = InvocationType.EXIT,
            method = @MethodSpec(className = "com.foo.ConnectionPool",
                        methodName = "createConnection",
                        parameterTypes = {},
                        returnType = "com.foo.Connection"))
    public static void openConnection(ProbeContext pc, @ReturnValue Connection c) {
        pc.openControlObject(c, c.getId());
    }

    @PayloadInterception(
            invokeOn = InvocationType.EXIT,
            method = @MethodSpec(className = "com.foo.ConnectionPool",
                        methodName = "createConnection",
                        parameterTypes = {"com.foo.Query", "com.foo.Connection"},
                        returnType = "com.foo.Connection"))
    public static Payload handleQuery(
        ProbeContext pc, @Parameter(0) Query query, @Parameter(1) Connection c) {
        return pc.createPayload(query.getVerbose(), c, MyEventTypes.QUERY);
    }

    ...

}
```

Control objects have a defined lifetime and the probe view records their open and close times in the timeline and the control objects view. If possible, you should open and close control objects explicitly by calling `ProbeContext.openControlObject()` and `ProbeContext.closeControlObject()`. Otherwise you have to declare a static method annotated with `@ControlObjectName` that resolves the display names of control objects.

Probe events can be associated with control objects if your handler method returns instances of `Payload` instead of `String`. The `ProbeContext.createPayload()` method takes a control object and a probe event type. The enum with the allowed event types has to be registered with the `customTypes` attribute of the `Probe` annotation.

Control objects have to be specified at the start of the time measurement which corresponds to the method entry. In some cases, the name of payload string will only be available at method exit, because it depends on the return value or other interceptions. In that case, you can use `ProbeContext.createPayloadWithDeferredName()` to create a payload object without a name. Define an interception handler annotated with `@AdditionalInterception(invokeOn = InvocationType.EXIT)` right below and return a `String` from that method, it will then automatically be used as the payload string.

Overriding the thread state

When measuring execution times for database drivers or native connectors to external resources, it sometimes becomes necessary to tell JProfiler to put some methods into a different thread state. For example, it is useful to have database calls in the "Net I/O" thread state. If the communication mechanism does not use the standard Java I/O facilities, but some native mechanism, this will not automatically be the case.

With a pair of `ThreadState.NETIO.enter()` and `ThreadState.exit()` calls, the profiling agent adjusts the thread state accordingly.

```
...

@Interception(invokeOn = InvocationType.ENTER, method = ...)
public static void enterMethod(ProbeContext probeContext, @ThisValue JComponent
component) {
    ThreadState.NETIO.enter();
}

@AdditionalInterception(InvocationType.EXIT)
public static void exitMethod() {
    ThreadState.exit();
}

...
```

Deployment

There are two ways to deploy injected probes, depending on whether you want to put them on the classpath or not. With the VM parameter

```
-Djprofiler.probeClassPath=...
```

a separate probe class path is set up by the profiling agent. The probe classpath can contain directories and class files, separated with ';' on Windows and ':' on other platforms. The profiling agent will scan the probe classpath and find all probe definitions.

If it's easier for you to place the probe classes on the classpath, you can set the VM parameter

```
-Djprofiler.customProbes=...
```

to a comma-separated list of fully qualified class names. For each of these class names, the profiling agent will try to load an injected probe.

A.4 Embedded Probes

If you control the source code of the software component that is the target of your probe, you should write an embedded probe instead of an injected probe.

Most of the initial effort when writing an injected probe goes into specifying the intercepted methods and selecting the injected objects as method parameters for the handler method. With embedded probes, this is not necessary, because you can call the embedded probe API directly from the monitored methods. Another advantage of embedded probes is that deployment is automatic. Probes ship together with your software and appear in the JProfiler UI when the application is profiled. The only dependency you have to ship is a small JAR file licensed under the Apache 2.0 License that mainly consists of empty method bodies serving as hooks for the profiling agent.

Development environment

The development environment is the same as for injected probes, with the difference that the artifact name is `jprofiler-probe-embedded` instead of `jprofiler-probe-injected` and that you ship the JAR file with your application instead of developing the probe in a separate project. The probe API that you need for adding an embedded probe to your software component is contained in the single JAR artifact. In the javadoc, start with the package overview for `com.jprofiler.api.probe.embedded` when you explore the API.

Just like for injected probes, there are two examples for embedded probes as well. In `api/samples/simple-embedded-probe`, there is an example that gets you started with writing an embedded probe. Execute `../gradlew` in that directory to compile and run it and study the gradle build file `build.gradle` to understand the execution environment. For more features, including control objects, go to the example in `api/samples/advanced-embedded-probe`.

Payload probes

Similar to injected probes, you still need a probe class for configuration purposes. The probe class must extend `com.jprofiler.api.probe.embedded.PayloadProbe` or `com.jprofiler.api.probe.embedded.SplitProbe`, depending on whether your probe collects payloads or splits the call tree. With the injected probe API, you use different annotations on the handler methods for payload collection and splitting. The embedded probe API, on the other hand, has no handler methods and needs to shift this configuration to the probe class itself.

```
public class FooPayloadProbe extends PayloadProbe {
    @Override
    public String getName() {
        return "Foo queries";
    }

    @Override
    public String getDescription() {
        return "Records foo queries";
    }
}
```

Whereas injected probes use annotations for configuration, you configure embedded probes by overriding methods from the base class of the probe. For a payload probe, the only abstract method is `getName()`, all other methods have a default implementation that you can override if required. For example, if you want to disable the events view to reduce overhead, you can override `isEvents()` to return `false`.

For collecting payloads and measuring their associated timing you use a pair of `Payload.enter()` and `Payload.exit()` calls

```
public void measuredCall(String query) {
    Payload.enter(FooPayloadProbe.class);
    try {
        performWork();
    } finally {
        Payload.exit(query);
    }
}
```

The `Payload.enter()` call receives the probe class as an argument, so the profiling agent knows which probe is the target of the call, the `Payload.exit()` call is automatically associated with the same probe and receives the payload string as an argument. If you miss an exit call, the call tree would be broken, so this should always be done in a finally clause of a try block.

If the measured code block does not produce any value, you can call the `Payload.execute` method instead which takes the payload string and a `Runnable`. With Java 8+, lambdas or method references make this method invocation very concise.

```
public void measuredCall(String query) {
  Payload.execute(FooPayloadProbe.class, query, this::performWork);
}
```

The payload string must be known in advance in that case. There is also a version of `execute` that takes a `Callable`.

```
public QueryResult measuredCall(String query) throws Exception {
    return Payload.execute(PayloadProbe.class, query, () -> query.execute());
}
```

The problem with the signatures that take a `Callable` is that `Callable.call()` throws a checked `Exception` and so you have to either catch it or declare it on the containing method.

Control objects

Payload probes can open and close control objects by calling the appropriate methods in the `Payload` class. They are associated with probe events by passing them to a version of the `Payload.enter()` or `Payload.execute()` methods that take a control object and a custom event type.

```
public void measuredCall(String query, Connection connection) {
    Payload.enter(FooPayloadProbe.class, connection, MyEventTypes.QUERY);
    try {
        performWork();
    } finally {
        Payload.exit(query);
    }
}
```

The control object view must be explicitly enabled in the probe configuration and custom event types must be registered in the probe class as well.

```
public class FooPayloadProbe extends PayloadProbe {
    @Override
    public String getName() {
        return "Foo queries";
    }

    @Override
    public String getDescription() {
        return "Records foo queries";
    }

    @Override
    public boolean isControlObjects() {
        return true;
    }

    @Override
    public Class<? extends Enum> getCustomTypes() {
        return Connection.class;
    }
}
```

If you do not explicitly open and close your control objects, the probe class must override `getControlObjectName` in order to resolve display names for all control objects.

Split probes

The split probe base class has no abstract methods, because it can be used to just split the call tree without adding a probe view. In that case, the minimal probe definition is just

```
public class FooSplitProbe extends SplitProbe {}
```

One important configuration for split probes is whether they should be reentrant. By default, only the top-level call is split. If you would like to split recursive calls as well, override `isReentrant()` to return `true`. Split probes can also create a probe view and publish the split strings as payloads if you override `isPayloads()` to return `true` in the probe class.

To perform a split, make a pair of calls to `Split.enter()` and `Split.exit()`.

```
public void splitMethod(String parameter) {
    Split.enter(FooSplitProbe.class, parameter);
    try {
        performWork(parameter);
    } finally {
        Split.exit();
    }
}
```

Contrary to to payload collection, the split string has to be passed to the `Split.enter()` method together with the probe class. Again, it is important that `Split.exit()` is called reliably, so it should be in a finally clause of a try block. `Split` also offers `execute()` methods with `Runnable` and `Callable` arguments that perform the split with a single call.

Telemetries

It is particularly convenient to publish telemetries for embedded probes, because being in the same classpath you can directly access all static methods in your application. Just like for injected probes, annotate static public methods in your probe configuration class with `@Telemetry` and

return a numeric value. See the chapter on probe concepts [p. 132] for more information. The `@Telemetry` annotations of the embedded and the injected probe APIs are equivalent, they are just in different packages.

Another parallel functionality between embedded and injected probe API is the ability to modify the thread state with the `ThreadState` class. Again, the class is present in both APIs with different packages.

Deployment

There are no special steps necessary to enable embedded probes when profiling with the JProfiler UI. However, the probe will only be registered when the first call into `Payload` or `Split` is made. Only at that point will the associated probe view be created in JProfiler. If you prefer the probe view to be visible from the beginning, as is the case for built-in and injected probes, you can call

