

The definitive guide to JProfiler

All you need to know as a performance professional

© 2025 ej-technologies GmbH. All rights reserved.

Index

Introduction	4
Architecture	5
Installing	7
Profiling a JVM	11
Recording data	27
Snapshots	40
Telemetries	45
CPU profiling	53
Method call recording	66
Memory profiling	71
The heap walker	80
Thread profiling	
Probes	104
GC analysis	118
MBean browser	124
Offline profiling	129
Comparing snapshots	134
IDE integrations	141
A Custom probes	152
A.1 Probe concepts	
A.2 Script probes	159
A.3 Injected probes	163
A.4 Embedded probes	168
B Call tree features in detail	172
B.1 Auto-tuning for instrumentation	172
B.2 Async and remote request tracking	175
B.3 Viewing parts of the call tree	
B.4 Splitting the call tree	
B.5 Call tree analyses	190
C Advanced CPU analysis views	195
C.1 Outlier detection	

C.2 Complexity analysis	199
C.3 Call tracer	201
C.4 Javascript XHR	203
D Heap walker features in detail	206
D.1 HPROF snapshots	206
D.2 Minimizing overhead	208
D.3 Filters and live interactions	210
D.4 Finding memory leaks	214
E JDK Flight Recorder (JFR)	221
E.1 JFR overview	221
E.2 Recording JFR snapshots	223
E.3 JFR event browser	227
E.4 JFR views	234
F Configuration in detail	
F.1 Trouble shooting connection problems	241
F.2 Scripts	
F.3 Custom help	
F.4 Profiling settings at startup	248
G Command line reference	251
G.1 Executables for profiling	251
G.2 Executables for snapshots	
G.3 Gradle tasks	
G.4 Ant tasks	

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. 11] will get you up and running.

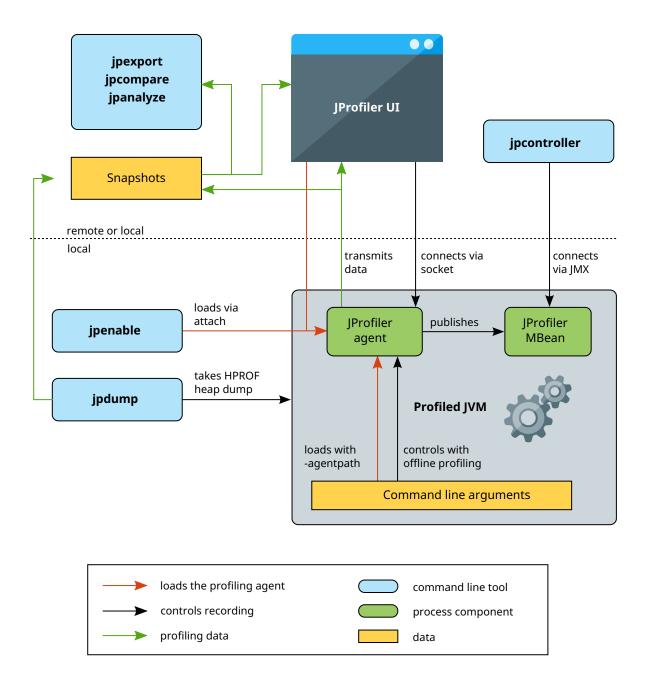
Following that, the discussion of data recording [p. 27] and snapshots [p. 40] take you to a level of understanding where you can explore JProfiler on your own.

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 here [p. 7].

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 <code>-agentpath:<path to native library></code> 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. For secure connections to remote servers, you can configure JProfiler to automatically create SSH tunnels.

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⁽¹⁾. 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://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.

🥑 Setup - JProfiler	_		×
	Welcome to the JProfiler Setup Wizard		
	This will install JProfiler on your computer.		
	A previous installation has been detected. Do you wish to installation?	update t	hat
	Yes, update the existing installation		
	○ No, install into a different directory		
	Click Next to continue, or Cancel to exit Setup.		
	Instal		Cancel

On macOS, JProfiler uses the standard installation procedure for UI applications: a DMG archive 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 the entire 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 website 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 number 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.

Integration Wizard - [Generic a	pplication] on Remote Linux X86/AMD64 X
1. Choose wizard 2. Local or remote	Specify the remote installation directory
3. Profiled JVM	The profiling agent must be available on the remote Linux X86/AMD64 machine.
 4. Startup mode 5. Remote connection 6. Remote installation directory 	Please specify the JProfiler installation directory on the remote machine, for example "/opt/jprofiler15".
 Choose profiling port Perform modifications Finished 	Remote installation directory: /opt/jprofiler15
	If JProfiler is not installed, you can download the profiling agent and extract it on the remote machine in the above directory.
	Direct Download Copy URL To Clipboard
	A web page with agent downloads for all supported platforms is also available.
	Download Overview Copy URL To Clipboard
	← Back Next ► Finish Cancel

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

https://www.ej-technologies.com/jprofiler/agent?version=15.0.2

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

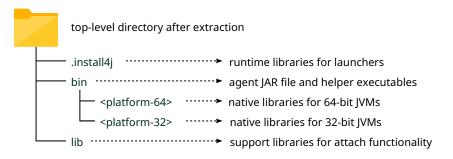
https://download.ej-technologies.com/jprofiler/jprofiler_agent_<platform>_15_0_2.<extension>

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_15_0_2.tar.gz

The agent archive contains the required native agent libraries together with the jpenable, jpdump and jpcontroller executables. The executables in the archive as well as the profiling agent only require Java 8 as a minimum version.

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.



Supported platforms

Because JProfiler utilizes the native profiling interface of the JVM (JVMTI), its profiling agent is a native library.

OS	Architecture	Supported JVMs	Versions
Windows 11/10	x86	Hotspot (OpenJDK)	1.8 - 24
Windows Server 2025/2022/2019/2016	x64/AMD64	IBM/OpenJ9	1.8 - 24
macOS 10.12 - 15	Intel, Apple	Hotspot (OpenJDK)	1.8 - 24
		IBM/OpenJ9	1.8 - 24
Linux	x86	Hotspot (OpenJDK)	1.8 - 24
	x64/AMD64	IBM/OpenJ9	1.8 - 24
Linux	PPC64LE	Hotspot (OpenJDK)	1.8 - 24
		IBM/OpenJ9	1.8 - 24
Linux	ARMv7	Hotspot (OpenJDK)	1.8 - 24
	ARMv8		

JProfiler supports profiling on the following platforms:

The JProfiler GUI frontend needs a Java 21 VM to run. A Java 21 JRE is bundled with JProfiler for that purpose on Windows, macOS and Linux x64. The attach command line tools jpenable, jdump and jpcontroller only require a Java 8 VM.

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 [p. 7]. 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/jprofiler15/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.

Under some circumstances, setting the profiling settings at startup [p. 248] is required and some manual work may be required to achieve this.

By default, the JProfiler agent binds the communication socket to the loopback interface. You can add the option address=[IP address] in order to select a specific interface or address= 0.0.0.0 to bind the communication socket to all available network interfaces. This can be necessary if you want to publish the profiling port from a docker container.

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.

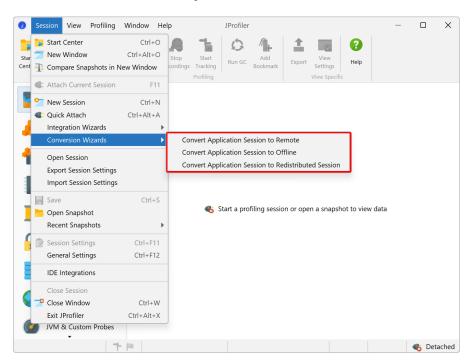
Session Settings		×
Application Settings	Session name: Animated Bezie	r Curve Demo Id: 101 💡
Profiled JVM	Session Type Attach to an alrea	dy running HotSpot/OpenJ9 JVM and profile it
Code Editor	Attach Attach type: •) Select from all local JVMs \bigcirc Attach to remote JVM \bigcirc Kubernete:
Call Tree Recording	Launch a new JVN Launch Launch type:	
Call Tree Filters	Application Settings	
Trigger Settings	Java VM:	21 [C:\Users\ingo\jdks\jbrsdk-21.0.5-wi Configure JREs
Databases	Working directory: VM options:	[startup directory] ····
HTTP, RPC & JEE	Main class or executable JAR:	bezier.BezierAnim
JVM & Custom Probes	Program arguments:	block
Advanced Settings	Java File Path	
	Class path Source path 2 Library path 2	lemo\bezien\classes
General Settings Copy Setti	ngs From	OK Cancel

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. This feature supports OpenWebStart⁽¹⁾, legacy WebStart from pre-Java 9 Oracle JREs is not supported.

Session Settings		
Application Settings	Session name:	Web Start Session Id: 162
	Session Type	
Profiled JVM		Attach to an already running HotSpot/OpenJ9 JVM and profile it
Code Editor		Attach type: Select from all local JVMs Attach to remote JVM Kubernet
Call Tree Recording		Launch a new JVM and profile it
	Launch	Launch type: 🔵 Application 🗿 Web Start
Call Tree Filters	Web Start Settin	gs
Trigger Settings	URL of the JNI	P file: http://www.jgoodies.com/download/jdiskreport2/jdiskreport.jnlp
Databases	WebStart sessi Java File Path —	ons require that <u>OpenWebStart</u> is installed.
HTTP, RPC & JEE		Note: the classpath is used for the bytecode viewer only.
JVM & Custom Probes	 Class pat Source pat 	-
Advanced Settings	0	
General Settings Copy Setti	ngs From	ОК Салсе

⁽¹⁾ https://openwebstart.com/

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. 129] 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. 141] 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

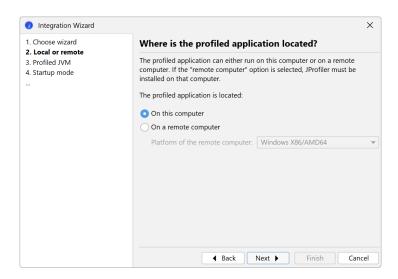
JProfiler's integration wizards handle many well-known third party containers with start scripts or config files that can be modified programmatically to include additional VM parameters. For some products, start scripts can be generated where VM parameters are passed as arguments or via environment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

1. Choose wizard 2. Local or remote 3. Profiled JVM 4. Startup mode	Integration Wizard	×
 3. Profiled JVM 4. Startup mode This wizard integrates your application server or remote application with JProfiler. Choose the appropriate wizard from the list below. If your application server is not listed, choose "[Generic application server]" to get step-by-step instructions for manual integration [Generic application server] [Generic application] install4j/exe4j project jsvc service ColdFusion Glassfish JBoss Jetty Jonas Netbeans RCP application 		Choose integration wizard
 [Generic application] install4j/exe4j project jsvc service ColdFusion Glassfish JBoss Jetty Jonas Netbeans RCP application Resin 	3. Profiled JVM 4. Startup mode	JProfiler. Choose the appropriate wizard from the list below. If your application server is not listed, choose "[Generic application server]" to
Next Finish Cancel		 Generic application] install4j/exe4j project jsvc service ColdFusion Glassfish JBoss Jetty Jonas Netbeans RCP application Resin

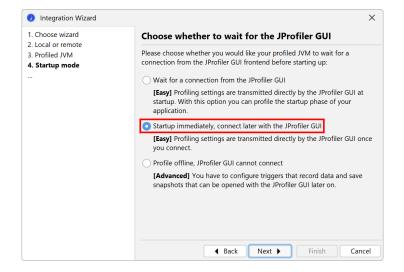
In all cases, you have to locate some specific file from the third-party product, so JProfiler has the necessary context to perform its modifications. Some generic wizards only give you instructions on what you have to do in order to enable profiling.

🥑 Integration Wizard - Tomcat	>	×
1. Choose wizard	Locate start script	
 Local or remote Profiled JVM Startup mode Locate start script Choose profiling port Chock modifications Finished 	Please locate the start script for Tomcat below.	···

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.

1. Choose wizard	Choose how to synchronize profiling settings
 Local or remote Profiled JVM Startup mode Remote connection Remote installation directory Config synchronization Choose profiling port Perform modifications Finished 	The profiling agent can receive its profiling settings when the connection is made from the JProfiler GUI. However, class retransformations can take a lot of time. For fast connections, you can specify the configuration at startup. Apply configuration when connecting with the JProfiler GUI Apply configuration at startup Directory for config file on remote computer: /home/build/config Manual synchronization Copy with SSH to remote directory Copy config file to directory: Execute command: Command:

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.

Integration is completed	
The integration of your profiled JVM has been completed successfully. To profile, you have to manually start your profiled JVM first. When you click on Finish, the remote session can be started immediately. Yes, start the session and wait for the profiled JVM No, I will start the session later Edit Session And Synchronize Config The created session has been named Application server on demo Back Next Finish	el
	 The integration of your profiled JVM has been completed successfully. io profile, you have to manually start your profiled JVM first. When you click on Finish, the remote session can be started immediately. Yes, start the session and wait for the profiled JVM. No, I will start the session later Edit Session And Synchronize Config the created session has been named Application server on demo

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.

Session Settings		×
Application Settings	Session name: Application server on demo	ld: 162 🕜
Profiled JVM	Session Type Attach to an already running HotSpot/OpenJ9 JVM and profile it	
Code Editor	Attach type: Select from all local JVMs S Attach to remote JVM Kuberne	etes
Call Tree Recording	Launch a new JVM and profile it Launch Launch type: Application () Web Start	
Call Tree Filters	Launch Launch type: Application Web Start Profiled JVM Settings	
Trigger Settings	If you have not yet prepared a JVM for profiling, it is recommended to run an integration wizard. It create the remote session for you.	will
Databases	SSH tunnel connection Direct SSH to demo:8849 Edit Use SOCKS proxy	0
HTTP, RPC & JEE	See Sec(3 ploty C:\Users\bob\appserver\startServer.bat	🕄
JVM & Custom Probes	Execute stop command c:\Users\bob\appserver\stopServer.bat Open browser with URL http://localhost:8080	
Advanced Settings	Connection timeout: 60 🗢 seconds Config Synchronization C	Options
	Java File Path	
	Oclass path Source path (2)	+ *
General Settings Copy Setti	ings From OK	Cancel

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.

	<pre>@ubuntu:~\$ jprofiler10/bin/jpintegrate</pre>	
elc	ome to the JProfiler console integration wizard!	
ow	do you want to find your integration wizard?	
ear	ch by keyword [1, Enter], List all wizards [2]	
	se enter a number of keywords separated by spaces (for example: Tomcat phere	5)
	se choose one of the following integration wizards:	
	Websphere 9.x Application Server [1]	
	Websphere 8.x Application Server [2]	
	Websphere 7.0 Application Server [3]	
	Websphere 6.1 Application Server [4]	
BW	WebSphere Community Edition 2.x [5]	

When an error occurs while starting a remote profiling session, see the trouble-shooting guide [p. 241] for a checklist 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.

G	General Settings						×
UI	Session Defaults	Snapshots	IDE Integrations	Updates	External Programs		
IDE	Integration						
То	integrate JProfiler	with an IDE, cl	hoose the target IDI	E and click o	on "Integrate" below.		
Γ	ntelliJ IDEA						-
	Integrate 🕜						_
_							
					0	K Can	cel

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. 141].

Attach mode

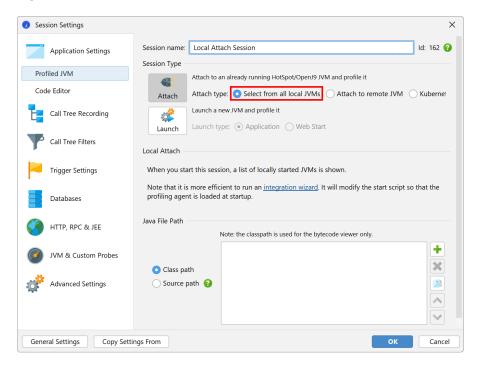
You do not necessarily have to decide beforehand that you intend to profile a JVM. With the attach functionality in JProfiler, you can select a running JVM and load the profiling agent on the fly. While attach mode is convenient, it has a couple of drawbacks that you should be aware of:

- 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 supported for OpenJDK JVMs, Oracle JVMs with version 6 or higher, recent OpenJ9 JVMs (8u281+, 11.0.11+ or Java 17+) or IBM JVMs that are based on such a release. The VM parameters -XX:+PerfDisableSharedMem and -XX:
 +DisableAttachMechanism must not be specified for the JVM.

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, Linux 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.

Ø JProfiler Star	art Center	×
Start Ce	nter	
	• On this computer On another computer On a Kubernetes cluster	
Open	Container: 📗 None, showing top level processes	Select Container
Session	Status: All detected HotSpot/OpenJ9 JVMs 🔻	Show Services
	PID A Process Name	
Quick	10620 org.gradle.launcher.daemon.bootstrap.GradleDaemon 8.13	
Attach	11196 org.gradle.wrapper.GradleWrapperMain screenshotsLightEn	
	14724 C:\Users\ingo\AppData\Local\JetBrains\Toolbox\bin\jre	
New Session Open Snapshots	21516 org.jetbrains.kotlin.daemon.KotlinCompileDaemondaemon-runFilesPath C:\Users\i	ngo\AppData\Local\K
	Legend: Profiling agent loaded JProfiler GUI connected Offline mode JFR ru Q* Filter	nning
	Start Heap Dump Only Start JF	R OClose

On Linux, JProfiler supports switching the user directly in the UI through PolicyKit that is part of most Linux distributions. By clicking *Switch user* in the attach dialog, you can enter a different user name and authenticate with the system password dialog.

Start Ce	enter		
	On this	computer 🔿 On another computer 🔿 On a Kubernetes cluster	
Open Session	User:	💄 Current user	Switch User
Session	Container:	III None, showing top level processes	Select Container
-	Status:	All detected HotSpot/OpenJ9 JVMs 🔻	
Quick Attach	PID 4	Process Name	
Attach	10620	org.gradle.launcher.daemon.bootstrap.GradleDaemon 8.13	
<u></u>	11196	org.gradle.wrapper.GradleWrapperMain screenshotsLightEn	
0	14724	C:\Users\ingo\AppData\Local\JetBrains\Toolbox\bin\jre	
New Session	21516	org.jetbrains.kotlin.daemon.KotlinCompileDaemondaemon-runFilesPath C:\Users\ingo\	AppData\Local\k

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. After that you can either connect with a local session from the JProfiler UI or an SSH connection that directly connects the port given by jpenable.

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 on which 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.

Ø Edit SSH Tunnel - Direct SS	H connection X
1. Tunnel mode	Configure the SSH host
2. Configure SSH host 3. SSH options	JProfiler will tunnel its connection to the profiling agent through the SSH connection configured below.
	User name: build
	Host: demo
	SSH port: 22 Default
	Authentication: O Password
	Private key C:\Users\ingo\.ssh\id_rsa
	\bigcirc Discover running JVMs and attach to selected process ?
	Use netcat instead of SSH port forwarding
	• Manually specify profiling port
	Exit host: 127.0.0.1
	Profiling port: 31775 Default
	← Back Next ► Finish Cancel

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.

Attach To Running JVM X						
Remote user:	Switch User					
Remote conta	iner: None, showing top level processes Select Container					
Status: Not profiled						
PID 🔺	Process Name					
767	/usr/share/jetty9/start.jar jetty.state=/var/lib/jetty9/jetty.state jetty-started.xml					
773	install4j.com.perfino.server.ServerMain_perfino_service start					
918	org.apache.catalina.startup.Bootstrap start					
1517	standalone_demo_service					
Legend:	Profiling agent loaded 🛑 JProfiler GUI connected 📁 Offline mode 💻 JFR running					
Q • Filter						
	Start Heap Dump Only Start JFR Starcel					

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".

I. Tunnel mode	Configure the SSH tunnel
2. Configure SSH tunnel 3. SSH options	SSH tunnel steps:
	SSH to gateway.mycorp.com:22 [private key C:\Users\ingo\.ssh\id_rsa]
	SSH to demo:22 [private key C:\Users\ingo\.ssh\id_rsa]
	User name: build
	Host: demo
	SSH port: 22 Default
	Authentication: O Password
	Private key C:\Users\ingo\.ssh\id_rsa ···
	Discover running JVMs and attach to selected process ??
	Use netcat instead of SSH port forwarding 😮
	Manually specify profiling port
	Exit host: 127.0.0.1
	Profiling port: 8849 Default

For other authentication mechanisms, you can use the OpenSSH tunnel mode. You enter the host name as you would enter it on the command line when using OpenSSH executables. This can also be an alias as configured in the OpenSSH config file. Apart from the host name, port and user name can optionally be specified. On Windows, only the built-in OpenSSH client by Microsoft is supported.

The SSH options text field takes arbitrary additional arguments to the OpenSSH executable that you would specify on the command line. This is especially useful when you are following instructions given by a tutorial, for example on how to tunnel an SSH connection through AWS session manager.

1. Tunnel mode	Configure t	he OpenSSH host	
2. Configure OpenSSH		nel its connection to the profiling agent through the S	5H connection configured
		nstallation and config file will be used to make the con optional and will be taken from the config file if availa	
	Host:	staging	Choose
	User name:		
	SSH port:	22 Default	
	SSH options:		
	 Discover run 	ning JVMs and attach to selected process 🔞	
	Use neto	at instead of SSH port forwarding 💡	
	 Manually sp 	ecify profiling port 😮	
	Exit host:	127.0.0.1	
	Profiling por	t: 8849 Default	

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 local Docker Desktop installation in Windows or macOS by selecting the Docker container in the quick attach dialog. By default, JProfiler detects the path to the docker executable automatically. Alternatively, you can configure it on the "External tools" tab of the general settings dialog.

	 On this 	computer On another computer On a Kubernetes cluster	
Dpen	Container:	None, showing top level processes	Select Contain
ession	Status:	All detected HotSpot/OpenJ9 JVMs 🔻	💛 Show Service
¢.	PID	Process Name	
uick	10620	org.gradle.launcher.daemon.bootstrap.GradleDaemon 8.13	
ttach	11196	org.gradle.wrapper.GradleWrapperMain screenshotsLightEn	
courti	14724	C:\Users\ingo\AppData\Local\JetBrains\Toolbox\bin\jre	
New ession			

After you select the container, all JVMs that run inside the Docker container will be shown. When you select a JVM, JProfiler will 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.

For remote Docker installations, you can use the SSH remote attach functionality and then select 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.

🧷 Attach T	o Running JVM X
Remote user:	2 root (via sudo) Switch User
Remote conta	iner: III None, showing top level processes Select Container
Status:	Not profiled 🔹
PID 🔺	Process Name
767	/usr/share/jetty9/start.jar jetty.state=/var/lib/jetty9/jetty.state jetty-started.xml
773	install4j.com.perfino.server.ServerMain_perfino_service start
918	org.apache.catalina.startup.Bootstrap start
1517	standalone_demo_service
Legend:	Profiling agent loaded 📁 JProfiler GUI connected 💳 Offline mode 💻 JFR running
Legena.	
Q• Filter	
	Start Heap Dump Only Start JFR Start JFR

With the *Select container* hyperlink in the remote attach dialog you can choose a running Docker container and show all JVMs that are running in it.

Attaching to JVMs running on Kubernetes clusters

To profile a JVM that is running on a Kubernetes cluster, JProfiler uses the kubectl command line tool, both for discovering pods and containers, as well as to connect to a container, list its JVMs and finally to connect to a selected JVM.

The kubectl command line tool may be available on your local computer or alternatively on a remote machine to which you have SSH access. JProfiler directly supports both scenarios. For local installations, JProfiler will try to detect the path to kubectl automatically, but you can configure an explicit path on the "External tools" tab of the general settings dialog.

🥑 JProfiler Sta	rt Center		×
Start Ce	nter		
Open Session	On this computer Where is kubectl located vubectl is on this con		
Quick Attach	 kubect is on another kubect is on another SSH connection: Use SOCKS proxy 	computer [not configured]	Edit
New Session Open Snapshots			
		Start	Close

JProfiler lists all detected containers in a tree with three levels. At the top are namespace nodes that contain child nodes with the detected pods. The leaf nodes are the containers themselves and one of them has to be chosen to proceed to the selection of a running JVM .