```
PayloadProbe.register(FooPayloadProbe.class);
```

for payload probes and

```
SplitProbe.register(FooSplitProbe.class);
```

for split probes.

You may be considering whether to call the methods of `Payload` and `Split` conditionally, maybe controlled by a command line switch in order to minimize overhead. However, this is generally not necessary, because the method bodies are empty. Without the profiling agent attached, no overhead is incurred apart from the construction of the payload string. Considering that probe events should not be generated on a microscopic scale, they will be created relatively rarely, so that building the payload string should be a comparatively insignificant effort.

Another concern for containers may be that you do not want to expose external dependencies on the class path. A user of your container could also use the embedded probe API which would lead to a conflict. In that case you can shade the embedded probe API into your own package. JProfiler will still recognize the shaded package and instrument the API classes correctly. If build-time shading is not practical, you can extract the source archive and make the classes part of your project.

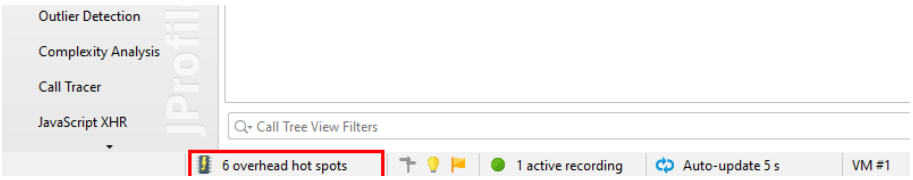
B Call Tree Features In Detail

B.1 Auto-Tuning And Ignored Methods

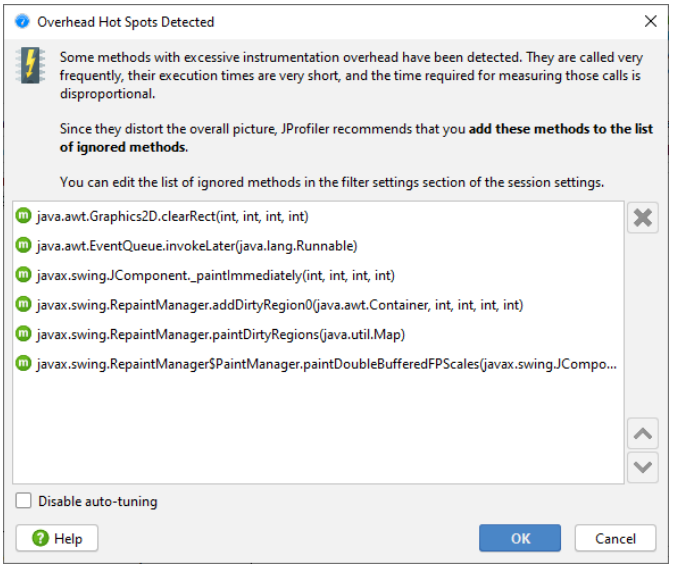
If the method call recording type is set to instrumentation, all methods of profiled classes are instrumented. This creates significant overhead for methods that have very short execution times. If such methods are called very frequently, the measured time of those method will be far to high. Also, due to the instrumentation, the hot spot compiler might be prevented from optimizing them. In extreme cases, such methods become the dominant hot spots although this is not true for an uninstrumented run. An example is the method of an XML parser that reads the next character. Such a method returns very quickly, but may be invoked millions of times in a short time span.

This problem is not present when the method call recording type is set to sampling. However, sampling does not provide invocations counts, only shows longer method calls and several views do not have their full functionality when sampling is used.

To alleviate the problem with instrumentation, JProfiler has a mechanism called auto-tuning. From time to time, the profiling agent checks for methods with high instrumentation overhead and transmits them to the JProfiler GUI. In the status bar, an entry alerting to the presence of overhead hot spots will be shown.

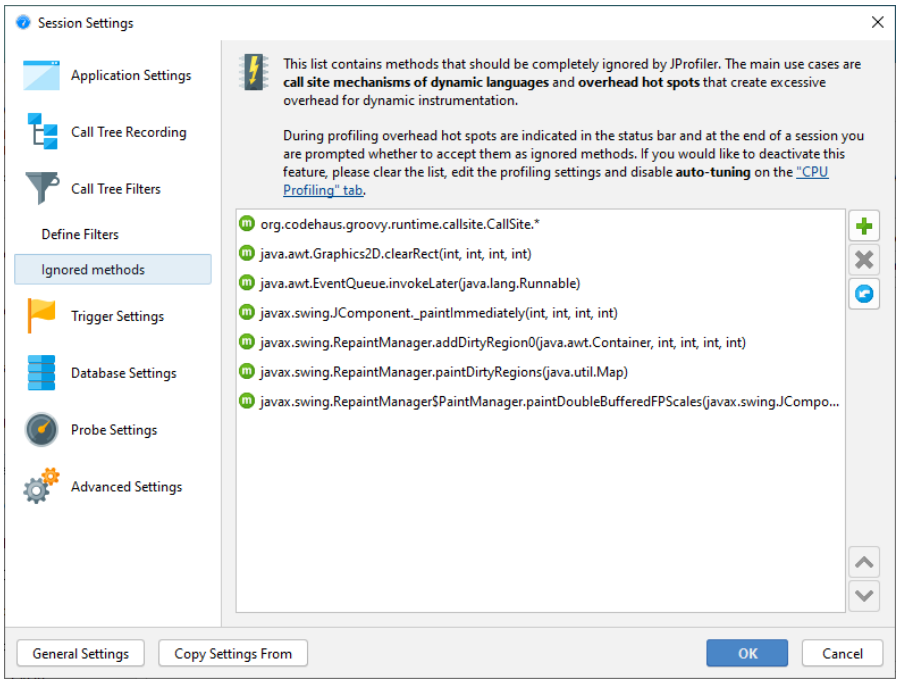


You can click on that status bar entry to review the detected overhead hot spots and choose to accept them into the list of ignored methods. These ignored methods will then not be instrumented. When a session is terminated, the same dialog is shown.



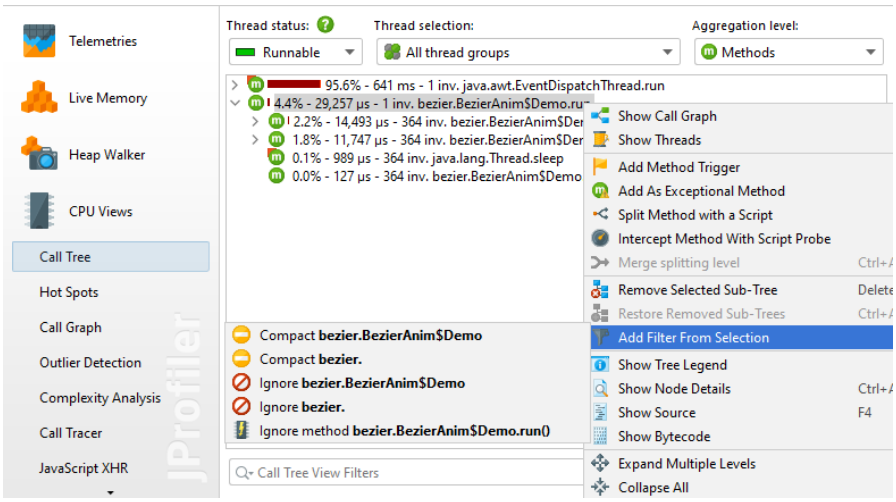
After you apply the new profiling settings, all ignored methods will be missing in the call tree. Their execution time will be added to the self time of the calling method. If later on you find that

some ignored methods are indispensable in the profiling views, you can remove them in the *Ignored Methods* tab in the session settings.



The default configuration for ignored methods includes the call site classes for Groovy that are used for the dynamic method dispatch, but make it difficult to follow the actual call chain.

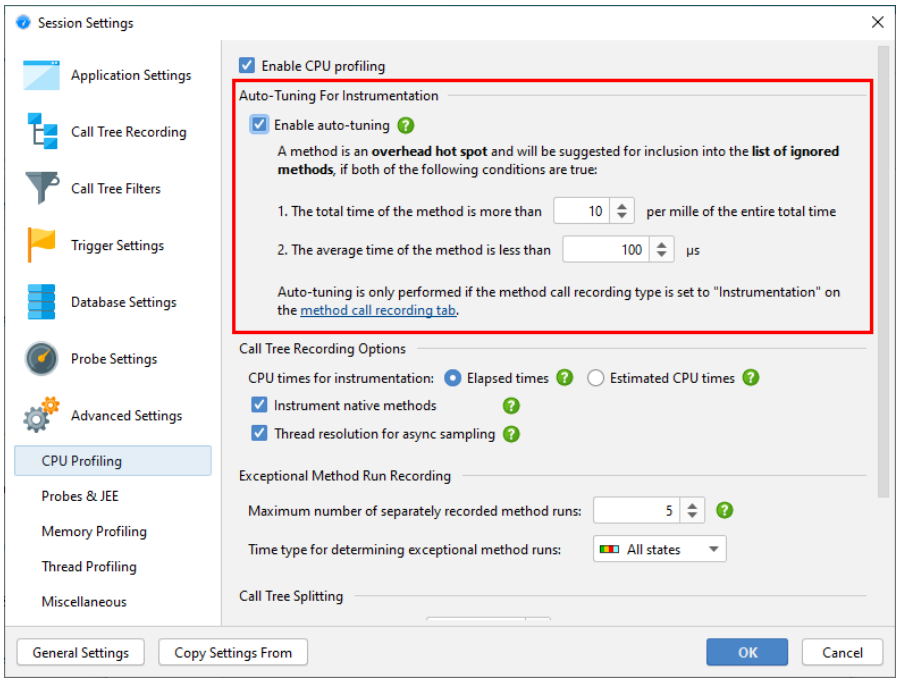
If you want to manually add ignored methods, you can do so in the session settings, but a much easier way is to select a method in the call tree and invoke the *Ignore Method* action from the context menu.



In the filter settings, you can also ignore entire classes or packages by setting the type of the filter entry to "Ignored". The *Add Filter From Selection* menu contains actions that depend on the selected node and suggest ignoring the class or packages up to the top-level package. Depending on whether the selected node is compact-profiled or profiled, you also see actions for changing the filter to the opposite type.

In case you don't want to see any messages about auto-tuning, you can disable it in the profiling settings. Also, you can configure the criteria for determining an overhead hot spot. A method is considered an overhead hot spot if both of the following conditions are met:

- The total time of all its invocations exceeds a threshold in per mille of the entire total time in the thread
- Its average time is lower than an absolute threshold in microseconds



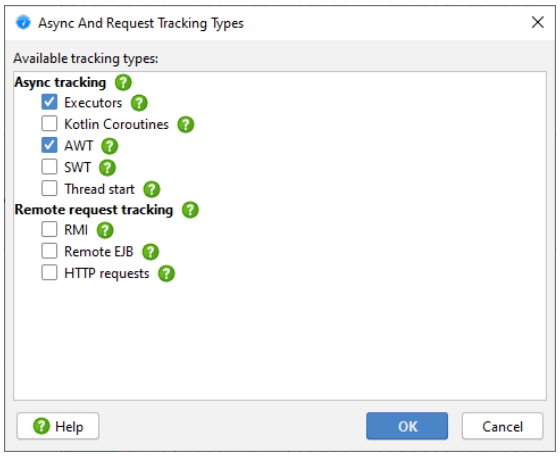
B.2 Async And Remote Request Tracking

Asynchronous execution of tasks is a common practice, both in plain Java code and even more so with reactive frameworks. Code that is adjacent in your source file is now executed on two or more different threads. For debugging and profiling, these thread changes present two problems: On the one hand, it is not clear how expensive an invoked operation is. On the other hand, an expensive operation cannot be traced back to the code that caused its execution.

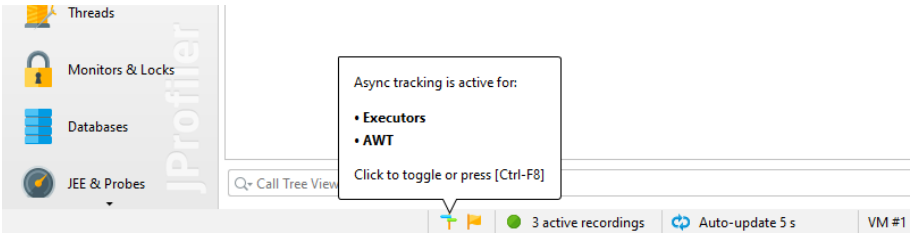
JProfiler provides different solutions to this problem depending on whether the call stays in the same JVM or not. If the async execution takes place in the same JVM that invokes it, the "Inline Async Executions" call tree analysis [p. 169] calculates a single call tree that contains both call sites as well as execution sites. If a request to a remote JVM is made, the call tree [p. 48] contains hyperlinks to call sites and execution sites, so you can seamlessly navigate both ways between different JProfiler top-level windows that show profiling sessions for the involved JVMs.

Enabling Async And Remote Request Tracking

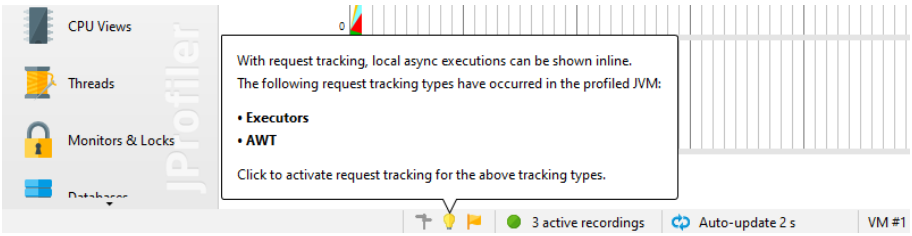
Async mechanisms can be implemented in various ways and the semantics of starting tasks on a separate thread or in a different JVM cannot be detected in a generic way. JProfiler explicitly supports several common asynchronous and remote request technologies. You can enable or disable them in the request tracking settings. By default, request tracking is not enabled. It is also possible to configure request tracking in the session startup dialog that is shown directly before a session is started.



In JProfiler's main window, the status bar indicates if some async and remote request tracking types are enabled and gives you a shortcut to the configuration dialog.

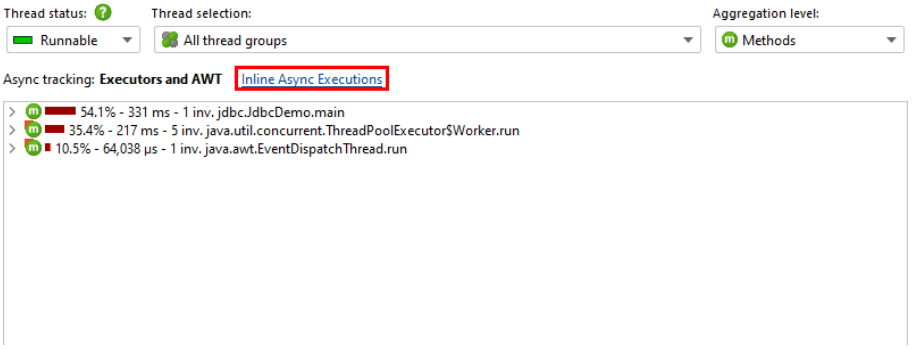


JProfiler detects if an async request tracking type that is not activated is used in the profiled JVM and shows you a 💡 notification icon next to the async and remote request tracking icon in the status bar. By clicking on the notification icon, you can activate the detected tracking types. Async and remote request tracking can produce substantial overhead and should only be activated if necessary.



Async Tracking

If at least one async tracking type is activated, the call tree and hot spot views for CPU, allocation and probe recording show information about all activated tracking types together with a button that calculates the "Inline Async Executions" call tree analysis. In the result views of that analysis, the call tree of all async executions is connected with the call sites by way of an "async execution" node. By default, the async execution measurements are not added to the ancestor nodes in the call tree. Because it is sometimes useful to see aggregated values, a check box at the top of the analysis allows you to do that where appropriate.



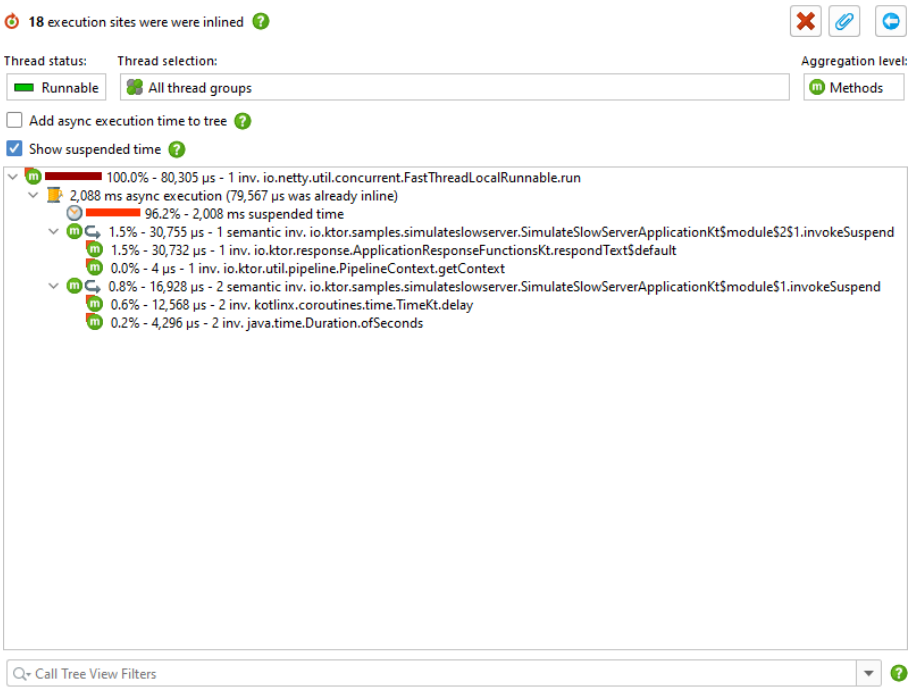
The simplest way to offload a task on another thread is to start a new thread. With JProfiler, you can follow a thread from its creation to the execution site by activating the "Thread start" request tracking type. However, threads are heavy-weight objects and are usually reused for repeated invocations, so this request tracking type is more useful for debugging purposes.

The most important and generic way to start tasks on other threads uses executors in the `java.util.concurrent` package. Executors are also the basis for many higher-level third party libraries that deal with asynchronous execution. By supporting executors, JProfiler supports a whole class of libraries that deal with multi-threaded and parallel programming.

Apart from the generic cases above, JProfiler also supports two GUI toolkits for the JVM: AWT and SWT. Both toolkits are single-threaded, which means that there is one special event dispatch thread that can manipulate GUI widgets and perform drawing operations. In order not to block the GUI, long-running tasks have to be performed on background threads. However, background threads often need to update the GUI to indicate progress or completion. This is done with special methods that schedule a `Runnable` to be executed on the event dispatch thread.

In GUI programming, you often have to follow multiple thread changes in order to connect cause and effect: The user initiates an action on the event dispatch thread, which in turn starts a background operation via an executor. After completion, that executor pushes an operation to the event dispatch thread. If that last operation creates a performance problem, it's two thread changes away from the originating event.

Finally, JProfiler supports Kotlin coroutines[1], Kotlin's multi-threading solution that is implemented for all Kotlin backends. The async execution itself is the point where a coroutine is launched. The dispatching mechanism of Kotlin coroutines is flexible and can actually involve starting on the current thread, in which case the "async execution" node has an inline part that is then reported separately in the text of the node.



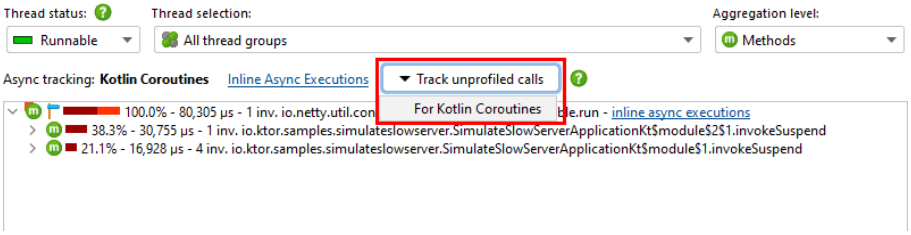
Suspending methods can interrupt the execution which is then possibly resumed on different threads. Methods where suspension was detected have an additional "suspend" icon with a tooltip that shows the number of actual calls versus the semantic invocations of the method. Kotlin coroutines can be suspended deliberately, but because they are not bound to threads, the waiting time will not appear anywhere in the call tree. To see the total time taken until a coroutine execution is finished, a "suspended" time node is added below the "async execution" node that captures the entire suspension time for the coroutine. Depending on whether you are interested in the CPU time or in the wall clock time of async executions, you can add or remove those nodes on the fly with the "Show suspended times" check box at the top of the analysis.

Tracking unprofiled call site

By default, both executor and Kotlin coroutine tracking only track async executions where the call site is in a profiled class. This is because frameworks and libraries can use these async mechanisms in a way that is not directly related to the execution of your own code and the added call and execution sites would just add overhead and distraction. However, there are use cases for tracking unprofiled call sites. For example, a framework can start a Kotlin coroutine on which your own code is then executed.

If such call sites in unprofiled classes are detected, the tracking information in the call tree and hot spot views shows a corresponding notification message. In live sessions, you can switch on tracking for unprofiled call sites separately for executor and Kotlin coroutine tracking directly from those views. These options can be changed at any time on the "CPU profiling" step of the session settings dialog.

[1] https://kotlinlang.org/docs/reference/coroutines.html



It is important to understand that Kotlin coroutines can only be tracked when their launch happened while CPU recording was active. If you start CPU recording later on, the async executions from Kotlin coroutines cannot be inlined. JProfiler will notify you just like for the detection of call sites in unprofiled classes. If you need to profile long-lived coroutines that are started at the beginning of the application, then using the attach mode is not an option. In that case, launch the JVM with the -agentpath VM parameter [p. 10] and start CPU recording at startup.

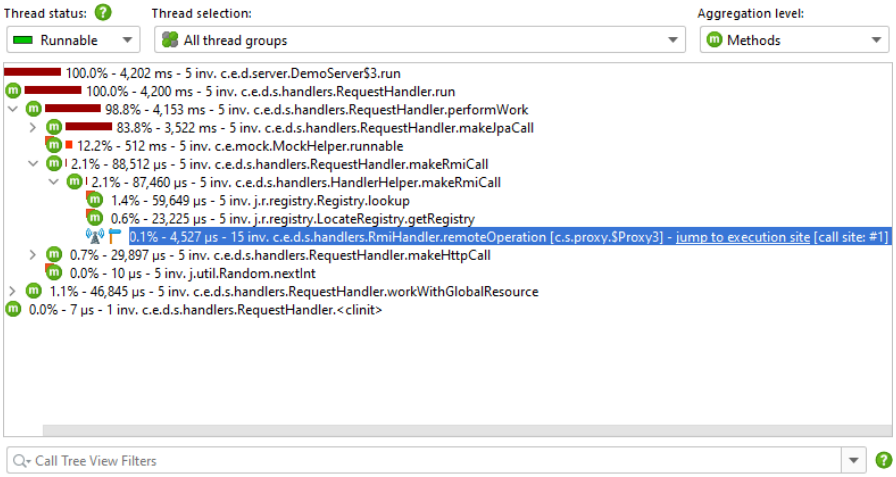
Remote Request Tracking

For selected communication protocols, JProfiler is able to insert meta-data and track requests across JVM boundaries. The supported technologies are:

- HTTP: HttpURLConnection, java.net.http.HttpClient, Apache Http Client 4.x, Apache Async Http Client 4.x, OkHttp 3.9+ on the client side, any Servlet-API implementation or Jetty without Servlets on the server side
- Additional support for async JAX-RS calls for Jersey Async Client 2.x, RestEasy Async Client 3.x, Cxf Async Client 3.1.1+
- Web services: JAX-WS-RI, Apache Axis2 and Apache CXF
- RMI
- Remote EJB calls: JBoss 7.1+ and Weblogic 11+

In order to be able to follow the request in JProfiler you have to profile both VMs and open them at the same time in separate JProfiler top-level windows. This works with both live sessions as well as with snapshots. If the target JVM is not currently open, or if CPU recording was not active at the time of the remote call, clicking on a call site hyperlink will show an error message.

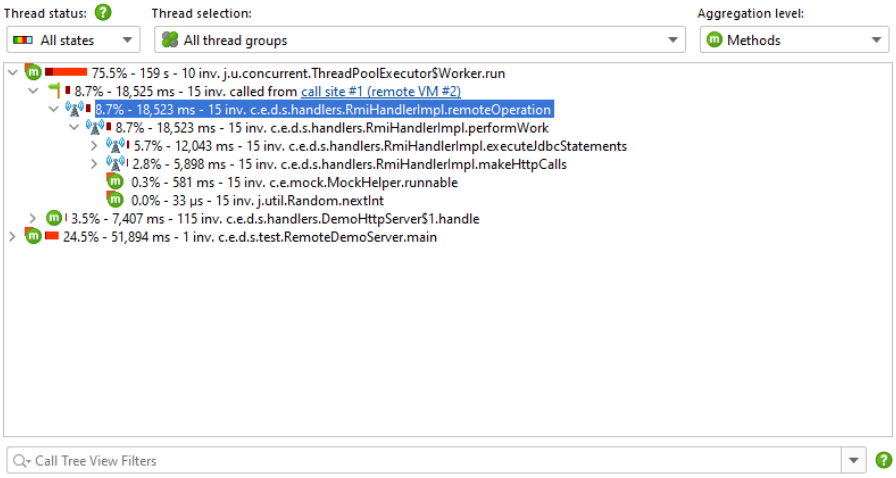
When tracking remote requests, JProfiler makes call sites and execution sites explicit in the call trees of the involved JVMs. A call site in JProfiler is the last profiled method call before a recorded remote request is performed. It starts a task at an execution site that is located in a different VM. JProfiler allows you to jump between call sites and execution sites by using hyperlinks that are shown in the call tree view.



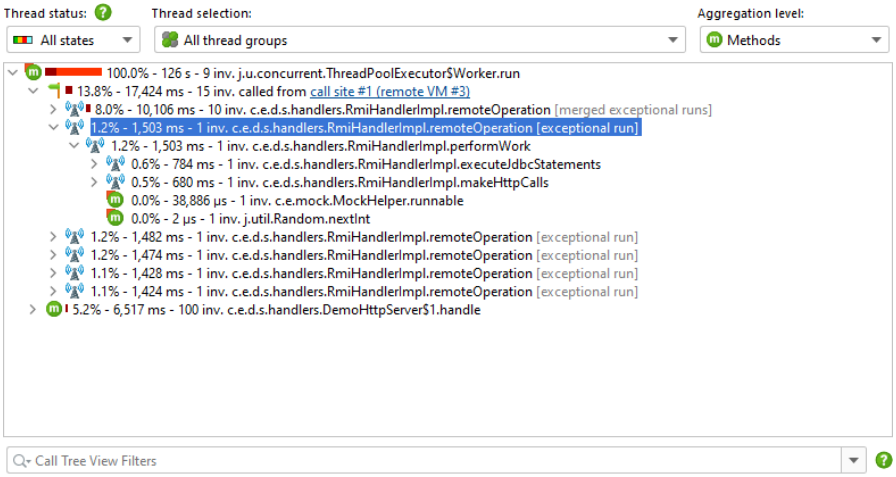
Call sites have the same identity with respect to remote request tracking for all threads. This means that when you jump from call sites to execution sites and vice versa, there is no thread-resolution and the jump always activates the "All thread groups" as well as the "All thread states" thread status selection, so that the target is guaranteed to be part of the displayed tree.

Call sites and execution sites are in a 1:n relationship. A call site can start remote tasks on several execution sites, especially if they are in different remote VMs. In the same VM, multiple execution sites for a single call site are less common, because they would have to occur at different call stacks. If a call site calls more than one execution site, you can choose one of them in a dialog.

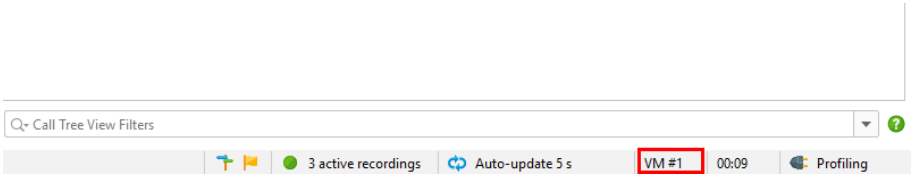
An execution site is a synthetic node in the call tree that contains all executions that were started by one particular call site. The hyperlink in the execution site node takes you back to that call site.



If the same call site invokes the same execution site repeatedly, the execution site will show the merged call tree of all its invocations. If that is not desired, you can use the exceptional methods [p. 174] feature to split the call tree further, as shown in the screen shot below.



Unlike execution sites which are only referenced from a single call site, call sites themselves can link to several execution sites. With the numeric ID of a call site, you can recognize the same call site if you see it referenced from different execution sites. In addition, a call site displays the ID of the remote VM. The ID of the profiled VM can be seen in the status bar. It is not the unique ID that JProfiler manages internally, but a display ID that starts at one and is incremented for each new profiled VM that is opened in JProfiler.



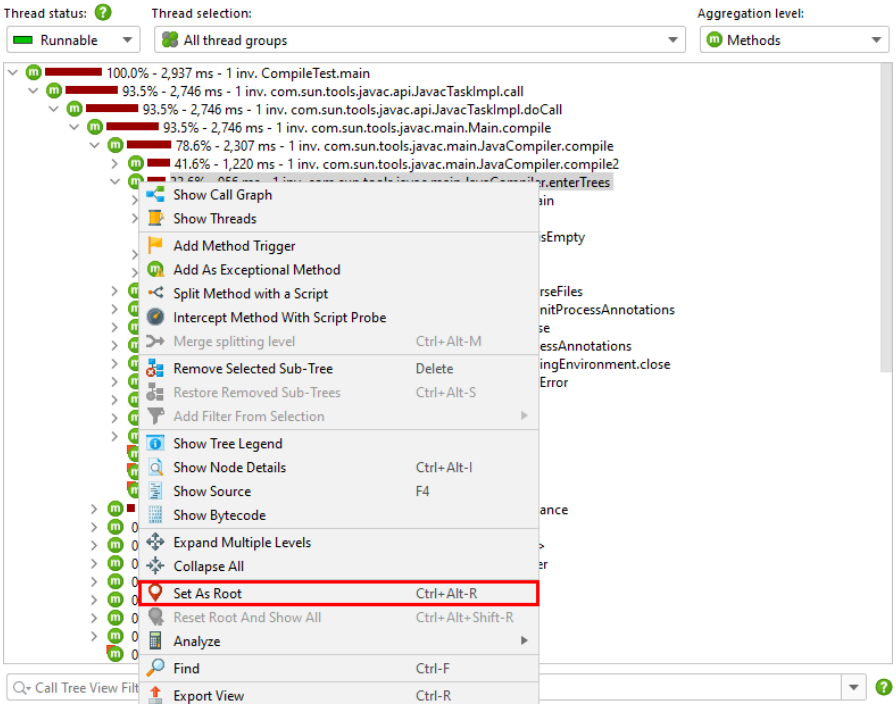
B.3 Viewing Parts Of The Call Tree

Call trees often contain too much information. When you want to reduce the displayed detail, there are several possibilities: you can restrict the displayed data to one particular sub-tree, remove all unwanted data, or use a more coarse-grained filter for displaying method calls. All of these strategies are supported by JProfiler.

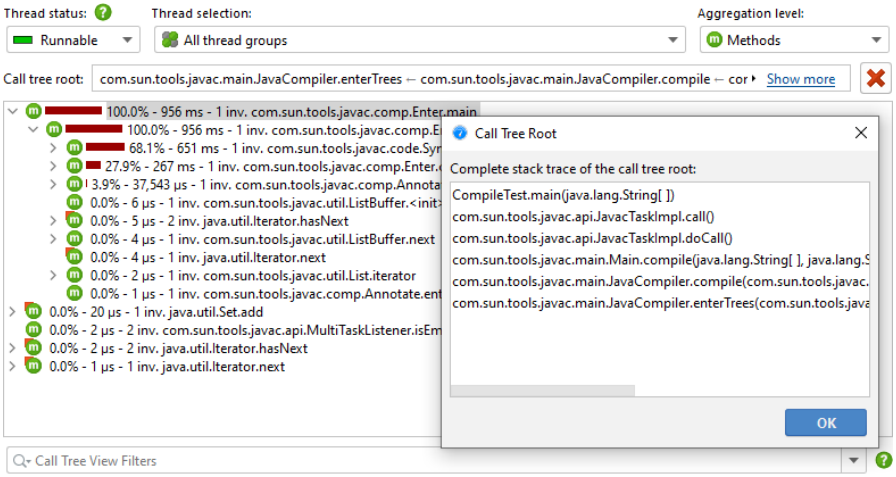
Setting call tree roots

If you profile a use case that consists of multiple tasks that run sequentially, each sub-tree can be analyzed separately. Once you have found the entry point to such a sub-task, the surrounding call tree is only a distraction and the timing percentages in the sub-tree inconveniently refer to the root of the entire call tree.

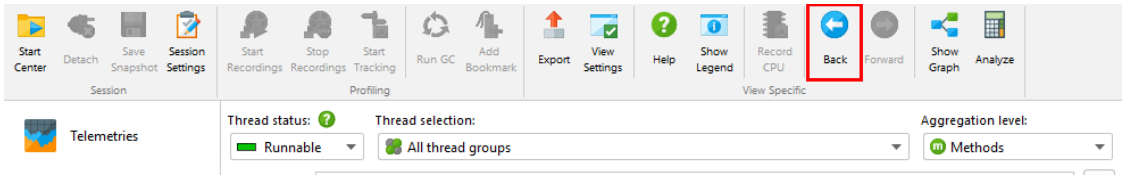
To focus on a particular sub-tree, JProfiler offers the *Set As Root* context action in the call tree and the allocation call tree views.



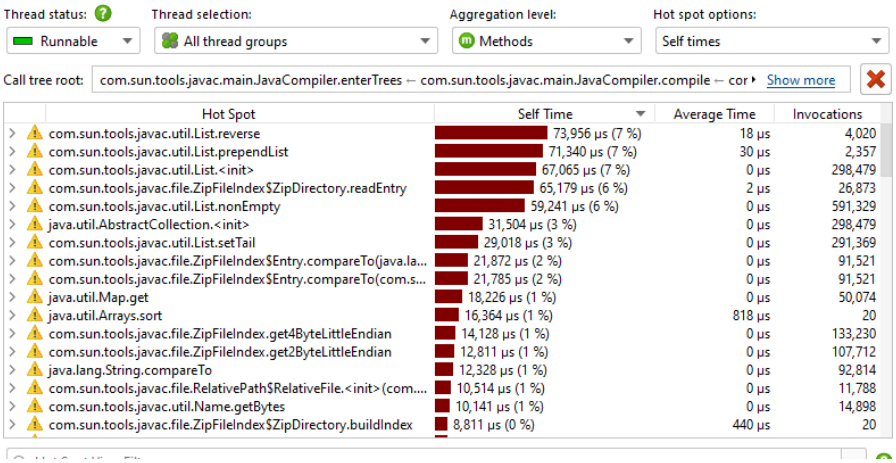
After setting a call tree root, information about the selected root is shown at the top of the view. A single scrollable label shows the last few stack elements leading up to the root and a detail dialog with the entire stack of the call tree root can be displayed by clicking on the *Show More* button.



When you use the set root action recursively, the call stack prefixes will simply be concatenated. To go back to the previous call tree, you can either use the *Back* button of the call tree history to undo one root change at a time, or the *Reset Root And Show All* action in the context menu to go back to the original tree in a single step.



What is most important about changing the call tree root, is that the hot spots view will show data that is calculated for the selected root only, and not for the entire tree. At the top of the hot spots view, you will see the current call tree root just like in the call tree view to remind of you the context of the displayed data.

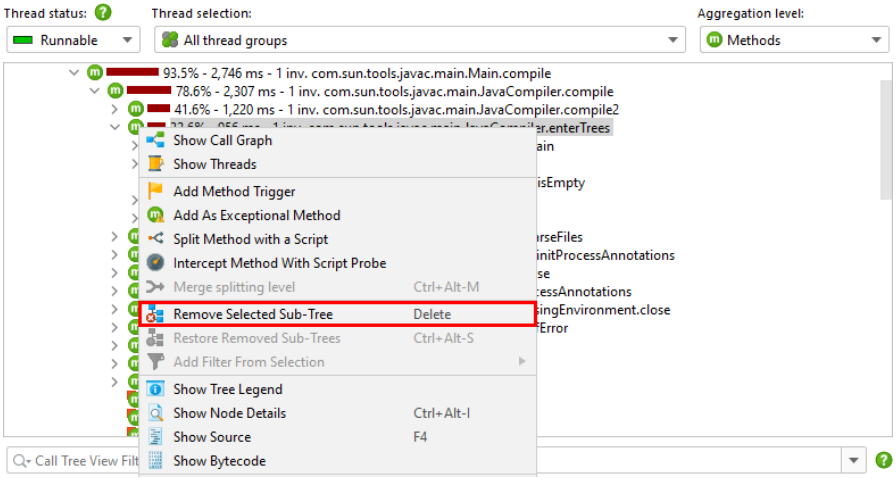


Removing parts of the call tree

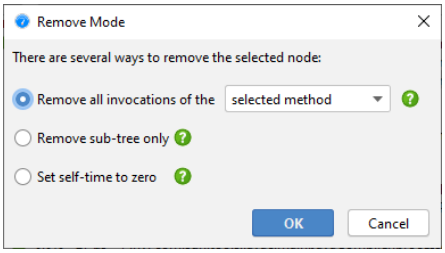
Sometimes it's helpful to see how the call tree would look like if a certain method was not present. For example, this can be the case when you have to fix several performance problems in one go, because you are working with a snapshot from a production system that cannot be iterated quickly like in your development environment. After solving the main performance problem, you

then want to analyze the second one, but that can only be seen clearly if the first one is eliminated from the tree.

Nodes in the call tree can be removed together with their sub-trees by selecting them and hitting the `Delete` key or by choosing *Remove Selected Sub-Tree* from the context menu. Times in ancestor nodes will be corrected accordingly as if the hidden nodes did not exist.

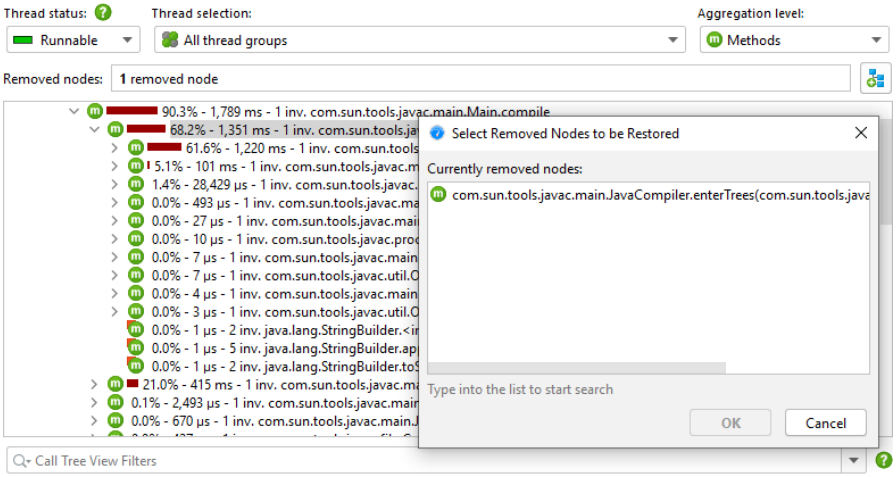


There are three removal modes. With the *Remove all invocations* mode, JProfiler searches for all invocations of the selected method in the entire call tree and removes them together with their entire sub-trees. The *Remove sub-tree only* option only removes the selected sub-tree. Finally, the *Set self-time to zero* leaves the selected node in the call tree bug sets its self-time to zero. This is useful for container nodes like `Thread.run` that may include a lot of time from unprofiled classes.



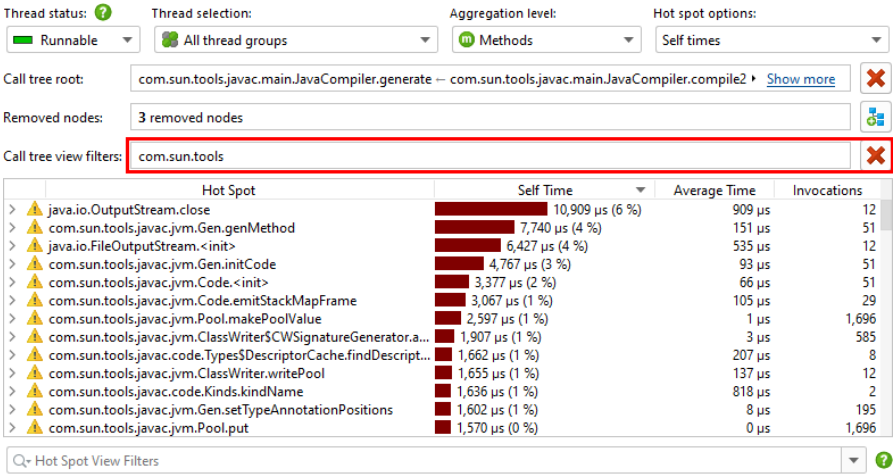
Just like for the *Set As Root* action, removed nodes influence the hot spots view. In this way, you can check what the hot spots would look like if those methods were optimized to the point of not being important contributions.

When you remove a node, the header area of both the call tree and the hot spots views will show a line with the count of the removed nodes and a *Restore Removed Sub-Trees* button. Clicking on that button will bring up a dialog where you can select removed elements that should be be shown again.



Call tree view filters

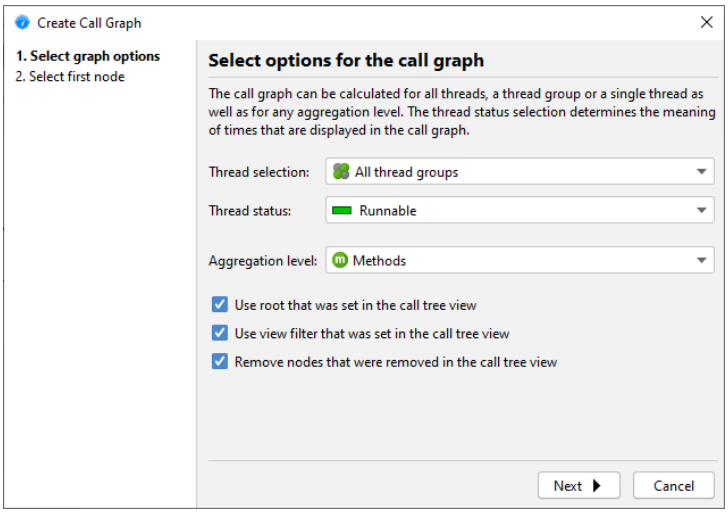
The third feature in the call that has an influence on the displayed data in the hot spots view is the view filter. When you change your call tree filters, it has a large effect on the calculated hot spots [p. 48]. To emphasize this interdependence with the call tree view, the hot spots view shows the call tree view filter in a line above the view together with a button to remove the additional filters.



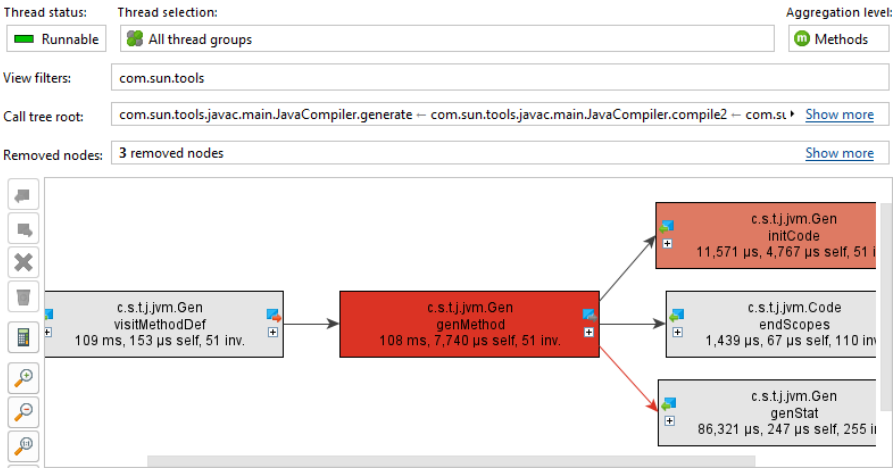
Setting a call tree root, removing parts of the call tree and view filters can be used together, with the limitation that view filters have to be set last. As soon as view filters are configured in the call tree, the *Set As Root* and *>Remove Selected Sub-Tree* actions do not work anymore.

Interaction with the call graph

Invoking the *Show Graph* action in either the call tree or the hot spots view will show a graph that is limited to the same call tree root, does not include the removed methods and uses the configured call tree view filters. At the top of the graph, the information about these changes is displayed in a similar form as in the call tree.



When creating a new graph in the graph view itself, check boxes in the wizard let you choose which of these call tree adjustment features should be taken into account for the calculation of the call graph. Each check box is only visible if the corresponding feature is currently used in the call tree view.



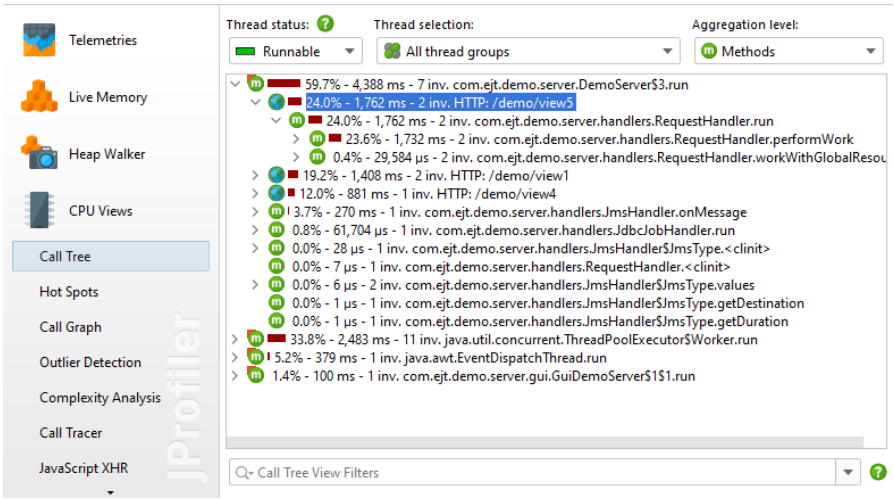
B.4 Splitting The Call Tree

Call trees are cumulated for repeated invocations of the same call stacks. This is necessary because of memory overhead and the need for consolidating data in order to make it understandable. However, sometimes you want to break the cumulation at selected points so you can view parts of the call tree separately.

JProfiler has a concept of splitting the call tree with special nodes that are inserted into the call stack and show semantic information that has been extracted from the method invocation above the inserted node. These splitting nodes allow you to see additional payload information directly inside the call tree and to analyze their contained sub-trees separately. Each splitting type can be merged and unmerged on the fly with the actions in the context menu and has a cap on the total number of splitting nodes so that the memory overhead is bounded.

Call tree splitting and probes

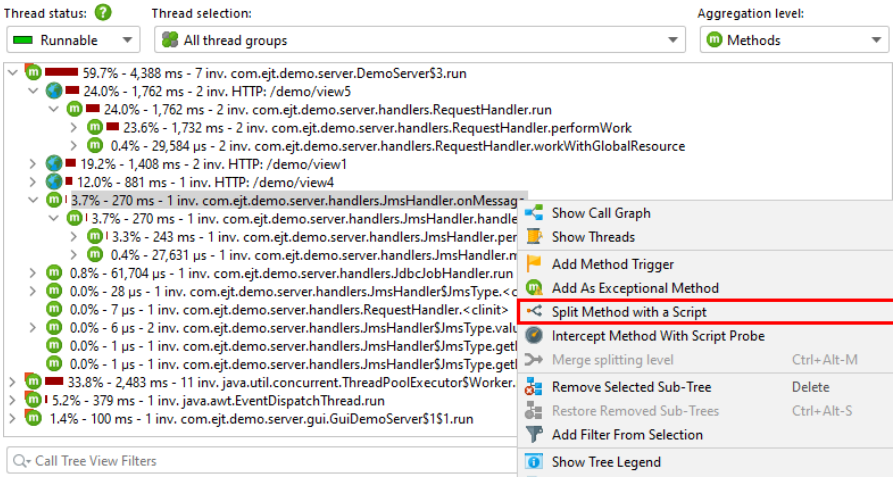
Probes [p. 97] can split the call tree according to the information that they collect at selected methods of interest. For example, the "HTTP server" probe splits the call tree for each different URL. The splitting in this case is highly configurable, so you can include only the desired parts of the URL, some other information from the servlet context or even produce multiple splitting levels.



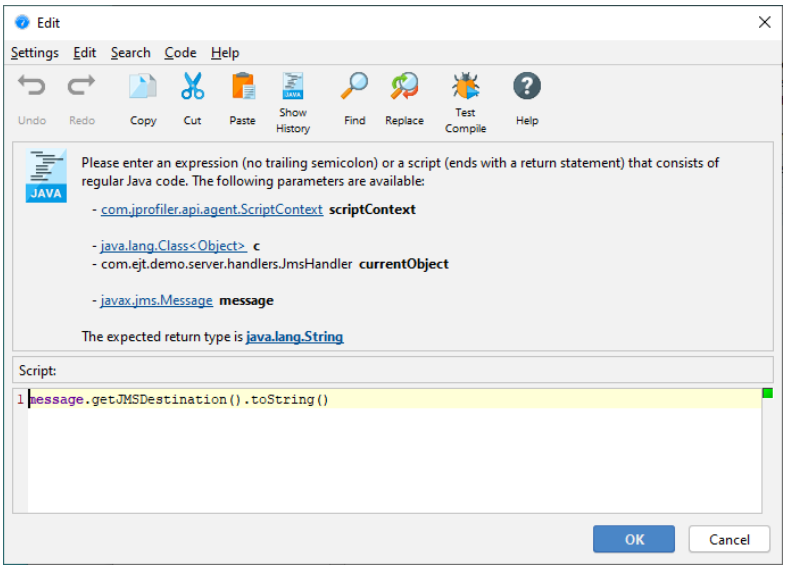
If you write your own probe, you can split the call tree in the same way, with both the embedded [p. 147] and the injected [p. 142] custom probe systems.

Splitting methods with scripts

The same splitting functionality that is available to probes can be used directly in the call tree, with the *Split Method With a Script* action. In the screen shot below, we want to split the call tree for a JMS message handler to see the handling of different types of messages separately.



Instead of writing a probe, you just enter a script that returns a string. The string is used for grouping the call tree at the selected method and is displayed in the splitting node. If you return `null`, the current method invocation is not split and added to the call tree as usual.

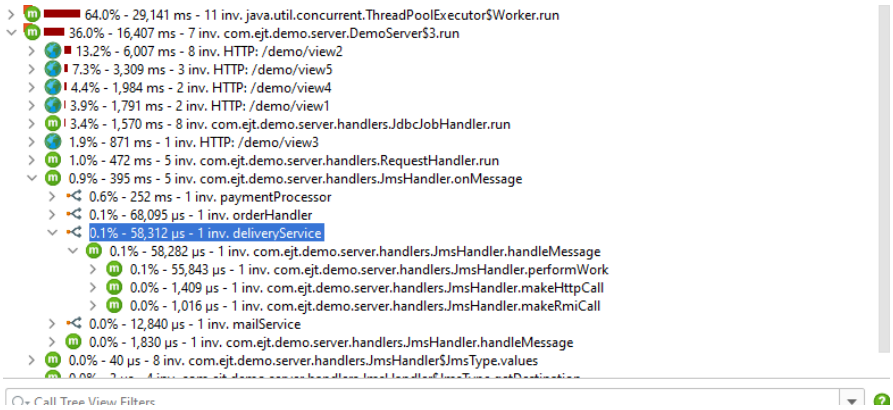


The script has access to a number of parameters. It is passed the class of the selected method, the instance for non-static methods, as well as all method parameters. In addition, you get a `ScriptContext` object that can be used to store data. If you need to recall some values from previous invocations of the same script, you can invoke the `getObject/putObject` and `getLong/putLong` methods in the context. For example, you may want to split only the first time a particular value for method parameter is seen. You could then use

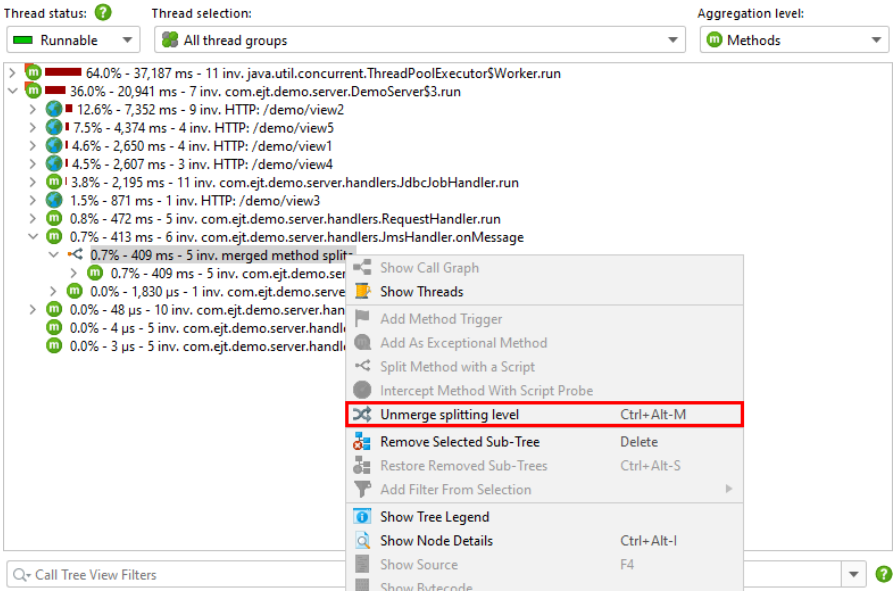
```
if (scriptContext.getObject(text) != null) {
    scriptContext.putObject(text);
    return text;
} else {
    return null;
}
```

as part of your splitting script.

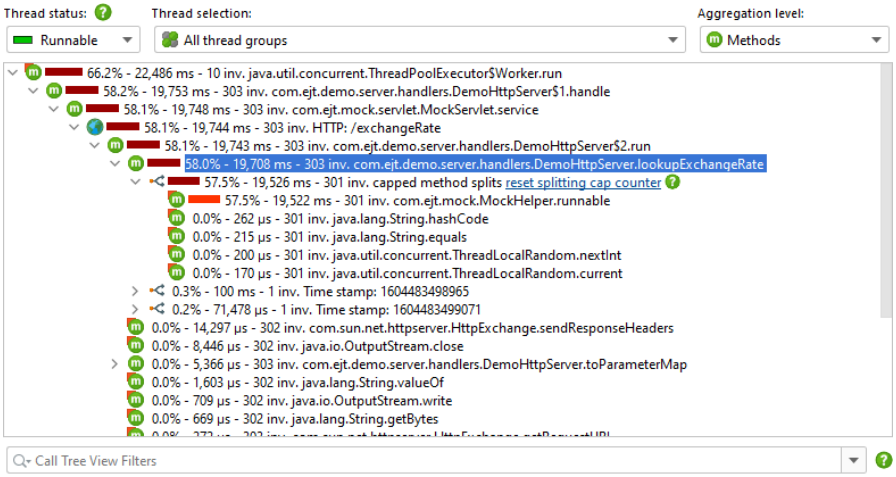
Splitting nodes are inserted below the selected method. For the example in the above screen shot, we now see the handling code for each JMS message destination separately.



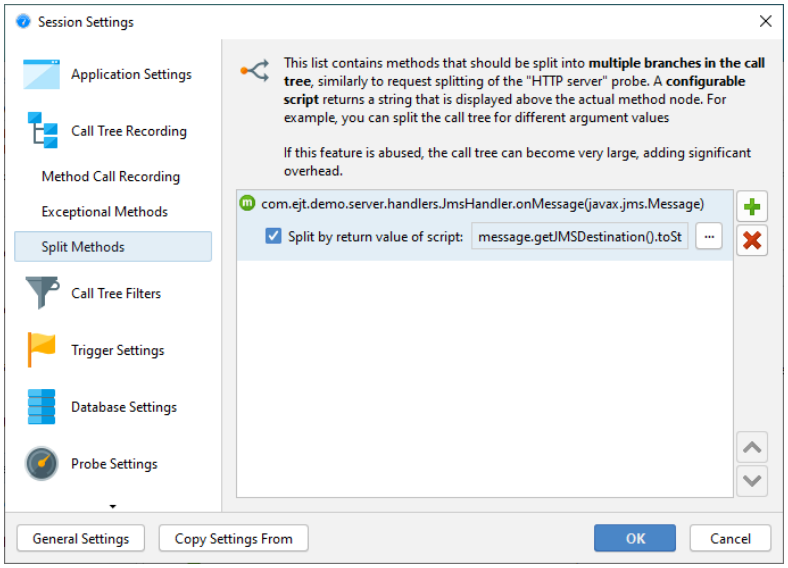
The splitting location is bound to a method, not to the selected call stack. If the same method is present somewhere else in the call tree, it will be split as well. If you use the *Merge splitting level* action, all splits will be merged into a single node. That node gives you a chance to unmerge the split again.



If you produce too many splits, a node labeled *capped method splits* will contain all further split invocations, cumulated into a single tree. With the hyperlink in the node, you can reset the cap counter and record some more splitting nodes. For a permanent increase in the maximum number of splits, you can increase the cap in the profiling settings.



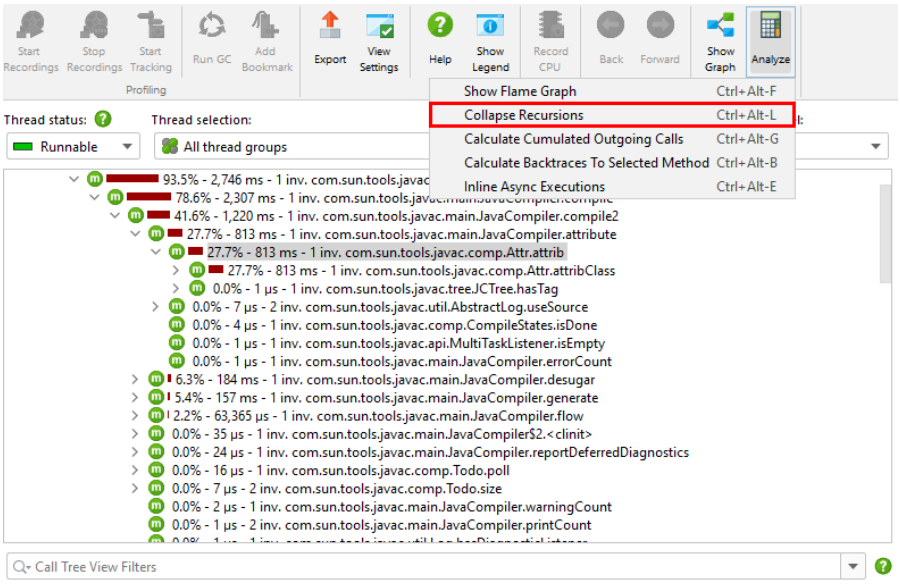
To edit split methods after you have created them, go to the session settings dialog. If you don't need a particular split method anymore, but want to keep it for future use, you can disable it with the check box in front of the script configuration. This is better than just merging it in the call tree, because the recording overhead may be significant.



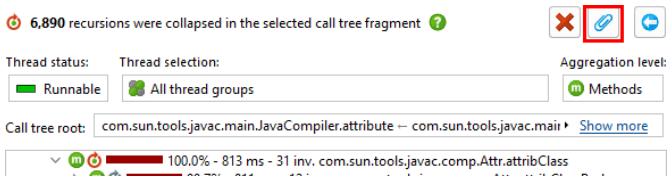
B.5 Call Tree Analyses

The call tree [p. 48] shows the actual call stacks that JProfiler has recorded. When analyzing the call tree, there are a couple of transformations that can be applied to the call tree to make it easier to interpret. These transformations can be time-consuming and change the output format in a way that is incompatible with the functionality in the call tree view, so new views with the results of the analyses are created.

To perform such an analysis, select a node in the call tree view and choose one of the call tree analysis actions from the tool bar or the context menu.



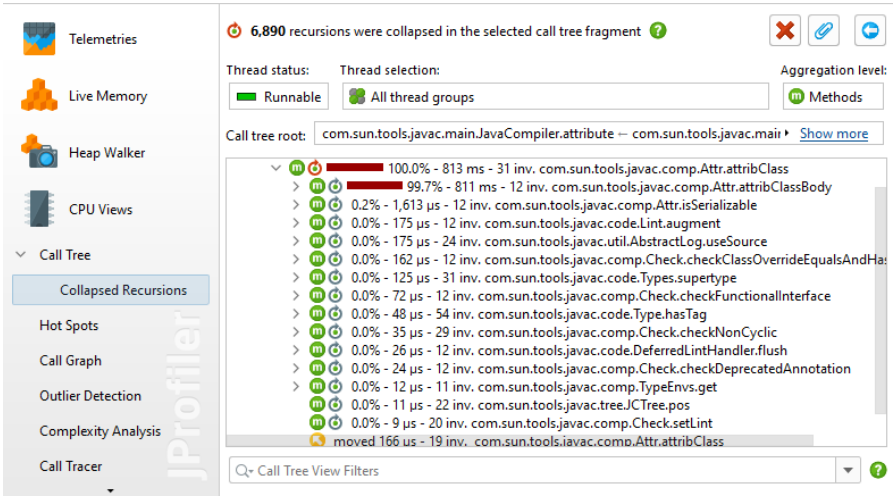
A nested view will be created below the call tree view. If you invoke the same analysis action again, the analysis will be replaced. To keep multiple analysis results at the same time, you can pin the result view. In that case, the next analysis of the same type will create a new view.



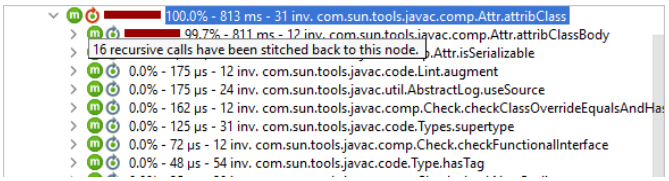
In live sessions, the result views are not updated together with the call tree and show data from the time when the analysis was made. To re-calculate the analysis for the current data, use the reload action. If the call tree itself has to be re-calculated, like in the allocation tree with disabled auto-updates, the reload action takes care of that as well.

Collapsing recursions

A programming style that makes use of recursions leads to call trees that are difficult to analyze. The "Collapse recursions" call tree analysis calculates a call tree where all recursions are folded. The parent node of the current selection in the call tree serves as the call tree root [p. 160] for the analysis. To analyze the entire call tree, select one of the top-level nodes.



A recursion is detected when the same method was already called higher up in the call stack. In that case, the sub-tree is removed from the call tree and stitched back to the first invocation of that method. That node in the call tree is then prefixed with an icon whose tool tip shows the number of recursions. Below that node, stacks from different depths are merged. The number of merged stacks is shown in the tool tip as well. The total number of collapsed recursions is shown in the header, above the information about call tree parameters that were set for the original call tree.

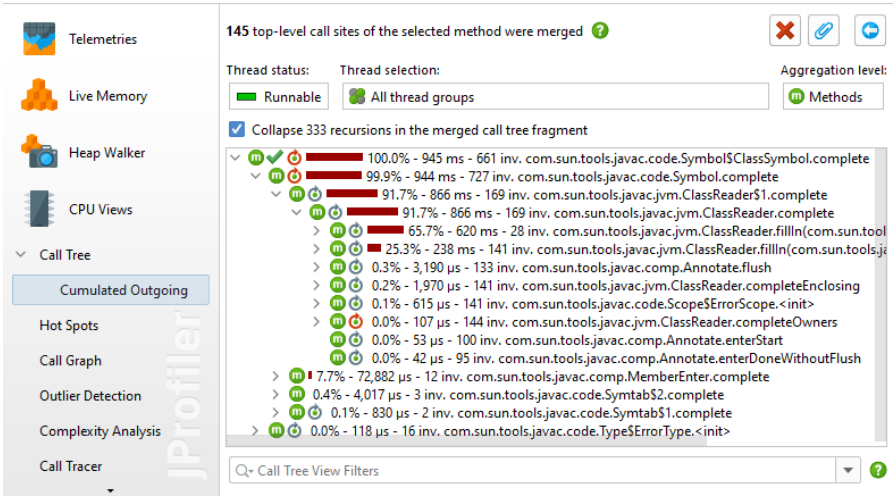


For a simple recursion, the number of merged stacks is the number of recursions plus one. So a node whose recursion tool tip shows "1 recursion" would contain a tree with nodes that show "2 merged stacks" in their recursion tool tip. In more complex cases, recursions are nested and produce overlapping merged call trees, so that the number of merged stacks varies from stack depth to stack depth.

At the point where a sub-tree is removed from the call tree to be merged higher up, a special "moved nodes" placeholder is inserted.

Analyzing cumulated outgoing calls

In the call tree, you can see the outgoing calls for a selected method, but only for one particular call stack where that method has been invoked. The same method of interest may have been invoked in different call stacks and it's often useful to analyze a cumulated call tree of all those invocations in order to get better statistics. The "Calculate cumulated outgoing calls" analysis shows a call tree that sums all outgoing calls of a selected method, regardless of how the method was invoked.

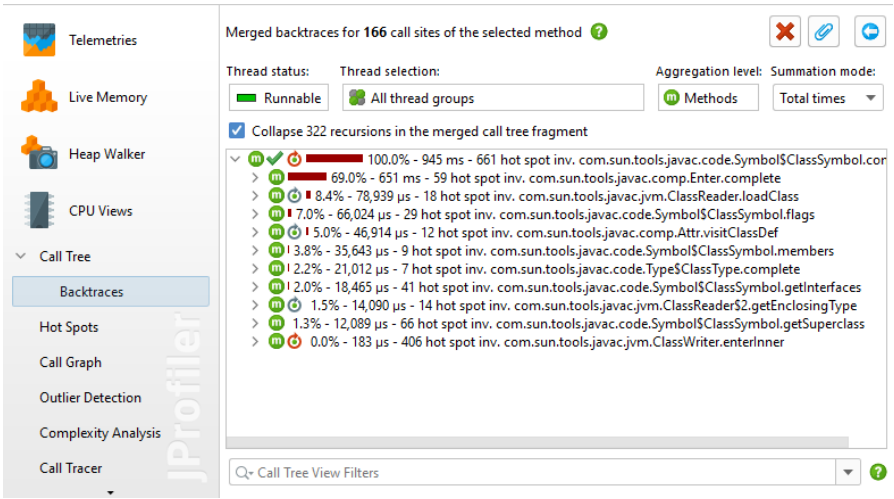


For the selected method, JProfiler collects all its top-level invocations without considering recursive calls and cumulates them in the result tree. The header shows how many such top-level call sites were summed in that process.

At the top of the view, there is a check box that allows you to collapse recursions in the result tree, similar to the "Collapse recursions" analysis. If recursions are collapsed, the top level node and the first level of outgoing calls show the same numbers as the method call graph.

Calculating backtraces

The "Calculate backtraces" analysis complements the "Calculate cumulated outgoing calls" analysis. Like the latter, it sums all top-level calls of the selected method without considering recursive calls. However, instead of showing outgoing calls, it shows the back traces that contribute to the invocations of the selected method. The call originates at the deepest node and progresses toward the selected method at the top.



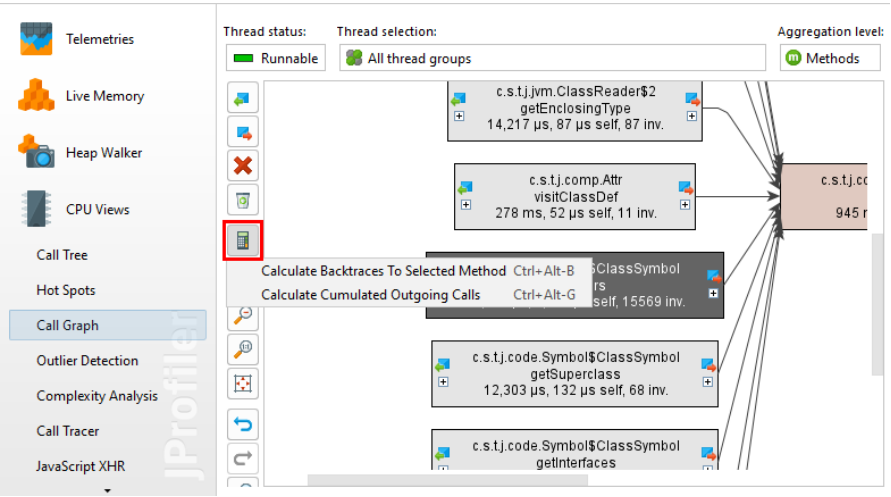
This analysis is similar to the hot spots view, only that by default it sums total times instead of self times for the selected method, and the hot spots view only shows methods whose self time is a significant fraction of the total time. At the top of the view there is a radio button group labeled *Summation mode* that can be set to *Self times*. With that selection, the summed values for the selected method match that of the default mode in the hot spots view.

In the back traces, the invocation counts and times on the back trace nodes are only related to the selected method. They show how much the invocations along that particular call stack have contributed to the values of the selected method. Similar to the "Calculate cumulated outgoing calls" analysis, you can collapse recursions and the first level in the backtraces is equivalent to the incoming calls in the method call graph.

Call tree analyses in the call graph

In the call graph, each method is unique while in the call tree methods can occur in multiple call stacks. For one selected method, the "Calculate cumulated outgoing calls" and the "Calculate backtraces" analyses are a bridge between the viewpoints of the call tree and the call graph. They put the selected method in the center and show the outgoing and incoming calls as trees. With the *Show Call Graph* action, you can switch to the full graph at any time.

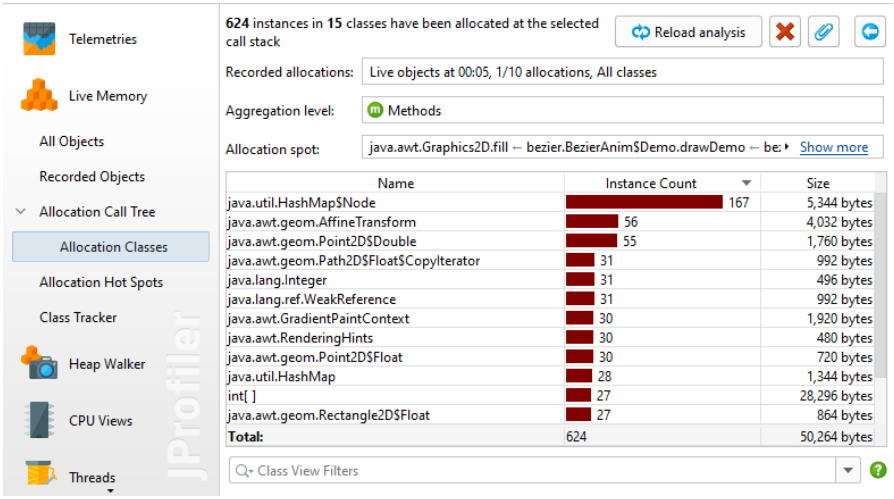
Sometimes, you want to switch the perspective in the opposite direction and change from graph to a tree view. When you are working in the call graph, you can show the cumulated outgoing calls and the backtraces as trees for any selected node in the graph with the same call tree analyses as in the call graph.



In the IntelliJ IDEA integration [p. 124], the call graph that is shown in the gutter of the editors contains actions to show these trees directly.

Showing classes for allocations

A little bit different from the previous call tree analyses is the "Show classes" analysis in the allocation call tree and the allocation hot spots views. It does not transform the call tree to another tree, but shows a table with all allocated classes. The result view is similar to the recorded objects view [p. 66], but restricted for a particular allocation spot.



In the analysis result views that show call trees, both the "Calculate cumulated outgoing calls" and the "Calculate backtraces to selected method" analyses are available. Invoking them creates new top-level analyses with independent parameters. Any call tree removals from the previous analysis result view are not reflected in the new top-level analysis.

The *Show Classes* action, on the other hand, does not create a new top-level analysis when used from a call tree analysis result view. Instead, it creates a nested analysis that is two levels below the original view.

C Advanced CPU Analysis Views

C.1 Outlier Detection And Exceptional Method Recording

In some situations, it's not the average invocation time of a method that is a problem, but rather that a method misbehaves every once in a while. In the call tree, all method invocations are cumulated, so a frequently called method that takes 100 times as long as expected once every 10000 invocations will not leave a distinct mark in the total times.

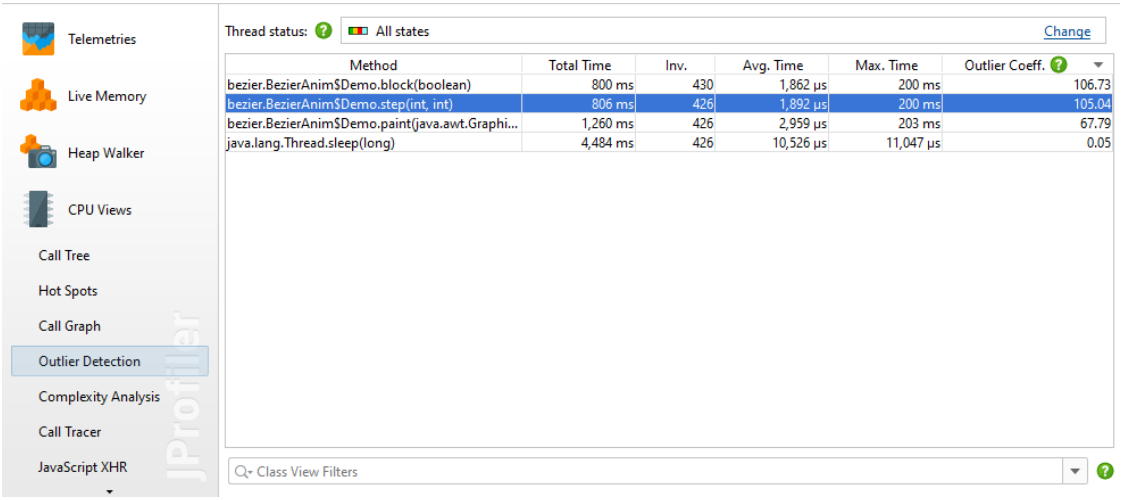
To tackle this problem, JProfiler offers the outlier detection view and the exceptional method recording feature in the call tree.

Outlier detection view

The outlier detection view shows information about the call durations and invocation counts of each method together with the maximum time that was measured for a single call. The deviation of the maximum call time from the average time shows whether all calls durations are in a narrow range or if there are significant outliers. The outlier coefficient that is calculated as

```
(maximum time - average time) / average time
```

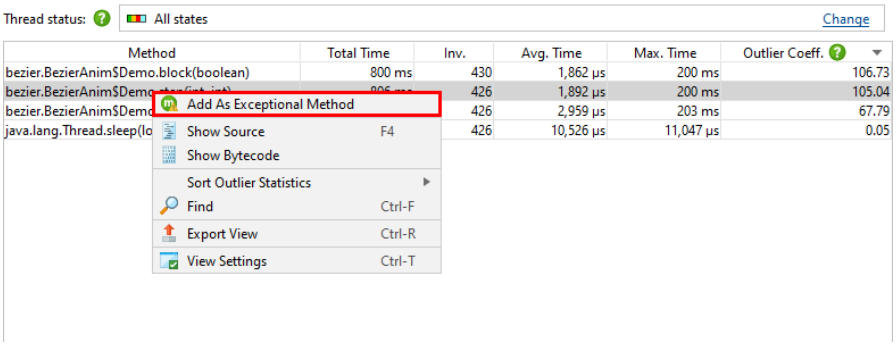
can help you to quantify methods in this respect. By default, the table is sorted such that the methods with the highest outlier coefficient are at the top. Data in the outlier detection view is available if CPU data has been recorded.



To avoid excessive clutter from methods that are only called a few times and from methods that are extremely short running, lower thresholds for the maximum time and the invocation count can be set in the view settings. By default, only methods with a maximum time of more than 10 ms and an invocation count greater than 10 are shown in the outlier statistics.

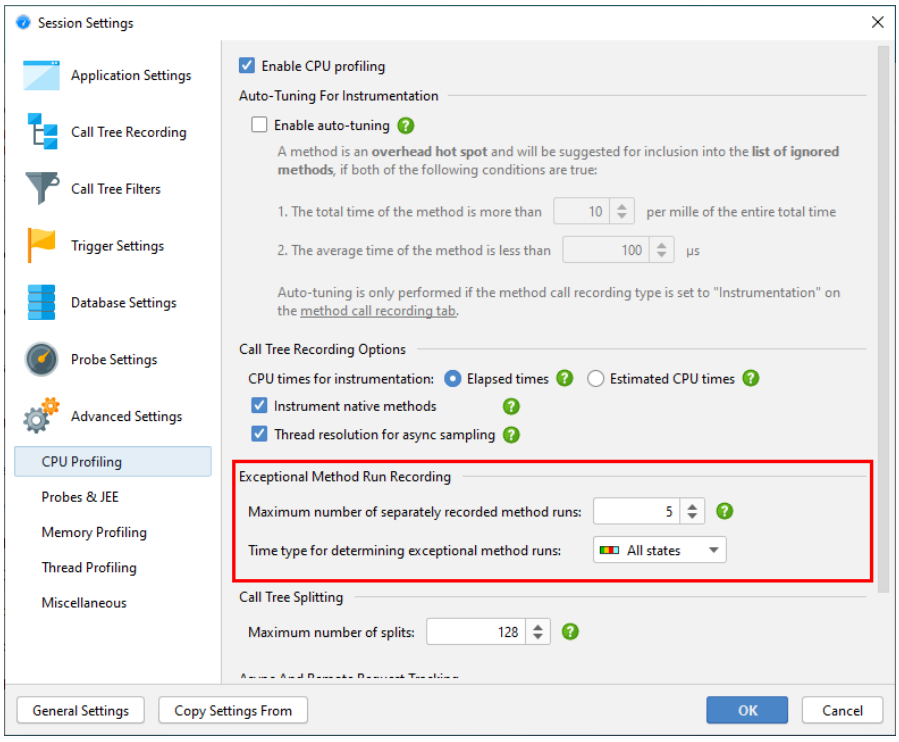
Configuring exceptional method recording

Once you have identified a method that suffers from exceptional call durations, you can add it as an exceptional method in the context menu. The same context menu action is also available in the call tree view.

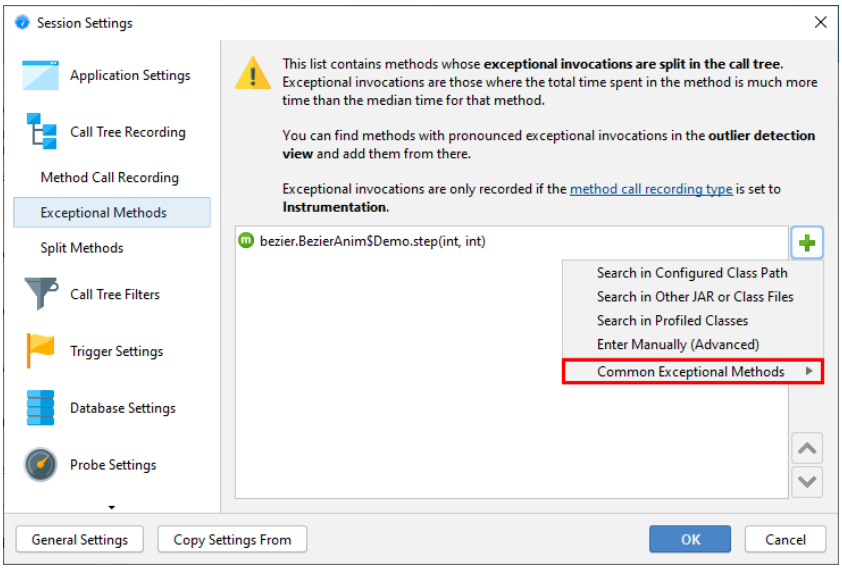


When you register a method for exceptional method recording, a few of the slowest invocations will be retained separately in the call tree. The other invocations will be merged into a single method node as usual. The number of separately retained invocations can be configured in the profiling settings, by default it is set to 5.

When discriminating slow method invocations, a certain thread state has to be used for the time measurement. This cannot be the thread status selection in the CPU views, because that is just a display option and not a recording option. By default, the wall clock time is used, but a different thread status can be configured in the profiling settings. The same thread state is used for the outlier detection view.

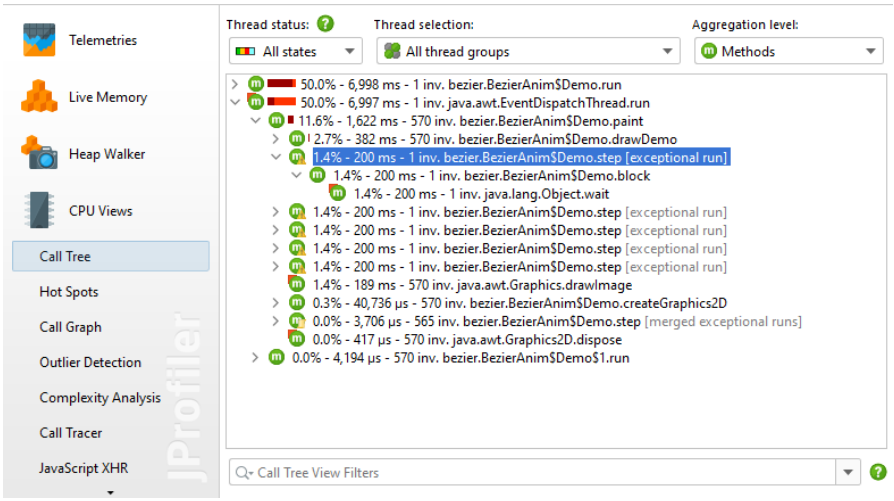


In the session settings, you can remove exceptional methods or add new ones without the context of the call tree or the outlier detection view. Also, the exceptional method configuration provides the option to add exceptional method definitions for well known systems, like the AWT and JavaFX event dispatch mechanisms where exceptionally long-running events are a major problem.



Exceptional methods in the call tree

Exceptional method runs are displayed differently in the call tree view.



The split method nodes have modified icons and show additional text:

- **[exceptional run]**

  Such a node contains an exceptionally slow method run. By definition, it will have an invocation count of one. If many other method runs are slower later on, this node may disappear and be added to the "merged exceptional runs" node depending on the configured maximum number of separately recorded method runs.

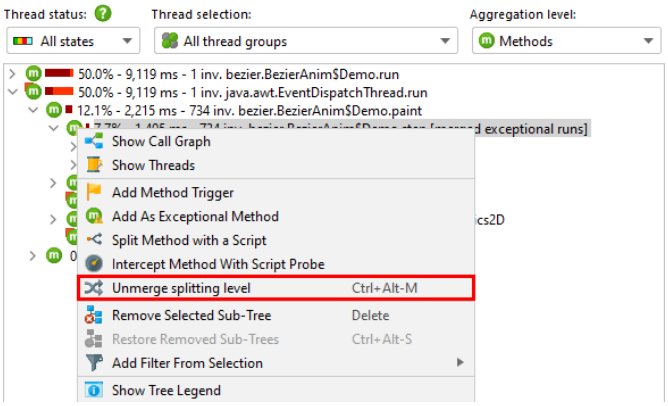
- **[merged exceptional runs]**

  Method invocations that do not qualify as exceptionally slow are merged into this node. For any call stack, there can only be one such node per exceptional method.

- **[current exceptional run]**

If an invocation was in progress while the call tree view was transmitted to the JProfiler GUI, it was not yet known whether the invocation was exceptionally slow or not. The "current exceptional run" shows the separately maintained tree for the current invocation. After the invocation completes, it will either be maintained as a separate "exceptional run" node or be merged into the "merged exceptional runs" node.

Like for call tree splitting by probes [p. 97] and split methods [p. 165], an exceptional method node has a *Merge Splitting Level* action in the context menu that lets you merge and unmerge all invocations on the fly.

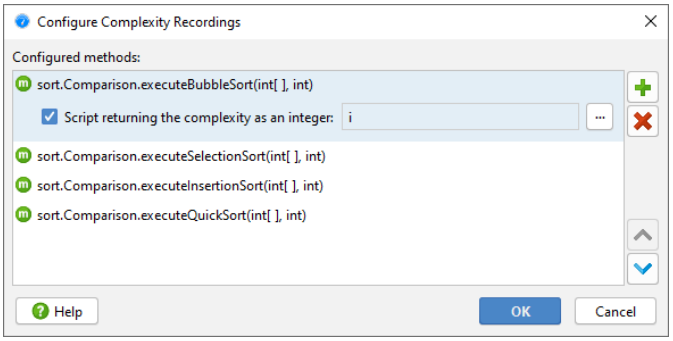


C.2 Complexity Analysis

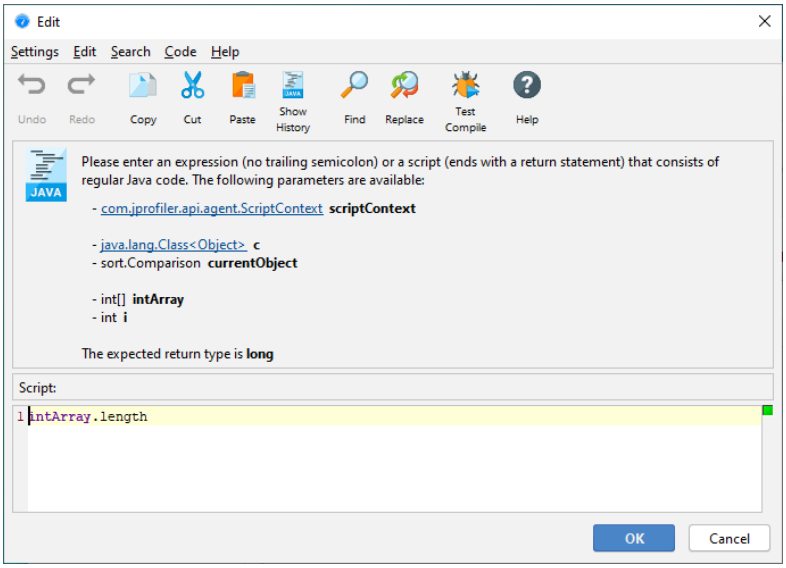
The complexity analysis view allows you to investigate the algorithmic complexity of selected methods depending on their method parameters.

To refresh the details on big O notation, an introduction to algorithmic complexity [1] and a comparative guide to complexities for common algorithms [2] are recommended readings.

First, you have to select one or more methods that should be monitored.



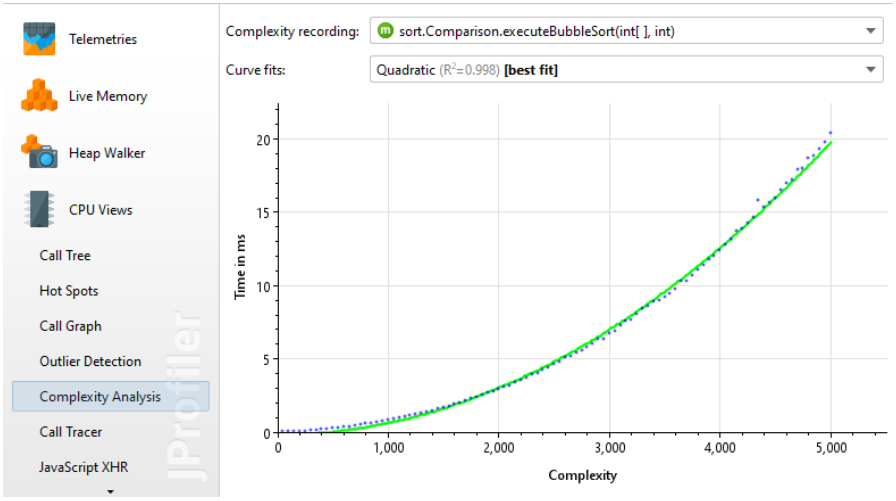
For each method, you can then enter a script whose return value of type `long` is used as the complexity for the current method call. For example, if one of the method parameters of type `java.util.Collection` is named `inputs`, the script could be `inputs.size()`.



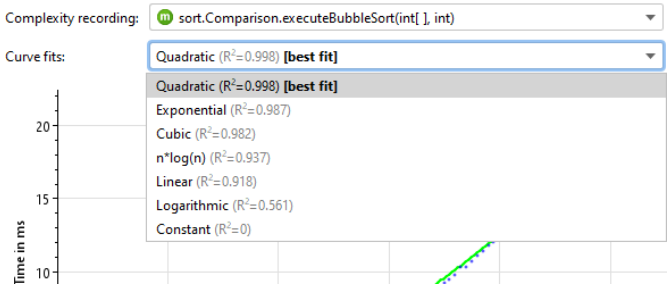
Complexity recording is independent of CPU recording. You can start and stop complexity recording directly in the complexity analysis view or by using a recording profile or a trigger action [p. 23]. After recording has been stopped, a graph with the results is displayed plotting the complexities on the x-axis against the execution times on the y-axis. To reduce memory requirements, JProfiler can combine different complexities and execution times into common buckets. The drop-down at the top allows you to switch between the different configured methods.

[1] https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
[2] http://bigocheatsheet.com/

The graph is a bubble chart, where the size of each data point is proportional to the amount of measurements in it. If all measurements are distinct, you will see a regular scatter chart. In the other extreme, if all method invocations have the same complexity and execution time, you will see a single large circle.



If there are at least 3 data points, a curve fit with common complexities is shown. JProfiler tries curve fits from several common complexities and initially shows you the best fit. The drop-down for the curve fits allows you to show other curve fit models as well. The R^2 value embedded in the description of the curve fit shows you how good the fit is. The models in the drop-down are sorted in descending order with respect to R^2, so the best model is always the first item.



Note that R^2 can be negative, because it is just a notation and not really the square of anything. Negative values indicate a fit that is worse than a fit with a constant line. The constant line fit always has an R^2 value of 0 and a perfect fit has a value of 1.

You can export the parameters of the currently displayed fit by choosing the "Properties" option in the export dialog. For automated analysis in a quality assurance environment, the command line export [p. 206] supports the properties format as well.

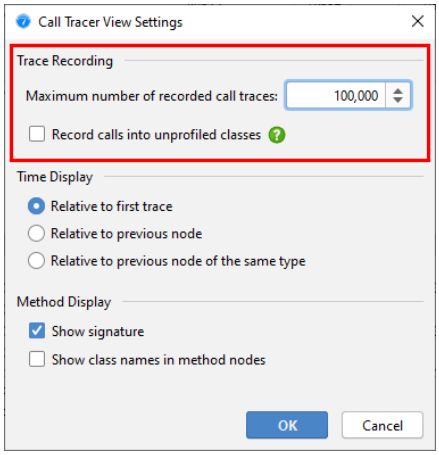
C.3 Call Tracer

Method call recording in the call tree cumulates calls with the same call stacks. Keeping precise chronological information is usually not feasible because the memory requirements are huge and the volume of the recorded data makes any interpretation quite difficult.

However, in limited circumstances, it makes sense to trace calls and keep the entire chronological sequence. For example, you may want to analyze the precise interlacing of method calls of several cooperating threads. A debugger cannot step through such a use case. Alternatively, you would like to analyze a series of method invocations, but be able to go back and forth and not just see them once like in the debugger. JProfiler provides this functionality with the call tracer.

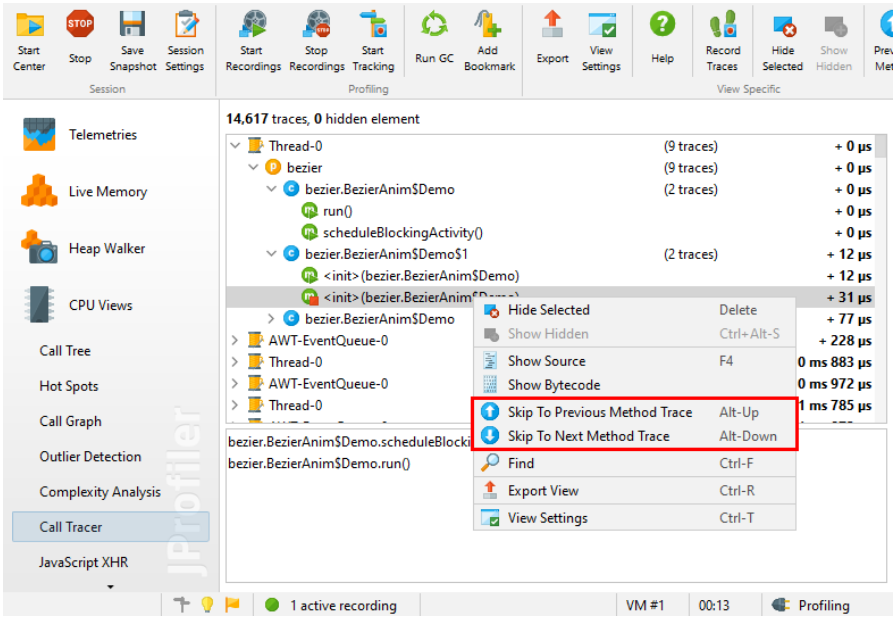
The call tracer has a separate recording action that can be activated in the call tracer view, with a trigger [p. 23] or with the profiling API [p. 113]. To avoid problems with excessive memory consumption, a cap is set on the maximum number of collected call traces. That cap is configurable in the view settings. The rate of collected traces heavily depends on your filter settings.

Call tracing only works when the method call recording type is set to instrumentation. Sampling does not keep track of single method calls, so it is technically not possible to collect call traces with sampling. Calls into compact-filtered classes are recorded in the call tracer, just like in the call tree. If you just want to focus on your own classes, you can exclude these calls in the view settings.



The traced method calls are displayed in a tree with three levels that make it easier to skip related calls by collapsing them. The three groups are threads, packages and classes. Each time the current value for any of these groups changes, a new grouping node is created.

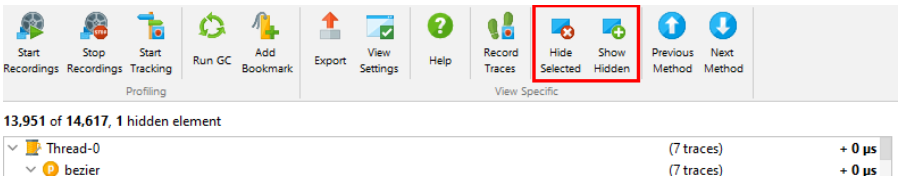
At the lowest level there are method entry and method exit nodes. Below the table with the call traces, the stack trace of the currently selected method trace is shown. If call traces into other methods have been recorded from the current method or if another thread interrupts the current method, the entry and exit nodes for the that method will not be adjacent. You can navigate on the method level only by using the *Previous Method* and *Next Method* actions.



The timing that is displayed on the traces and all grouping nodes refers to the first trace by default, but can be changed to show relative times since the previous node. If the previous node is the parent node, that difference will be zero. Also available is the option to show relative times with respect to the previous node of the same type.

Even with appropriate filters, a huge number of traces can be collected in a very short time. To eliminate traces that are of no interest, the call tracer allows you to quickly trim the displayed data. For example, certain threads might not be relevant or traces in certain packages or classes might not be interesting. Also, recursive method invocations can occupy a lot of space and you might want to eliminate those single methods only.

You can hide nodes by selecting them and pressing the delete key. All other instances of the selected nodes and all associated child nodes will be hidden as well. At the top of the view you can see how many call traces out of all the recorded traces are still shown. To show hidden nodes again, you can click on the *Show Hidden* tool bar button.

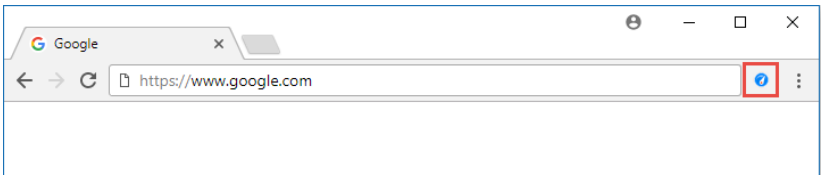


C.4 JavaScript XHR Origin Tracking

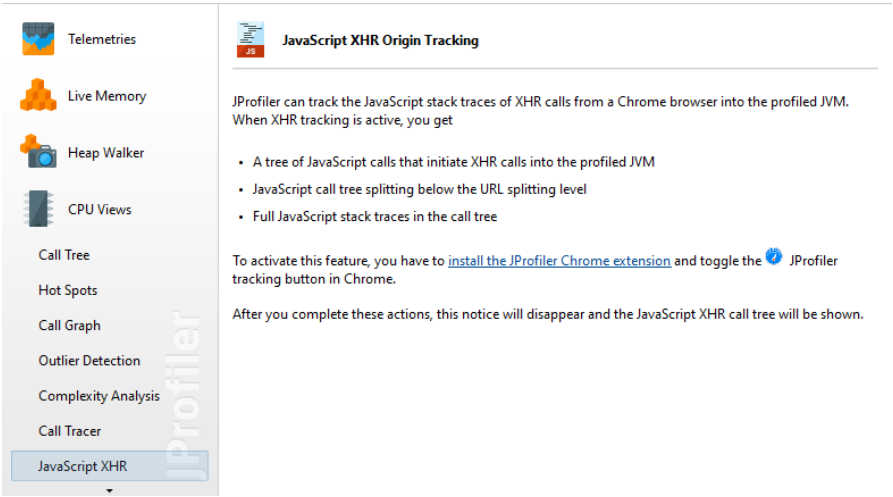
With JavaScript XHR origin tracking, you can split servlet invocations for different stack traces in the browser during XMLHttpRequest [1] or Fetch [2] requests, so you can better correlate the activity in the profiled JVM with actions in the browser. in the following, "XHR" designates both the XMLHttpRequest and the Fetch mechanisms.

Browser plugin

To use this feature, you have to use Google Chrome [3] as the browser and install the JProfiler origin tracker extension [4].



The Chrome extension adds a button with a JProfiler icon to the tool bar that starts tracking. When you start tracking, the extension will intercept all XHR calls and report them to a locally running JProfiler instance. As long as tracking has not been started, JProfiler will show an information page that tells you how to set up JavaScript XHR origin tracking.



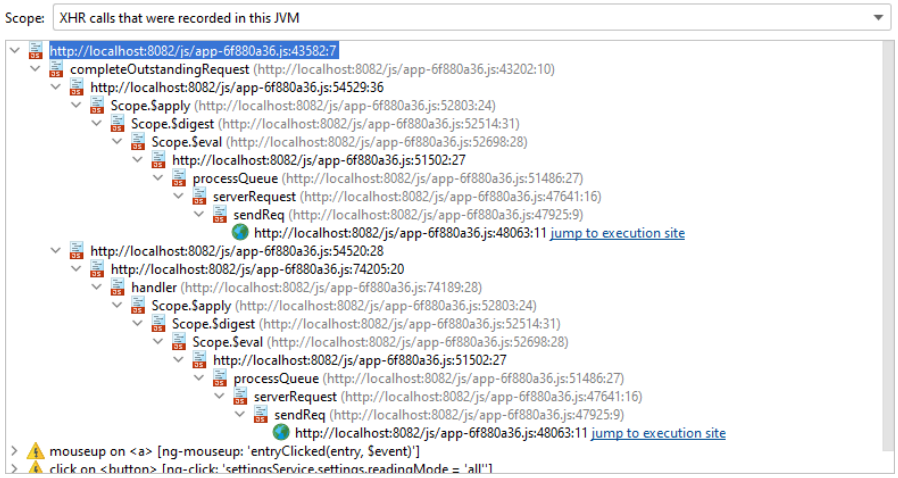
When tracking is activated, the JProfiler extension will ask you to reload the page. This is necessary for adding instrumentation. If you choose to not reload the page, event detection may not work.

The tracking status is persistent on a per-domain basis. If you restart the browser while tracking is active and visit the same URL, tracking will automatically be enabled, without the need to reload the page.

[1] https://xhr.spec.whatwg.org/
[2] https://fetch.spec.whatwg.org/
[3] http://www.google.com/chrome/
[4] https://chrome.google.com/webstore/detail/jprofiler-origin-tracker/mnicmpklpjkhohdbcdkflhochdfnmmbm

JavaScript XHR tree

If the XHR calls are handled by a JVM that is profiled by an active profiling session in JProfiler, the JavaScript XHR view will show a cumulated call tree of these calls. If the view remains empty, you can switch the "Scope" at the top of the view to "All XHR calls" to check if any XHR calls have been made.



Javascript call stack nodes include information on the source file and the line number. The function where the XHR call is made has a special icon and and adjacent hyperlink in case the XHR call was handled by the profiled JVM. The hyperlink will take you to the Javascript splitting node in the call tree view [p. 48] where you can see the server side call tree that was responsible for handling requests of this type.

At the top of the tree you find browser event nodes that show event name and element name together with important attributes that help you pin down the source of the event. Not all requests have an associated event.

The extension is aware of several popular JavaScript frameworks and walks the ancestor hierarchy between the target node of an event up to the node where the event listener is located, looking for attributes that are suitable for display and splitting the call tree. Failing to find framework-specific attributes, it stops at an `id` attribute. In the absence of an ID, it searches for "control elements" like `a`, `button` or `input`. All failing, the element where the event listener is registered will be shown.

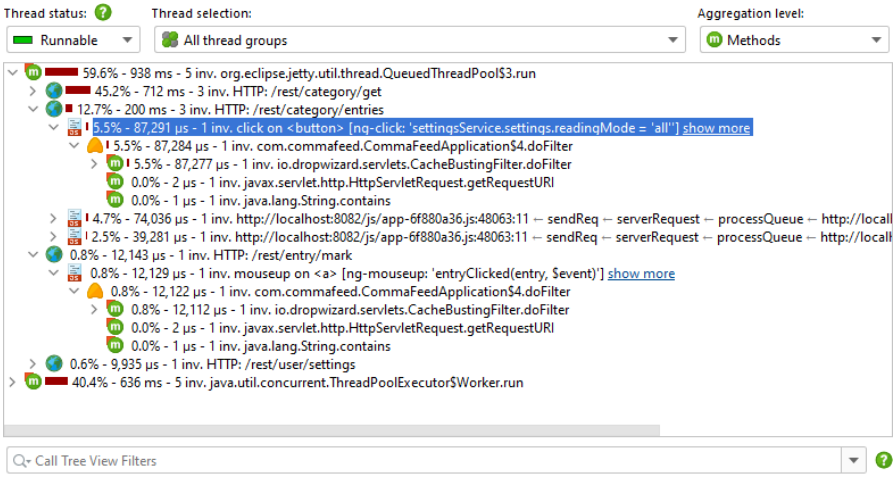
In some cases, the automatic detection of interesting attributes may not be suitable and you may prefer a different call tree splitting. For example, some frameworks assign automatic IDs, but it would be more readable to group all elements together with a semantic description of the action. To achieve a different call tree splitting, add the HTML attribute