🤣 Select Remote C		×	
Options for kubectl:	None		Change
	Remember across restarts 😯		
 database (nai default (nai- openjdc- openjdc- openjc- spring-pe 	espace] pp [pod]		
Type into the tree to	start quick search		
		ОК	Cancel

kubect1 may require additional command line options for authentication in order to be able to connect to the Kubernetes cluster. These options can be entered at the top of the container selection dialog. Because these options may be sensitive information, they are only saved to disk if you explicitly select the checkbox to remember them across restarts. Deselecting this checkbox will clear any previously saved information immediately.

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.

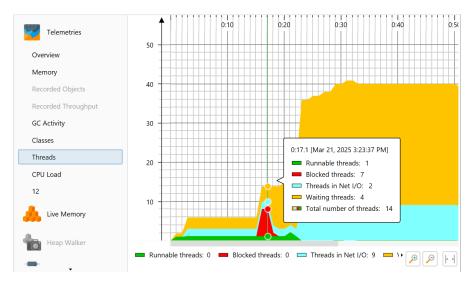
Recording Data

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 always recorded 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.

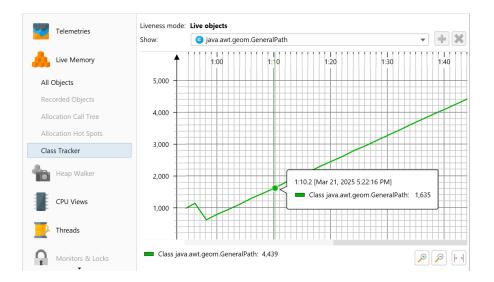
This is why JProfiler offers fine-grained mechanisms for controlling the recording of information that you are actually interested in.

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. 45]. 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. This is why the "All objects" is not updated automatically and you need to manually create new dumps of all objects.

Telemetries	All objects dumps:	📥 📝 🗙	Aggregation level:	Classes	
	at 1:44.570.220		Name	Instance Count	Size
Live Memory			java.awt.Rectan	50,265 (10 %)	1,608 kt
			java.util.HashM	37,549 (7 %)	1,201 k
ojects			java.security.Ac	33,479 (6 %)	1,339 kt
Jujects			sun.java2d.pipe	23,918 (4 %)	956 kE
orded Objects			java.awt.geom	20,930 (4 %)	1,506 kE
			char[]	17,528 (3 %)	1,062 kl
cation Call Tree			float[]	16,145 (3 %)	1,225 k
			sun.java2d.d3d	15,622 (3 %)	312 k
cation Hot Spots			int[]	13,243 (2 %)	30,237 k
ss Tracker			java.lang.String	13,148 (2 %)	315 k
STIdCKEI			sun.java2d.Sun	12,937 (2 %)	2,794 k
			java.lang.Integer	12,570 (2 %)	201 k
Heap Walker			java.lang.ref.We	12,153 (2 %)	388 k
			sun.java2d.State	11,745 (2 %)	187 k
			java.lang.Object[]	9,993 (2 %)	412 k
CPU Views			sun.awt.EventQ	8,777 (1 %)	210 k
			java.awt.EventQ	8,215 (1 %)	197 k
			java.util.ArrayList		191 k
Threads			Total from 1.0		50,151 k
				475,557 (10076)	50,151 K
Monitors & Locks			Q- Class View Filte	ers	-

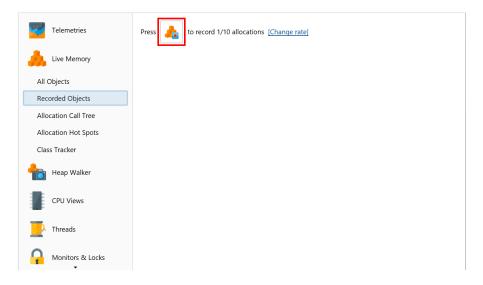
Some measurements capture enum-like values, such as the execution status a thread is currently in. This kind of measurement can be displayed as a colored time line and consumes a lot less memory than numerical telemetries. In the cases of thread statuses, the "Thread history" view shows the timelines 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.

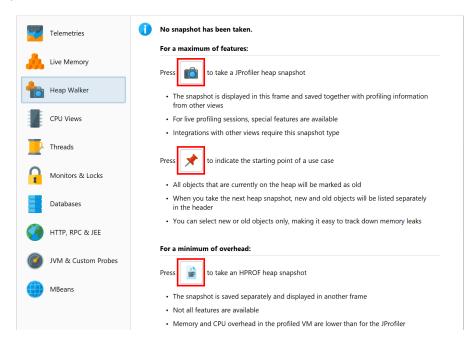
Telemetries	Recorded allocations:	Live objects at , 1/10 allocations, java.lang.String	<u>Change</u>
	Aggregation level:	Methods	•
Live Memory		216 bytes - 134 alloc, java util.concurrent.ThreadPoolExecutor\$Worker.run 1,872 bytes - 78 alloc, called from call site #3 (remote VM #1)	
All Objects	🔻 🙀 🖬 28.79	6 - 1,872 bytes - 78 alloc. com.ejt.demo.server.handlers.RmiHandlerImpl.rer 8.7% - 1,872 bytes - 78 alloc. com.ejt.demo.server.handlers.RmiHandlerImpl	
Recorded Objects		28.7% - 1,872 bytes - 78 alloc. com.ejt.demo.server.handlers.RmiHandlerf	mpl.make
-II Allocation Call Tree		 	erHelper.
Allocation Hot Spots		 15.5% - 360 bytes - 15 alloc, java.ioc.lbut.ict.NRL.vint> 15.5% - 360 bytes - 4 alloc, java.io.BufferedReader.readLine 	coponsec
Class Tracker		0 1.5% - 96 bytes - 4 alloc. java.net.URL.openConnection	
Heap Walker	▼ 🙀 🖬 19.9%	0.7% - 48 bytes - 2 alloc. java.lang.StringBuilder.toString ,296 bytes - 54 alloc. called from call site #1 (remote VM #1) - 1,296 bytes - 54 alloc. com.ejt.demo.server.handlers.RmilAndlerImpl.rem 2000 bytes - 54 alloc. com.ejt.demo.server.handlers.RmilAndlerImpl.rem	
CPU Views	₩ ((<u>k</u>))	.9% - 1,296 bytes - 54 alloc. com.ejt.demo.server.handlers.RmiHandlerImpl., 119.9% - 1,296 bytes - 54 alloc. com.ejt.demo.server.handlers.RmiHandlerIm	npl.makeł elper.mak
Threads		15.4% - 1,006 bytes - 42 alloc, java.net.HttpURLConnection.getR 12.6% - 168 bytes - 7 alloc, java.net.URL. <init> 27% - 48 bytes - 7 alloc, java.net.URL.<init></init></init>	
Monitors & Locks	Q- Call Tree View Filt	ers	- 3

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.

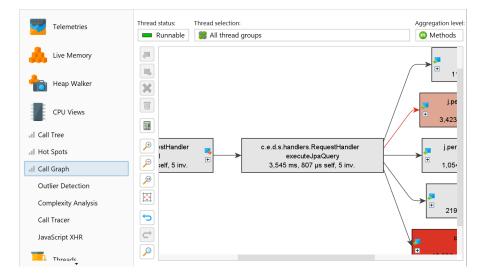
A more lightweight operation is marking all objects on the heap before you start a use case, so that you can find all newly allocated objects when you take a heap snapshot later on.

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 fewer 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.

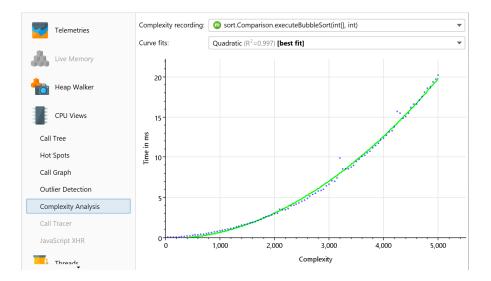


Under particular circumstances it may be useful to see the chronological sequence of method calls, especially if multiple threads are involved. For these special cases, JProfiler offers the "Call tracer" view. That view has a separate recording type that is not tied to the more general CPU recording. 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.

Telemetries	744 traces, 0 hidden element		
Telemetries	V TCP Connection(5)-127.0.0.1	(6 traces)	+ 0 µs
	🔻 🕑 java.util.concurrent	(1 trace)	+ 0 µs
Live Memory	java.util.concurrent.ThreadPoolExecutor\$Worker	(1 trace)	+ 0 µs
	😱 run()		+ 0 µs
	v (2) com.ejt.demo.server.handlers	(5 traces)	+ 0 µs
Heap Walker	🔻 😉 com.ejt.demo.server.handlers.RmiHandlerImpl	(3 traces)	+ 0 µs
	🐢 remoteOperation()		+ 0 µs
-	performWork()		+ 0 µs
CPU Views	🐢 makeHttpCalls()		+ 0 µs
	com.ejt.demo.server.handlers.HandlerHelper	(2 traces)	+ 0 µs
I Call Tree	pool-1-thread-3	(17 traces)	+ 690 µs
	RMI TCP Connection(5)-127.0.0.1	(2 traces)	+ 52 ms 903 µs
Hot Spots	pool-1-thread-3	(16 traces)	+ 53 ms 339 µs
	RMI TCP Connection(5)-127.0.0.1	(2 traces)	+ 95 ms 14 µs
II Call Graph	pool-1-thread-3	(16 traces)	+ 95 ms 466 µs
		/ 4	
Outlier Detection	com.ejt.demo.server.handlers.RmiHandlerImpl.performWork()		
Complexity Analysis	com.ejt.demo.server.handlers.RmiHandlerImpl.remoteOperation	0	
complexity Analysis	java.util.concurrent.ThreadPoolExecutor\$Worker.run()		
I Call Tracer			
JavaScript XHR			
Threads			

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 number 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.

Ø Monitor History Graph View	Settings	×
Recording Time line		
Recording Thresholds		
Monitor blocking threshold:	1,000	🗢 μs
Monitor waiting threshold:	100,000	¢μs
All events with a duration the configured threshold will be a Warning: If you lower the th recorded. Please note that th overhead grows linearly in tir	discarded. resholds, more data e associated memo	
	ОК	Cancel

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.

Session View Profiling	Window Help Demo server - JProfiler -	×
Start Center Stop Save Session Session	Start Stop Change Add Image: Change Add Image: Change Image: Change	
Three de	Time Line JDB Connections JDBC connections and execution of statement	-
Threads	Both open and closed 💌 Sort by start time 💌 🔍 Filter	•
Monitors & Locks	Physical Connections 0:10 0:20 0:30	
Databases	jdbc:demo://remote_host/test [ID 1]	
.I JDBC	jdbc:demo://remote_host/test [ID 4]	
JPA/Hibernate	jdbc:demo://remote_host/test [ID 5]	
MongoDB	jdbc:demo://remote_host/test [ID 7]	
Cassandra	jdbc:demo://remote_host/test [ID 9]	
HTTP, RPC & JEE	jdbc:demo://remote_host/test [ID 10]	
JVM & Custom Probes		
MBeans	— Idle 💻 Statement execution 💻 Prepared statement execution 💻 Batch exe 🕨 🔊	
7 9	3 active recordings Auto-update 2 s VM #1	ofiling

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.

Start enter Stop Start Stop Save Save Session Session	Window Help E Start Stop Cha Recordings Recordings Trace Profil	king Run GC	Add Bookmark	Export Settings	Help Stop Probe JDBC View Specific	Events
Threads	🖣 Leaks 🗧 Telem	etries	★ Event	s JDBC col	nnections and execution	JDBC of statements
4	All types	•	Filter in all te	xt columns 💌	Q . Filter	
Monitors & Locks	Start Time 🔺 Eve	nt Type 🛛 🛛	Duration	Connection ID	Description	Thread
4	0:01.965 [M 🗔 Co	nnectio	0 µs 1		jdbc:demo://rem	Servlet requ
-	0:02.010 [M 💻 Pr	epared st	179 ms 1		SELECT * FROM	
Databases	0:02.645 [M 💻 Pr	epared st	69,372 µs 1		INSERT INTO CUS.	. Servlet requ
	0:02.795 [M 💻 Pr	epared st	54,981 µs 1		INSERT INTO OR	Servlet requ
JDBC	0:02.929 [M 💻 Pr	epared st	47,105 µs 1		INSERT INTO OR	Servlet requ
	0:03.774 [M 🗖 Co		0 µs 2		jdbc:demo://rem	
JPA/Hibernate	0:03.806 [M Pr		119 ms 2		SELECT * FROM	
Manage	0:04.001 [M 🖵 Co		0 µs 3		jdbc:demo://rem	
MongoDB	0:04.001 [M 🖿 Sta		515 ms 3		SELECT i.id, i.avai	
Cassandra	0:04.577 [M 💻 Pr		60,289 µs 2		INSERT INTO CUS.	
	Total from 25 rows:	atement	776 ms 3 6,474 ms		SELECT i id i avai	RMI TOP CO
HTTP, RPC & JEE	Selection 🔢 D	uration				
	Stack trace:					
JVM & Custom Probes	javax.persistence.Typed	Query.getResul	ltList()			
MBeans	com.ejt.demo.server.hai com.ejt.demo.server.hai com.ejt.demo.server.hai	ndlers.RequestI	Handler.make	JpaCall()	.persistence.EntityMa	anager)
	3 active record		uto-update 2		1 #1	🔹 Profilin

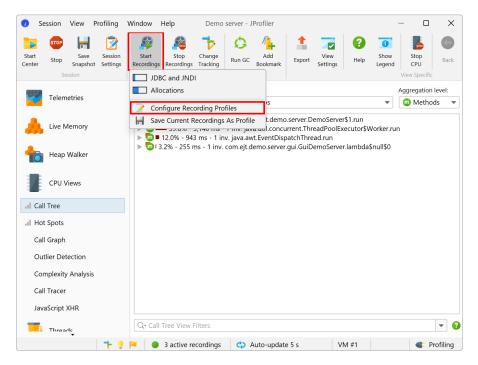
The JDBC probe has a third recording action for recording JDBC connection leaks. The associated overhead with looking for connection leaks is only incurred if you are actually trying to investigate such a problem. Just like the event recording action, the selection state of the leak recording action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

Session View Profiling	Vindow Help JDBC demo - J	Profiler	– 🗆 X
Start Center Stop Save Session Session Session	Start Stop Start Recordings Recordings Tracking Profiling	Add Export Settings	Stop Probe JDBC Stop Leaks Freeze View View Specific View View
Telemetries	Hot Spots	Connection Leaks	JDBC Enclose and execution of statements
ive Memory	This view shows all virtual connections are what you get from connection pool:		
Heap Walker	Connections of type "Unclosed collecter candidates.	d" are definite leaks while "Unclo	sed" connections are strong
CPU Views	All types Filter in all Opened At Open Since	text columns 🔹 🔍 Filter	Thread Class Name
Threads	0:08.855 [Mar 13,320 ms■	Unclo jdbc:hsqldb:hsql://l	pool-1-threa jdk.proxy2.\$Prox
Monitors & Locks	1 row		
Databases	Stack trace:		
.i JDBC	javax.sql.DataSource.getConnection() jdbc.JdbcTestWorker.call()		
JPA/Hibernate	jdbc.JdbcTestWorker.call() java.util.concurrent.ThreadPoolExecut	or\$Worker.run()	
MongoDB			
🕂 💡	ڬ 🥚 3 active recordings	Auto-update 2 s VM	#1

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 allocation recording will distort the timings of CPU data significantly.

Onfigure Recording Profiles	×
Configured recording profiles:	
JDBC and JNDI	
Allocations CPU data Call tracer Complexity data Custom probes Record database probes: [none] Record JVM & custom probes: [none] Recording overhead:	***
	
С Неір	Cancel

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.

.I Call Tree				
Hot Spots				
Call Graph		J		
Outlier Detecti	The following data is being recorded:			
Complexity An	JDBC			
Call Tracer				
JavaScript XHF	You can start and stop recording with view-specific			
Threads	tool bar buttons or recording profiles.			~ ?
	👎 💡 🎮 🧔 3 active recordings	🗘 Auto-update 5 s	VM #1	Profiling

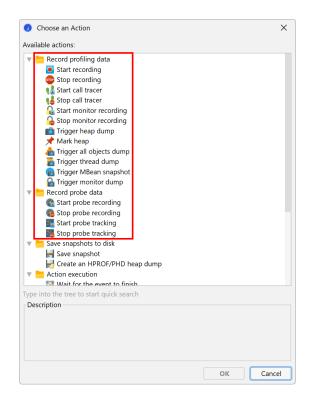
A recording profile can also be activated directly when you start profiling. The "Session startup" dialog has an *Initial recording profile* drop-down. By default, no recording profile is selected, but if you need data from the startup phase of the JVM, this is the place to configure the required recordings.

Call tree recording:	Instrumentation, 1 exceptional method		
	For low-overhead CPU profiling, <u>switch to sampling</u> .		
Call tree filters:	1 filter rule for method call recording		
	Profiled packages have been defined. If the overhead is too high, make you specific or <u>switch to sampling</u> .	ur filters more	
Trigger settings:	No active triggers	Edit	
Database probe settings:	4 enabled probes	Edit	
HTTP, RPC & JEE probe settings:	9 enabled probes	Edit	
JVM & custom probe settings:	5 enabled probes	Edit	
	nd JNDI Confi	gure	
erformance			

Recording with triggers

Sometimes you want to start a recording when a particular condition occurs. JProfiler has a system for defining triggers [p. 129] 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.

7 Trigger Wizard - Method	d invocation	×
1. Trigger type	Configure actions for this trigger	
 Specify methods Actions 	Configured actions:	
4. Description 5. Group ID	Start recording	+
6. Finished	 CPU data Allocation data Reset data Thread data VM telemetry data 	×
	Complexity analysis	
	🕇 Back Next 🕨 Finish Ca	ncel

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.

rread status: 🕜 Thread selection:				Aggregation leve	el:
💻 Runnable 🔻 🛛 🍔 All thread grou	os		▼	Methods	
61.8% - 24,364 ms - 9 inv. java					
🔻 🔟 💳 35.3% - 13,915 ms - 7 inv. com.e		noServer\$1.run			
9.4% - 3,720 ms - 4 inv. HTTP: /					
9.2% - 3,606 ms - 3 inv. HTTP: /					
6.8% - 2,680 ms - 3 inv. HTTP: /					
6.4% - 2,511 ms - 3 inv. HTTP: /					
P = 1.0% - 1,012 ms - 3 inv. com.ejt. P = 1.0% - 275 ms - 2 inv. com.eit.dv					
Show Call Graph		ers.JmsHandler.handleMessage			
		ndlers.JmsHandler.performWork			
Show Threads		andlers.JmsHandler.makeHttpCall			
Add Method Trigger		andlers.JmsHandler.makeRmiCall			
		sHandler\$JmsType. <clinit></clinit>			
🕞 🧃 🧑 Add As Exceptional Method		nsHandler\$JmsType.values			
🧕 < Split Method with a Script		juestHandler. <clinit></clinit>			
🖳 🕥 Intercept Method With Script	Probe	Handler\$JmsType.getDestination			
Q		;Handler\$JmsType.getDuration			
🕨 🛅 🗩 Merge splitting level	Ctrl+Alt+M	1			
C Remove Selected Subtree	Delete	Server.lambda\$null\$0			
Restore Removed Subtrees	Ctrl+Alt+S				
0.0	Carryacio				
P Add Filter From Selection	•	·			_
Q+ Call Show Tree Legend				-	1

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 rigger recording state icon in the status bar.

JavaScript XHR		
	Q- Call Tree View Filters	
	📫 💡 📃 🛛 🛛 🗧 3 active recordings	🗘 Auto-update 5 s

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

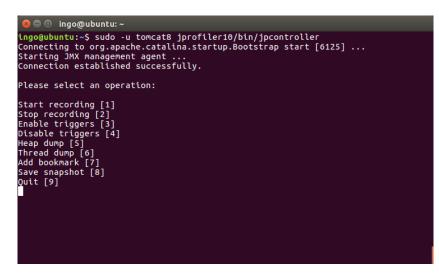
JProfiler has a command line executable for controlling the recordings in any JVM that is already being profiled. jpcontroller requires that the JProfiler MBean is published, otherwise it will not be able to connect to the profiled JVM. This is only the case if the profiling agent has already received 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. For more information, see the help topic on setting profiling settings at startup [p. 248].

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. 129] 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

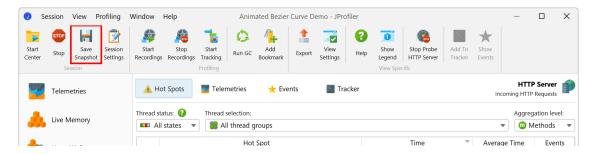
Until now, we have only looked at live sessions where the JProfiler GUI obtains the data from the profiling agent that is running inside the profiled JVM. JProfiler also supports snapshots where all profiling data is written to a file. This can be of advantage in several scenarios:

- 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.

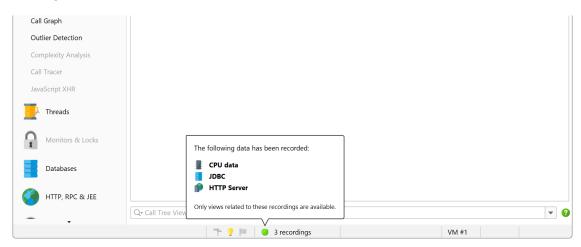
🕖 JP	rofiler	×
	Current Window Is In Use lo you wish to proceed?	
	Open a new window	
►	Use this window	
	If you select this option, the currently active profiling session will be stopped.	
	Cancel	

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

File View Window He	elp	Snapshot Comparison - JProfiler	-	×
Memory COPU COmparison Comparison	Probe Start Comparison Center	Export View Settings		
Available Snapshots	+ J ^A			
test1.jps 2025-03-03 14:26:52 test2.jps 2025-03-03 14:26:55 test3.jps 2025-03-03 14:26:57		Please select snapshots on the left and create a	comparison	

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.

Call tree recording:		Instrumentation		Edit	
5		For low-overhead CPU profiling, <u>switch to sampling</u> .			
Call tree filters:		1 filter rule for method call recording		Edit	
		Profiled packages have been defined. If the overhead is too high, n specific or <u>switch to sampling</u> .	nake your filters	s more	
Trigger settings:		No active triggers		Edit	
Database probe settings:		enabled probes enabled probes			
HTTP, RPC & JEE probe s	settings:				
JVM & custom probe settings:		5 enabled probes			
tartup And Exit	Ino roc	ordinas]	Configure		
initial recording profile:	[no rec	stangsj •	Configure	J	
JVM exit action:	Let the	JVM exit and disconnect	•	More •	
erformance		JVM exit and disconnect e VM alive for profiling			
Overhead:	Save an	d immediately open a snapshot			

- 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.

1. Trigger type	Configure actions for this trigger
2. Threshold 3. Actions	Configured actions:
4. Description 5. Group ID	Start recording
6. Finished	I Sleep
	Stop recording
	H Save snapshot
	Snapshot file: test
	Add a unique number to the snapshot name
	Note: If the JProfiler GUI is connected, the saved snapshot will only have partial content.

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.

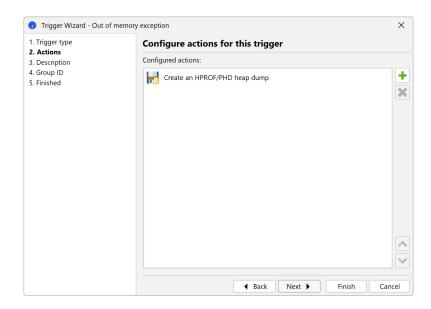
📀 Si	ession	View	Profiling Window Help	Animated B	ezie	r Curve D	Demo - JPro	ofiler					-		×
Start Center	Stop Stop	Save Snapst		► F8 equest Tracking Ctrl+F8	ark	1 Export	View Settings	? Help	Show Legend View Spe	Stop Probe HTTP Server cific	Add To Tracker	Show Events			
	Telem	etries	 Disable Triggers And Custo Enable Trigger Groups Save HPROF Snapshot 	m Probes F6 Shift+F6 Ctrl+Shift+S	Eve	ents	🎫 Tr.	acker				Inc	HTT oming HTTF	P Server Requests	
	Live N	1emory	Mark Heap	Shift+F4										gation lev Aethods	
	Неар	Walker		F3	-					Time	~	Averag	je Time	Ever	nts
	CPU V	/iews	Edit Bookmarks Show Global Filters for Met	Shift+F3 hod Call Recording											

• 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

JProfile fully supports opening snapshots saved by Java Flight Recorder (JFR). The UI is notably different in this case and is adjusted to the capabilities of JFR. See the JFR help topics [p. 221] for more details.

⁽²⁾ https://docs.oracle.com/en/java/javase/11/tools/jmap.html#GUID-D2340719-82BA-4077-B0F3-2803269B7F41

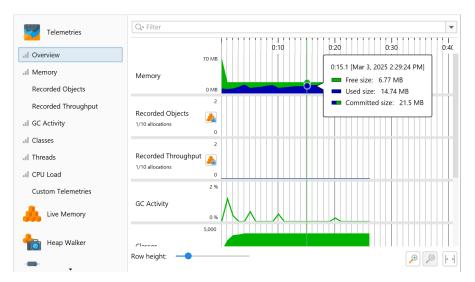
Telemetries

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.

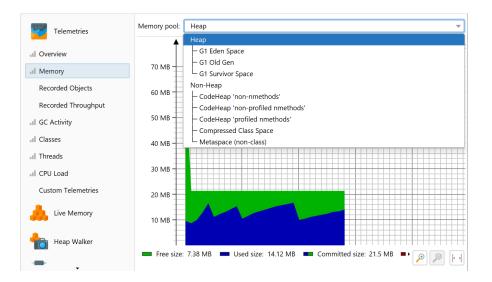
Standard telemetries

In the "Telemetries" section of the JProfiler UI, a number of telemetries are recorded by default. For interactive sessions, they are always enabled. Some telemetries require that a special type of data is recorded. In that case, a recording action will be shown in the telemetry.

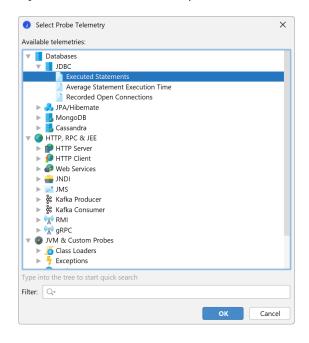
To compare multiple telemetries on the same time axis, the overview shows multiple small-scale telemetries on top of each other with a configurable row height. Clicking on the telemetry title activates the full telemetry view. The default order of the telemetries in the overview may not be suitable, for example, because you want to correlate selected telemetries side by side. In that case you can reorder them with drag and drop in the overview.



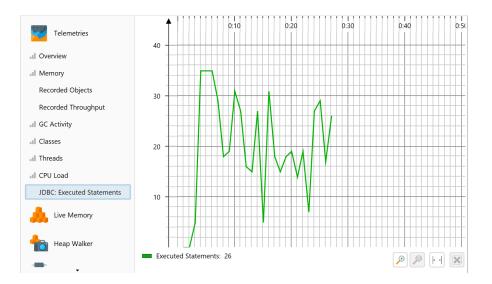
The full view shows a legend with current values and may have more options than what is visible in the overview. For example, the "Memory" telemetry allows you to select single memory pools.



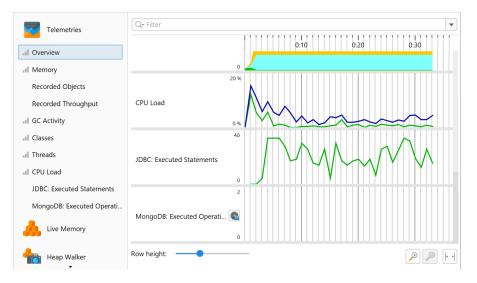
JProfiler has a large number of probes [p. 104] that record events from high-level systems in the JVM and important frameworks. Probes have telemetries that are displayed in the corresponding probe views. To compare these telemetries to the system telemetries, you can add selected probe telemetries to the top-level telemetries section. From the toolbar, choose + Add telemetries->Probe Telemetry and select one or more probe telemetries.



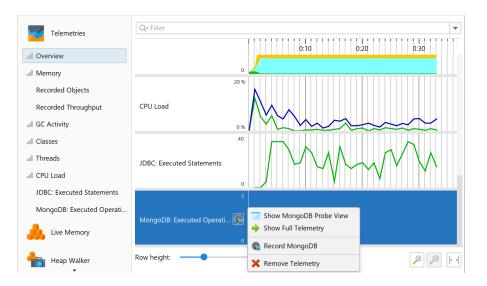
Each added probe telemetry gets its own view in the telemetry section and is also displayed in the overview.



Once a probe telemetry has been added, data is only shown if probe data has been recorded. If not, the telemetry description contains an inline button to start recording.

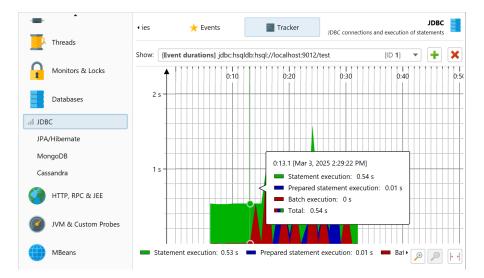


The context menu for probe telemetries contains the recording actions as well an action to show the corresponding probe view.



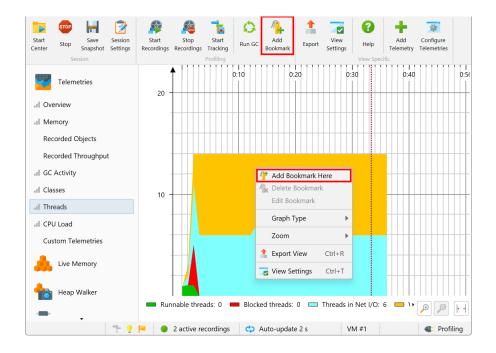
Similar to the probe views, the VM telemetries for the recorded objects depend on memory recording and also have a recording button and a similar context menu.

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 programmatically.

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.

🥑 Edit Bookr	nark X
Bookmark Pro	perties
Description:	Stopped JDBC recording
Color:	 Default
	Custom
	• 0, 0, 0
	🗸 Draw dashed line
	OK Cancel

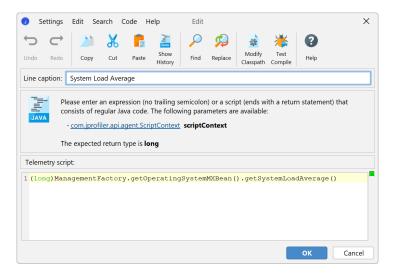
If right-clicking several bookmarks in a telemetry is too inconvenient, you can use the *Profiling->Edit Bookmarks* action from the menu to get a list of bookmarks. This is also the place where you can export bookmarks to HTML or CSV.

Time	Bookmark	1
0:12.998 [Mar 3, 2025 2:29:21 PM]	Unnamed bookmark	
0:33.398 [Mar 3, 2025 2:29:42 PM]	Stopped JDBC recording	3
0:38.748 [Mar 3, 2025 2:29:47 PM]	Unnamed bookmark	

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.

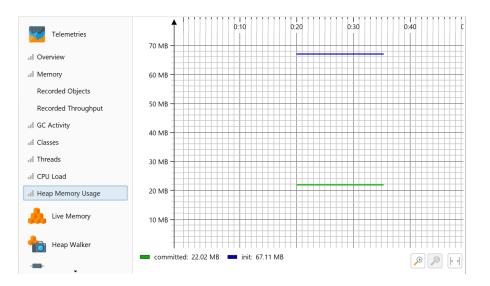
₹ Filter	•	Q . Filter		
com.jprofiler.api.agent.mbea com.sun.management	n	Name ▼ HeapMemoryUsage	Value [java.lang.management.MemoryUsage]	
 java.lang GarbageCollector [type] MemoryManager [type] ClassLoading [type] Compilation [type] OperatingSystem [type] Runtime [type] Threading [type] java.nio java.util.logging jdk.management.jfr 		committed init max used NonHeapMemoryUsage ObjectName ObjectPendingFinalizatio Verbose	22020096 67108864 67108864 13680448 [java.lang:management.MemoryUsage] java.lang:type=Memory 0 false	

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.

🕖 Configure Tel	emetries		\times
Several types of te	elemetries are sh	own in the Telemetries section:	
• The set of s	tandard JVM te	lemetries that is always shown	
Probes tele	metries that are	only shown in the probe views by default, but can be added here as well	
Custom tele	emetries from N	1Beans or scripts	
Probe telemetries		m Telemetry X	+
committed	Name:	Heap Memory Usage	
init [MBear	Unit:	bytes 💌	×
	Scale (10^-n):	0 🗢 😮	
	Stack all lin	es in the telemetry and show an area graph	
	🕜 Help	OK Cancel	
🕜 Help		ок	Cancel

Custom telemetries are added at the end of the "Telemetries" section in the order in which they are configured. To reorder them, drag them to the desired position in the overview.



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

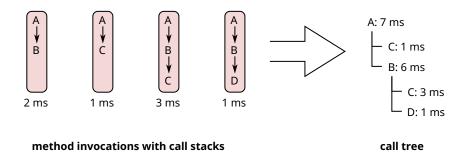
When JProfiler measures the execution times of method calls together with their call stacks, we call it "CPU profiling". This data is presented in a variety of ways. Depending on the problem you are trying to solve, one or the other presentation will be most helpful. CPU data is not recorded by default, you have to switch on CPU recording [p. 27] to capture interesting use cases.

Call tree

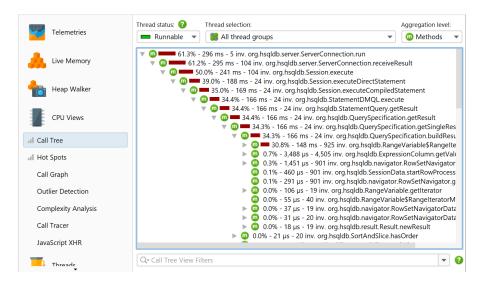
Keeping track of all method calls and their call stacks would consume a considerable amount of memory and could only be kept up for a short time until all memory is exhausted. Also, it is not easy to intuitively grasp the number of method calls in a busy JVM. Usually, that number is so great that locating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

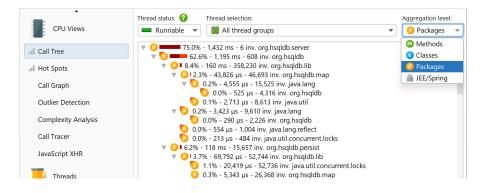
This is why JProfiler builds a cumulated tree of all observed call stacks, annotated with the observed timings and invocation counts. The chronological aspect is eliminated and only the total numbers are kept. Each node in the tree represents one call stack that was observed at least once. Nodes have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



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 "JEE/Spring components" mode. If your application uses JEE or Spring, you can use this mode to see only JEE and Spring components on a class level. Splitting nodes like URLs are retained in all aggregation levels.

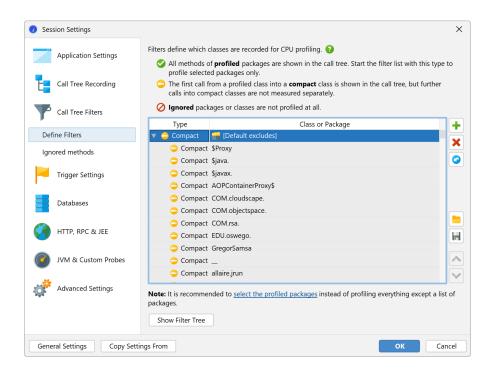


Call tree filters

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 fewer 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. 66] and the default filters is so undesirable that JProfiler suggests changing it in the session startup dialog.

Session Startup						
Settings						
Call tree recording:		Instrumentation, 1 exceptional method	Edit			
		For low-overhead CPU profiling, <u>switch to sampling</u> .				
Call tree filters:		1 filter rule for method call recording				
	4	The configured exclusive filters may be too broad. In that case, the overf nstrumentation may be very high, and CPU times will be distorted. Pleas packages or switch to sampling.				
Trigger settings:		No active triggers	Edit			
Database probe settings:		3 enabled probes	Edit			
HTTP, RPC & JEE probe settings:		9 enabled probes				
JVM & custom probe settings:		5 enabled probes	Edit			
tartup And Exit						
Initial recording profile:	JDBC an	d JNDI 🔹 Con	ifigure			
JVM exit action:	Let the J	VM exit and disconnect	▼ <u>More</u>			
erformance						
Overhead:						
The overhead is	compose	d of the selected profiling settings and the selected recording profile.				

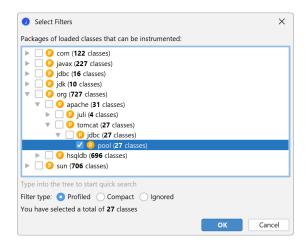
The filter expressions are compared against the fully qualified class name, so <code>com.mycorp</code>. matches classes in all nested packages, like <code>com.mycorp.myapp.Application</code>. There are three types of filters, called "profiled", "compact" and "ignored". All methods in "profiled" classes 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

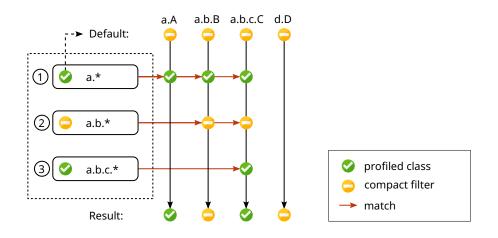
JRE. For example, when calling hashMap.put(a, b) you probably want to see HashMap.put() in the call tree, but not more than that - its inner workings should be treated as opaque unless you are the developer of the map implementation.

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.

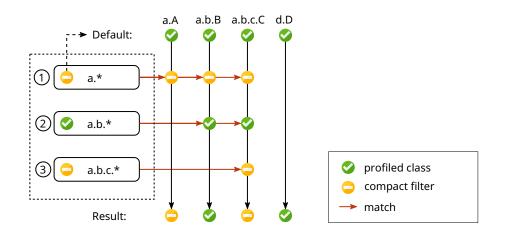
Entering packages manually is error prone, so you can use the package browser. Before you start the session, the package browser can only show you packages in the configured class path which often does not cover all the classes that are actually loaded. At runtime, the package browser will show you all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



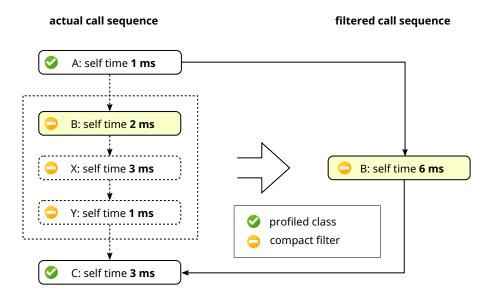
If you start it with a "compact" filter, the initial filter type of a class is "profiled". In this case, all classes are profiled except for explicitly excluded classes.



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.

Node Description
Show percentage bar
✓ Show time
Show self time
Show invocation count
Show average times in brackets 🛛 😮
🗹 Always show fully qualified names 🛛 😮
Always show signature 😗
Shorten packages
Time Scale
● Automatic 😢 ─ Mixed units ─ s ─ ms ─ µs
Display Threshold
Hide calls with less than 0.01 % 😯
Percentage Calculation
🗌 Relative 😮 💿 Absolute 😮
OK Cancel

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.

• •	Thread status: 😯 Thread selection:	Aggregation level:
CPU Views	Runnable 🔻 🧊 All thread groups 👻	💿 Methods 🔻 🔻
.II Call Tree	Image: Weight of the second	
I Hot Spots	🔻 🔟 💻 26.0% 😻 HSQLDB Connections @79e2c065	
Call Graph	▼ 00 ■ 26.09 ● HSQLDB Connection @22a9860f (67) ▼ 00 ■ 16 ● HSQLDB Connection @48f4ecb1 (66)	
Outlier Detection	 HSQLDB Connection @60665068 (65) HSQLDB Connection @64a338c (64) 	
Complexity Analysis	HSQLDB Connection @75a81785 (68)	executeOuerv

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 be waiting on a monitor, for example, by calling <code>Object.wait()</code> or <code>Thread.sleep()</code> 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 <code>synchronized</code> 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.

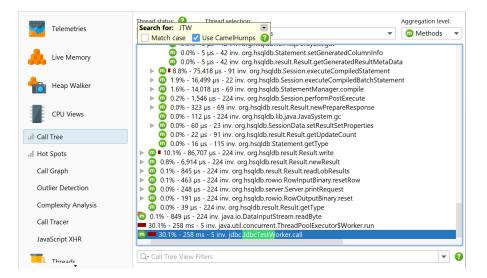
•	Thread status: 😮 Thread selection: Aggregati	on level:
CPU Views	Runnable 🔻 🥞 All thread groups 🔹 🐨	hods 💌
	All states - 1,207 ms - 5 inv. org.hsgldb.server.ServerConnection.run	
.1 Call Tree	Runnable 4 ms - 5 inv. java.util.concurrent.ThreadPoolExecutor\$Worker.run	
II Hot Spots	Waiting 424 ms - 5 inv. jdbc.JdbcTestWorker.call	
	Blocked % - 424 ms - 5 inv. jdbc.JdbcTestWorker.call	
Call Graph	Net I/O 5.5% - 268 ms - 27 inv. jdbc.JdbcTestWorker.testStatementsPath1 9.2% - 150 ms - 27 inv. jdbc.JdbcTestWorker.testStatement	
Outlier Detection	🔻 🗓 I 5.2% - 85,210 μs - 47 inv. java.sql.Statement.executeQuery	
Complexity Analysis	 JDBC calls <u>Show in probe call tree</u> Internet and the second of the second	erv

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. 168] or injected [p. 163] probe APIs.

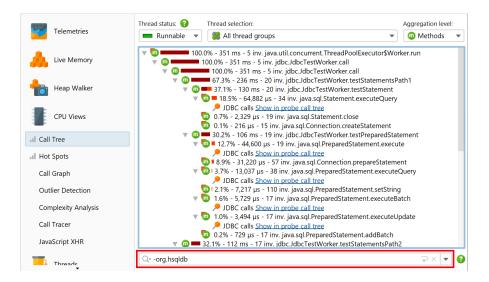
Finding nodes in the call tree

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. Expressions 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. 190].

	Start Tracking Run GC	Add Bookmark	1 Export	View Settings	? Help	Show Legend	Stop CPU	G Back	Forward	Show Graph	Analyze	
Thread status: 👔	Profiling Thread selection:					how Flame					+Alt+F	n level:
Runnable 🔻	🝔 All thread g	Jroups			-	alculate Cu		5 5	Calls ed Method		+Alt+G +Alt+B	ods
🕨 💼 💻 24.6% - 48	- 1,487 ms - 5 inv. 6 ms - 5 inv. java.u s - 1 inv. org.hsqldl	til.concurre	nt.Thread	dPoolExecu	ci to li	nline Async reate Call	: Executio	ns	ed Method	Ctrl	+Alt+E +Alt+S	
Q- Call Tree View Fi	ilters											•

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.



If the tool tip near the mouse cursor disturbs your analysis, you can lock it with the button in its upper right corner and then move it to a convenient location with the gripper at the top of the tool tip. The same button or a double click on the flame graph close the tool tip.

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. Colorizations are persisted across profiling sessions and are used globally for all sessions and snapshots.

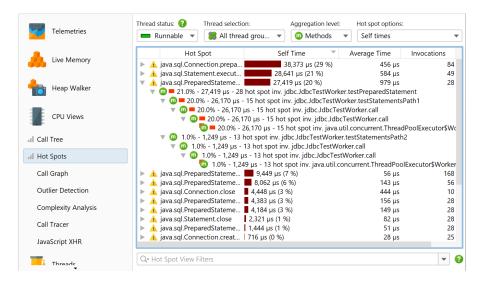
org.hsqldb.server. [match mode "Starts with", case sensitive]	
org.hsqldb.Expression [match mode "Starts with", case sensitive]	Ľ
	~
olorization actions based on the text of the selected node are available in t	the context
olorization actions based on the text of the selected node are available in t enu of the flame graph.	the context

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.

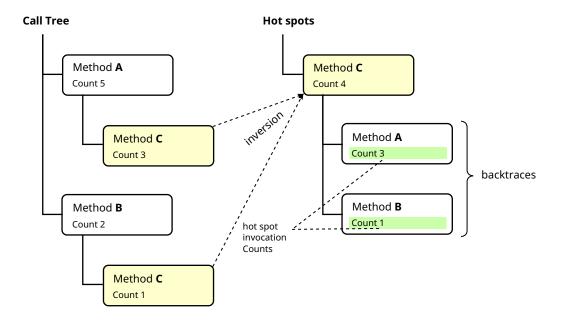
Hot spots

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.

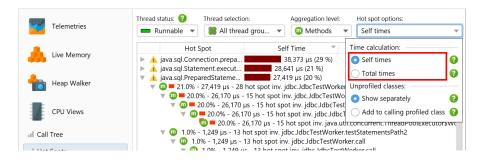
In that case, you need the inverse of the call tree: A list of all methods sorted by their total self time, cumulated from all different call stacks and with back traces that show how the methods were called. In a hot spot tree, the leafs are the entry points, like the main method of the application 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 be 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.

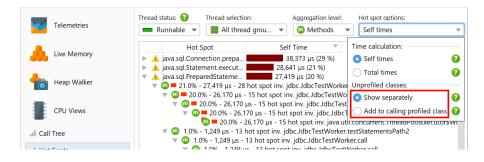


Hot spots and filters

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.

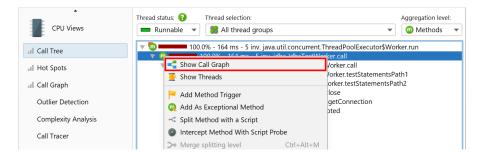
Telemetries	Thread status: Thread selection: Image: Runnable Image: Run	Aggregation level:
ive Memory		
Heap Walker		
CPU Views	stWorker	j.
II Call Tree	ntsPath2	
II Hot Spots	jdbc.JdbcTestWorker	- ¹
.il Call Graph	stWorker	37,550 μs
Outlier Detection	■ ntsPath1 self, 25 inv.	. j.s
Complexity Analysis		cr
Call Tracer		1,160 μs
JavaScript XHR		
Threade		

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.

 Call Graph View Settings
Display Options
Show signature tooltips
Shorten packages
Show average times in brackets ?
Color Information
• Self time 🔿 Total time
Color Scale Base
• Displayed methods only • All methods
Time Scale
Automatic 2
Display Threshold
✓ Initially hide outgoing calls with less than 1.0 % (?)
OK Cancel

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. 172]. Also, there are other advanced CPU views that are presented separately [p. 195].

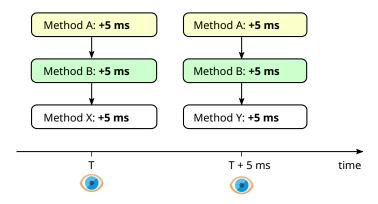
Method Call Recording

Recording method calls 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.

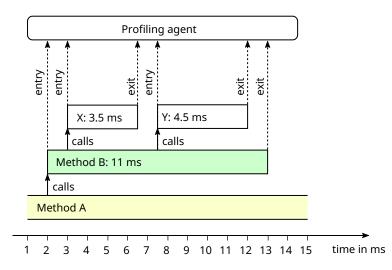
Sampling versus instrumentation

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.



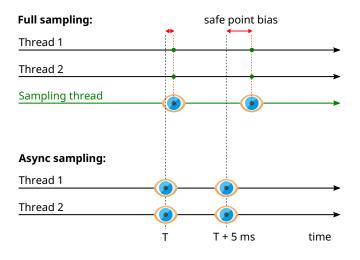
Full sampling versus async sampling

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. 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 suffers from truncated traces where only the end of the call stack is available. This is why the call tree is often not as useful for async sampling as the hot spots view. Async sampling is only supported on Linux and macOS.