```
data-jprofiler="..."
```

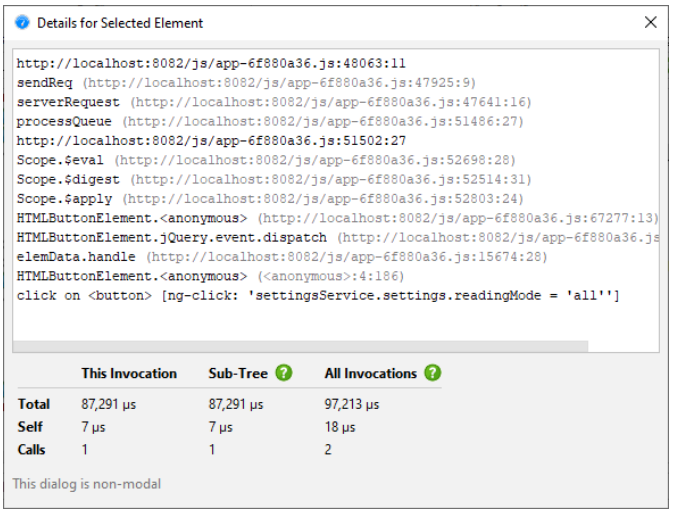
to the target element or an element between the target and the location of the event listener. The text in that attribute will be used for splitting and other attributes will be ignored.

Call tree splitting

In the call tree view, XHR calls will split the call tree for each separate combination of browser event and call stack. The splitting nodes show information about the browser event. If no event is in progress, like in a call to `setTimeout()`, the last few stack frames are displayed inline.



The "show more" hyperlink on these nodes opens the same detail dialog that is opened by the *View->Show Node Details* action. For JavaScript splitting nodes, the detail dialog does not show the text of the node, but the entire browser call stack. To inspect the call stack of other JavaScript splitting nodes, leave the non-modal detail dialog open and click on those nodes, the detail dialog will update its contents automatically.



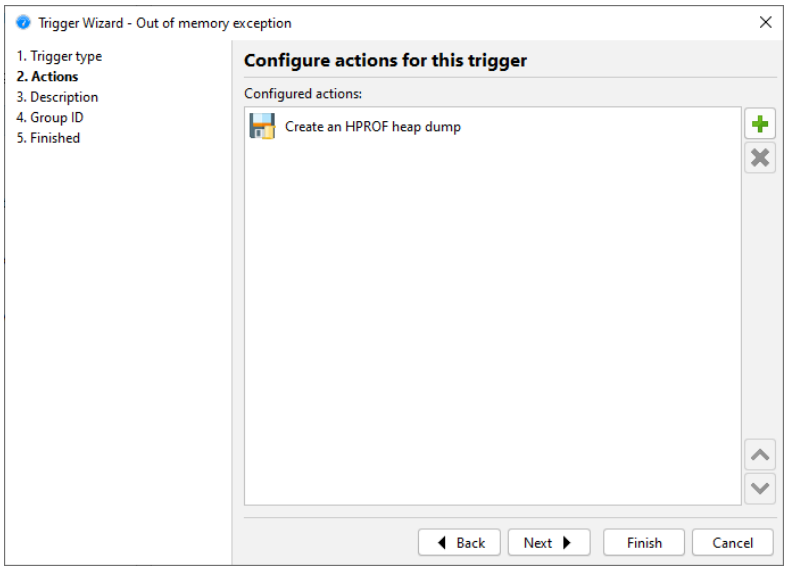
D Heap Walker Features In Detail

D.1 HPROF And PHD Heap Snapshots

The HotSpot JVM and the Android Runtime both support heap snapshots in the HPROF format, The IBM J9 JVM writes such snapshots in the PHD format. PHD files do not contain garbage collector roots, so JProfiler simulates classes as roots. Finding class loader memory leaks may be difficult with a PHD file.

Native heap snapshots can be saved without the profiling agent and incur a lower overhead than JProfiler heap snapshots, because they are saved without the constraints of a general purpose API. On the flip side, the native heap snapshots support less functionality than JProfiler heap snapshots. For example, allocation recording information is not available, so you cannot see where objects have been allocated. HPROF and PHD snapshots can be opened in JProfiler with*Session->Open Snapshot*, just like you would open a JProfiler snapshot. Only the heap walker will be available, all other sections will be grayed out.

In a live session, you can create and open an HPROF/PHD heap snapshot by invoking *Profiling->Save HPROF/PHD Heap Snapshot*. For offline profiling [p. 113], there is a "Create an HPROF heap dump" trigger action. It is usually used with the "Out of memory exception" trigger to save an HPROF snapshot when an `OutOfMemoryError` is thrown.



This corresponds to the VM parameter [(1)]

```
-XX:+HeapDumpOnOutOfMemoryError
```

that is supported by HotSpot JVMs.

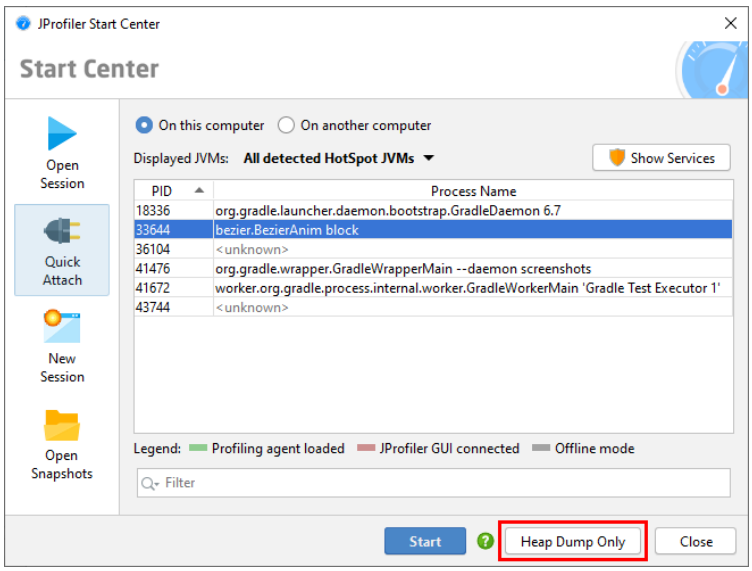
An alternative way to extract an HPROF heap dump from a running system is via the command line tool `jmap` that is part of the JRE. Its invocation syntax