Starting with Java 17, JProfiler can avoid using a global safe point for sampling on Hotspot JVMs and operate full sampling with near-zero overhead. Compared to async sampling, it still introduces some kind of safe point bias for single threads, but no longer the overhead of a global safe point for all threads in the JVM. Considering the drawbacks of async sampling, using full sampling is recommended for Java 17+.



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.

Ø Session Settings	
Application Settings	Method Call Recording Type
, appreciation occurringo	There are important trade-offs to be considered. Check out the <u>in-depth explanation</u> in the documentation.
Call Tree Recording	Instrumentation 2
Method Call Recording	All features Invocation counts Ideal for I/O bound code Careful with CPU bound code
Exceptional Methods	Adjust call tree filters
Split Methods	C Full sampling 3
Call Tree Filters	Low overhead 17+ Ideal for finding CPU hot spots Better accuracy for CPU times
	Not all features
Trigger Settings	Sample unmounted virtual threads
Databases	○ Async sampling ?
	Low overhead Best accuracy for CPU times Native sampling Only CPU times
MTTP, RPC & JEE	Not all features Experimental HotSpot-API
	Enable sampling of native libraries
JVM & Custom Probes	Async buffer size: 100 🗢 % 🎯
Advanced Settings	Common Options For Sampling
¥	Disable all filters for sampling
	efi = e
General Settings Copy Set	tings From OK Cance

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.

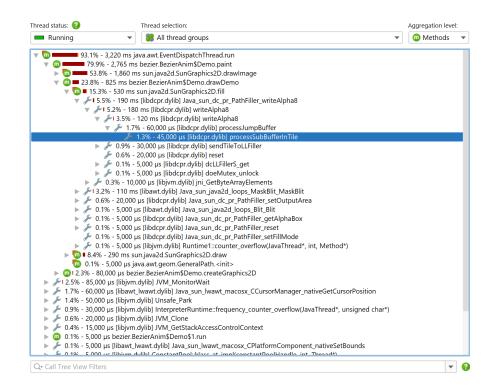
Otherwise, "Full sampling" is generally the most suitable option and is suggested as the default for new sessions.

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.

Session Settings		×
Application Settings	Method Call Recording Type There are important trade-offs to be considered. Check out the <u>in-depth explanation</u> in the documentation.	
Call Tree Recording	 documentation. Instrumentation (2) 	
Method Call Recording	All features Invocation counts Ideal for I/O bound code Careful with CPU bound code	
Exceptional Methods	Adjust call tree filters	
Split Methods	Full sampling	
Call Tree Filters	Low overhead 17+ Ideal for finding CPU hot spots Better accuracy for CPU times	
Trigger Settings	Not all features	
Databases	Async sampling Async sampling Dest accuracy for CPU times Native sampling Only CPU times	
HTTP, RPC & JEE	Not all features Experimental HotSpot-API	
JVM & Custom Probes	Enable sampling of native libraries ? Async buffer size: 100 % ?	
Advanced Settings	Common Options For Sampling	1-
General Settings Copy Setti		el

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.

Thread status: 😮	Thread selection:	A	ggregation leve	el:
Running 💌	🝔 All thread groups	•	Classes	•
	va.awt.EventDispatchThread bezier.BezierAnim\$Demo			
79.3% - 2,745	ms sun.java2d.SunGraphics2D			
►	ibdcpr.dylib			
►	μs libjvm.dylib s libsystem_m.dylib			
<i>F</i> 0.1% - 5,000 μ <i>F</i> 0.1% - 5,000 μ 6.8% - 235 ms lib	s libsystem_malloc.dylib awt.dylib			
🧿 0.1% - 5,000 μs java.	awt.geom.GeneralPath			
 ▶ № 6.4% - 220 ms libjvm.dy ▶ № 1.9% - 65,000 µs libawt_ 				
▶ ③ 0.1% - 5,000 µs bezier.Be				

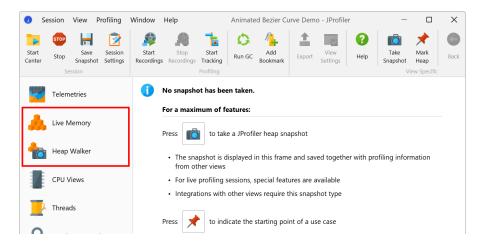
To eliminate selected native libraries, you can remove a node [p. 181] from that native library and choose to remove the entire class.

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. 27] 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. 214], 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 this view is not updated automatically with the current values.

Session View Profiling V	Window Help		r Curve Demo - JProfiler		- 🗆
tart stop Save Save Save Session Session	Start Stop Start Recordings Recordings Profiling	Run GC Add Bookmark	Export Settings View Specific	Show In Heap Walker	
Telemetries	All objects dumps:	A 🔀 🖌	ggregation level: 🕒 Classe	S	
	at 0:11.105.169		Name	Instance Count	Size
Live Memory		ł	oyte[]	16,855 (9 %)	938 kB
		j	ava.lang.String	14,771 (8 %)	354 kB
		i	ava.util.HashMap\$Node	9,860 (5 %)	315 ki
All Objects		į	ava.security.AccessControlCo.	8,994 (4 %)	359 k
Recorded Objects		j	ava.awt.geom.AffineTransform	n 8,169 (4 %)	588 k
Recorded objects		ĺ	ava.awt.Rectangle	6,503 (3 %)	208 k
Allocation Call Tree		j	ava.lang.Object[]	5,336 (2 %)	279 k
			double[]	4,939 (2 %)	406 k
Allocation Hot Spots		9	un.java2d.pipe.Region	4,835 (2 %)	193 k
		5	un.java2d.loops.RenderCach	4,431 (2 %)	106 k
Class Tracker		j	ava.util.concurrent.Concurre	3,656 (2 %)	116 k
		ĺ	ava.lang.Class	3,323 (1 %)	409 k
Heap Walker		ĺ	ava.lang.Integer	3,228 (1 %)	51,648 byte
		j	ava.lang.ref.WeakReference	3,153 (1 %)	100 k
		l j	ava.util.HashMap	2,896 (1 %)	139 k
CPU Views		5	un.java2d.SunGraphics2D	2,448 (1 %)	509 k
-		j	ava.awt.geom.Path2D\$Float\$.	2,397 (1 %)	76,704 byte
		[i	nt[]	2,312 (1 %)	3,092 kl
Threads			Fotal from 1,534 rows:	180,757 (100 %)	17,254 ki
Monitors & Locks			Q . Class View Filters		•
•					

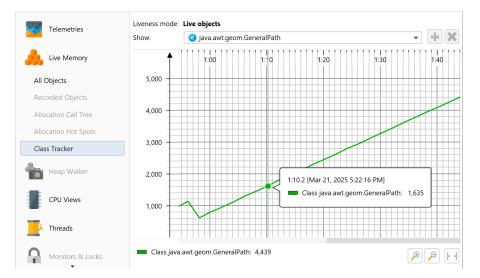
When looking for a memory leak, you often want to compare instance counts over time. To do that for all classes, you can use the differencing functionality of the view. When two dumps of all objects are selected at the same time, a *Difference* column is inserted and the histogram of the instance counts shows the baseline values at the time of the marking in green color. When taking a new dump of all objects, the oldest selected dump will remain selected and the difference with the new dump of all objects will be shown.

Session View Profiling V	Window Help Anin	nated Bezier Curve Demo - JProfiler	f	—	
tart enter Stop Snapshot Session Session	Start Stop Start Recordings Recordings Tracking Profiling	Add View	3 Show In Help Heap Walker		
Telemetries	All objects dumps:	Aggregation level: O	Classes		
	at 0:14.726.087	Name	Instance Count	Difference	Size
Live Memory	at 0:11.105.169	byte[]	17,187 (7 %)	+332 (+2.0 %)	968 kB
	at 0.11.105.109	java.lang.String	14,790 (6 %)	+19 (+0.0 %)	354 ki
		java.security.AccessC	12,281 (5 %)	+3,287 (+37.0 %)	491 k
All Objects		java.util.HashMap\$N	12,257 (5 %)	+2,397 (+24.0 %)	392 k
Recorded Objects		java.awt.geom.Affine	11,164 (5 %)	+2,995 (+37.0 %)	803 k
- Notonata o bjeta		java.awt.Rectangle	8,897 (4 %)	+2,394 (+37.0 %)	284 k
Allocation Call Tree		double[]	6,739 (3 %)	+1,800 (+36.0 %)	505 k
		sun.java2d.pipe.Region	6,630 (3 %)	+1,795 (+37.0 %)	265 k
Allocation Hot Spots		sun.java2d.loops.Ren	6,055 (2 %)	+1,624 (+37.0 %)	145 k
Class Tracker		java.lang.Object[]	5,937 (2 %)	+601 (+11.0 %)	300 k
		java.lang.Integer	4,128 (1 %)	+900 (+28.0 %)	66,048 byte
		java.lang.ref.WeakRef	4,051 (1 %)	+898 (+28.0 %)	129 k
Heap Walker		java.util.HashMap	3,796 (1 %)	+900 (+31.0 %)	182 k
		java.util.concurrent.C	3,656 (1 %)	0 (±0 %)	116 k
CPU Views		sun.java2d.SunGraphi	3,346 (1 %)	+898 (+37.0 %)	695 k
		java.lang.Class	3,328 (1 %)	+5 (+0.0 %)	410 k
		java.awt.geom.Path2	3,297 (1 %)	+900 (+38.0 %)	105 k
		java.util.HashMap\$N	2,646 (1 %)	+601 (+29.0 %)	191 k
Threads		Total from 1,539 ro	217,342 (100 %)	+36,585 (+20.0 %)	19,352 k
Monitors & Locks		Q- Class View Filters			•

The dump selector shows time stamps when the dump was taken. By double-clicking on a dump, you can add a label for easier identification. Dumps of all objects can also be triggered with a trigger action [p. 27] or the Controller API, where a label can also be specified.

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. As with most views that are updated dynamically a *Freeze* tool bar button is available to stop updating the displayed data.

With the *Mark Current* toolbar button, you can show a difference column with respect to a selected baseline in the "Recorded objects" view as well. 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

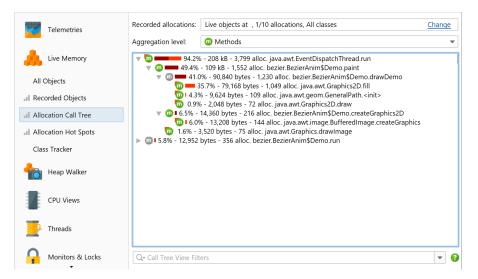
When allocation recording is active, JProfiler takes note of the call stack each time an object is allocated. It does not use the exact call stack, for example, from the stack-walking API, because that would be prohibitively expensive. Instead, the same mechanism is used that is configured for CPU profiling. This means that the call stack is filtered according to the call tree filters [p. 53] 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.

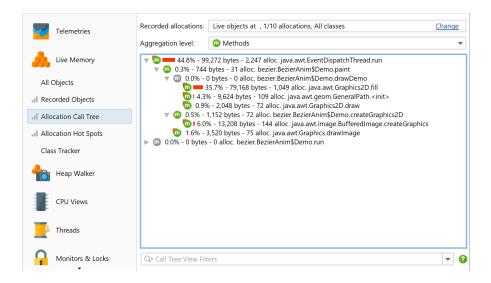
Just like for CPU profiling, the allocation call stacks are presented as a call tree, only with allocation counts and allocated memory rather than invocation counts and time. Unlike for the CPU call tree, the allocation call tree is not displayed and updated automatically because the calculation of the 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.

Allocation Options	×						
Type of Allocations to be Shown							
 Allocations cumulated for all classes 							
 Allocations for a selected class or package 							
java.awt.Rectangle							
Live objects	• 😧						
Update Options							
Auto-update the allocation views periodically 💡							
ок	Cancel						

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 screen shot above. 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 dialog 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.

Telemetries	Recorded allocations:	Live objects at , 1/1	0 allocations, All clas	sses		<u>Change</u>
-	Aggregation level:	💿 Methods 🛛 🔻	Hot spot options:	Self allocat	ed memory	•
Live Memory		ot Spot	Self Allocated N		Allocations	Difference
All Objects	 java.awt.Graph java.awt.image 	BufferedImage.cre	135 kB (3 22,072 bytes (5 %)	1	2,044 273	+2,078 (+92 +995 (+95.0 +129 (+90.0
Recorded Objects			17,080 bytes (4 %) 13,712 bytes (3 %)			+343 (+191 +82 (+75.0
 Allocation Call Tree 	java.awt.Graph		5,280 bytes (1 %)			+60 (+80.0
Allocation Classes	 A java.awt.Graph A bezier.BezierAr 	nim\$Demo.schedul	3,832 bytes (0 %) 2,416 bytes (0 %)			+70 (+97.0 +132 (+695
Allocation Hot Spots	📃 🕨 🔥 bezier.BezierAi	nim\$Demo.createG nim\$Demo.paint	2,176 bytes (0 %) 2,136 bytes (0 %)			+64 (+89.0 +58 (+187.0
Class Tracker	🕨 🛕 bezier.BezierA	nim\$Demo.schedul	512 bytes (0 %)		16	-141 (-90.0
Heap Walker						
CPU Views						
Threads	Q- Hot Spot View Filt	ters				• •
-	C Hot spot view Filt					

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 fewer 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.

Telemetries	Recorded allocations: Live object	ts, 1/10 allocations	<u>Change</u>
-	Aggregation level: Classes		
Live Memory	Name	Instance Count	Size
	java.awt.Rectangle	59 (9 %)	1,888 bytes
All Objects	java.awt.geom.AffineTransform	53 (8 %)	3,816 bytes
	java.util.HashMap\$Node	40 (6 %)	1,280 bytes
Recorded Objects	double[]	33 (5 %)	1,808 bytes
	sun.java2d.pipe.Region	30 (4 %)	1,200 bytes
Allocation Call Tree	java.security.AccessControlContex	29 (4 %)	1 160 hytes
Allocation Hot Spots	java.awt.geom.Path2D\$Float\$Cor	Ø Allocation Options	×
Allocation Hot spots	sun.awt.EventQueueltem		
Class Tracker	sun.java2d.loops.RenderCache\$K	Liveness Mode	
	java.util.HashMap		
<u></u>	sun.java2d.SunGraphics2D	Live objects	• •
Heap Walker	java.awt.EventQueue\$4		
	java.lang.Integer	Recording Rate	
	java.lang.ref.WeakReference	Record all objects	
CPU Views	java.lang.Long		
_	sun.awt.AWTThreading\$\$Lambda	 Sample one out of 10 (\$) allocation 	ions
5 - 1	java.util.HashMap\$Node[]	•	
Threads	iava awt geom Point2D\$Double		
	Total from 51 rows:	ОК	Cancel
Monitors & Locks	Q. Class View Filters		-

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. 134], 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.

Telemetries	Recorded allocation	ns:	Live objects, 1/10 allocations Change					
	Aggregation level:		Classes					
Live Memory	Na	me	Ir	nstance Cou	unt 🔻	Size		
	java.awt.geom.Affi	noT.	rancform		770 (8 %)	55,440 bytes		
All Objects	java.util.HashMap	Ď	Show Selection In Heap Walker		701 (7 %)	22,432 bytes		
,	java.awt.Rectangle		Show Allocation Tree for Selecti	ion	35 (5 %)	17,120 bytes		
Recorded Objects	sun.java2d.loops.F		Show Allocation Hot Spots for S	loction	5 %)	11,496 bytes		
	sun.java2d.pipe.Re			selection	5 %)	18,520 bytes		
Allocation Call Tree	java.security.Acce:		Add Selection To Class Tracker		; %)	18,440 bytes		
Allocation Classes	double[]	24	Show Source	F4	6)	22,984 bytes		
	java.lang.ref.Weak	£		F4		9,888 bytes		
Allocation Hot Spots	sun.java2d.SunGra		Show Bytecode			44,304 bytes		
	java.awt.geom.Pat					6,752 bytes		
Class Tracker	java.lang.Integer	2	Mark Current Values			3,296 bytes		
•	java.util.HashMap		Remove Mark			13,056 bytes		
Heap Walker	java.util.HashSet				-	3,232 bytes		
	sun.awt.EventQue		Change Liveness Mode	•		4,608 bytes		
_	bezier.BezierAnim		Sort Classes			2,768 bytes		
CPU Views	java.util.HashMap		Sort Classes	•		7,872 bytes		
	java.lang.Long	ρ	Find	Ctrl+F		3,816 bytes		
	iava util IdentityHa					6 360 hvtes		
Threads	Total from 54 rov	1	Export View	Ctrl+R		442 kB		
	Q- Class View Filt	-	View Settings	Ctrl+T		-		

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.

Vindow	Help			Anima	ted Bezie	r Curve De	emo - JPro	ofiler							_		×
Start Recording	Stop Recordings	Start Tracking	C Run GC	Add Bookmark	1 Export	View Settings	? Help	Show Legend	Stop Memory	Calculate Call Tree	Back	Forward	Show I Heap Wa	In a	nalyze		
		Profiling						— C	Show C	lasses			(Ctrl+Alt	t+C		
Recorde	d allocations	: Live o	bjects at	, 1/10 alloc	ations, A	l classes				ame Grap				Ctrl+Al		<u>Cha</u>	ange
Aggrega	tion level:	🔟 Me	ethods							e Recursic		ioing Calls		Ctrl+Al Ctrl+Alt			•
	🝺 💶 49.4°	% - 109 kE	3 - 1,552 a	alloc. java. Illoc. beziel - 1,230 allo	r.BezierAr	im\$Demo	.paint		Calculat		ces To Se	elected Me	thod (Ctrl+Alt	t+B		
▶ @1	• 00 • 6.5% 00 • 6.	- 14,360 ł 0% - 13,2 - 3,520 by 2 bytes -	oytes - 21 08 bytes - tes - 75 a	144 alloc. lloc. java.av	tier.Bezier java.awt.i wt.Graphi	Anim\$De mage.Buf cs.drawIm	mo.create feredImag										• 2
				-	• P	9 1a	active reco	ording				V	VI #1			🐠 Pro	filing

The histogram of the allocated classes is shown as a call tree analysis [p. 190]. This action also works from other call tree analyses.

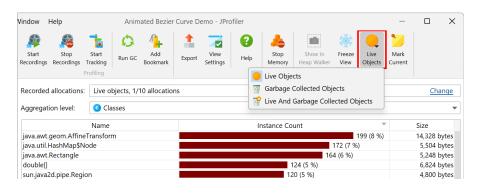
Telemetries	selected call stack	classes have been allocated at th	Reload analysis	× 🖉 🤇
_	Recorded allocations:	Live objects at , 1/10 allocatio	ns, All classes	
Live Memory	Aggregation level:	Methods		
All Objects	Allocation spot:	java.awt.Graphics2D.fill ← bezi	er.BezierAnim\$Demo.drawDemo	← b ► Show more
Recorded Objects		Name	Instance Count	Size
	java.util.HashMap\$N	ode	215 (20 %)	6,880 bytes
Allocation Call Tree	double[]		142 (13 %)	6,968 bytes
Allocation Classes	sun.java2d.loops.Ren	derCache\$Key	93 (8 %)	2,232 byte
Allocation classes	java.awt.geom.Affine	Transform	82 (7 %)	5,904 byte
Allocation Hot Spots	java.awt.geom.Path2	D\$Float\$CopyIterator	70 (6 %)	2,240 byte
a vilocation not spots	java.awt.geom.Point2	2D\$Double	69 (6 %)	2,208 byte
Class Tracker	java.lang.Integer		41 (3 %)	656 byte
	java.util.HashMap		41 (3 %)	1,968 byte
Heap Walker	sun.java2d.loops.Gra	phicsPrimitiveMgr\$PrimitiveSpec	41 (3 %)	656 byte
	int[]		39 (3 %)	40,872 byte:
	java.awt.geom.Point2	2D\$Float	33 (3 %)	792 byte
CPU Views	java.awt.GradientPair	ntContext	32 (3 %)	2,048 byte
CFU VIEWS	java.awt.RenderingHi	ints	32 (3 %)	512 byte
	iava awt deom Rectai	nale2D\$Double	32 (3 %)	1 536 hvte
Threads	Total from 17 rows:		1,049 (100 %)	79,168 bytes
	Q- Class View Filters			•

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 changing 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.

Session Settings		×
Application Settings	Allocation Tree Recording Type Record allocations of: O Live objects	
Call Tree Recording	Live and GCed objects without class resolution Live and GCed objects	
Call Tree Filters	Recording options	
Trigger Settings	Recording rate: O Record all objects 😧 O Sample one out of 10 💠 allocations	
Databases	Record object allocation times 💡	
HTTP, RPC & JEE		
JVM & Custom Probes		
Advanced Settings		
CPU Profiling		
Probes & JEE		
Memory Profiling		
Thread Profiling		
Miscellaneous		
General Settings Copy Setti	ngs From OK Cance	

Next stop: heap walker

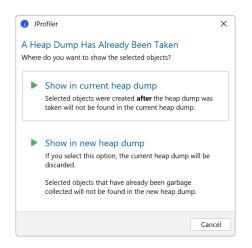
Any more advanced type of question will involve references between objects. For example, the sizes that are displayed in the recorded objects, allocation tree and allocation hot spot views are **shallow sizes**. They just include the memory layout of the class, but not any referenced classes. To see 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.

/indow ⊢	lelp		Animated Bezier Curve Demo - JProfiler											×
Start Recordings	Stop Recordings	Start Tracking Profiling	CRun GC	Add Bookmark	1 Export	View Settings	? Help	Stop Memory	Show In Heap Walker	Unfreeze View	Live Objects	Mark Current		
Recorded	allocations	Live of	ojects, 1/1	iu allocatio	ns								Chi	ange
Aggregatio	on level:	🕝 Cla	sses											•
		Name					li	nstance Cou	unt		Ŧ		Size	
java.awt.g	geom.Affin	eTransfor	n							6	81 (8 %)		49,032 l	bytes
java.awt.F	Rectangle								535	(6 %)			17,120 8	oytes
java.util.H	lashMap\$N	lode							521 (6	5 %)			16,672	oytes
double[]					413 (5 %) 22,624 byte							oytes		
sun.java2d.pipe.Region 410 (5 %) 16,400 byte							oytes							

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.

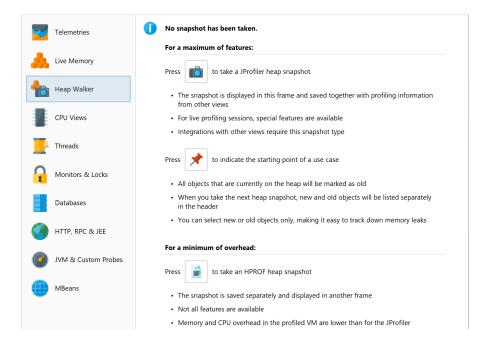
The Heap Walker

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. HotSpot JVMs can create HPROF snapshots, and Eclipse OpenJ9 JVMs provide PHD snapshots.

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. 206].



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.

Classes	2 Allocations	🕂 Biggest Objects	🔀 References	🚫 Time	🏟 Inspections 🔸
Current object set:		t s in 1,428 classes. 670 kB shallow size			
Classes 🔻 Use	e 🔻 🛛 🧴 Gro	oup By Class Loaders		Calcul	ate estimated retained sizes
	Name		Insta	ance Count	▼ Size
byte[]				13,721 (19 %	688 kB
java.lang.String				12,476 (18 %)	299 kB
 Annual and Annual and 				e 60	44010

Initially, you are looking at the "Classes" view which is similar to the "All objects" view in the live memory section [p. 71]. 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 afterward.

🥑 New Object Set		×
A new object set has been created	l. It consists	of 4,465 instances of java.util.HashMap\$Node.
Please choose the initial view for th	e object set:	
Classes Allocations Cumulated allocation tree Biggest objects References		This view mode of the references view shows trees of outgoing references from the single instances in the current object set. You can navigate to other instances in the reference tree.
Outgoing references Time Inspections	▼ ◎ ◎	Do not show this dialog again
		OK Cancel

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.