```
jmap -dump:live,format=b,file=<filename> <PID>
```

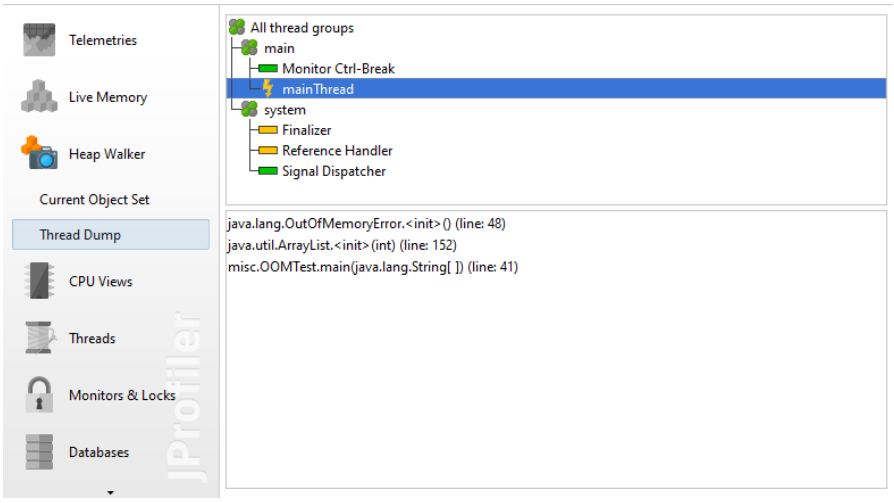
[(1)] http://docs.oracle.com/javase/9/troubleshoot/command-line-options1.htm#JSTGD592

is difficult to remember and requires you to use the `jps` executable to find out the PID first. JProfiler ships with an interactive command line executable `bin/jpdump` that is much more convenient. It lets you select a process, can connect to processes running as a service on Windows, has no problems with mixed 32-bit/64-bit JVMs and auto-numbers HPROF snapshot files. Execute it with the `-help` option to get more information.

Taking HPROF heap snapshots without loading the profiling agent is also supported in the JProfiler GUI. When attaching to a process, locally or remotely, you always have the possibility to just take an HPROF heap snapshot.



HPROF snapshots can contain thread dumps. When an HPROF snapshot was saved as a consequence of an `OutOfMemoryError`, the thread dump may be able to convey what part of the application was active at the time of the error. The thread that triggered the error is marked with a special icon.



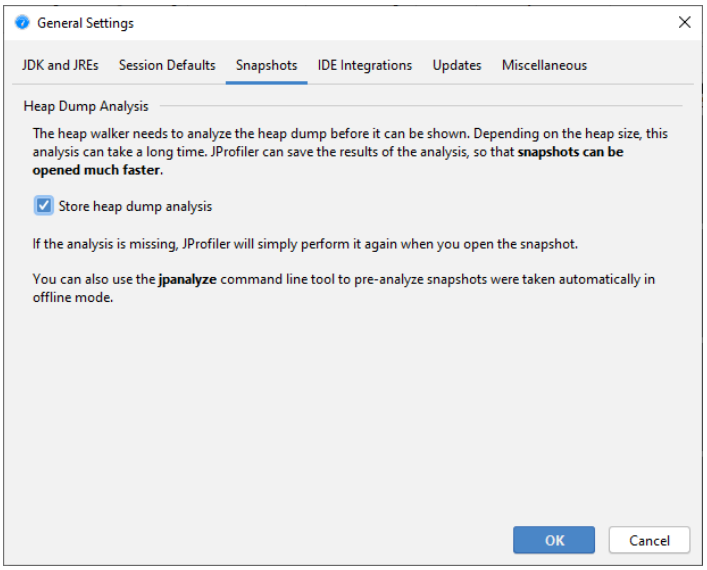
D.2 Minimizing Overhead In The Heap Walker

For small heaps, taking a heap snapshot takes a couple of seconds, but for very large heaps, this can be a lengthy process. Insufficient free physical memory can make the calculations a lot slower. For example, if the JVM has a 50 GB heap and you are analyzing the heap dump on your local machine with only 5 GB of free physical memory, JProfiler cannot hold certain indices in memory and the processing time increases disproportionately.

Because JProfiler mainly uses native memory for the heap analysis, it is not recommended to increase the `-Xmx` value in the `bin/jprofiler.vmoptions` file unless you have experienced an `OutOfMemoryError` and JProfiler has instructed you to make such a modification. Native memory will be used automatically if it is available. After the analysis has completed and the internal database has been built, the native memory will be released.

For a live snapshot, the analysis is calculated immediately after taking the heap dump. When you save a snapshot, the analysis is saved to a directory with the suffix `.analysis` next to the snapshot file. When you open the snapshot file, the heap walker will be available very quickly. If you delete the `.analysis` directory, the calculation will be performed again when the snapshot is opened, so if you send the snapshot to somebody else, you don't have to send the analysis directory along with it.

If you want to save memory on disk or if the generated `.analysis` directories are inconvenient, you can disable their creation in the general settings.

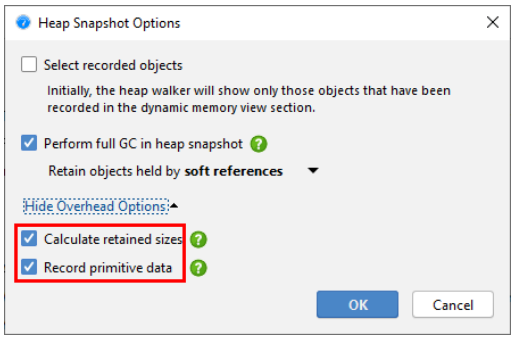


HPROF snapshots and JProfiler snapshots that were saved with offline profiling [p. 113] do not have an `.analysis` directory next to them, because the analysis is performed by the JProfiler UI and not by the profiling agent. If you do not want to wait for the calculation when opening such snapshots, the `jpanalyze` command line executable can be used to pre-analyze [p. 206] snapshots.

It is advisable to open snapshots from writable directories. When you open a snapshot without an analysis, and its directory is not writable, a temporary location is used for the analysis. The calculation then has to be repeated each time the snapshot is opened.

A big part of the analysis is the calculation of retained sizes. If the processing time is too long and you don't need the retained sizes, you can disable their calculation in the overhead options of the heap walker options dialog. In addition to retained sizes, the "Biggest objects" view will not be available either in that case. Not recording primitive data makes the heap snapshot

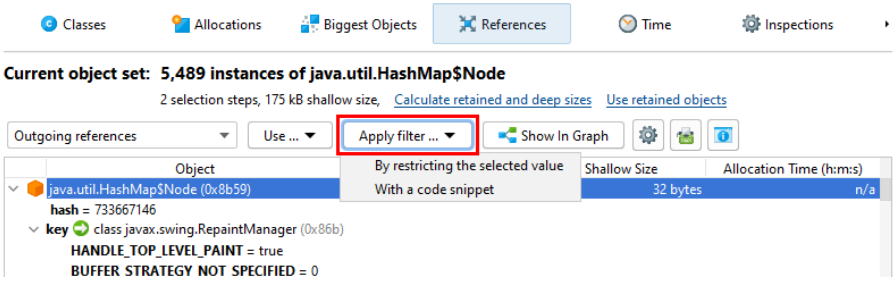
smaller, but you will not be able to see them in the reference views. The same options are presented when opening snapshots if you choose *Customize analysis* in the file chooser dialog.



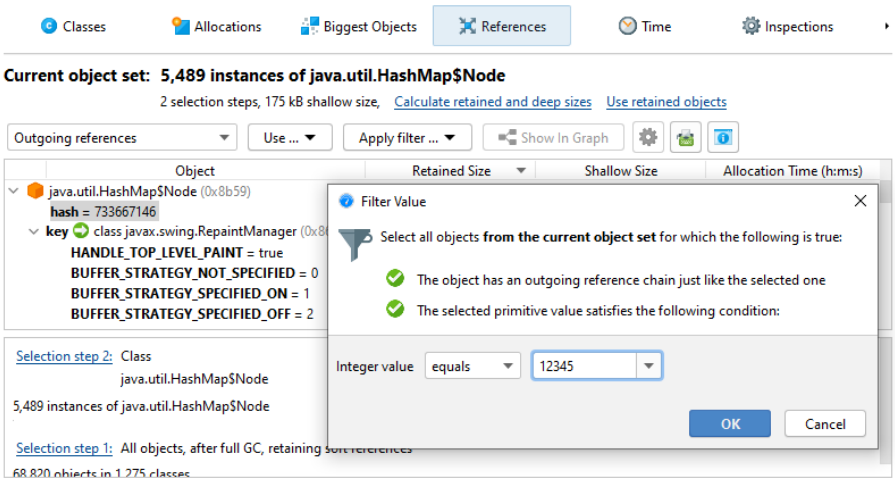
D.3 Filters And Live Interactions

When looking for objects of interest in the heap walker, you often arrive at an object set that has too many instances of the same class in it. To further trim the object set according to your particular focus, the selection criteria could then involve their properties or references. For example, you may be interested in HTTP session objects that contain a particular attribute. In the merged outgoing reference view of the heap walker you can perform selection steps that involve chains of references for the entire object set.

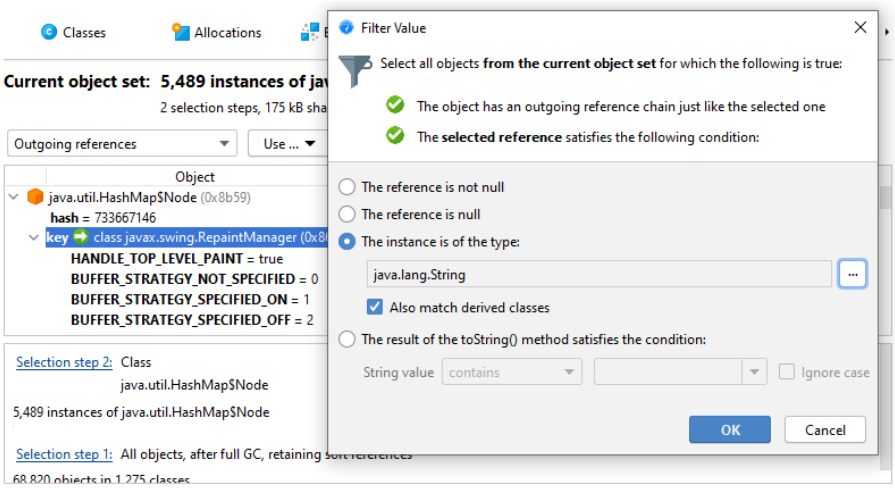
However, the outgoing references view where you see individual objects offers much more powerful functionality to make selection steps that constrain references and primitive fields.



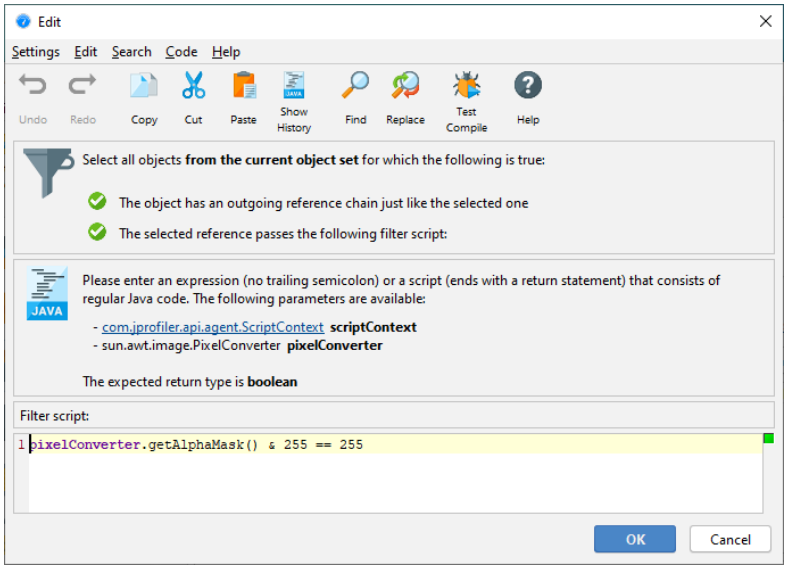
When you select a top-level object, a primitive value or a reference in the outgoing references view, the *Apply Filter->By Restricting The Selected Value* action becomes enabled. Depending on the selection, the filter value dialog offers different options. Whatever options you configure, you always implicitly add the constraint that objects in the new object set must have outgoing reference chains like the selected one. Filters always work on the top-level objects by restricting the current set of objects into a possibly smaller set.



Constraining primitive values works in both HPROF and JProfiler heap snapshots. For reference types, you can ask JProfiler to filter non-null values, null values, and values of a selected class. Filtering by the result of the `toString()` method is only available in live sessions, except for `java.lang.String` and `java.lang.Class` objects where JProfiler can figure this out by itself.

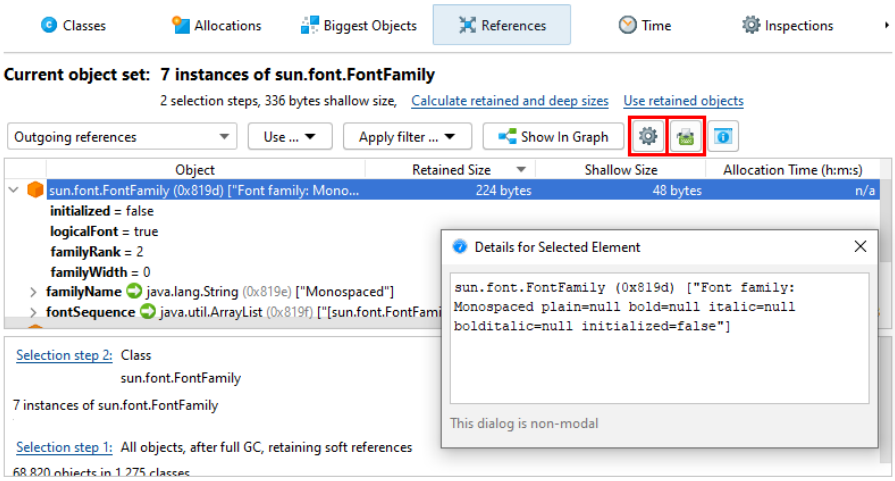


The most powerful filter type is the code filter snippet. In the script editor, you have access to the object or reference and can write an expression or script whose boolean return value decides whether an instance should be retained in the current object set or not.

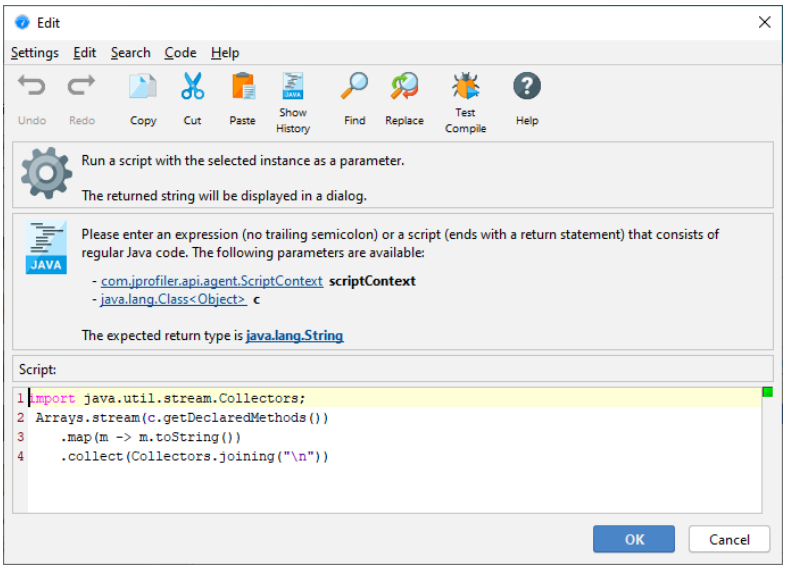


Of course this feature can only work for live sessions, because JProfiler needs access to the live objects. Another consideration is that an object may have been garbage collected since the heap snapshot was taken. In that case, such an object would not be included in the new object set when a code snippet filter is executed.

Apart from filters, there are two other features in the outgoing references view for interacting with individual objects: The *Show toString() Values* action invokes the `toString()` method on all objects that are currently visible in the view and shows them directly in the reference nodes. The nodes can become very long and the text may be cut off. Using the *Show Node Details* action from the context menu helps you to see the entire text.



A more general method of obtaining information from an object than calling the `toString()` method is to run an arbitrary script that returns a string. The *Run Script* action next to the *Show toString() Values* action allows you to do that when a top-level object or a reference is selected. The result of the script execution is displayed in a separate dialog.

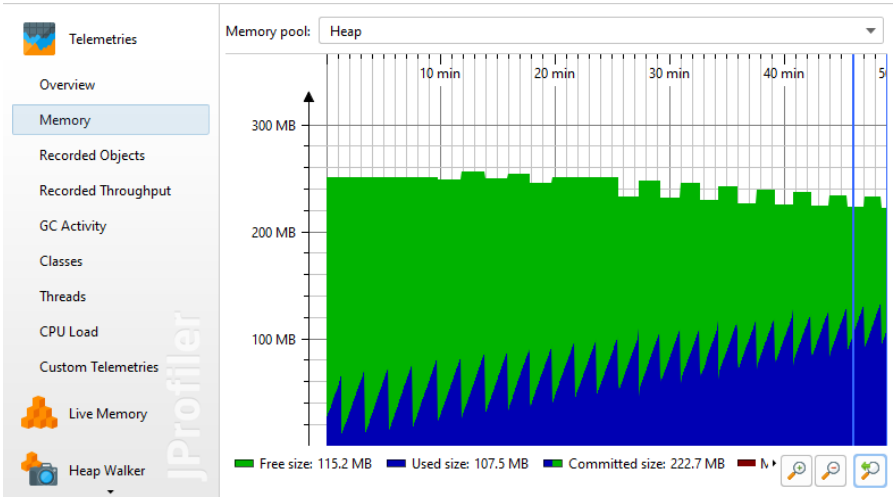


D.4 Finding Memory Leaks

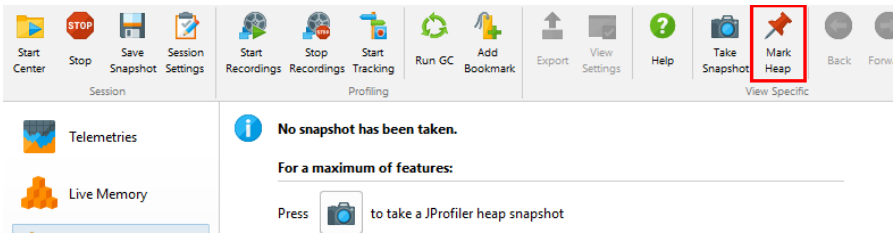
Distinguishing regular memory usage from a memory leak is often not quite simple. However, both excessive memory usage and memory leaks have the same symptoms and so they can be analyzed in the same way. The analysis proceeds in two steps: Locating suspicious objects and finding out why those objects are still on the heap.

Finding new objects

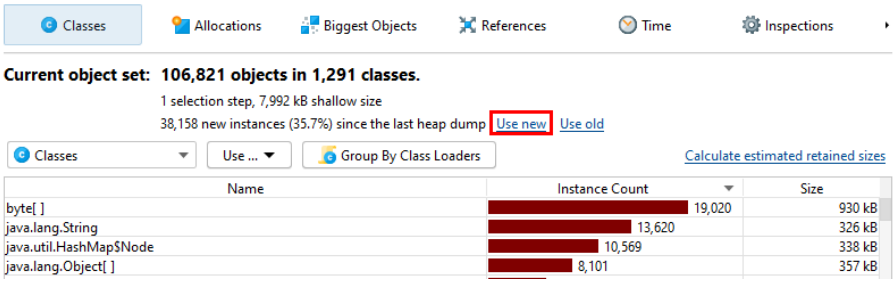
When an application with a memory leak is running, it consumes more and more memory over time. Detecting the growth of memory usage is best done with the VM telemetries and the differencing functionality [p. 66] in the "All objects" and the "Recorded objects" views. With these views you can determine if you have a problem and how severe it is. Sometimes, the difference column in the instance tables already gives you an idea what the problem is.



Any deeper analysis of a memory leak requires the functionality in the heap walker. To investigate a memory leak around a particular use case in detail, the "Mark heap" functionality [p. 75] is best suited. It allows you to identify new objects that have remained on the heap since a particular previous point in time. For these objects, you have to check whether they are still legitimately on the heap or if a faulty reference keeps them alive even though the object serves no further purpose.



Another way to isolate a set of objects that you are interested in is through allocation recording. When taking a heap snapshot, you have the option to show all recorded objects. However, you may not want to limit allocation recording to just a particular use case. Also, allocation recording has a high overhead, so the *Mark Heap* action will have a comparatively much smaller impact. Finally, the heap walker lets you select old and new objects at any selection step with the *Use new* and *Use old* hyperlinks in the header if you have marked the heap.



Analyzing the biggest objects

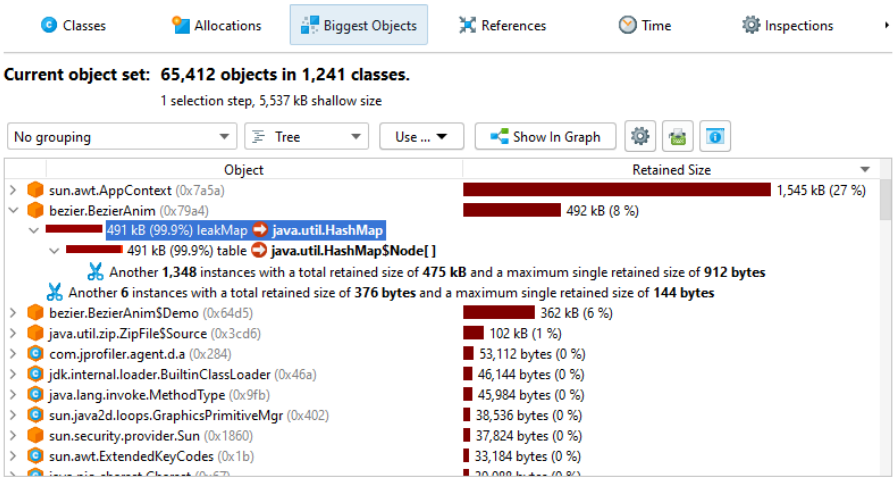
If a memory leak fills up the available heap, it will dwarf other types of memory usage in the profiled application. In that case, you don't have to examine new objects, but simply analyze what objects are most important.

Memory leaks can have a very slow rate and may not become dominant for a long time. Profiling such a memory leak until it becomes visible may not not practicable. With the built-in facility in the JVM to automatically save an HPROF snapshot [p. 185] when an `OutOfMemoryError` is thrown, you can get a snapshot where the memory leak is more important than the regular memory consumption. In fact, it's a good idea to always add