	Profiling			ew Specific	
 Classes 	P Allocations	📑 Biggest Objects	🔀 References	🕑 Time	🄯 Inspections
urrent object s	et: 4,465 instance	s of java.util.Has	hMap\$Node		
		2 kB shallow size, Cal		eep sizes Use retained o	bjects
Outgoing references	s ▼ Use	Apply filter	Show	n Graph 🔠 👼	
Juigoing references					
• • • • • • • • •	Object	Reta	ined Size	Shallow Size	Allocation Time (h:m:s)
11	Map\$Node (0x8825)		1,508 kB	32 bytes	n/:
	Map\$Node (0x8827)		1,508 kB	32 bytes	n/:
1.1	Map\$Node (0x68e9)		1,507 kB 96,504 bytes	32 bytes	n/i
1	Map\$Node (0x8194) Map\$Node (0x826d)		51,992 bytes	32 bytes 32 bytes	n/i n/i
	Map\$Node (0x862b)		5.344 bytes	32 bytes	n/i
	Map\$Node (0xb57a)		2,752 bytes	32 bytes	n/i
	Map\$Node (0xb58a)		2,752 bytes	32 bytes	n/i
Selection step 2: Cl					
iav	va.util.HashMap\$Node				

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.

Classes	P Allocations	Biggest Objects	🔀 References	🚫 Time	🄯 Inspections
Current object se	-	s in 1,428 classes.			
	1 selection step, 6,	670 kB shallow size			
Classes 🔻	Use 🔻 📄 🥫 Gro	oup By Class Loaders		Calcu	ulate estimated retained size:
	Name		Insta	ance Count	▼ Size
byte[]				13,721 (19	%) 688 kB
java.lang.String				12,476 (18 %)	299 kB
java.util.HashMap\$Noc	le		4,465 (6 %)	142 kB
java.lang.Class			3,304 (4 9	%)	1,057 kB
java.util.concurrent.Cor	ncurrentHashMap\$Nod	le	3,298 (4 9	105 kB	
fer a sub-sub-	1.1				

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.

 Classes 	P Allocations	Biggest Objects	🔀 References	🚫 Time	iği I	Inspections
bject groups:						
Priority 🔺		Class Loader		Instan	ce Count	Shallow Size
1 Defau	ult class loader				68,708	6,666 kl
2 jdk.int	ternal.loader.ClassLoaders	s\$AppClassLoader (0x10df	1		24	3,632 byte
urrent object s	set: 68,708 object	ts in 1,421 classes.				
	3 selection steps, 6	5,666 kB shallow size, <u>Cal</u>	culate retained and deep			ated retained siz
	3 selection steps, 6					lated retained siz
Generation → Classes →	3 selection steps, 6 Use ▼	5,666 kB shallow size, <u>Cal</u>		<u>c</u> ance Count	Calculate estim	
 G Classes ▼ Oyte[] 	3 selection steps, 6 Use ▼	5,666 kB shallow size, <u>Cal</u>		Q	alculate estim	Size
Classes Classes	3 selection steps, 6 Use ▼ SG Gro Name	5,666 kB shallow size, <u>Cal</u>		<u>C</u> ance Count 13,721 12,476 (1	alculate estim	Size 688 kB
Classes Vite[] ava.lang.String ava.util.HashMap\$N	3 selection steps, 6 Use ▼ SG Gro Name	5,666 kB shallow size, <u>Cal</u>	Insta	ance Count 13,721 12,476 (1 6 %)	alculate estim	Size 688 kB 299 kB
Classes v pyte[] ava.lang.String ava.util.HashMap\$N ava.util.concurrent.C	3 selection steps, 6 Use	5,666 kB shallow size, <u>Cal</u>	4,465 (ance Count 13,721 12,476 (1: 6 %) %)	alculate estim	Size 688 kB 299 kB 142 kB
Classes	3 selection steps, 6 Use	5,666 kB shallow size, <u>Cal</u>	4,465 (3,298 (4 9	c ance Count 13,721 12,476 (1 6 %) %)	alculate estim	Size 688 kB 299 kB 142 kB 105 kB

Allocation recording views

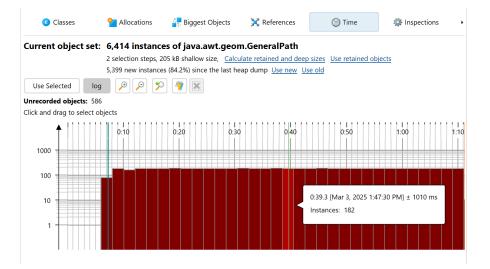
The information where objects have been allocated can be important when narrowing down suspects for a memory leak or when trying to reduce memory consumption. For JProfiler heap snapshots, the "Allocations" view shows the allocation call tree and the allocation hot spots for those objects where allocations 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.

 Classes 	2 Allocations	Biggest Objects	🔀 References	🚫 Time	🄯 Inspections	,
Current object s	et: 68,732 object	s in 1,428 classes.				
	1 selection step, 6,	670 kB shallow size				
Cumulated allocation	n tree 🔻 of 🔟 Met	hods 🔻 🧻 Use	Selected			
		. java.awt.EventDispatchT				
		oc. bezier.BezierAnim\$De				
		alloc. bezier.BezierAnim 7 alloc. java.awt.geom.Ge				
	.8% - 21,936 bytes - 430		inerair ath. Sinit2			
		loc. java.lang.Long.value0	Df			
	% - 32 bytes - 1 alloc. jav					
🔞 0.0% - 6,531 k	B - 66,576 alloc. unrecor	ded objects				
Recorded allocations:	All allocations					

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.



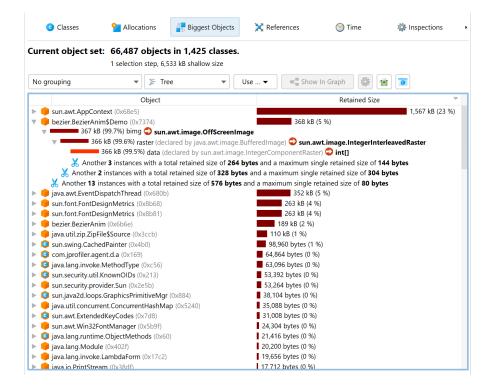
For a more precise selection of a time interval, you can specify a range of bookmarks [p. 45]. All objects between the first and last selected bookmark will then be marked.

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.

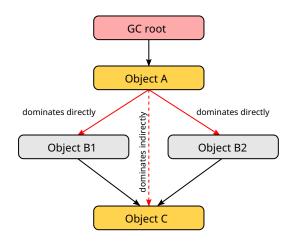
Biggest objects view

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.

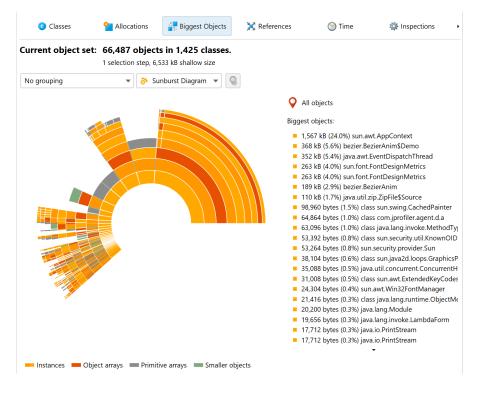
The dominator tree has a built-in cutoff that eliminates all objects that have a retained size that is lower than 0.5% of the retained size of the parent object. This is to avoid excessively long lists of small dominated objects that distract from the important objects. If such a cutoff occurs, a special "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 checkbox 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.

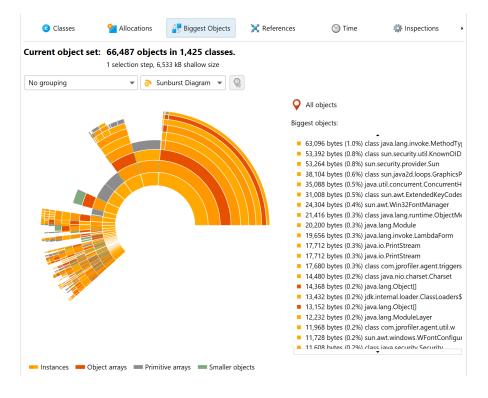
Classes	2 Allocations	Biggest Objects	🔀 References	🚫 Time	🌼 Inspections	,
Current object set:	66,487 object 1 selection step, 6,5					
No grouping	▼ = Tre	ee 🔹	Use 🔻 Show	v In Graph	i	
Group by class loade	er 🕜 Dbject			Retained Size		•
Group by class	0				1,567 kB (23 9	%)
	B74)		368 kB	(5 %)		
🕨 🕨 java.awt.EventDis	patchThread (0x680b))	352 kB	(5 %)		
🕨 🛑 sun.font.FontDesi	gnMetrics (0x8b68)		263 kB (4	%)		
🗆 🕨 💼 sun font FontDesi	anMetrics (0x8b81)		263 kB (4	%)		

The view mode selector above the biggest objects view allows you to switch to a sunburst diagram. The diagram is composed of a series of concentric segmented rings and shows the entire content of the dominator tree up to a maximum depth in one single image. References originate in the innermost ring and propagate towards the outer rim of the circle. This visualization gives you a flattened perspective with high information density that allows you to discover reference patterns and see large primitive and object arrays at a glance through their special color coding.

If the current object set is the entire heap, the total circumference of the circle corresponds to the used heap size. Because the biggest object view only shows objects that retain more than 0.1% of the total heap, this means that a substantial sector will be empty, corresponding to all objects that are not retained by those biggest objects.



Clicking on any ring segment sets a new root for the circle, thereby expanding the maximum depth that you can see in the diagram. Clicking on the hollow center of the diagram restores the previous root. If a new root has been set, the total circumference of the circle corresponds to the retained size of the root object. An empty sector represents the self-size of the root object and additional objects that are not present in the list of biggest retained objects. If the current object set is not the entire heap, the total circumference of the circle corresponds to the sum of all displayed biggest objects and no empty sector is shown.



More information about instances and their immediately retained objects is displayed on the right side of the diagram when you hover over them with the mouse. When the mouse is outside any ring segment, the list on the right side shows the biggest objects in the innermost ring. Hovering over that list highlights the corresponding ring segments and clicking on a list item sets a new root for the diagram. To create a new object set, you can choose from the actions in the context menu, both on the ring segments as well on the list items.

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.

Classes	2 Allocations	Biggest Objects	×	References	🚫 Time	🔅 Inspections 🔸
Current object set:	-	es of java.util.Hasl 42 kB shallow size, <u>Calc</u>			o sizes <u>Use retained</u>	objects
Outgoing references	▼ Use		•	< Show In C	Graph 🔯 📹	
	Object Se	lected Objects		~	Shallow Size	Allocation Time (h:m:s)
🔻 🛑 java.util.HashMap\$1	Node (0x88 Se	lected java.lang.Class Ob	jects	508 kB	32 bytes	n/a
hash = 2028549414 ▼ key ⊃ java.lang.Stri		tained Objects				_
		actStringBuilder) = 0				
		g.AbstractStringBuilder)	= false			
		actStringBuilder) = 10 :tStringBuilder) ᄋ byte[]	1 (0v104)	16)		
 next <i>java.util.Has</i> 			(0x104	16)		
Selection step 2: Class						
	HashMap\$Node					
4,465 instances of java.uti	l.HashMap\$Node					
Selection step 1: All object		taining soft				
68 722 objects in 1 128 cla	0000					

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.

🕖 Heap V	Valker Viev	v Settings				×
General	Classes	Allocations	Biggest Objects	References	Time	Graph
Size Scale	For Cumu	lated Views				
💽 Auto	omatic 🔞		nits 🔿 MB 🔿 k	B Obytes		
Instance	/iews					
🗹 Sho	w object ID)s				
🔽 Sho	w declaring	g class if differ	ent from actual cla	ss 🕜		
Sho	w fields wit	h null values i	n outgoing referen	ces view 📀		
	in neids in					
Instance	block size	: 100	€			
Common	Options –					
🔽 Com	npact repre	sentation of ir	ncoming references	to collections	0	
				ок		Cancel

Beside the simple selection of displayed instances, the outgoing references view has powerful filtering capabilities [p. 210]. 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. 214] has detailed information on this important topic.

Classes	Parallocations	Biggest Objects	🔀 References	🕑 Time	🄯 Inspections
urrent object set:	608 instances	of java.awt.geom	.GeneralPath		
	2 selection steps, 1	9 kB shallow size, Calc	ulate retained and de	ep sizes Use retained o	bjects
ncoming references	▼ Use	🔻 🔤 📢 🕹	Graph 🔯 👹	💿 📋 Show Path	ns To GC Root
	Object	Reta	ined Size 👻	Shallow Size	Allocation Time (h:m:s)
🏮 java.awt.geom.Ge	neralPath (0x73dc)		248 bytes	32 bytes	n/
value of java.u	util.HashMap\$Node	0x73da)			
🔻 🔇 element of	java.util.HashMap\$N	lode[] (0xcb72)			
🔻 🔇 table o	f java.util.HashMap (0x73d9)			
	Map of bezier.Bezie				
👻 😡 1	arget of sun.awt.wir	dows.WPanelPeer (decla	red by sun.awt.windo	ows.WObjectPeer) (0x6b6	5f)
(🕽 🖞 JNI global refe	rence		-	
😑 java.awt.geom.Ge	neralPath (0x73df)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x73e8)		248 bytes	32 bytes	n/
😑 java.awt.geom.Ge	neralPath (0x73e5)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73e2)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73f1)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73ee)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73eb)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73f4)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73fd)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73fa)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x73f7)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x7400)		248 bytes	32 bytes	n/
🛛 🛑 java.awt.geom.Ge	neralPath (0x7403)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x7406)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x7409)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x740c)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x740f)		248 bytes	32 bytes	n/
🏮 java.awt.geom.Ge	neralPath (0x7415)		248 bytes	32 bytes	n/
	neralPath (0x7412)		248 bytes	32 bytes	

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 ⁽³⁾ special node is inserted that shows the original instances plus the instances

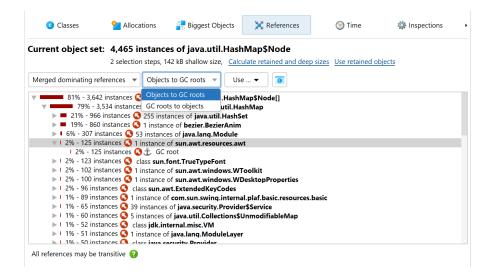
from these class-recursive references. This mechanism automatically collapses internal reference chains in common data structures, such as in a linked list.

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.

Classes	P Allocations	Biggest Objects	🔀 References	🚫 Time	🌼 Inspections
Current object se	t: 4,465 instanc	es of java.util.Hasl	hMap\$Node		
	2 selection steps, 1	42 kB shallow size, Calo	ulate retained and dee	p sizes Use retained o	bjects
Merged incoming refe	erences 💌 Aggree	gate by class 🔻 🛛 Use	e 🔻 🧕 🧕	_	
v 11 81% - 3,64	2 instances 🔇 435 ins	tances of java.util	Referenced Objects		
	3,534 instances 🔇 434		Inreferenced Objects		
	6 instances 🔇 255 inst	ances of java.util.	Reference Holders		
) instances 🔇 1 instar	ce of bezier.Bezie			
		tances of java.lang.Obje			
		tances of java.util.Hash			
		tance of sun.awt.image			
		tance of sun.awt.windo			
		tance of bezier.BezierA			
		tance of javax.swing.JR			
		tance of bezier.BezierA			
		tance of java.awt.Light			
- 60/ 207 in	oou instances 🕥 T instances	stance of java.awt.Borde es of java.lang.Module	rLayout		
		es of java.iang.iviodule es of java.util.HashMap	t Vau Cat		
		es of java.util.HashMap			
		of sun.awt.resources.av			
	istances 🔇 class sun.		**		
		of and and mind and M	Toollia		

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.

Classes	Parallocations	Biggest Objects	🔀 References	🚫 Time	🕸 Inspections
Current object set: Available Inspections:	-	ts in 1,425 classes. 533 kB shallow size			
 Duplicate objects Duplicate strings Duplicate primitive Duplicate arrays Collections & Arr Reference & field Weak references Stack references Thread locals Classes & Class I Custom inspection 	wrappers ays 1 analysis oaders	After the heap wa analyze I Note: If n set, the i Configura Minimun Status	licate java.lang : inspection is calcu ker views where yo the corresponding s no java.lang.St nspection will retur tion n length:	.String objects in the cur lated, you will see a statistic u can select each duplicate string objects separately. tring objects are containe n the empty object set. 20 Calculate inspection and c	ts table at the top of all string value and d in the current object

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.

💿 Classes 🛛 💡 Al	llocations 📑 Biggest	Objects 🔀 References	s 🕑 Tim	ne 🄅	Inspections
Dbject groups:					
Priority 🔺	Duplicate String	Ins	tance Count S	String Length	Total Size
1 makeConcatWithC	makeConcatWithConstants			23	782 byte
2 C:\Users\ingo\proj	C:\Users\ingo\projects\jprofiler\dist\bin			41	164 byte
3 file:///C:/Users/ing	file:///C:/Users/ingo/projects/jprofiler/dist/demo/bezier/classes/			66	132 byte
	iects\jprofiler\dist\bin\windo		2	66	132 byte
	m32\WindowsPowerShell\v		3	42	126 byte
	ojects/jprofiler/dist/demo/be		2	59	118 byte
	ects\jprofiler\dist\demo\be		2	57	114 byte
	ijbrsdk-21.0.5-windows-x64		2	56	112 byte
	▼ Use ▼ Ap	ow size, <u>Calculate retained a</u>	In Graph	*	
					n Time (h:m:s)
Outgoing references	ct	ply filter 💌 🛋 Show	In Graph 🙀		
Outgoing references Object	ct ["makeConcatWithCon	ply filter ▼ Show	In Graph Shallow Size 24 B	Allocatio	n
Outgoing references Objec • • • java.lang.String (0xb810) [ct ["makeConcatWithCon ["makeConcatWithCon	Ply filter Retained Size 64 bytes	In Graph Shallow Size	Allocatio	n
Outgoing references Objec java.lang.String (0xb810) [java.lang.String (0xb81a) [java.lang.String (0xb81a) [java.lang.String (0xb822) [rt "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Retained Size 64 bytes 64 bytes	In Graph 🐲 Shallow Size 24 B 24 B 24 B	Allocatio bytes bytes	n n n
Dutgoing references Objec java.lang.String (0xb810) [java.lang.String (0xb81a) [java.lang.String (0xb822) [ct ("makeConcatWithCon ("makeConcatWithCon ("makeConcatWithCon ("makeConcatWithCon	Ply filter Retained Size 64 bytes 64 bytes 64 bytes 64 bytes 64 bytes	In Graph Shallow Size 24 B 24 B 24 B 24 B	Allocatio bytes bytes bytes	n n n
Outgoing references java.lang.String (0xb810) [java.lang.String (0xb81a) [java.lang.String (0xb82) [java.lang.String (0xb82) [java.lang.String (0xb82) [ct "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Retained Size 64 bytes 64 bytes 64 bytes 64 bytes 64 bytes	In Graph Shallow Size	Allocatio oytes oytes oytes oytes	n n n n
Outgoing references bigava.lang.String (0xb810) [bigava.lang.String (0xb81a) [bigava.lang.String (0xb822) [bigava.lang.String (0xb822) [bigava.lang.String (0xb822) [bigava.lang.String (0xb82) [bigava.lan	t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Retained Size 64 bytes 64 bytes	In Graph Size Shallow Size 24 I 24 I 24 I 24 I 24 I 24 I 24 I	Allocatio oytes oytes oytes oytes oytes oytes	n Time (h:m:s) ກ, ກ, ກ, ກ, ກ, ກ, ກ,
Outgoing references	t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size 64 bytes 64 by	In Graph Size Shallow Size 24 t 24 t 24 t 24 t 24 t 24 t 24 t 24 t	Allocatio oytes oytes oytes oytes oytes oytes oytes	n n n n n n n
Dutgoing references Dutgoing references Java.lang.String (0xb810) [Java.lang.String (0xb822) [Java.lang.String (0xb822) [Java.lang.String (0xb823) [Java.lang.String (0xb883) [Java.lang.String (0xb883) [Java.lang.String (0xb883) [Java.lang.String (0xb883) [t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size 64 bytes 64 by	In Graph 5 Shallow Size 24 I 24 I 24 I 24 I 24 I 24 I 24 I 24 I	Allocatio oytes oytes oytes oytes oytes oytes oytes oytes oytes	n n n n n n n n
Dutgoing references Dutgoing references Dutgoing Ang. String (0xb810) [Java.lang. String (0xb812) [Java.lang. String (0xb822) [Java.lang. String (0xb823) [Java.lang. String (0xb823) [Java.lang. String (0xb824) [Java.lang. String (0xb824) [Java.lang. String (0xb824) [Java.lang. String (0xb824) [t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size G4 bytes G4 by	In Graph 5 Shallow Size 24 24 24 24 24 24 24 24 24 24 24 24 24	Allocatio oytes oytes oytes oytes oytes oytes oytes oytes oytes	n, n, n, n, n,
Dutgoing references java.lang.String (0xb810) [java.lang.String (0xb81a) [java.lang.String (0xb82a) [java.lang.String (0xb82a) [java.lang.String (0xb82a) [java.lang.String (0xb82b) [java.lang.String (0xb8bf) [java.lang.String (0xb8bf) [java.lang.String (0xb8bf) [t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size G4 bytes G4 by	In Graph 55 Shallow Size 24 t 24 t 24 t 24 t 24 t 24 t 24 t 24 t	Allocatio oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes	n, n, n, n, n, n, n, n, n, n,
Outgoing references Object java.lang.String (0xb810) [java.lang.String (0xb81a) [java.lang.String (0xb82a) [java.lang.String (0xb82a) [java.lang.String (0xb82a) [java.lang.String (0xb82a) [java.lang.String (0xb82b) [t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size 64 bytes 64 by	In Graph 55 Shallow Size 24 1 24 1 24 1 24 1 24 1 24 1 24 1 24 1	Allocatio Allocatio oytes o	, , , , , , , , , , , , , , , , , , ,
Dutgoing references Dutgoing references Dutgoing ava.lang.String (0xb810) [Java.lang.String (0xb812) [Java.lang.String (0xb822) [Java.lang.String (0xb824) [t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size 64 bytes 64 by	In Graph 5 Shallow Size 24 I 24 I	Allocatio oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes oytes	ת ת ת ת ת ת ת
Outgoing references Outgoing references Java.lang.String (Xxb810) [Java.lang.String (Xxb820) Java.lang.String (Xxb822) [Java.lang.String (Xxb822) [Java.lang.String (Xxb820) [Java.	t "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon "makeConcatWithCon	Ply filter Retained Size G4 bytes G4 by	In Graph 5 Shallow Size 24 24 24 24 24 24 24 24 24 24	Allocatio opytes opytes opytes opytes opytes opytes opytes opytes opytes opytes opytes opytes opytes opytes opytes	, , , , , , , , , , , , , , , , , , ,

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.

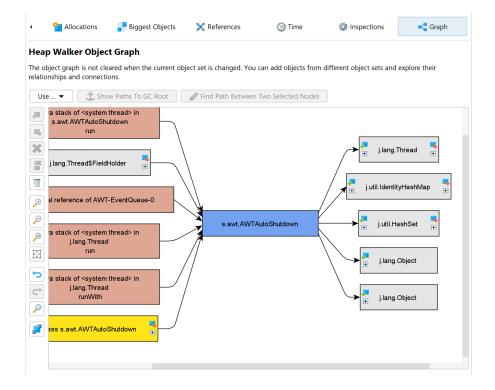
Heap walker graph

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.