```
-XX:+HeapDumpOnOutOfMemoryError
```

to the VM parameters or production systems so you have a way to analyze memory leaks that may be hard to reproduce in development environments.

If the memory leak is dominant, the top objects in the "Biggest objects" view of the heap walker will contain the memory that was retained by mistake. While the biggest objects themselves may be legitimate objects, opening their dominator trees will lead to the leaked objects. In simple situations, there is a single object that will contain most of the heap. For example, if a map is used to cache objects and that cache is never cleared, then the map will show up in the dominator tree of the biggest object.

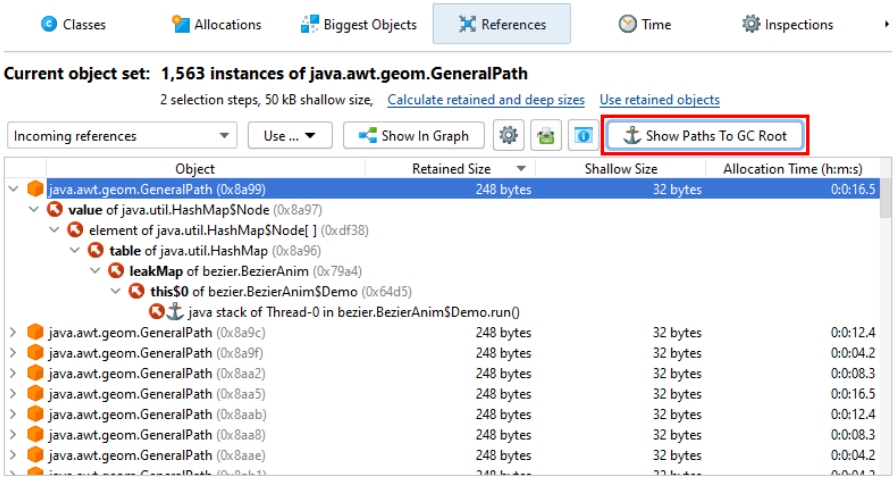


Finding strong references chains from garbage collector roots

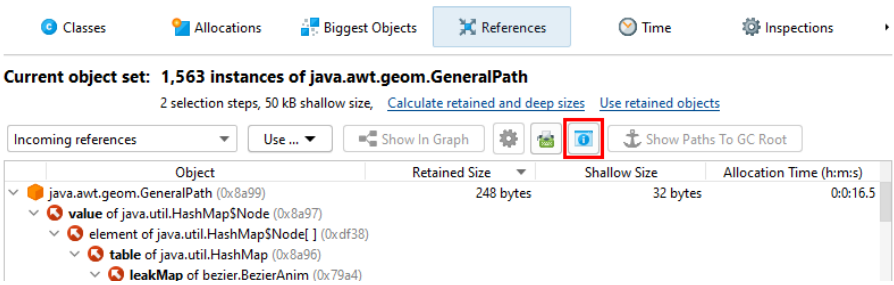
An object can only be a problem if it is strongly referenced. "Strongly referenced", means that there is at least one chain of references from a garbage collector root to the object. "Garbage

collector" roots (in short GC roots) are special references in the JVM that the garbage collector knows about.

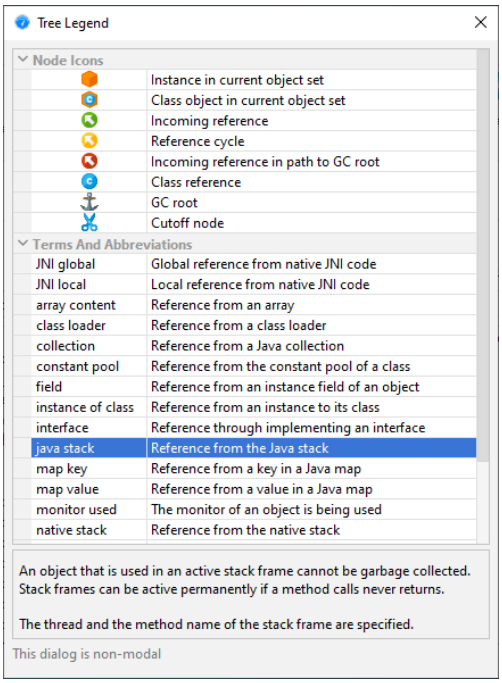
To find a reference chain from a GC root, you can use the *Show Path To GC Root* actions in the "Incoming references" view or in the heap walker graph. Such reference chains may by very long in practice, so they can generally be interpreted more easily in the "Incoming references" view. The references point from the bottom towards the object at the top level. Only the reference chains that are the result of the search are expanded, other references on the same levels are not visible until a node is closed and opened again or the *Show All Incoming References* action in the context menu is invoked.



To get an explanation for types of GC roots and other terms that are used in the reference nodes, use the tree legend.



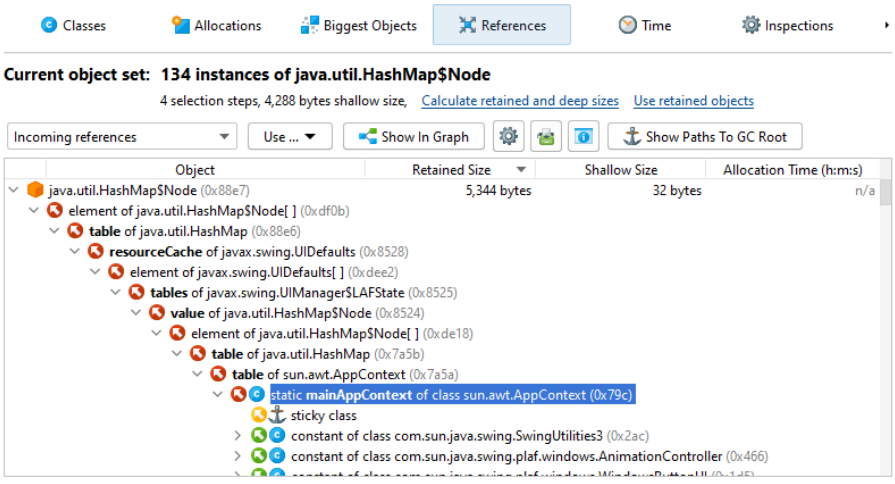
When you select nodes in the tree, the non-modal tree legend highlights all used icons and terms in the selected node. Clicking on a row in the dialog will show an explanation at the bottom.



Important types of garbage collector roots are references from the stack, references created by native code through JNI and resources like live threads and object monitors that are currently being used. In addition, the JVM adds in a couple of "sticky" references to keep important systems in place.

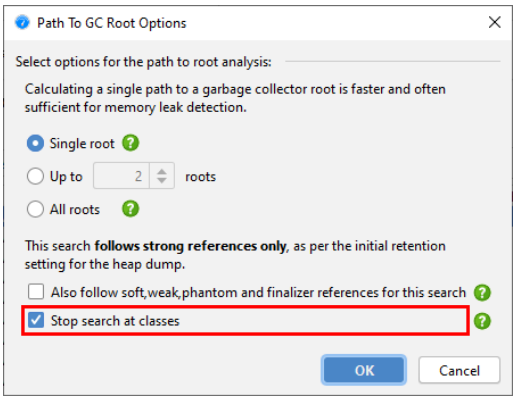
Classes and classloaders have a special circular reference scheme. Classes are garbage collected together with their classloader when

- no class loaded by that classloader has any live instances
- the classloader itself is unreferenced except by its classes
- none of the `java.lang.Class` objects are referenced except in the context of the classloader



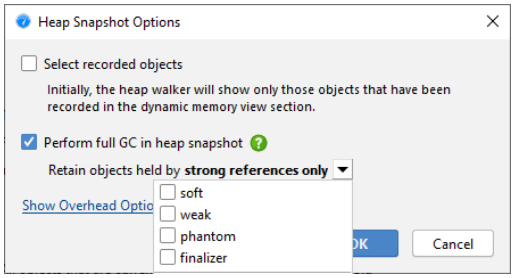
In most circumstances, classes are the last step on the path to the GC root that you are interested in. Classes are not GC roots by themselves. However, in all situations where no custom

classloaders are used, it is appropriate to treat them as such. This is JProfiler's default mode when searching for garbage collector roots, but you can change it in the path to root options dialog.

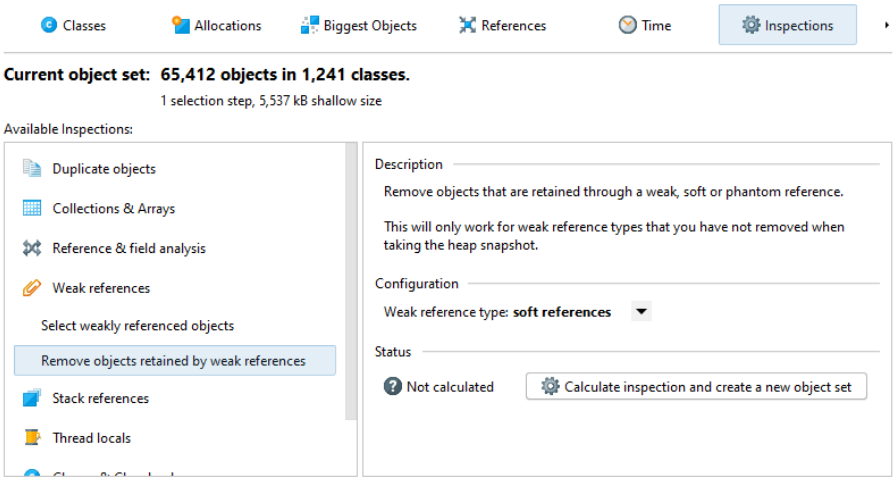


If you have problems interpreting the shortest path to a GC root, you can search for additional paths. Searching for all paths to GC roots is not recommended in general because it can produce a large number of paths.

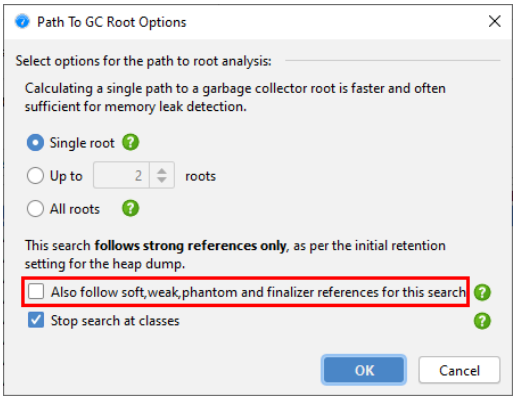
In contrast to the live memory views, the heap walker never shows unreferenced objects. However, the heap walker may not only show strongly referenced objects. By default, the heap walker also retains objects that are only referenced by soft references, but eliminates objects that are only referenced by weak, phantom or finalizer references. Because soft references are not garbage collected unless the heap is exhausted, they are included because otherwise you might not be able to explain large heap usages. In the options dialog which is shown when you take a heap snapshot, you can adjust this behavior.



Having weakly referenced objects in the heap walker may be interesting for debugging purposes. If you want to remove weakly referenced objects later on, you can use the "Remove objects retained by weak references" inspection.

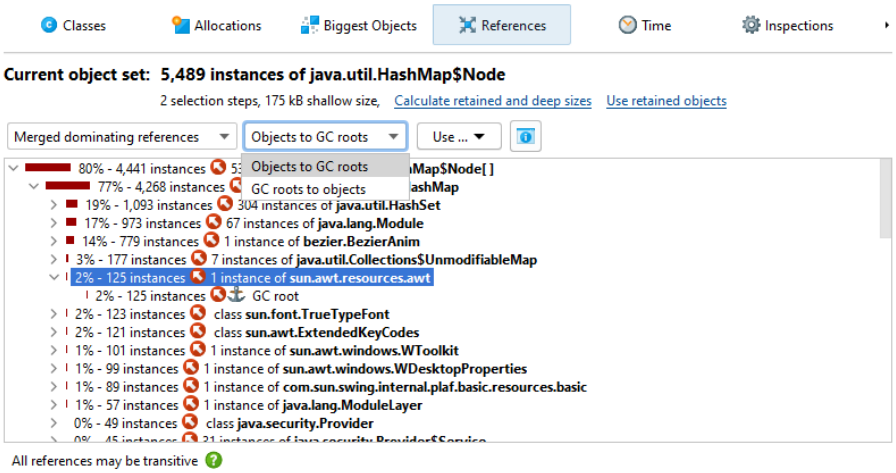


When searching for paths to GC roots, the reference types that were selected to retain objects in the heap walker options dialog are taken into account. In that way, the path to GC root search can always explain why an object was retained in the heap walker. In the options dialog for the path to GC root search you can widen the acceptable reference types to all weak references.



Eliminating entire object sets

Until now we have only looked at single objects. Often you will have many objects of the same type that are part of a memory leak. In many cases, the analysis of a single object will also be valid for the other objects in the current object set. For the more general case where the objects of interest are referenced in different ways, the "Merged dominating references" view will help you to find out which references are responsible for holding the current object set on the heap.



Each node in the dominating reference tree tells you how many objects in the current object set will be eligible for garbage collection if you eliminate that reference. Objects that are referenced by multiple garbage collector roots may not have any dominating incoming reference, so the view may only help you with a fraction of the objects, or it may even be empty. In that case, you have to use the merged incoming reference view and eliminate garbage collector roots one by one.

E Configuration In Detail

E.1 Trouble Shooting Connection Problems

When a profiling session cannot be established, the first thing to do is to have a look at the terminal output of the profiled application or application server. For application servers, the stderr stream is often written to a log file. This may be a separate log file and not the main log file of the application server. For example, the Websphere application server writes a `native_stderr.log` file where only the stderr output is included. Depending on the content of the stderr output, the search for the problem takes different directions:

Connection problems

If stderr contains "`Waiting for connection ...`", the configuration of the profiled application is ok. The problem might then be related to the following questions:

- Did you forget to start the "Attach to remote JVM" session in the JProfiler GUI on your local machine? Unless the profiling agent is configured to start up immediately with the "nowait" option, it will wait until the JProfiler GUI connects before letting the VM continue to startup.
- Is the host name or the IP address configured correctly in the session settings?
- Did you configure a wrong communication port? The communication port has nothing to do with HTTP or other standard port numbers and must not be the same as any port that is already in use. For the profiled application, the communication port is defined as an option to the profiling VM parameter. With the VM parameter `-agentpath:<path to jprofilerti library>=port=25000`, a port of 25000 would be used.
- Is there a firewall between the local machine and the remote machine? There may be firewalls for incoming as well as for outgoing connections or even firewalls on gateway machines in the middle.

Port binding problems

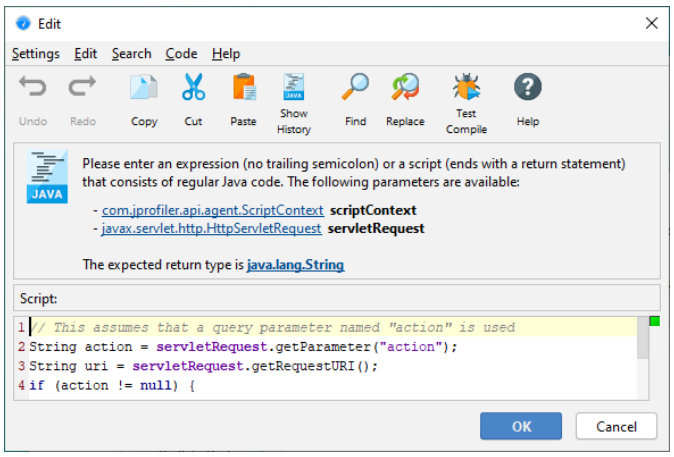
If stderr contains an error message about not being able to bind a socket, the port is already in use. In that case, check the following questions:

- Did you start the profiled application multiple times? Each profiled application needs a separate communication port.
- Are there any zombie java processes of previous profiling runs that are blocking the port?
- Is there a different application that is using the communication port?

If there are no lines in stderr that are prefixed with `JProfiler>` and your application or application server starts up normally, the `-agentpath:[path to jprofilerti library]` VM parameter has not been included in the Java call. You should find out which java call in your startup script is actually executed and add the VM parameters there.

E.2 Scripts In JProfiler

JProfiler's built-in script editor allows you to enter custom logic in various places in the JProfiler GUI, including custom probe configuration, split methods, heap walker filters and many more.



The box above the edit area shows the available parameters of the script as well as its return type. By invoking *Help->Show Javadoc Overview* from the menu you can get more information on classes from the `com.jprofiler.api.*` packages.

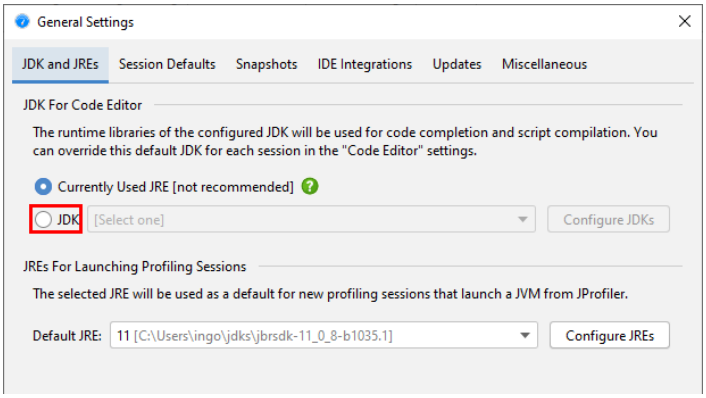
A number of packages can be used without using fully-qualified class names. Those packages are:

- java.util.*
- java.io.*

You can put a number of import statements as the first lines in the text area in order to avoid using fully qualified class names.

All scripts are passed an instance of `com.jprofiler.api.agent.ScriptContext` that allows you to save state between successive invocations of the script.

To get the maximum editor functionality, it is recommended to configure a JDK in the general settings. By default, the JRE that JProfiler runs with is used. In that case, code completion does not offer parameter names and Javadoc for classes in the JRE.



Script types

Scripts can be expressions. An expression doesn't have a trailing semicolon and evaluates to the required return type. For example,

```
object.toString().contains("test")
```

would work as a filter script in the outgoing reference view of the heap walker.

Alternatively, a script consists of a series of Java statements with a return statement of the required return type as the last statement:

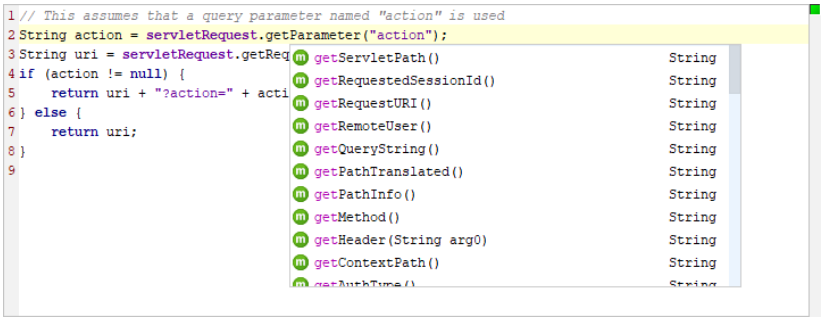
```
import java.lang.management.ManagementFactory;
return ManagementFactory.getRuntimeMXBean().getUptime();
```

The above example would work for a script telemetry. JProfiler automatically detects whether you have entered an expression or a script.

If you want to reuse a script that you have entered previously, you can select it from the script history. If you click on the  *Show History* tool bar button, all previously used scripts are shown. Scripts are organized by script signature and the current script signature is selected by default.

Code completion

Pressing `CTRL-Space` brings up a popup with code completion proposals. Also, typing a dot (".") shows this popup after a delay if no other character is typed. The delay is configurable in the editor settings. While the popup is being displayed, you can continue to type or delete characters with `Backspace` and the popup will be updated accordingly. "Camel-hump" completion is supported. For example, typing `NPE` and hitting `CTRL-Space` will propose `java.lang.NullPointerException` among other classes. If you accept a class that is not automatically imported, the fully qualified name will be inserted.



The completion popup can suggest:

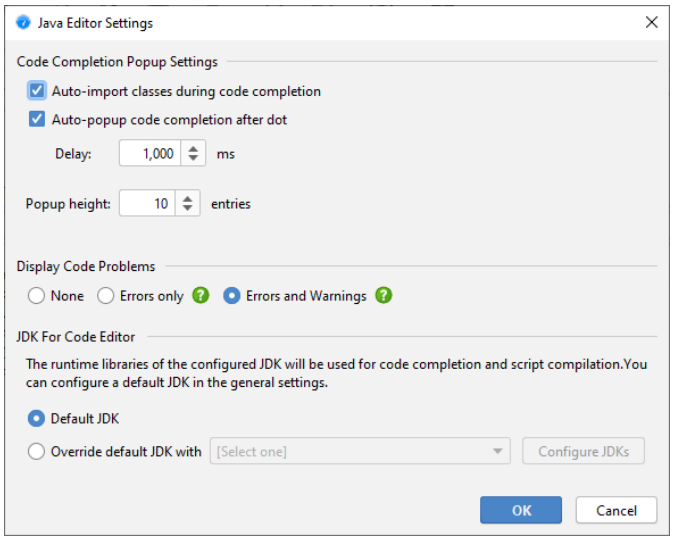
- variables and script parameters. Script parameters are displayed in bold font.
- packages, when typing an import statement
- classes
- fields, when the context is a class
- methods, when the context is a class or the parameter list of a method

Parameters with classes that are neither contained in the configured session class path nor in the configured JDK are marked as `[unresolved]` and are changed to the generic `java.lang.Object` type. To be able to call methods on such parameters and get code completion for them, add the missing JAR files to the class path in the application settings.

Problem analysis

The code that you enter is analyzed on the fly and checked for errors and warning conditions. Errors are shown as red underlines in the editor and red stripes in the right gutter. Warnings such as an unused variable declaration are shown as a yellow backgrounds in the editor and yellow stripes in the gutter. Hovering the mouse over an error or warning in the editor as well as hovering the mouse over a stripe in the gutter area displays the error or warning message.

The status indicator at the top of the right gutter is green if there are no warnings or errors in the code, yellow if there are warnings and red if errors have been found. You can configure the threshold for problem analysis in the editor settings.



If the gutter icon in the top right corner of the dialog is green, your script is going to compile unless you have disabled error analysis in the editor settings. In some situations, you might want to try the actual compilation. Choosing *Code->Test Compile* from the menu will compile the script and display any errors in a separate dialog. Saving your script with the *OK* button will not test the syntactic correctness of the script unless the script is used right away.

Key bindings

Pressing `SHIFT-F1` opens the browser at the Javadoc page that describes the element at the cursor position. Javadoc for the Java runtime library can only be displayed if a JDK with a valid Javadoc location is configured for the code editor in the general settings.

All key bindings in the Java code editor are configurable. Choose *Settings->Key Map* from the window menu to display the key map editor. Key bindings are saved in the file `$HOME/.jprofiler12/editor_keymap.xml`. This file only exists if the default key map has been copied. When migrating a JProfiler installation to a different computer, you can copy this file to preserve your key bindings.

F Command Line Reference

F.1 Command Line Executables For Profiling

JProfiler includes a number of command line tools for setting up the profiling agent and controlling profiling actions from the command line.

Loading the profiling agent into a running JVM

With the command line utility `bin/jpenable`, you can load the profiling agent into any running JVM with a version of 6 or higher. With command line arguments you can automate the process so that it requires no user input. The supported arguments are:

```
Usage: jpenable [options]

jpenable starts the profiling agent in a selected local JVM, so you can connect
to it from a different computer. If the JProfiler GUI is running locally, you
can attach directly from the JProfiler GUI instead of running this executable.

* if no argument is given, jpenable attempts to discover local JVMs that
  are not being profiled yet and asks for all required input on the command
  line.
* with the following arguments you can partially or completely supply all
  user input on the command line:

  -d  --pid=<PID>     The PID of the JVM that should be profiled
  -n  --noinput       Do not ask for user input under any circumstances
  -h  --help          Show this help
      --options=<OPT> Debugging options passed to the agent

  GUI mode: (default)
  -g  --gui           The JProfiler GUI will be used to attach to the JVM
  -p  --port=<nnnnn>  The port on which the profiling agent should listen for
                      a connection from the JProfiler GUI

  Offline mode:
  -o  --offline       The JVM will be profiled in offline mode
  -c  --config=<PATH> Path to the config file with the profiling settings
  -i  --id=<ID>       ID of the session in the config file. Not required, if
                      the config file holds only a single session.

Note that the JVM has to be running as the same user as jpenable, otherwise
JProfiler cannot connect to it.
An exception are Windows services running under the local system account if you
list them interactively with jpenable.
```

Saving HPROF snapshots

If you just need a heap snapshot, consider using the `bin/jpdump` command line tool that saves an HPROF snapshot [p. 185] without loading the profiling agent into the VM:

```
Usage: jpdump [options]

jpdump saves an HPROF heap dump from a locally running JVM to a file.
The HPROF file can then be opened in the JProfiler GUI.

* if no argument is given, jpdump lists all locally running JVMs.
* with the following arguments you can partially or completely supply all
  user input on the command line:

  -p  --pid=<PID>    The PID of the JVM whose heap should be dumped
                     With a specified PID, no further questions will by asked.
  -a  --all          Save all objects. If not specified, only live objects are
                     dumped
  -f  --file=<PATH>  Path to the dump file. If not specified, the dump file
                     <VM name>.hprof is written in the current directory.
                     If the file already exists, a number is appended.
  -h  --help         Show this help

Note that the JVM has to be running as the same user as jpdump, otherwise
JProfiler cannot connect to it.
An exception are Windows services running under the local system account if you
list them interactively with jpdump.
```

This has a lower overhead than loading the profiling agent and saving a JProfiler heap snapshot. Also, because the profiling agent can never be unloaded, this method is suitable for JVMs running in production.

Controlling the profiling agent

When you start the `bin/jpcontroller` executable without arguments, it attempts to connect to a profiled JVM on the local machine. If multiple profiled JVMs were discovered, you can select one from a list. Because the discovery mechanism uses the attach API of the Oracle JVM, this only works for Oracle JVMs starting with Java 6.

`jpcontroller` can only connect to JVMs where the profiling settings have been set, so it does not work if the JVM was started with the "nowait" option for the `-agentpath` VM parameter. That option is set when choosing the *Startup immediately, connect later with the JProfiler GUI* radio button on the "Startup mode" screen of an integration wizard and no JProfiler GUI has yet connected to the agent. Using `jpenable` without the `--offline` argument also requires a connection from the JProfiler GUI before `jpcontroller` can connect to the profiled process.

If you want to connect to a process on a remote computer, or if the JVM is not a HotSpot JVM with a version of 6 or higher, you have to pass the VM parameter `-Djprofiler.jmxServerPort=[port]` to the profiled JVM. An MBean server will be published on that port and you can specify the chosen port as an argument to `jpcontroller`. With the additional VM parameter `-Djprofiler.jmxPasswordFile=[file]` you can specify a properties file with key-value pairs of the form `user=password` to set up authentication. Note that these VM parameters are overridden by the `com.sun.management.jmxremote.port` VM parameter.

With the explicit setup of the JMX server, you can use the command line controller to connect to a remote server by invoking `jpcontroller host:port`. If the remote computer is only reachable via an IP address, you have to add `-Djava.rmi.server.hostname=[IP address]` as a VM parameter to the remote VM.

By default, `jpcontroller` is an interactive command line utility, but you can also automate profiling sessions with it without the need for manual input. An automated invocation would pass `[pid | host:port]` to select a profiled JVM as well as the `--non-interactive` argument. In addition, a list of commands is read, either from stdin, or from a command file that is specified

with the `--command-file` argument. Each command starts on a new line, lines that are blank or start with a "#" comment character are ignored.

Commands for this non-interactive mode are the same as the method names in the JProfiler MBean [(1)]. They require the same number of parameters, separated by spaces. String must be surrounded by double quotes if they contain spaces. In addition, a `sleep <seconds>` command is provided that pauses for a number of seconds. This allows you to start recording, wait for some time and then save a snapshot to disk.

The supported arguments of jpcontroller are shown below:

```
Usage: jpcontroller [options] [host:port | pid]

* if no argument is given, jpcontroller attempts to discover local JVMs that
   are being profiled
* if a single number is specified, jpcontroller attempts to connect to the JVM
   with process ID [pid]. If that JVM is not profiled, jpcontroller cannot
   connect. In that case, use the jpenable utility first.
* otherwise, jpcontroller connects to "host:port", where port is the value
   that has been specified in the VM parameter -Djprofiler.jmxServerPort=[port]
   for the profiled JVM.

The following options are available:
  -n --non-interactive      Run an automated session where a list of commands
                            is read from stdin.
  -f --command-file=<PATH>  Read commands from a file instead of stdin. Only
                            applicable together with --non-interactive.

Syntax for non-interactive commands:
  See the javadoc for RemoteControllerMBean (https://bit.ly/2DimDN5) for a
  list of operations. Parameters are separated by spaces and must be quoted if
  they contain spaces. For example:

    addBookmark "Hello world"
    startCPURecording true
    sleep 10
    stopCPURecording
    saveSnapshot /path/to/snapshot.jps

  The sleep <seconds> command pauses for the specified number of seconds.
  Empty lines and lines starting with # are ignored.
```

[(1)] https://www.ej-technologies.com/resources/jprofiler/help/api/javadoc/com/jprofiler/api/agent/mbean/RemoteControllerMBean.html

F.2 Command Line Executables For Working With Snapshots

When using offline profiling [p. 113] to save snapshots programmatically, it may also be necessary to programmatically extract data or reports from those snapshots. JProfiler offers two separate command line executables, one for exporting views from a snapshot and one for comparing snapshots.

Exporting views from a snapshot

The executable `bin/jpexport` exports view data to various formats. If you execute it with the `-help` option, you will get information on the available view names and view options. For reasons of conciseness, duplicate help texts in the output below have been omitted.

```
Usage: jpexport "snapshot file" [global options]
               "view name" [options] "output file"
               "view name" [options] "output file" ...

where "snapshot file" is a snapshot file with one of the extensions:
           .jps, .hprof, .hpz, .phd
      "view name" is one of the view names listed below
      [options] is a list of options in the format -option=value
      "output file" is the output file for the export

Global options:
    -obfuscator=none|proguard|yguard
        Deobfuscate for the selected obfuscator. Defaults to "none", for other
        values the mappingFile option has to be specified.
    -mappingfile=<file>
        The mapping file for the selected obfuscator.
    -outputdir=<output directory>
        Base directory to be used when the output file for a view is a
        relative file.
    -ignoreerrors=true|false
        Ignore errors that occur when options for a view cannot be set and
        continue with the next view. The default value is "false", i.e. the
        export is terminated, when the first error occurs.
    -csvseparator=<separator character>
        The field separator character for the CSV exports. Defaults to ','.

Available view names and options:
* TelemetryHeap, TelemetryObjects, TelemetryThroughput, TelemetryGC,
  TelemetryClasses, TelemetryThreads, TelemetryCPU
  options:
    -format=html|csv
        Determines the output format of the exported file. If not present, the
        export format will be determined from the extension of the output
        file.
    -minwidth=<number of pixels>
        Minimum width of the graph in pixels. The default value is 800.
    -minheight=<number of pixels>
        Minimum height of the graph in pixels. The default value is 600.

* Bookmarks, ThreadMonitor, CurrentMonitorUsage, MonitorUsageHistory
  options:
    -format=html|csv

* AllObjects
  options:
    -format=html|csv
    -viewfilters=<comma-separated list>
        Sets view filters for the export. If you set view filters, only the
        specified packages and their sub-packages will be displayed by the
```

```
        exported view.
    -viewfiltermode=startswith|endswith|contains|equals
        Sets the view filter mode. The default value is contains.
    -viewfilteroptions=casesensitive
        Boolean options for the view filter. By default, no options are set.
    -aggregation=class|package|component
        Selects the aggregation level for the export. The default value is
        classes.
    -expandpackages=true|false
        Expand package nodes in the package aggregation level to show
        contained classes. The default value is "false". Has no effect for
        other aggregation levels and with csv output format.