Classes	Participation Allocations	Biggest Objects	🔀 References	🚫 Time	🄯 Inspections
Current object set	: 608 instances of	of java.awt.geom.G	eneralPath		
	2 selection steps, 19	kB shallow size, Calculat	e retained and deep	sizes Use retained o	bjects
Incoming references	▼ Use	▼ Show In Gra	ph 🏟 📹	🚺 🧘 Show Pat	hs To GC Root
	Object	Retained	d Size 🔻	Shallow Size	Allocation Time (h:m:s)
🕨 🏓 java.awt.geom.Ge	eneralPath (0x73dc)		248 bytes	32 bytes	n/a
java.awt.geom.Ge	eneralPath (0x73df)		248 bytes	32 bytes	n/a
java.awt.geom.Ge	eneralPath (0x73e8)		248 bytes	32 bytes	n/a
java.awt.geom.Ge	eneralPath (0x73e5)		248 bytes	32 bytes	n/a
java.awt.geom.Ge	eneralPath (0x73e2)		248 bytes	32 bytes	n/a
java.awt.geom.Ge	eneralPath (0x73f1)		248 bytes	32 bytes	n/a
🕨 🛑 java.awt.geom.Ge	eneralPath (0x73ee)		248 bytes	32 bytes	n/a
	manalBath (0.70 als)		240 butor	22 hitor	- 1-

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. 214] 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.

Path Search Options	×
Search Directions	
Search for directed path from first to second object	
Search for directed path from second to first object	
Search for undirected path ??	
Options	
This search follows soft , as per the initial retention setting for the heap dump.	
\square Also follow weak, phantom and finalizer for this search $\ argeta$	
Stop search at classes 3	
ОК Салс	el

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.

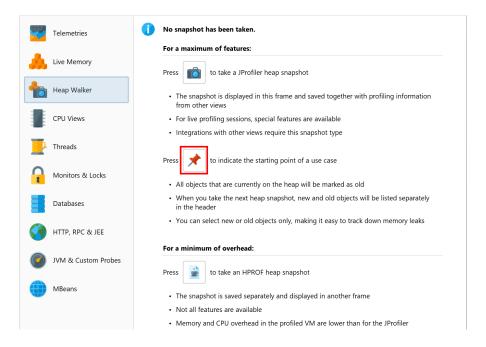
In addition, the heap walker performs a garbage collection and removes weakly referenced objects, except for soft references. This is usually desirable because weakly referenced objects are distracting when looking for memory leaks where only strongly referenced objects are relevant. However, in those 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 choose which of them should be sufficient for retaining an object in the heap snapshot.

Heap Snapshot Options			×
Select recorded objects Initially, the heap walker will show or memory view section.	nly those objects that have been rec	orded in the dynamic	
✓ Perform full GC in heap snapshot	0		
Retain objects held by references	of type soft		
Show Overhead Options -	✓ soft weak phantom finalizer	OK Cance	1

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.



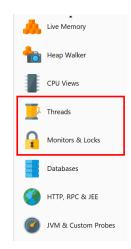
If there was a previous heap snapshot or a mark heap invocation, the title area of the heap walker shows the new instance count and two links titled *Use new* and *Use old* that allow you to select either the instances that have been allocated since that point in time, or the surviving instances that 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.

Classes	Parallocations	Biggest Objects	🔀 References	🕑 Time	🄯 Inspe	ections
Current object se	t: 107,534 obje	cts in 1,436 classes.				
	1 selection step, 8,	788 kB shallow size				
		ces (36.4%) since the last h	ean dumn Use new U	ise old		
ⓒ Classes ▼		oup By Class Loaders			culate estimated	retained siz
Classes						I retained siz
	Use 🔻 🛛 🧕 Gro			Cal	▼ 5	
byte[]	Use 🔻 🛛 🧕 Gro			Cal	▼ 5	Size
oyte[] ava.lang.String	Use 👻 🧴 🧔 Gro			Cal tance Count 22,293 (2	▼ 5	Size 1,025 kB
Classes Classes byte[] java.lang.String java.util.HashMap\$Nod java.lang.Long	Use 👻 🧴 🧔 Gro			Cal tance Count 22,293 (2 15,634 (14 %) 1,060 (10 %)	▼ 5	5ize 1,025 kB 375 kB

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, by name or by their thread group and can be filtered by name. You can also rearrange the order of threads yourself via drag and drop. 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

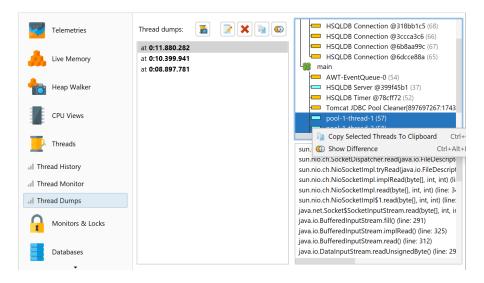
in the table. At the bottom, the stack trace of the creating thread is shown. For performance reasons, no actual stack trace is requested from the JVM, but the current information from CPU recording is used. This means that the stack traces will only show those classes that satisfy the filter settings for call tree collection.

	Name	ID 🕜	Group	Start Time 🔺	Creating Thread	Status
Telemetries	HSQLDB Server	37	main	0:00.306	main (1) [main]	🔲 Net I/O
	HSQLDB Timer	52	main	0:00.753	HSQLDB Server @39	💻 Waiting
Live Memory	AWT-EventQue	54	main	0:01.393	main (1) [main]	💻 Waiting
Eive Memory	pool-1-thread-1	57	main	0:01.513	<not recorded=""></not>	💻 Net I/O
	pool-1-thread-2	58	main	0:01.514	<not recorded=""></not>	🔲 Net I/O
Heap Walker	pool-1-thread-3	59	main	0:01.514	<not recorded=""></not>	🔲 Net I/O
	pool-1-thread-4	60	main	0:01.514	<not recorded=""></not>	🔲 Net I/O
_	pool-1-thread-5	61	main	0:01.514	<not recorded=""></not>	🔲 Net I/O
CPU Views	Tomcat JDBC P	63	main	0:01.520	pool-1-thread-1 (57)	💻 Waiting
	HSQLDB Conne	64	HSQLDB Conne	0:01.551	HSQLDB Server @39	💻 Waiting
	HSQLDB Conne	65	HSQLDB Conne	0:01.674	HSQLDB Server @39	💻 Waiting
Threads	HSQLDB Conne	66	HSQLDB Conne	0:01.778	HSQLDB Server @39	💻 Waiting
-	HSQLDB Conne	67	HSQLDB Conne	0:01.881	HSQLDB Server @39	💻 Waiting
Thread History	HSQLDB Conne	68	HSQLDB Conne	0:01.983	HSQLDB Server @39	💻 Waiting
Thread Monitor	Filtered stack trace	or thread crea	tion: 🙆			
Thread Dumps			•••••			
Monitors & Locks			No stack trac	e was recorded		
Databases						

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.

	×
Enable CPU profiling Auto-Tuning For Instrumentation	
A method is an overhead hot spot and will be suggested for inclusion into the list of	
ignored methods, if both of the following conditions are true: 1. The total time of the method is more than 10 + per mille of the entire total time	
2. The average time of the method is less than $100~\blacklozenge~\mu s$	
Auto-tuning is only performed if the method call recording type is set to "Instrumentation" on the <u>method call recording tab</u> .	
Call Tree Recording Options CPU times for instrumentation: Elapsed times CPU times for instrumentation:	
 Instrument native methods Thread resolution for async sampling (2) 	
Exceptional Method Run Recording	
Call Tree Splitting Maximum number of splits: 128 🗢 😧	
	Auto-Tuning For Instrumentation ✓ Enable auto-tuning 𝔅 A method is an overhead hot spot and will be suggested for inclusion into the list of ignored methods, if both of the following conditions are true: 1. The total time of the method is more than 10 ♀ per mille of the entire total time 2. The average time of the method is less than 100 ♀ µs Auto-tuning is only performed if the method call recording type is set to "Instrumentation" on the method call recording tab. Call Tree Recording Options CPU times for instrumentation: ● Elapsed times 𝔅 ● Estimated CPU times 𝔅 ✓ Instrument native methods 𝔅 ✓ Thread resolution for async sampling 𝔅 Exceptional Method Run Recording Maximum number of separately recorded method runs: ⑤ ♀ 𝔅 Time type for determining exceptional method runs: Call Tree Splitting

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.

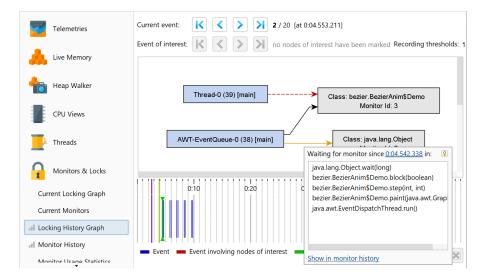


Thread dumps can also be taken with the "Trigger thread dump" trigger action or via the API.

Analyzing locking situations

Every Java object has an associated monitor that can be 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.

Start Stop Recordings Recordings	Start Tracking Profiling	Add Bookmark	Export Settings	Help View Specific	Freeze View Heap Walker	
All types 🔹	Filter in all text	columns 💌	Q• Filter			•
Time 🔺	Duration	Туре	Monitor ID	Monitor Class	Waiting Thread	Owning Thread
0:04.542 [Mar 3,	200 ms 🗖	Waiting	2	java.lang.Object	AWT-EventQueue-0 (38)	
0:04.553 [Mar 3,	190 ms	Blocked	3	bezier.BezierAnim\$D	Thread-0 (39) [main]	AWT-EventQueue-0 (38)
0:05.810 [Mar 3,	200 ms 🗖	Waiting	2	java.lang.Object	AWT-EventQueue-0 (38)	
0:05.821 [Mar 3,	190 ms 💻	Blocked	3	bezier.BezierAnim\$D	Thread-0 (39) [main]	AWT-EventQueue-0 (38)
0:07.077 [Mar 3,	200 ms 🗖	Waiting	2	java.lang.Object	AWT-EventQueue-0 (38)	
0:07.088 [Mar 3,	190 ms 💻	Blocked	3	bezier.BezierAnim\$D	Thread-0 (39) [main]	AWT-EventQueue-0 (38)
Tatal from 6 vo	1 172					
Total from 6 ro	1,172 ms					
Total from 6 ro	1,172 ms			Recording thresho	lds: 1,000 µs blocking / 10	0,000 µs waiting [Change
Total from 6 ro		0		-	lds: 1,000 µs blocking / 10 ace for owning thread:	10,000 µs waiting [Change

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.

Start Recordings	Stop Recordings	Start Tracking Profiling	CRun GC	Add Bookmark	1 Export	View Settings	? Help	Stop Monitors View Specifi	Freeze View	Show In Heap Walker	Show In Graph		
All types	•	Filter in a	ll text co	olumns 🔻	Q . Fi	lter							•
Time	e 🔺	Duration		Туре	Monite	or ID	Monito	or Class	W	/aiting Thread		Owning Thread	
0:04.542 [[Mar 3,	200	ms 💻	Waiting		2 j	ava.lang.Ol	oject	AWT-E	/entQueue-0 (38)		
0:04.553 [[Mar 3,	190	ms 💻	Blocked		3 k	bezier.Bezie	rAnim\$D	Thread	0 (39) [main]	1	AWT-EventQueue-0 (3	8)
0:05.810 [[Mar 3,	200	ms 💻	Waiting		2 ji	ava.lang.Ol	oject	AWT-E	/entQueue-0 (38)		

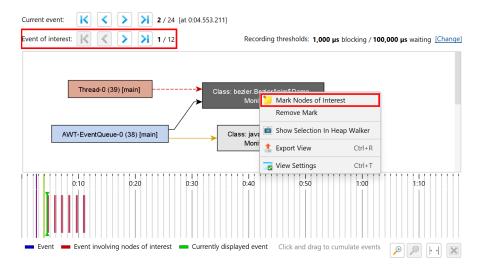
Limiting the events of interest

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.

Current event: 🔀 💙 🔰 2 / 44 [a	at 0:04.553.211]
Event of interest: 🔣 💙 🔰 1 / 22	Recording thresholds: 1,000 µs blocking / 100,000 µs waiting [Change
Thread-0 (39) [main]	Monitor History Graph View Settings X
	Recording Time line
AWT-EventQueue-0 (38) [main]	Recording Thresholds
	Monitor blocking threshold: 1,000 🜩 µs
	Monitor waiting threshold: 100,000 🗢 µs
0:10 0:20	All events with a duration that is lower than the configured threshold will be discarded.
	Warning: If you lower the thresholds, more data will be recorded. Please note that the associated memory overhead grows linearly in time.
Event Event involving nodes of interest	Currently displayed event Click and drag to cumulate 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.



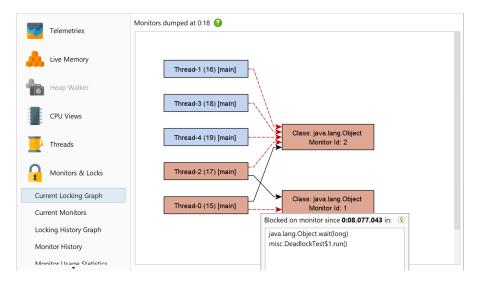
Another strategy to reduce the number of events that need your attention is to cumulate locking situations. In the locking history graph, there is a time line at the bottom that shows all recorded events. Clicking and dragging in it selects a time range and data from all contained events is shown in 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

The "Current locking graph" and the "Current monitors" views operate on a "monitor dump" that is triggered with an action in the JProfiler UI. With a monitor dump, you can inspect events that are still in progress. This includes deadlocks which are events that never finish and cannot be shown in the history views.

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.

Taking a new monitor dump will replace the data in the both views. You can also trigger monitor dumps with the "Trigger monitor dump" trigger action or via the API.



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.

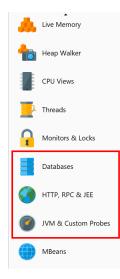
Start Stop Save Session Center Snapshot Settings Session	Start Stop Start Recordings Recordings Tracking Profiling Profiling	Run GC Add Bookma	rk Export Settings	Help Calculate Statistics View Specific	
Telemetries	Monitor Usage Statistics Gro	ouped by Monito	rs		
Telemenes	Monitors	Block Count	Block Duration	Wait Count	Wait Duration
	bezier.BezierAnim\$Demo	13	2,470 ms	0	0 µ
Live Memory	java.lang.Object (id: 2)	0	0 µs	13	2,607 m
	java.util.concurrent.locks	0	0 µs	1,262	11,548 m
Heap Walker	java.util.concurrent.locks	0	0 µs	4	66 µ
CPU Views		Maritan Usana Ch	<i>tistica Octiona</i>	~	
Threads		Monitor Usage Sta	itistics Options	×	
-	Sele	ct the desired mor	itor usage statistics: -	×	
Threads	Sele		itor usage statistics: -	×	
Monitors & Locks	Sele	ct the desired mor	nitor usage statistics: – rs	×	
Threads	Sele	ct the desired mor Group by Monito	nitor usage statistics: – rs s	×	
Monitors & Locks	Sele	ct the desired mor Group by Monito Group by Thread	nitor usage statistics: - rs s of monitors		
Threads Monitors & Locks Current Locking Graph Current Monitors	Sele	ct the desired mor Group by Monito Group by Thread	nitor usage statistics: – rs s	Cancel	

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. 163] for it. Some libraries, containers or database drivers may ship with their own embedded probe [p. 168] 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. 27] 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.

0:02.181 [Mar 3, 20 Co 0:02.185 [Mar 3, 20 St	Filter in Event Type onnection op	all text columns Duration			
0:02.181 [Mar 3, 20 C 0:02.185 [Mar 3, 20 St		Duration	e		
0:02.181 [Mar 3, 20 C 0:02.185 [Mar 3, 20 St			Connection ID	Description	Thread
		0 µs 2		jdbc:demo://remote_host/test	RMI TCP Connectio.
		860 ms 2		SELECT i.id, i.availability, i.name FRO.	RMI TCP Connectio.
0:02.701 [Mar 3, 20 C		0 µs 1		jdbc:demo://remote_host/test	Servlet request simu
	repared state	141 ms 1		SELECT * FROM ORDER O WHERE O	
0:03.640 [Mar 3, 20 P	repared state	62,083 µs 1		INSERT INTO CUSTOMER (ID, NAME	
0:03.752 [Mar 3, 20 💻 Pi		59,534 µs 1		INSERT INTO ORDER (ID, NAME, OP	
0:03.838 [Mar 3, 20 💶 St		720 ms 2		SELECT i.id, i.availability, i.name FRO.	
0:03.881 [Mar 3, 20 💻 Pr	repared state	59,601 µs 1		INSERT INTO ORDER_CUSTOMER (O	
0:04.111 [Mar 3, 20 C		0 µs 3		jdbc:demo://remote_host/test	RMI TCP Connectio.
0:04.111 [Mar 3, 20 💻 St	tatement exe	659 ms 3		SELECT i.id, i.availability, i.name FRO.	RMI TCP Connectio.
0.05 282 [Mar 2 20	tatement eve	860 ms 2		SELECT i id i availability i name ERO	RMI TCP Connectio
Total from 115 rows:		42,617 ms			

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 row that shows the total number of displayed events and 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 text field columns, but you can choose a specific filter column from the drop-down before the text field. Filter options are also available from the context menu, for example, to filter all events with a duration larger than that of the selected event.

All types	▼ Filter in a	ll text columns 💌	Q . ▼ Filter			
Start Tim	e 🔺 Event Type	Duration Conn	ection ID		Description	Thread
0:02.181 [N	1ar 3, 20 🗖 Connection op	0 µs 2		jdbc:demo:	//remote_host/test	RMI TCP Connectio
0:02.185	ta= 2 20 - Statement ava	060 mc 7		SELECT i.id	, i.availability, i.name FRO	RMI TCP Connectio
0:02.701	Show Connections For Selected	Events Ctrl+Alt+C		jdbc:demo:	//remote_host/test	Servlet request simu
0:02.717	Filter Equals This		Dur	ation	DM ORDER O WHERE O. CUSTOMER (ID, NAME.	
0:03.752	Filter Greater Than This		Con	nection ID	ORDER (ID, NAME, OP.	
0:03.838	Filter Less Than This		Des	cription	availability, i.name FRO	RMI TCP Connectio
0:03.881	The Less that this				ORDER_CUSTOMER (O.	
0:04.111	a Stop Recording Probe Events		Thre	ad	remote_host/test	RMI TCP Connectio
0:04.111	Sort Events			SELECT i.id	i.availability, i.name FRO	RMI TCP Connectio
0.02 585	~		-	SELECT LIN	i availability i name FRO	RMI TCP Connectio
Total from 🍃	🏳 Find	Ctrl+F				
🗸 Selecti	文 Show Row Details	Ctrl+Alt+I				
Stack trace:	靠 Export View	Ctrl+R				
java.sql.Sta 🗕	🛃 View Settings	Ctrl+T				

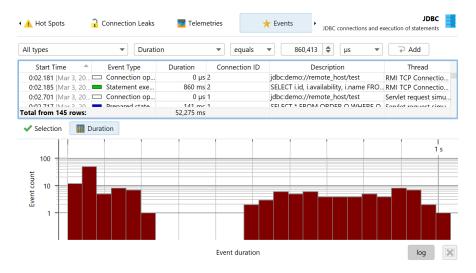
Other probe views also offer options to filter probe events: In the probe telemetries view you can select a time range, in the probe call tree view you can filter events from the selected call stack, the probe hot spots view offers a probe event filter based on the selected back trace or

hot spot and the control object and time line views offer actions to filter probe events for the selected control object.

Stack traces of selected probe events are shown at the bottom. If multiple probe events are selected, the stack traces are cumulated and shown either as a call tree, as probe hot spots with back traces or as CPU hot spots with back traces.

All types	 Duration 	n	▼ equals	▼ 860,413 \$ µs ▼	Add
Start Time 🔺	Event Type	Duration	Connection ID	Description	Thread
0:02.181 [Mar 3, 20	Connection op	0 µs 2		jdbc:demo://remote_host/test	RMI TCP Connectio
0:02.185 [Mar 3, 20	Statement exe	860 ms 2		SELECT i.id, i.availability, i.name FRO	RMI TCP Connectio
0:02.701 [Mar 3, 20	Connection op	0 µs 1		jdbc:demo://remote_host/test	Servlet request simi
0:02.717 [Mar 3, 20	Prepared state	141 ms 1		SELECT * FROM ORDER O WHERE O.	.Servlet request simi
0:03.640 [Mar 3, 20	Prepared state	62,083 µs 1		INSERT INTO CUSTOMER (ID, NAME	Servlet request sim
0:03.752 [Mar 3, 20	 Prepared state 	59,534 µs 1		INSERT INTO ORDER (ID, NAME, OP.	Servlet request sim
0:03.838 [Mar 3, 20	Statement exe	720 ms 2		SELECT i.id, i.availability, i.name FRO.	RMI TCP Connectio
0:03.881 [Mar 3, 20	Prepared state	59,601 µs 1		INSERT INTO ORDER_CUSTOMER (O	Servlet request sim
0:04.111 [Mar 3, 20	Connection op	0 µs 3		jdbc:demo://remote_host/test	RMI TCP Connectio
otal from 133 rows:	Chatamant and	48,971 ms		CELECT 114 Canadiability Casara EDO	DM TCD C
Probe call tree from select	ration cted events 0 ms - 2 evt. java.util. - 3 evt. com.ejt.demo.:			orker.run	

Next to the stack trace views, histogram views for event durations and optionally for recorded throughput are shown. You can select a duration range in these histograms with the mouse in order to filter probe events in the table above.



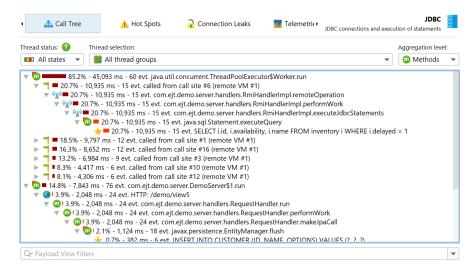
Probes can record different kinds of activities and associate an event type with their probe events. For example, the JDBC probe shows statements, prepared statements and batch executions as event types with different colors.

🔥 Hot Spots 🛛 🔒 Connec	tion Leaks	🕎 Telem	etries	🛨 Events 🔹	JDBC connections and ex	JDBC xecution of statements
All types 🔹	Filter in	all text column	S ▼ Q Filter			
Connection opened	ype	Duration	Connection ID	De	escription	Thread
Connection closed	tion op	0 µs	2	jdbc:demo://rer	note_host/test	RMI TCP Connectio.
Statement execution	ent exe	860 ms	2	SELECT i.id, i.av	ailability, i.name FRO	RMI TCP Connectio.
Prepared statement execution	tion op	0 µs	1	jdbc:demo://rer	note_host/test	Servlet request simu
Batch execution	d state	141 ms	1	SELECT * FROM	ORDER O WHERE O.	Servlet request simu
Select All Select None	d state	62,083 µs	1	INSERT INTO C	USTOMER (ID, NAME.	Servlet request simu
Select All Select None	d state	59,534 µs	1	INSERT INTO O	RDER (ID, NAME, OP.	Servlet request simi
OK Cancel	ent exe	720 ms	2	SELECT i.id, i.av	ailability, i.name FRO	RMI TCP Connectio
0.03.001 [IVIAL 3, 20] FTEPA	d state	59,601 µs	1	INSERT INTO O	RDER_CUSTOMER (O.	Servlet request sime
0.04 111 BAse 2: 20 CORRE		0.116	2	idhc:demo://rer	note host/test	DMI TCD Connectio

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.