* RecordedObjects
  like AllObjects, but with additional options:
    -liveness=live|gc|all
        Selects the liveness mode for the export, i.e. whether to display live
        objects, garbage collected objects or both. The default value is live
        objects.

* AllocationTree
  options:
    -format=html|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
        Selects the aggregation level for the export. The default value is
        methods.
    -viewmode=tree|treemap
        Selects the view mode for the export. The default value is "tree".
    -width=<number of pixels>
        Minimum width of the tree map in pixels. The default value is 800.
        Only relevant if "viewmode" is set to "tree".
    -height=<number of pixels>
        Minimum height of the tree map in pixels. The default value is 600.
        Only relevant if "viewmode" is set to "tree".
    -class=<fully qualified class name>
        Specifies the class for which the allocation data should be
        calculated. If empty, allocations of all classes will be shown. Cannot
        be used together with the package option.
    -package=<fully qualified package name>
        Specifies the package for which the allocation data should be
        calculated. If empty, allocations of all packages will be shown.
        Cannot be used together with the class option.
    -liveness=live|gc|all

* AllocationHotSpots
  options:
    -format=html|csv|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -class=<fully qualified class name>
    -package=<fully qualified package name>
    -liveness=live|gc|all
    -unprofiledclasses=separately|addtocalling
        Selects if unprofiled classes should be shown separately or be added
        to the calling class. The default value is to show unprofiled classes
        separately.
    -valuesummation=self|total
        Determines how the times for hot spots are calculated. Defaults to
        "self".
```

```
    -expandbacktraces=true|false
        Expand backtraces in HTML or XML format. The default value is "false".

* ClassTracker
  like TelemetryHeap, but with additional options:
    -class
        The tracked class. If missing, the first tracked class is exported.

* CallTree
  options:
    -format=html|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -viewmode=tree|treemap
    -width=<number of pixels>
    -height=<number of pixels>
    -threadgroup=<name of thread group>
        Selects the thread group for the export. If you specify thread as well
        , the thread will only be searched in this thread group, otherwise the
        entire thread group will be shown.
    -thread=<name of thread>
        Selects the thread for the export. By default, the call tree is merged
        for all threads.
    -threadstatus=all|running|waiting|blocking|netio
        Selects the thread status for the export. The default value is
        "running".

* HotSpots
  options:
    -format=html|csv|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<name of thread group>
    -thread=<name of thread>
    -threadstatus=all|running|waiting|blocking|netio
    -expandbacktraces=true|false
    -unprofiledclasses=separately|addtocalling
    -valuesummation=self|total

* MethodStatistics
  options:
    -format=html|csv
    -threadstatus=all|running|waiting|blocking|netio
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive

* Complexity
  options:
    -format=html|csv|properties
    -fit=best|constant|linear|quadratic|cubic|exponential|logarithmic|n_log_n
        The fit that should be exported. The default value is "best". No curve
        fitting data is exported to CSV.
    -method=<method name>
        The method name for which the complexity graph should be exported. If
        not given, the first method will be exported. Otherwise, the first
        method name that starts with the given text will be exported.
    -width=<number of pixels>
    -height=<number of pixels>
```

```
* ThreadHistory
  like TelemetryHeap, but with changed options:
    -format=html

* MonitorUsageStatistics
  options:
    -format=html|csv
    -type=monitors|threads|classes
        Selects the entity for which the monitor statistics should be
        calculated. The default value is "monitors".

* ProbeTimeLine
  like ThreadHistory, but with additional options:
    -probeid=<id>
        The internal ID of the probe that should be exported. Run "jpexport
        --listProbes" to list all available built-in probes and for an
        explanation of custom probe names.

* ProbeControlObjects
  options:
    -probeid=<id>
    -format=html|csv

* ProbeCallTree
  options:
    -probeid=<id>
    -format=html|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -aggregation=method|class|package|component
    -viewmode=tree|treemap
    -width=<number of pixels>
    -height=<number of pixels>
    -threadgroup=<name of thread group>
    -thread=<name of thread>
    -threadstatus=all|running|waiting|blocking|netio
        Selects the thread status for the export. The default value is "all".

* ProbeHotSpots
  options:
    -probeid=<id>
    -format=html|csv|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -aggregation=method|class|package|component
    -threadgroup=<name of thread group>
    -thread=<name of thread>
    -threadstatus=all|running|waiting|blocking|netio
    -expandbacktraces=true|false

* ProbeTelemetry
  like TelemetryHeap, but with additional options:
    -probeid=<id>
    -telemetrygroup
        Sets the one-based index of the telemetry group that should be
        exported. This refers to the entries that you see in the drop-down
        list above the probe telemetry view. The default value is "1".

* ProbeEvents
  options:
    -probeid=<id>
    -format=html|csv|xml
```

```
* ProbeTracker
  like TelemetryHeap, but with additional options:
    -probeid=<id>
    -index=<number>
        Sets the zero-based index of the drop-down list that contains the
        tracked elements. The default value is 0.
```

Some examples for using the export executable are:

```
jpexport test.jps TelemetryHeap heap.html

jpexport test.jps RecordedObjects -aggregation=package -expandpackages=true objects.html

jpexport test.jps -ignoreerrors=true -outputdir=/tmp/export
         RecordedObjects objects.csv
         AllocationTree -class=java.lang.String allocations.xml
```

Comparing snapshots

The executable `bin/jpcompare` compares different snapshots [p. 118] and exports them to HTML or a machine-readable format. Its `-help` output is reproduced below, again without any duplicate explanations.

```
Usage: jpcompare "snapshot file"[,"snapshot file",...] [global options]
              "comparison name" [options] "output file"
              "comparison name" [options] "output file" ...

where "snapshot file" is a snapshot file with one of the extensions:
           .jps, .hprof, .hpz, .phd
      "comparison name" is one of the comparison names listed below
      [options] is a list of options in the format -option=value
      "output file" is the output file for the export

Global options:
    -outputdir=<output directory>
        Base directory to be used when the output file for a comparison is a
        relative file.
    -ignoreerrors=true|false
        Ignore errors that occur when options for a comparison cannot be set
        and continue with the next comparison. The default value is "false",
        i.e. the export is terminated, when the first error occurs.
    -csvseparator=<separator character>
        The field separator character for the CSV exports. Defaults to ','.
    -sortbytime=false|true
        Sort the specified snapshot files by modification time. The default
        value is false.
    -listfile=<filename>
        Read a file that contains the paths of the snapshot files, one
        snapshot file per line.

Available comparison names and options:
* Objects
  options:
    -format=html|csv
        Determines the output format of the exported file. If not present, the
        export format will be determined from the extension of the output
        file.
    -viewfilters=<comma-separated list>
        Sets view filters for the export. If you set view filters, only the
```

```
        specified packages and their sub-packages will be displayed by the
        exported view.
    -viewfiltermode=startswith|endswith|contains|equals
        Sets the view filter mode. The default value is contains.
    -viewfilteroptions=casesensitive
        Boolean options for the view filter. By default, no options are set.
    -objects=all|recorded|heapwalker
        Compare all objects (JVMTI only) or recorded objects, or objects in
        the heap walker. The default is all objects for .jps files and
        heapwalker for HPROF files.
    -aggregation=class|package|component
        Selects the aggregation level for the export. The default value is
        classes.
    -liveness=live|gc|all
        Selects the liveness mode for the export, i.e. whether to display live
        objects, garbage collected objects or both. The default value is live
        objects.

* AllocationHotSpots
  options:
    -format=html|csv
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -classselection
        Calculate the comparison for a specific class or package. Specify a
        package with a wildcard, like 'java.awt.*'.
    -aggregation=method|class|package|component
        Selects the aggregation level for the export. The default value is
        methods.
    -liveness=live|gc|all
    -unprofiledclasses=separately|addtocalling
        Selects if unprofiled classes should be shown separately or be added
        to the calling class. The default value is to show unprofiled classes
        separately.
    -valuesummation=self|total
        Determines how the times for hot spots are calculated. Defaults to
        "self".

* AllocationTree
  options:
    -format=html|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -classselection
    -aggregation=method|class|package|component
    -liveness=live|gc|all

* HotSpots
  options:
    -format=html|csv
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -firstthreadselection
        Calculate the comparison for a specific thread or thread group.
        Specify thread groups like 'group.*' and threads in specific thread
        groups like 'group.thread'. Escape dots in thread names with
        backslashes.
    -secondthreadselection
        Calculate the comparison for a specific thread or thread group. Only
        available if 'firstthreadselection' is set. If empty, the same value
        as for 'firstthreadselection' will be used. Specify thread groups like
```

```
        'group.*' and threads in specific thread groups like 'group.thread'.
        Escape dots in thread names with backslashes.
    -threadstatus=all|running|waiting|blocking|netio
        Selects the thread status for the export. The default value is
        "running".
    -aggregation=method|class|package|component
    -differencecalculation=total|average
        Selects the difference calculation method for call times. The default
        value is total times.
    -unprofiledclasses=separately|addtocalling
    -valuesummation=self|total

* CallTree
  options:
    -format=html|xml
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals
    -viewfilteroptions=casesensitive
    -firstthreadselection
    -secondthreadselection
    -threadstatus=all|running|waiting|blocking|netio
    -aggregation=method|class|package|component
    -differencecalculation=total|average

* TelemetryHeap
  options:
    -format=html|csv
    -minwidth=<number of pixels>
        Minimum width of the graph in pixels. The default value is 800.
    -minheight=<number of pixels>
        Minimum height of the graph in pixels. The default value is 600.
    -valuetype=current|maximum|bookmark
        Type of the value that is calculated for each snapshot. Default is the
        current value.
    -bookmarkname
        If valuetype is set to 'bookmark', the name of the bookmark for which
        the value should be calculated.
    -measurements=maximum,free,used
        Measurements that are shown in the comparison graph. Concatenate
        multiple values with commas. The default value is 'used'.
    -memorytype=heap|nonheap
        Type of the memory that should be analyzed. Default is 'heap'.
    -memorypool
        If a special memory pool should be analyzed, its name can be specified
        with this parameter. The default is empty, i.e. no special memory
        pool.

* TelemetryObjects
  options:
    -format=html|csv
    -minwidth=<number of pixels>
    -minheight=<number of pixels>
    -valuetype=current|maximum|bookmark
    -bookmarkname
    -measurements=total,nonarrays,arrays
        Measurements that are shown in the comparison graph. Concatenate
        multiple values with commas. The default value is 'total'.

* TelemetryClasses
  like TelemetryObjects, but with changed options:
    -measurements=total,filtered,unfiltered

* TelemetryThreads
  like TelemetryObjects, but with changed options:
```

```
    -measurements=total,runnable,blocked,netio,waiting

* ProbeHotSpots
  options:
    -format=html|csv
    -viewfilters=<comma-separated list>
    -viewfiltermode=startswith|endswith|contains|equals|wildcard|regex
    -viewfilteroptions=exclude,casesensitive
    -firstthreadselection
    -secondthreadselection
    -threadstatus=all|running|waiting|blocking|netio
    -aggregation=method|class|package|component
    -differencecalculation=total|average
    -probeid=<id>
        The internal ID of the probe that should be exported. Run "jpexport
        --listProbes" to list all available built-in probes and for an
        explanation of custom probe names.

* ProbeCallTree
  like ProbeHotSpots, but with changed options:
    -format=html|xml

* ProbeTelemetry
  like TelemetryObjects, but with additional or changed options:
    -measurements
        The one-based indices of the measurements in the telemetry group that
        are shown in the comparison graph. Concatenate multiple values with
        commas, like "1,2". The default value is to show all measurements.
    -probeid=<id>
    -telemetrygroup
        Sets the one-based index of the telemetry group that should be
        exported. This refers to the entries that you see in the drop-down
        list above the probe telemetry view. The default value is "1".
```

Automatic output formats

Most views and comparisons support multiple output formats. By default, the output format is deduced from the extension of the output file:

- **.html**

  The view or comparison is exported as an HTML file. A directory named `jprofiler_images` will be created that contains images used in the HTML page.

- **.csv**

  The data is exported as CSV data where the first line contains the column names.

  When using Microsoft Excel, CSV is not a stable format. Microsoft Excel on Windows takes the separator character from the regional settings. JProfiler uses a semicolon as the separator in locales that use a comma as a decimal separator and a comma in locales that use a dot as a decimal separator. If you need to override the CSV separator character you can do so by setting the global `csvseparator` option.

- **.xml**

  The data is exported as XML. The data format is self-descriptive.

If you would like to use different extensions, you can use the `format` option to override the choice of the output format.

Analyzing snapshots

If the generated snapshots have heap dumps in them, you can use the `bin/jpanalyze` executable to prepare the heap dump analysis in advance [p. 75]. Opening the snapshot in the JProfiler GUI will then be very fast. The usage information of the tool is shown below:

```
Usage: jpanalyze [options] "snapshot file" ["snapshot file" ...]

where "snapshot file" is a snapshot file with one of the extensions:
           .jps, .hprof, .hpz, .phd
      [options] is a list of options in the format -option=value

Options:
    -obfuscator=none|proguard|yguard
        Deobfuscate for the selected obfuscator. Defaults to "none", for other
        values the mappingFile option has to be specified.
    -mappingfile=<file>
        The mapping file for the selected obfuscator.
    -removeunreferenced=true|false
        If unreferenced or weakly referenced objects should be removed.
    -retained=true|false
        Calculate retained sizes (biggest objects). removeunreferenced will be
        set to true.
    -retainsoft=true|false
        If unreferenced objects are removed, specifies if soft references
        should be retained.
    -retainweak=true|false
        If unreferenced objects are removed, specifies if weak references
        should be retained.
    -retainphantom=true|false
        If unreferenced objects are removed, specifies if phantom references
        should be retained.
    -retainfinalizer=true|false
        If unreferenced objects are removed, specifies if finalizer references
        should be retained.
```

The `removeUnreferenced`, the `retained` and all the `retain*` command line options correspond to the options in the heap walker options dialog.

F.3 Gradle Tasks

JProfiler supports profiling from Gradle with special tasks. In addition. JProfiler offers a number of command line executables for working with snapshots [p. 206] that have corresponding Gradle tasks.

Using Gradle tasks

To make the JProfiler Gradle tasks available in a Gradle build file, you can use the `plugins` block

```
plugins {
    id 'com.jprofiler' version 'X.Y.Z'
}
```

If you do not want to use the Gradle plugin repository for this purpose, the Gradle plugin is distributed in the file `bin/gradle.jar`.

Next, you have to tell the JProfiler Gradle plugin where JProfiler is installed.

```
jprofiler {
    installDir = file('/path/to/jprofiler/home')
}
```

Profiling from Gradle

With tasks of type `com.jprofiler.gradle.JavaProfile` you can profile any Java process. This class extends Gradle's built-in `JavaExec`, so you can use the same arguments for configuring the process. For profiling tests, use tasks of type `com.jprofiler.gradle.TestProfile` that extend the Gradle `Test` task.

Without any further configuration, both tasks start an interactive profiling session where the profiling agent waits on the default port 8849 for a connection from the JProfiler GUI. For offline profiling, you have to add a couple of attributes that are shown in the table below.

Attribute	Description	Required
offline	Whether the profiling run should be in offline mode.	No, `offline` and `nowait` cannot both be `true`.
nowait	Whether profiling should start immediately or whether the profiled JVM should wait for a connection from the JProfiler GUI.	
sessionId	Defines the session ID from which profiling settings should be taken. Has no effect if neither `nowait` nor offline are set because in that case the profiling session is selected in the GUI.	Required if • `offline` is set • `nowait` is set for a 1.5 JVM
configFile	Defines the config file from which the profiling settings should be read.	No
port	Defines the port number on which the profiling agent should listen for a connection from the JProfiler GUI. This must be the same as the port configured in the remote	No

Attribute	Description	Required
	session configuration. If not set or zero, the default port (8849) will be used. Has no effect if `offline` is set because in that case there is no connection from the GUI.	
debugOptions	If you want to pass any additional library parameters for tuning or debugging purposes, you can do that with this attribute.	No

An example for profiling a Java class with a main method that is compiled by the containing project is given below:

```
task run(type: com.jprofiler.gradle.JavaProfile) {
    main = 'com.mycorp.MyMainClass'
    classpath sourceSets.main.runtimeClasspath
    offline = true
    sessionId = 80
    configFile = file('path/to/jprofiler_config.xml')
}
```

You can see a runnable example of this task in the `api/samples/offline` sample project. Unlike the standard `JavaExec` task, the `JavaProfile` task can also be started in the background by calling `createProcess()` on it. See the `api/samples/mbean` sample project for a demonstration of this feature.

If you need the VM parameter that is required for profiling, the `com.jprofiler.gradle.SetAgentpathProperty` task will assign it to a property whose name is configured with the `propertyName` attribute. Applying the JProfiler plugin automatically adds a task of this type named `setAgentPathProperty` to your project. For getting the VM parameter that would be used in the previous example, you can simply add

```
setAgentPathProperty {
    propertyName = 'profilingVmParameter'
    offline = true
    sessionId = 80
    configFile = file('path/to/jprofiler_config.xml')
}
```

to your project and add a dependency to `setAgentPathProperty` to some other task. Then you can use the project property `profilingVmParameter` in the execution phase of that task. When assigning the property to other task properties, surround its usage with a `doFirst {...}` code block in order to make sure that you are in the Gradle execution phase and not in the configuration phase.

Exporting data from snapshots

The `com.jprofiler.gradle.Export` task can be used to export views from a saved snapshot and replicates the arguments of the `bin/jpexport` command line tool [p. 206]. It supports the following attributes:

Attribute	Description	Required
snapshotFile	The path to the snapshot file. This must be a file with a `.jps` extension.	Yes
ignoreErrors	Ignore errors that occur when options for a view cannot be set and continue with the next view. The default value is `false`, meaning that the export is terminated when the first error occurs.	No
csvSeparator	The field separator character for the CSV exports. Defaults to ",".	No
obfuscator	Deobfuscate class and method names for the selected obfuscator. Defaults to "none", for other values the `mappingFile` option has to be specified. One of `none`, `proguard` or `yguard`.	No
mappingFile	The mapping file for the selected obfuscator. May only be set if the `obfuscator` attribute is specified.	Only if `obfuscator` is specified

On the export task, you call the `views` method and pass a closure to it in which you call `view(name, file[, options])` one or multiple times. Each call to `view` produces one output file. The `name` argument is the view name. For a list of available view names, please see the help page on the `jpexport` command line executable [p. 206]. The argument `file` is the output file, either an absolute file or a file relative to the project. Finally, the optional `options` argument is a map with the export options for the selected view.

An example for using the export task is:

```
task export(type: com.jprofiler.gradle.Export) {
    snapshotFile = file('snapshot.jps')
    views {
        view('CallTree', 'callTree.html')
        view('HotSpots', 'hotSpots.html',
            [threadStatus: 'all', expandBacktraces: 'true'])
    }
}
```

Comparing snapshots

Like the `bin/jpcompare` command line tool [p. 206], the `com.jprofiler.gradle.Compare` task can compare two or more snapshots. It attributes are:

Attribute	Description	Required
snapshotFiles	The snapshot files that should be compared. You can pass any `Iterable` containing objects that gradle resolves to file collections.	Yes
sortByTime	If set to `true` all supplied snapshots files are sorted by their file modification time, otherwise they are compared in the order they were specified in the `snapshotFiles` attribute.	No

Attribute	Description	Required
ignoreErrors	Ignore errors that occur when options for a comparison cannot be set and continue with the next comparison. The default value is `false`, meaning the export is terminated when the first error occurs.	No

Just like exported views are defined for the `Export` task, the `Compare` task has a `comparisons` method where nested calls to `comparison(name, file[, options])` define the comparisons that should be performed. The list of available comparison names is available on the help page of the `jpcompare` command line executable [p. 206].

An example for using the compare task is:

```
task compare(type: com.jprofiler.gradle.Compare) {
    snapshotFiles = files('snapshot1.jps', 'snapshot2.jps')
    comparisons {
        comparison('CallTree', 'callTree.html')
        comparison('HotSpots', 'hotSpots.csv',
            [valueSummation: 'total', format: 'csv'])
    }
}
```

or, if you want to create a telemetry comparison for multiple snapshots:

```
task compare(type: com.jprofiler.gradle.Compare) {
    snapshotFiles = fileTree(dir: 'snapshots', include: '*.jps')
    sortByTime = true
    comparisons {
        comparison('TelemetryHeap', 'heap.html', [valueType: 'maximum'])
        comparison('ProbeTelemetry', 'jdbc.html', [probeId: 'JdbcProbe'])
    }
}
```

Analyzing heap snapshots

The gradle task `com.jprofiler.gradle.Analyze` has the same functionality as the `bin/jpanalyze` command line tool [p. 206].

The task has a `snapshotFiles` attribute like the `Compare` task to specify the processed snapshots and `obfuscator` and `mappingfile` attributes like the `Export` task for deobfuscation. The attributes `removeUnreferenced`, `retainSoft`, `retainWeak`, `retainPhantom`, `retainFinalizer` and `retained` correspond the arguments of the command line tool.

An example for using the `Analyze` task is given below:

```
task analyze(type: com.jprofiler.gradle.Analyze) {
    snapshotFiles = fileTree(dir: 'snapshots', include: '*.jps')
    retainWeak = true
    obfuscator = 'proguard'
    mappingFile = file('obfuscation.txt')
}
```

F.4 Ant Tasks

The Ant[1] tasks provided by JProfiler are very similar to the Gradle tasks. This chapter highlights the differences to the Gradle tasks and shows examples for each Ant task.

All Ant tasks are contained in the archive `bin/ant.jar`. In order to make a task available to Ant, you must first insert a `taskdef` element that tells Ant where to find the task definition. All examples below include that taskdef. It must occur only once per build file and can appear anywhere on the level below the project element.

It is not possible to copy the `ant.jar` archive to the `lib` folder of your Ant distribution, you have to reference a full installation of JProfiler in the task definition.

Profiling from Ant

The `com.jprofiler.ant.ProfileTask` is derived from the built-in `Java` task and supports all its attributes and nested elements. The additional attributes are the same as for the `ProfileJava` Gradle task [p. 215]. Ant attributes are case-insensitive and usually written in lower case.

```
<taskdef name="profile"
         classname="com.jprofiler.ant.ProfileTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="profile">
  <profile classname="MyMainClass" offline="true" sessionid="80">
    <classpath>
      <fileset dir="lib" includes="*.jar" />
    </classpath>
  </profile>
</target>
```

Exporting data from snapshots

With the `com.jprofiler.ant.ExportTask` you can export view from snapshots, just like with the `Export` Gradle task [p. 215]. Views are specified differently than in the Gradle task: they are nested directly below the task element and options are specified with nested `option` elements.

```
<taskdef name="export"
         classname="com.jprofiler.ant.ExportTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="export">
  <export snapshotfile="snapshots/test.jps">
    <view name="CallTree" file="calltree.html"/>
    <view name="HotSpots" file="hotspots.html">
      <option name="expandbacktraces" value="true"/>
      <option name="aggregation" value="class"/>
    </view>
  </export>
</target>
```

Comparing snapshots

The `com.jprofiler.ant.CompareTask` corresponds to the `Compare` Gradle task and performs comparisons between two ore more snapshots. Like for the `com.jprofiler.ant.ExportTask`, comparisons are directly nested below the element and options are nested for each `comparison` element. The snapshot files are specified with a nested file set.

[1] http://ant.apache.org

```
<taskdef name="compare"
         classname="com.jprofiler.ant.CompareTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="compare">
  <compare sortbytime="true">
    <fileset dir="snapshots">
      <include name="*.jps" />
    </fileset>
    <comparison name="TelemetryHeap" file="heap.html"/>
    <comparison name="TelemetryThreads" file="threads.html">
      <option name="measurements" value="inactive,active"/>
      <option name="valuetype" value="bookmark"/>
      <option name="bookmarkname" value="test"/>
    </comparison>
  </compare>
</target>
```

Analyzing heap snapshots

Like the `Analyze` Gradle task, the equivalent `com.jprofiler.ant.AnalyzeTask` for Ant prepares the heap snapshot analysis in snapshots that have been saved with offline profiling for faster access in the GUI. The snapshots that should be processed are specified with a nested file set.

```
<taskdef name="analyze"
         classname="com.jprofiler.ant.AnalyzeTask"
         classpath="<path to JProfiler installation>/bin/ant.jar"/>

<target name="analyze">
  <analyze>
    <fileset dir="snapshots" includes="*.jps" />
  </analyze>
</target>
```