Ø Session Settings				×
Application Settings	Payload Options			
Call Tree Recording	Payloads are consolidated if there are too many different strin call tree, payloads are consolidated into an [Earlier calls] node.		otatin	g payloads into the
	Maximum number of distinct payloads for probe hot spots:	16,384	\$	0
Call Tree Filters	Maximum number of annotated payloads per call stack: Cutoff payload strings after:	50 8,192		😯 characters 😱
Trigger Settings	Record payload call stacks in sampling mode	0,152	•	
Databases	Retain call stacks when consolidating hot spots 💡			
HTTP, RPC & JEE	Event Options Maximum number of recorded events: 5,000 🔶	9		
JVM & Custom Probes	JEE/Spring Options			
Advanced Settings	Show request URLs without a recorded call stack ?			
CPU Profiling	Note: Probes are individually configured on the "Database pro	he settings" "	нттр	RPC & IFF probe
Probes & JEE	settings" and "JVM & custom probe settings" tabs. The			
Memory Profiling				
Thread Profiling				
Miscellaneous				
General Settings Copy Sett	ings From		0	K Cancel

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. 53]. 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.

read status: (7 Threa	d selection:						Aggregati	on level:
All state:	s 🔻 🎇	All thread groups					•	🔟 Met	hods
		Hot Spot			Time	-	Average Ti	ime	Events
🔻 🛅 I 5.2	2% - 3,019 ms	DER O WHERE O.DATE : - 19 hot spot evt. javax.	persistence.TypedQue	3,019 ms ery.getResultl	ist		1	158 ms	1
V	 □ I 5.2% - 3,(▼ □ I 5.2% - ▼ □ I 5.2 ▶ ③ ▶ ③ ▶ ③ ▶ ⑤ ▶ ⑤ ▶ ⑤ ▶ ⑤ 	119 ms - 19 hot spot evt 3,019 ms - 19 hot spot % - 3,019 ms - 19 hot sp 1.8% - 1,027 ms - 6 hot 1.6% - 923 ms - 6 hot s 0.7% - 421 ms - 3 hot s 0.6% - 327 ms - 2 hot s 0.5% - 318 ms - 2 hot s	evt. com.ejt.demo.ser pot evt. com.ejt.demo spot evt. HTTP: /dem pot evt. HTTP: /demo, pot evt. HTTP: /demo, pot evt. HTTP: /demo, pot evt. HTTP: /demo,	handlers.Rec ver.handlers. .server.handl o/view2 /view5 /view4 /view3 /view1	juestHandler.ma RequestHandle ers.RequestHan	akeJpaCall r.performW	fork		
SELEC	 □ 1 5.2% - 3,(○ □ 1 5.2% - ○ □ 1 5.2 ○ □ 1 5.2 ○ ○ ○	119 ms - 19 hot spot evt 3,019 ms - 19 hot spot % - 3,019 ms - 19 hot sp 1.8% - 1,027 ms - 6 hot s 1.6% - 923 ms - 6 hot s 0.7% - 421 ms - 3 hot s 0.6% - 327 ms - 2 hot s 0.5% - 318 ms - 2 hot s * 0.quantity) FROM cus	. com ejt.demo.server. evt. com ejt.demo.ser pot evt. com ejt.demo spot evt. HTTP: /demo, pot evt. HTTP: /demo,	handlers.Rec ver.handlers. server.handl o/view2 /view5 /view4 /view3 /view1 r] 2,173 ms	juestHandler.ma RequestHandle ers.RequestHan (3 %)	akeJpaCall r.performW	fork	724 ms	
SELEC	1 5.2% - 3,(▼ 1 5.2% - ▼ 1 5.2 ↔ ▼ 1 5.2 ■ 0 ■ 0 ■ 0 ■ 0 ■ 0 ■ 0 ■ 0 ■ 0 ■ 0 ■ 0	119 ms - 19 hot spot evt 3,019 ms - 19 hot spot % - 3,019 ms - 19 hot spot 1.6% - 9,23 ms - 6 hot s 0,7% - 421 ms - 3 hot s 0,6% - 327 ms - 2 hot s 0,5% - 318 ms - 2 hot s < o,quantity) FROM cus R (ID, NAME, OPTIONS)	. com.ejt.demo.server. evt. com.ejt.demo.ser pot evt. com.ejt.demo spot evt. HTTP: /demo, pot evt. HTTP: /demo, pot evt. HTTP: /demo, pot evt. HTTP: /demo, tomers c LEFT JOIN o VALUES (?, ?, ?)	handlers.Rec ver.handlers. server.handl o/view2 /view5 /view4 /view3 /view1 r 2,173 ms 1,180 ms	juestHandler.ma RequestHandle ers.RequestHan (3 %) (2 %)	akeJpaCall r.performW	'ork 7	724 ms 577 μs	1
SELEC	 1 5.2% - 3,(1 5.2% - 3,(1 5.2% - 1 5.2% - 1 5.2 1 1 1 1 1 5.2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	119 ms - 19 hot spot evt 3,019 ms - 19 hot spot % - 3,019 ms - 19 hot sp 1,8% - 1,027 ms - 6 hot s 1,6% - 923 ms - 6 hot s 0,7% - 421 ms - 3 hot s 0,6% - 327 ms - 2 hot s 0,5% - 318 ms - 2 hot s * o.quantity) FROM cus R (ID, NAME, OPTIONS) MER (ID, NAME, OPTIONS)	. com.ejt.demo.server. evt. com.ejt.demo.server. opt evt. com.ejt.demo.ser spot evt. HTTP: /demo pot evt. HTTP: /demo, pot evt. HTTP: /demo, pot evt. HTTP: /demo, totmers c LEFT JOIN o VALUES (?, ?, ?) NNS) VALUES (?, ?, ?)	handlers.Rec ver.handlers. server.handl o/view2 /view5 /view4 /view3 /view1 r 2,173 ms 1,180 ms 1,111 ms	juestHandler.ma RequestHandler ers.RequestHan (3 %) (2 %) (1 %)	akeJpaCall r.performW	'ork 7 65,: 61,:	577 μs 737 μs	1 1
SELEC SELEC INSER INSER	1 5.2% - 3,0 1 5.2% - 3,0 1 5.2% - 0 1 5.2 0 0 1 5.2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	119 ms - 19 hot spot evt 3,019 ms - 19 hot spot % - 3,019 ms - 19 hot spot 1.6% - 9,219 ms - 6 hot s 1.6% - 9,23 ms - 6 hot s 0,7% - 4,21 ms - 3 hot s 0,6% - 3,27 ms - 2 hot s 0,5% - 3,18 ms - 2 hot s % o,quantity) FROM cus R (ID, NAME, OPTIONS)	. com.ejt.demo.server. evt. com.ejt.demo.server. opt evt. com.ejt.demo.ser spot evt. HTTP: /demo pot evt. HTTP: /demo, pot evt. HTTP: /demo, pot evt. HTTP: /demo, totmers c LEFT JOIN o VALUES (?, ?, ?) NNS) VALUES (?, ?, ?)	handlers.Rec ver.handlers. server.handl o/view2 /view5 /view4 /view3 /view1 r 2,173 ms 1,180 ms 1,111 ms	juestHandler.ma RequestHandlei ers.RequestHan (3 %) (2 %) (2 %) (1 %)	akeJpaCall r.performW	'ork 7 65,: 61,: 61,:	577 µs	

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,

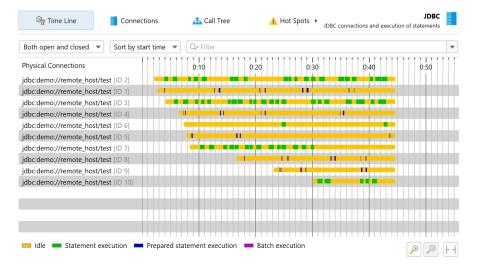
socket operations or process executions where it is important to look at the total time and not only on the time that the current JVM has spent working on it.

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.

2 jdbc:demo://remote_host/test 0:02.173 [Mar 3, 2025 2 24 18,644 3 jdbc:demo://remote_host/test 0:04.103 [Mar 3, 2025 2 24 17,174 4 jdbc:demo://remote_host/test 0:06.503 [Mar 3, 2025 2 20 1,827 5 jdbc:demo://remote_host/test 0:07.813 [Mar 3, 2025 2 21 1,064 5 jdbc:demo://remote_host/test 0:07.803 [Mar 3, 2025 2 6 2,243 7 jdbc:demo://remote_host/test 0:07.803 [Mar 3, 2025 2 15 11,559 3 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2 17 1,439 3 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2 13 1,828						
jdbc:demo://remote_host/test 0:02.693 [Mar 3, 2025 2 24 2,035 jdbc:demo://remote_host/test 0:02.173 [Mar 3, 2025 2 24 18,644 jdbc:demo://remote_host/test 0:04.103 [Mar 3, 2025 2 24 17,174 jdbc:demo://remote_host/test 0:06.503 [Mar 3, 2025 2 20 1,827 jdbc:demo://remote_host/test 0:07.303 [Mar 3, 2025 2 20 1,827 jdbc:demo://remote_host/test 0:07.303 [Mar 3, 2025 2 6 2,243 jdbc:demo://remote_host/test 0:07.303 [Mar 3, 2025 2 6 2,243 jdbc:demo://remote_host/test 0:08.423 [Mar 3, 2025 2 15 11,559 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2 17 1,439 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2 13 1,283	Both ope	n and closed 🔻 Filter in all text colum	nns 🔻 🔍 🔍			•
2 jdbc:demo://remote_host/test 0:02.173 [Mar 3, 2025 2 24 18,644 3 jdbc:demo://remote_host/test 0:04.103 [Mar 3, 2025 2 24 17,174 4 jdbc:demo://remote_host/test 0:06.503 [Mar 3, 2025 2 20 1,827 5 jdbc:demo://remote_host/test 0:07.813 [Mar 3, 2025 2 21 1,064 5 jdbc:demo://remote_host/test 0:07.803 [Mar 3, 2025 2 6 2,243 7 jdbc:demo://remote_host/test 0:07.803 [Mar 3, 2025 2 15 11,559 3 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2 17 1,439 3 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2 13 1,828	ID 🔺	Connection String	Start Time	End Time	Event Count	Event Duration
3 jdbc:demo://remote_host/test 0:04.103 [Mar 3, 2025 2 24 17,174 4 jdbc:demo://remote_host/test 0:06.503 [Mar 3, 2025 2 20 1,827 5 jdbc:demo://remote_host/test 0:07.313 [Mar 3, 2025 2 12 1,064 5 jdbc:demo://remote_host/test 0:07.303 [Mar 3, 2025 2 6 2,243 7 jdbc:demo://remote_host/test 0:08.423 [Mar 3, 2025 2 15 11,559 3 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2 17 1,439 9 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2 13 1,283	1	jdbc:demo://remote_host/test	0:02.693 [Mar 3, 2025 2		24	2,035 m
4 jdbc:demo://remote_host/test 0:06.503 [Mar 3, 2025 2 20 1,827 5 jdbc:demo://remote_host/test 0:07.813 [Mar 3, 2025 2 12 1,064 5 jdbc:demo://remote_host/test 0:07.303 [Mar 3, 2025 2 6 2,243 7 jdbc:demo://remote_host/test 0:08.423 [Mar 3, 2025 2 15 11,559 3 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2 17 1,439 9 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2 13 1,283	2	jdbc:demo://remote_host/test	0:02.173 [Mar 3, 2025 2		24	18,644 m
5 jdbc:demo://remote_host/test 0.07.813 [Mar 3, 2025 2 12 1,064 5 jdbc:demo://remote_host/test 0.07.303 [Mar 3, 2025 2 6 2,243 7 jdbc:demo://remote_host/test 0.08.423 [Mar 3, 2025 2 15 11,559 3 jdbc:demo://remote_host/test 0.16.753 [Mar 3, 2025 2 17 1,439 9 jdbc:demo://remote_host/test 0.23.273 [Mar 3, 2025 2 13 1,283	3	jdbc:demo://remote_host/test	0:04.103 [Mar 3, 2025 2		24	17,174 m
5 jdbc:demo://remote_host/test 0:07.303 [Mar 3, 2025 2] 6 2,243 7 jdbc:demo://remote_host/test 0:08.423 [Mar 3, 2025 2] 15 11,559 3 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2] 17 1,439 3 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2] 13 1,283	4	jdbc:demo://remote_host/test	0:06.503 [Mar 3, 2025 2		20	1,827 m
jdbc:demo://remote_host/test 0:08.423 (Mar 3, 2025 2 15 11,559 3 jdbc:demo://remote_host/test 0:16.753 (Mar 3, 2025 2 17 1,439 jdbc:demo://remote_host/test 0:23.273 (Mar 3, 2025 2 13 1,283	5	jdbc:demo://remote_host/test	0:07.813 [Mar 3, 2025 2		12	1,064 m
3 jdbc:demo://remote_host/test 0:16.753 [Mar 3, 2025 2] 17 1,439 jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2] 13 1,283	5	jdbc:demo://remote_host/test	0:07.303 [Mar 3, 2025 2		6	2,243 m
jdbc:demo://remote_host/test 0:23.273 [Mar 3, 2025 2 13 1,283	7	jdbc:demo://remote_host/test	0:08.423 [Mar 3, 2025 2		15	11,559 m
	3	jdbc:demo://remote_host/test	0:16.753 [Mar 3, 2025 2		17	1,439 m
10 jdbc:demo://remote_host/test 0:30.143 [Mar 3, 2025 2] 6 4,579	9	jdbc:demo://remote_host/test	0:23.273 [Mar 3, 2025 2		13	1,283 m
	10	jdbc:demo://remote_host/test	0:30.143 [Mar 3, 2025 2		6	4,579 m
	10	jdbc:demo://remote_host/test	0:30.143 [Mar 3, 2025 2]		6	4,579

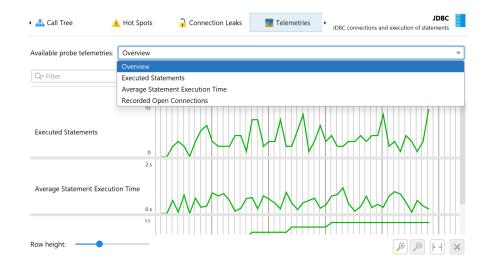
A probe can publish certain properties in a nested table. This is done to reduce the information overload in the main table and give more space to table columns. If a nested table is present, such as for the file and process probes, each row has an expansion handle at the left side that opens a property-value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

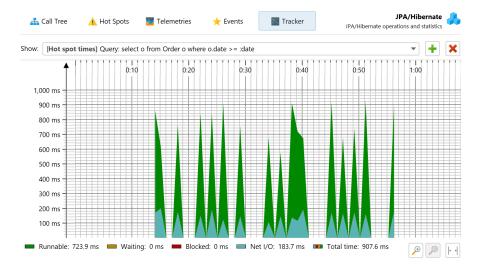
Connee	ctions	📥 Call Tree	⚠️ Hot Spots → JD	BC connections and execution c	JDBC of statements
Both open and closed 💌 Sort by st	tart time 💌	Q . Filter			•
Physical Connections jdbc:demo://remote_host/test [ID 2]		0:10	0:20 0:30	0:40	0:50
jdbc Show Selected Connection jdbc Show Events For Selected Con jdbc.	nection				
jdbc:demo://remote_host/test [ID 6] jdbc:demo://remote_host/test [ID 5]					
jdbc:demo://remote_host/test [ID 7]			** <mark>********</mark> ***		

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.



Probe telemetries can be added to the "Telemetries" section [p. 45] in order to compare them to system telemetries or to custom telemetries. You then also have control over probe recording with the context menu actions in the telemetry overview.

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.

All types Filter in all text columns	▼ Q• Filter		
Start Time 🔶 Event Type	Duration	Description	Thread
🔻 0:02.716 [Mar 3, 2025 2 🔎 Query	922 ms	select o from Order o where o.date >= :date	Servlet request simulato
★ JDBC [Prepared statement execution]	141 ms	SELECT * FROM ORDER O WHERE O.DATE >= 3	Servlet request simulato
🛚 0:03.640 [Mar 3, 2025 2 🕂 Insert	112 ms	com.ejt.demo.server.entities.Customer	Servlet request simulato.
★ JDBC [Prepared statement execution]	62,083 µs	INSERT INTO CUSTOMER (ID, NAME, OPTIONS	Servlet request simulato.
• 0:03.752 [Mar 3, 2025 2 🕂 Insert	188 ms	com.ejt.demo.server.entities.Order	Servlet request simulato.
▶ 0:06.549 [Mar 3, 2025 2 🔎 Query	785 ms	select o from Order o where o.date >= :date	Servlet request simulato.
0:07.334 [Mar 3, 2025 2 + Insert	123 ms	com.ejt.demo.server.entities.Customer	Servlet request simulato.
0:07.457 [Mar 3, 2025 2 + Insert	200 ms	com.ejt.demo.server.entities.Order	Servlet request simulato.
▶ 0:07.848 [Mar 3, 2025 2 🔎 Query	711 ms	select o from Order o where o.date >= :date	Servlet request simulato.
▶ 0:08.559 [Mar 3, 2025 2 🕂 Insert	119 ms	com.ejt.demo.server.entities.Customer	Servlet request simulato.
▶ 0.08 679 [Mar 3, 2025 2] 📥 Insert	217 ms 28,621 ms	com eit demo server entities Order	Servlet request simulato
Selection			
itack trace:			

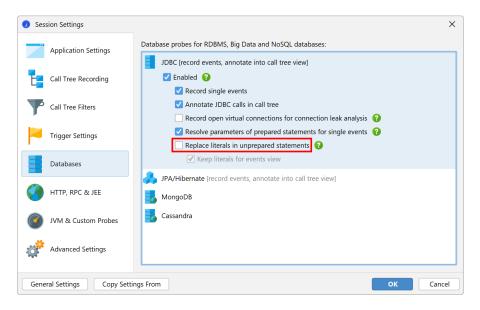
Similarly, the hot spots view adds a special "JDBC calls" node to all hot spots that contains the JDBC calls that were triggered by the JPA operation. Some JPA operations are asynchronous and are not executed immediately, but at some arbitrary later point in time when the session is flushed. When looking 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 labeled "Deferred operations", otherwise a "Direct operations" node is inserted.

read status: 🕜	Thread selection:			Aggreg	gation level:
All states	🝔 All thread groups			- 💿 N	1ethods
	Hot Spot	Time	T	Average Time	Events
▼	o from Order o where o.date >= :date ms - 24 evt. SELECT * FROM ORDER O WHERE	0.DATE >= ?	41 ms (71 %)	776 ms	24
	 % - 18,641 ms - 24 hot spot evt. Direct operatic 71.4% - 18,641 ms - 24 hot spot evt. javax.persis 71.4% - 18,641 ms - 24 hot spot evt. com.git 71.4% - 18,641 ms - 24 hot spot evt. com.git 71.4% - 18,641 ms - 24 hot spot evt. com 71.4% - 18,641 ms - 24 hot spot evt. 71.4% - 18,641 ms - 24 hot spot evt. 71.4% - 18,641 ms - 24 hot spot evt. 8.4% - 2,463 ms - 3 hot spot evt. HTT 8.7% - 2,285 ms - 3 hot spot evt. HTT 8.7% - 2,285 ms - 3 hot spot evt. HTT 9.7% - 18,78 ms - 2 hot spot evt. HTT 	stence.TypedQuery.getResu .demo.server.handlers.Requ n.ejt.demo.server.handlers.R com.ejt.demo.server.handlet vt. com.ejt.demo.server.han HTTP: /demo/view2 HTTP: /demo/view4 TP: /demo/view4 TP: /demo/view3	estHandler.exe equestHandler. rs.RequestHand	makeJpaCall ler.performWork	

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.

Connections		击 Call Tree	🔥 Hot S	Spots	2 Connection Leaks	n Leaks JDBC connections and execution of		
This view shows all pools and block a p				or more thar	n 10 seconds. Virtual con	nections are what you ge	et from connection	
Connections of type	e "Unclo	osed collected" are	definite leaks wh	ile "Unclose	d" connections are stron	g candidates.		
All types	▼	Filter in all text c	olumns 💌 🖸	∖ • Filter			•	
Opened At		Open Since	Туре		Description	Thread	Class Name	
0:02.080 [Mar 3	, 202	18,014 ms	Unclosed c	jdbc:hsqldb:	hsql://localhost:9012/tes	t pool-1-thread-2 (58)	jdk.proxy2.\$Proxy2	
0:08.843 [Mar 3	, 202	11,252 ms	Unclosed c	jdbc:hsqldb:	hsql://localhost:9012/tes	t pool-1-thread-2 (58)	jdk.proxy2.\$Proxy2	
0:14.916 [Mar 3	, 202	5,179 ms	Unclosed c	jdbc:hsqldb:	hsql://localhost:9012/tes	t pool-1-thread-2 (58)	jdk.proxy2.\$Proxy2	
3 rows								
Stack trace:								
javax.sql.DataSou jdbc.JdbcTestWor jdbc.JdbcTestWor java.util.concurrer	rker.call(rker.call()	orker.run()					

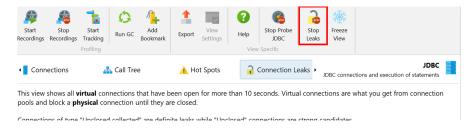
There are two types of leak candidates, "unclosed" connections and "unclosed collected" connections. Both types are virtual connections where the connection objects that have been handed out by the database pool are still on the heap, but close() has not been called on

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.

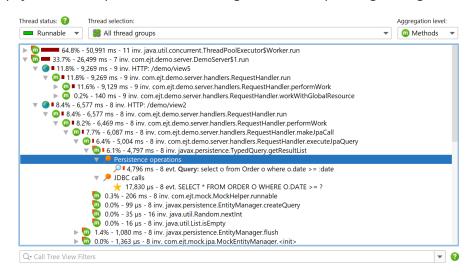


Payload data in the call tree

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 view analyses where the analysis is automatically repeated in the context of the probe call tree view when you click on the probe link.

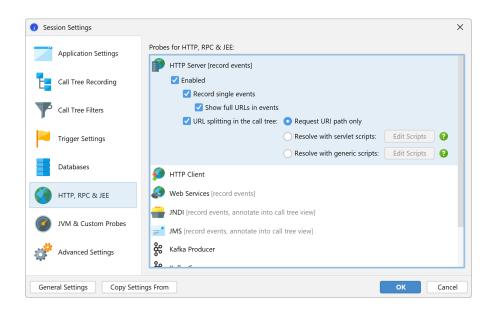
Thread status: 😮 Thread selection:		Aggregation leve	el:
Runnable 💌 😫 All thread groups	•	Methods	
 ■ ■ 63.1% - 757 ms - 1 inv. jdbc.JdbcDemo.main ■ ■ 30.0% - 360 ms - 5 inv. java.util.concurrent.ThreadPoolExecutor\$Worker.run ■ ■ 30.0% - 360 ms - 5 inv. jdbc.JdbcTestWorker.call ■ ■ 20.0% - 360 ms - 5 inv. jdbc.JdbcTestWorker.call ■ ■ 20.0% - 360 ms - 5 inv. jdbc.JdbcTestWorker.testStatementsPath1 ■ ■ 12.3% - 13. inv. jdbc.JdbcTestWorker.testStatement ■ ■ 6.2% - 73.908 µs - 13 inv. jdbc.JdbcTestWorker.testStatement 			
 ▼ 14.4% - 53.018 µs - 26 inv. java sql.Statement.executeQuery JDBC calls Show in probe call tree 1.1% - 12.867 µs - 13 inv. java sql.Connection.createStatement 0.7% - 7.949 µs - 10 inv. java sql.Connection.createStatement 0.6% - 82.418 µs - 13 inv. java.sql.DataSource.getConnection 1.6% - 82.5361 µs - 7 inv. java.sql.Connection.close 0.14.5% - 48.971 µs - 10 inv. java.sql.Connection.close 0.14.5% - 53.671 µs - 10 inv. java.sql.Conlection.createStatementsPath2 0.2% - 25.878 µs - 1 inv. java.lang.Thread.interrupted 0.0% - 14 µs - 13 inv. java.lang.Thread.interrupted 0.9% - 82.959 µs - 1 inv. java.awt.EventDispatchThread.run 			
Q- Call Tree View Filters		-] (

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 path without any parameters.



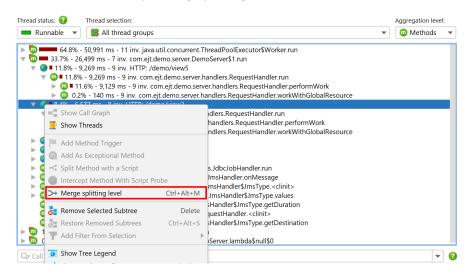
For more flexibility, you can define a script that determines the split string. In the script, you get the current <code>javax.servlet.http.HttpServletRequest</code> as a parameter and return the desired string.

Settings Edit Search Code Help Edit	×
Show Modify Test	? Help
Please enter an expression (no trailing semicolon) or a script (ends with a return state code. The following parameters are available: - com.jprofiler.api.agent.ScriptContext - javax.servlet.http.HttpServletRequest The expected return type is java.lang.String	atement) that consists of regular Java
Script:	
1 import javax.servlet.http.HttpSession;	
2	
3 // This is just an example that shows how to work with the session 4 // in this context. It will most likely not work directly in your	
5HttpSession session = servletRequest.getSession(false);	case.
6if (session != null) {	
<pre>7 Object user = session.getAttribute("user");</pre>	
<pre>8 if (user != null) {</pre>	
<pre>9 return user.toString();</pre>	
10 } else {	
<pre>11 return "Unauthenticated";</pre>	
12 }	
13 } else {	
14 return null; // Do not split	
15 }	
	OK Cancel

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 path 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.

J Edit Servlet Scripts	:
You can split requests on multiple nested levels . For example, you c and then split by the request path.	an split by the request method firs
The grouping expression for each level is defined by the return value cripts, some example entries help you to get started.	e of a script. When adding new
Script	4
servletRequest.getRequestURI()	Edit Script
servletRequest.getMethod()	Edit Script
import javax.servlet.http.HttpSession; []	Edit Script
	i
😧 Help	OK Cancel

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.

Garbage Collector Analysis

Understanding and analyzing the runtime characteristics of the garbage collector (GC) is important for several reasons. Firstly, GC pauses can directly impact the responsiveness of your application. By understanding how the garbage collector is performing, you can optimize its settings to reduce these pauses. In general, frequent long GC cycles may indicate that the heap is too small, or that too many temporary objects are being created.

With the help of the garbage collector probe you can solve these problems and make more informed decisions when tuning your JVM settings, such as selecting the appropriate garbage collector, heap size, or other JVM parameters.

The garbage collector probe has different views than the other probes and also uses a different data source. It does not obtain its data from the profiling interface of the JVM but uses JFR streaming to analyze GC-related events from the JDK flight recorder ⁽¹⁾. Because of the dependency on JFR event streaming, the GC probe is only available when you profile Java 17 or higher on a Hotspot JVM. When you open JFR snapshots [p. 221], the exact same probe is available, regardless of the used Java version.

Garbage collections view

The main view in the garbage collector probe is the "Garbage collections" table. It shows all recorded garbage collections as rows with their most important metrics as columns.

< Garbag	ge Collections	🕎 Teleme	tries	🗴 GC Summary	🐺 GC Confi	guration	GC Flags			and configuration		
Filter in all text columns Qx Filter Image: Second Sec												
GC ID 🔺	Start Time	Duration	Cause	Collector	Longest Pause	Sum Of Pauses	Final Refs	Weak Refs	Soft Refs	Phantom Refs		
▶ 41	0:01.997.517 [2,265 µs	G1 Evacuation	G1New	2,265 µs	2,265 µs	4	44	0	65		
▶ 42	0:01.999.852 [22,885 µs	G1 Evacuation	G1Old	5,688 µs	5,810 µs	1	1	0	1		
▶ 43	0:03.520.570 [1,365 µs	G1 Humongou	G1New	1,365 µs	1,365 µs	2	77	0	52		
▶ 44	0:03.521.951 [24,998 µs	G1 Humongou	G10ld	7,477 µs	7,632 µs	0	7	0	1		
▶ 45	0:03.655.470 [1,776 µs	G1 Evacuation	G1New	1,776 µs	1,776 µs	3	34	0	37		
▶ 46	0:03.809.613 [1,672 µs	G1 Evacuation	G1New	1,672 µs	1,672 µs	1	70	0	38		
▶ 47	0:03.811.333 [19,640 µs	G1 Evacuation	G1Old	4,167 µs	4,286 µs	0	0	0	0		
▶ 48	0:03.881.874 [20,034 µs	System.gc()	G1Full	20,034 µs	20,034 µs	6	1,691	0	344		
▶ 49	0:04.555.097 [1,920 µs	G1 Evacuation	G1New	1,920 µs	1,920 µs	1	72	0	39		
▶ 50	0:04.557.035 [20,714 µs	G1 Evacuation	G1Old	3,917 µs	4,035 µs	0	0	0	0		
▶ 51	0:05.606.811 [G1 Evacuation		2,043 µs	2,043 µs	4	46	0	15		
▶ 52	0:05.772.998 [1,548 µs	G1 Humongou	G1New	1,548 µs	1,548 µs	4	13	0	9		
▶ 53	0:05.774.563 [24,473 µs	G1 Humongou	G1Old	7,541 µs	7,665 µs	0	0	0	0		
▶ 54	0:05.885.318 [944 µs	G1 Humongou	G1New	944 µs	944 µs	0	0	0	0		
▶ 55	0:05.886.278 [21,066 µs	G1 Humongou	G1Old	4,363 µs	4,447 µs	0	0	0	0		
▶ 56	0:06.030.645 [G1 Humongou		1,053 µs	1,053 µs	0	0	0	0		
▶ 57	0:06.031.711 [G1 Humongou		6,388 µs	6,518 µs	0	0	0	0		
► 58	0.06 137 906 T	1 867 115	G1 Humondou	G1New	1 867 us	1 867 115	0	0	0	٥		
Total from	112 rows:	1,618 ms				645 ms	152	12,588	4,731	3,539		

The "Cause" column shows you why a garbage collection was triggered. For example, a call to System.gc() triggered a full garbage collection. You can see that from the associated "G1Full" value in the "Collector" column. It also caused a substantial pause of 20 ms which is why it is generally not a good idea to call System.gc(). Other causes trigger the collection of the young generation space ("G1New") or the old GC collection of the G1 collector ("G1Old") that cleans up unreferenced objects in the old generation. You can see that the old GC collections consistently take longer than the young generation collections although the young generation collections collections collections collect more objects.

Collected references with special GC handling are shown as "final", "weak", "soft" and "phantom" references in separate columns.

The reason there are separate columns for the longest pause and the sum of pauses is that each garbage collection is composed of multiple phases that produce separate pauses. Also, the "Duration" of a garbage collection is not equal to the sum of pauses, because a garbage ⁽¹⁾ https://en.wikipedia.org/wiki/IDK Flight Recorder

collection only partially pauses the JVM while it is executing. You can see that the "G1Old" collections in the screenshot only pause for about a fifth of their duration.

To inspect the various phases of a garbage collection, you can toggle the tree icon in the "GC ID" column.

ళ Garbage C		_		- '			GC Flags		GC collection	s and configu	ration
Filter in all text	t columns 💌	Q• Filter									
GC ID 🔺	Start Time	Duration	Cause	Collector	Longest Pause	Sum Of Pauses	Final Refs	Weak Refs	Soft Refs	Phantor	n Refs
41 0:0	01.997.517 [2,265 µs (G1 Evacuation	G1New	2,265 µs	2,265 µs		4 44	0		e
42 0:0	01.999.852 [22,885 µs (G1 Evacuation	G1Old	5,688 µs	5,810 µs		1 1	0		
Phase Le	evel	Duration	1	F	Phase Na	ame	c	ommitted metaspace	e: 58.392 kB →	58.458 kB	(+0.1 %
1 4,265 µs (68 %)			Class Unloadir	ng			Class metadata:	8.650 kB → 8.6		(±0%	
			Purge Metasp	ace			Other data:	49,741 kB → 49		(+0.1 9	
	1 📕 397 μ	is (6 %)		Reference Processing Notify and keep alive finalizable				sed metaspace:	57.895 kB →		
	2 📕 303 µ	s (4 %)									
	1 📕 209 µs	(3 %)		Finalize Marking Weak Processing				Class metadata:	8,365 kB → 8,36		(±0 9
	1 110 µs	(1%)						Other data:	49,530 kB → 49		(+0.1 9
	1 99 µs (Irrent Mark Cleanup		R	eserved metaspace:	1,124 MB →	1,124 MB	(±0 %
	1 74 µs (Reclaim Empt	v Regions			Class metadata:	1,073 MB → 1,0	73 MB	(±0 9
	1 49 µs (0				, mbered Set Tracking B	efore Rebuild		Other data:	50,331 kB \rightarrow 50,331 kB		(±0 9
	2 47 µs (0) %)			eakReferences						
	2 35 µs (0			Notify Phanto			C	ommitted heap:	65,011 kB →	65,011 kB	(±0 %
	1 34 µs (0			Flush Task Caches				sed heap:	36,331 kB →	36,331 kB	(±0 %
	2 30 µs (0			ClassLoaderDa	ata		R	eserved heap:	209 MB → 2	9 MB	(±0 %
	1 2 µs (0	,		Update Reme	mbered Set Tracking A	fter Rebuild		eserved neup.	205 100 - 21		(10)
tal from 112		1.618 ms			-	645 ms	15	2 12.588	4,731		3.5

In the screenshot above, a mixed GC collection of the G1 collector ("G1Old") was expanded. You can see that most of the time is spent in "Class Unloading", which does not pause the JVM. On the right, you can see further statistics for the garbage collection. Here, the used heap stayed the same while the used metaspace went up by 0.1%.

Filter in all	text columns 💌	Q . Filter							
GC ID 🔺	Start Time	Duration Cause	Collector	Longest Pause	Sum Of Pauses	Final Refs	Weak Refs	Soft Refs Phant	tom Refs
▶ 47	0:03.811.333 [19,640 µs G1 Evacuation		4,167 µs	4,286 µs	0	0	0	
48	0:03.881.874 [20,034 µs System.gc()	G1Full	20,034 µs	20,034 µs	6	1,691	0	3-
Phas	e Level	Duration	~	Phase Na	me	Com	mitted metaspace	: 59,965 kB → 59,965 kB	B (±0%
	1	11,366 µs (48 %) Phase 1: Mark	live objects		Cla	ass metadata:	8,716 kB → 8,716 kB	(±0 %
	2	6,577 µs (27 %)	Phase 1: Class	Unloading and Clean	qu	Ot	her data:	51,249 kB → 51,249 kB	(±0 %
		3,582 µs (15 %)	Phase 3: Adjus			Usec	metaspace:	59,486 kB → 59,486 kB	B (±0%
	1 📕 922 μ			are for compaction			ass metadata:	8.483 kB → 8.483 kB	(±0 %
	1 📕 905 µ		Phase 4: Comp			Ot	her data:	51,003 kB → 51,003 kB	(±0 9
	2 195 µs			ence Processing			rved metaspace:	1.132 MB → 1.132 MB	
	2 130 µs	(0 %)	Phase 1: Weak Processing				iss metadata:	1.073 MB → 1.073 MB	
							her data:	58.720 kB → 58.720 kB	(±0 %) (±0 %
						Com	mitted heap:	50,720 kB → 50,720 kB 70,254 kB → 70,254 kB 44,714 kB → 37,705 kB	B (±0 9
							rved heap:	209 MB → 209 MB	(±0 %
49	0:04.555.097 [1,920 µs G1 Evacuation	G1New	1,920 µs	1,920 µs	1	72	0	
otal from	112 rows:	1.618 ms			645 ms	152	12,588	4,731	3,53

The phases of each collector are different. In the screenshot above, a full collection is shown. It spends a lot of time marking live objects in the entire heap. At the end of the collection, the used heap was reduced by 15.7%, while the metaspace remained the same.

While analyzing garbage collections, **filtering** is an important tool to compare different subsets of garbage collections. At the top of the table, there is a filter selector that lets you choose any column and configure a corresponding filter. An easier way to see similar garbage collections is to use the context menu on the table and select a filter condition based on the column values in the selected row.

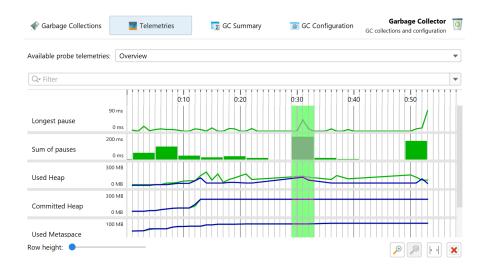
🞻 Ga	rbage Collections	🕎 Tele	metries	🛐 GC Summary	GC Confi	guration	GC Flags			and configuration
	om Refs ▼ on ≥ 2,265 μs × Pha	≥ antom Refs	 ▼ 65 ≥ 65 × 	Adv	d					
GC ID	Start Time	Duration	Cause	Collector	Longest Pause	Sum Of Pauses	Final Refs	Weak Refs	Soft Refs	Phantom Refs
▶ 41	0:01.997.517 [2,265	us G1 Evacuation	. G1New	2,265 µs	2,265 µs	4	44	0	65
48	0.02 001 074 1	20.024	us Evistom as/	G1Full	20,034 µs	20,034 µs	6	1,691	0	344
▶ 11'	Filter Equals This	▶	GC ID	G1Full	40,249 µs	40,249 µs	10	2,139	0	369
► 12.	Filter Greater Than	This 🕨	Duration	G1Full	43,426 µs	43,426 µs	10	2,432	1,113	304
▶ 12!	Filter Less Than Thi		Cause	G1Full	35,537 µs	35,537 µs	8	2,431	229	394
► 13	Filter Less Than Thi	s P		G1New	2,398 µs	2,398 µs	17	46	0	78
▶ 15	Sort Garbage Colle	ctions 🕨	Collector	G1Full	76,258 µs	76,258 µs	8	2,223	1,062	421
	-		Longest Pause							
	🔎 Find	Ctrl+F	Sum of Pauses							
	1 Export View	Ctrl+R	Final Refs							
1	View Settings	Ctrl+T	Weak Refs							
C			Soft Refs							
			Phantom Refs							
Total fr	rom 7 rows:	220	ms			220 ms	63	11,006	2,404	1,975

You can add multiple filters to narrow down the garbage collections of interests. Active filters are shown as labels at the top of the table. It is also possible to add filters from the nested GC phases tables.

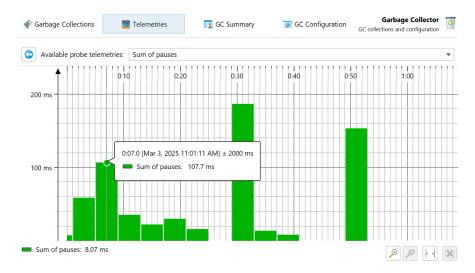
	arbage Co				gs	GC Flag	3	figuration	🐞 GC Conf	у	🔽 GC Summar	ries	🕎 Teleme	ns	e Collectio	< Garbaç
₹													Q ~ 4 8	¥		GC ID
														3 µs \times	ispace ≥ 53	urge Meta
om Refs	Phant	Soft Refs	s	Weak Refs	Refs	Final R	f Pauses	Sum Of	est Pause	Long	Collector	Cause	Duration	me	Start Ti	GC ID 🔺
	0		1		1		5,810 µs		5,688 µs		G1Old	G1 Evacuation	22,885 µs	52 [0:01.999.8	42
	0		7		0		7,632 µs		7,477 µs		G1Old	G1 Humongou	24,998 µs	51 [0:03.521.9	44
(+0.1 %	58.982 kB	: 58,916 kB →	pace	mitted metas	Com			ame	Phase Na		-	۰ ۱	Duratio		e Level	Phas
(±0 %		8.650 kB → 8.6	·	ass metadata:						ing) Class Unload	6,224 µs (81 %			1	
(+0.1 %),331 kB	50,266 kB → 50		her data:	Ot				n	0000	Durgo Motor		(Q 0/)	626	- 1	
(+0.1 %	58.479 kB	58.425 kB →		metaspace:	Used					iration	and a Longer Du	/ith This Phase A	Collections \	Garbage	Filter	
(+0.0 %	· · · · ·	8.398 kB → 8.4		ass metadata:					'k Cleanup	uration	And a Shorter D	/ith This Phase A	Collections \	Garbage	Filter	
(+0.1 9	0.078 kB	50.027 kB → 50		her data:	Ot											
(±0 %	1 124 MB	1,124 MB →	ace.	rved metaspa	Rese					•				Phases		
(±0 %		1.073 MB → 1.0		ass metadata:							Reclaim Emp			83 µs (1		
(±0 %		50,331 kB → 50		her data:			oulia	setore Rep			Update Rem			82 µs (1		
(10 /	,551 80	30,331 KD - 30		inci dutu.	00						Notify Soft/V Notify Phant			39 µs (0 33 µs (0		
(+8.1 %	70,254 kB	65,011 kB →		mitted heap:	Com				ices		Flush Task Ca			19 µs (0		
(+114%	40.883 kB	36,689 kB →		heap:							ClassLoader			15 µs (0		
(+0 %		209 MB → 2		wed heap:			ild	After Rebui	et Tracking A		Update Rem			2 us (0 9		
. +11 %		2.32	19		8		305 ms	inter nebui	ce riscking P		obdate nem		" 1.181 ms	2 03 10 7	33 rows:	tal from

Telemetries

The GC probe produces a number of telemetries which are available in the "Telemetries" probe view.



If you are interested in minimizing GC pauses, the "Longest pause" telemetry at the top will be the most interesting one. You can drag along the time axis of the telemetry to select the corresponding garbage collections in the "Garbage Collections" view. For better vertical resolution, you can select a single telemetry from the drop-down at the top or by clicking on the name of the telemetry.



In the screenshot above you can see the sum of pauses over time. JProfiler presents summable measurements by building a **histogram of the recorded data**. The bin width depends on the available horizontal space, so histogram bins will change depending on the zoom level and, if "scale to fit" is enabled, depending on the width of the window. What stays the same is the total area under all histogram bins.

The **heap and metaspace telemetries** are based on the statistics that you can see when expanding a garbage collection. This means that the data is not regularly sampled like for the memory telemetries in a full profiling session. If no garbage collection occurs during a time period, there will be no data. For a JVM with little allocation activity, there can be long stretches along the time axis where the graph is just interpolated between two garbage collections.

Each of these telemetries has two data lines: "Before GC" and "After GC". The differences are typically large for the "Used Heap" telemetry. At each time, you can see how much work the garbage collection has performed by comparing the values of the two data lines. You can look

at the tooltip to get the precise values. For the "Committed heap" telemetry and the metaspace telemetries, the differences between both lines will often be small.

If you are analyzing a JFR snapshot [p. 221], the same data from the jdk.GCHeapSummary JFR event type is also used in the "Memory" telemetry in the telemetry section. In that case, however, both the "Before GC" and "After GC" values are shown in the same data line and data is not aggregated to a once per-second granularity as in the GC probe telemetries, so the graph will look different.

GC Summary

The GC summary shows you measurements that are aggregated over the entire recording period. Each measurement provides the number of garbage collections, as well as the average, maximum and the total values. The most important data at the top are the "Pause times" that directly affect the liveness of your application.

Garbage Collections	Telemetries	🔽 GC Summary	🐺 GC Configuration	Garbage Collector GC collections and configuration	
Pause Times					
Pause Count		143			
Average Pause		4,512 µs			
Maximum Pause		76,258 µs			
Sum of Pauses		645 ms			
All Collections Total Times	5				
Average GC Time		14,451 µs			
Maximum GC Time		76,258 µs			
Total GC Time		1,618 ms			
Young Collections Tota	al Times				
GC Count		70			
Average GC Time		1,712 µs			
Maximum GC Time		3,460 µs			
Total GC Time		119 ms			
▼ Old Collections Total T	imes				
GC Count		42			
Average GC Time		35,682 µs			
Maximum GC Time		76,258 µs			
Total GC Time		1,498 ms			

The other top-level category shows the total times of all collections which is then split into two subcategories for young and old collections.

GC Configuration

When you tune your garbage collector, you may want to inspect the common properties that can either be set explicitly or that are set implicitly by the garbage collector itself.

🔹 🎻 Garbage Collections 🛛 🧧 Telemetries	Garbage Collector GC collections and configuration GC collections and configurati
▼ GC Configuration	
Young Garbage Collector	G1New
Old Garbage Collector	G1Old
Concurrent GC Threads	3
Parallel GC Threads	13
Concurrent Explicit GC	false
Disabled Explicit GC	false
Uses Dynamic GC Threads	true
GC Time Ratio	12
GC Heap Configuration	
Initial Size	209 MB
Minimum Heap Size	8,388 kB
Maximum Heap Size	209 MB
If Compressed Oops Are Used	true
Compressed Oops Mode	32-bit
Heap Address Size	32
Object Alignment	8 bytes
Young Generation Configuration	
Minimum Young Generation Size	1,363 kB
Maximum Young Generation Size	125 MB

These properties are common to all garbage collectors and help you understand the differences between garbage collectors.

GC Flags

Finally, the GC-specific flags give you an idea what properties of a garbage collector can be tuned and lets you check their actual values.

 Telemetries GC Su 	mmary 🛛 🚊 GC Conf	iguration 😨 GC F	Garbage Collector GC collections and configuration
Q• Filter			
Flag Name		Flag Value	Origin
AlwaysPreTouch	false		Default
ClassUnloading	true		Default
ClassUnloadingWithConcurrentMark	true		Default
G1ConcMarkStepDurationMillis	10.0		Default
G1ConcRSHotCardLimit	4		Default
G1ConcRSLogCacheSize	10		Default
G1ConcRefinementGreenZone	0		Default
G1ConcRefinementRedZone	0		Default
G1ConcRefinementServiceIntervalMillis	300		Default
G1ConcRefinementThreads	13		Ergonomic
G1ConcRefinementThresholdStep	2		Default
G1ConcRefinementYellowZone	0		Default
G1ConfidencePercent	50		Default
G1DummyRegionsPerGC	0		Default
G1EvacuationFailureALot	false		Default
G1EvacuationFailureALotCount	1000		Default
G1EvacuationFailureALotDuringConcMark			Default
G1EvacuationFailureALotDuringConcurrer	ntStart true		Default
G1EvacuationFailureALotDuringMixedGC	true		Default
and the second			

The "Origin" column shows you how the flag was set. "Default" values have not been modified from the standard settings while "Ergonomic" flags have been adjusted automatically by the garbage collector. If you set specific GC flags on the command line, they will be reported as "Command line" in origin.

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.

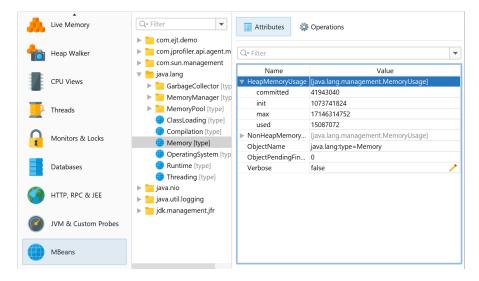
JProfiler includes an MBean browser that shows all registered MBeans in the profiled VM. The remote management level of JMX for accessing MBean servers is not required, because the JProfiler agent is already running in-process and has access to all registered MBean servers.

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

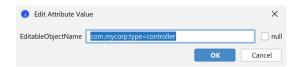
⁽²⁾ https://docs.oracle.com/javase/7/docs/technotes/guides/management/mxbeans.html

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 \checkmark 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 checkbox 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. 45] 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.

Q - Filter	Attributes	🌣 Opera	ations		
Q- Filter	C Filter ReapMemoryl committed init max used NonHeapM ObjectNam ObjectPend Verbose	e Usage Edit Attrib	(java.lang. 41943040 10737418; 174462447 vute Value metry For V v Details	24	Value MemoryUsage] /emoryUsage]
		View Sett	Ings	Ctrl+1	

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.

Q- Filter	Attributes discons	
 Test [type] com.jprofiler.api.agent.mbean com.sun.management DiagnosticCommand [type] HotSpotDiagnostic [type] 	Q+ Filter JumpThreads(java.lang.String p0, java.lang.String p1) → void getVMOption(java.lang.String p0) → [Composite] JumpHeap(java.lang.String p0, java.lang.String petVMOption(java.lang.String p0, java.lang.String view Ctrl+F View Settings Ctrl+T View Settings Ctrl+T	

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.

Q ∗ Filter		
Name	Value	
Return value	[com.sun.management.VMOption]	
name	HeapDumpOnOutOfMemoryError	
origin	DEFAULT	
value	false	
writeable	true	

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

Enter Operation	n Parameters	×
java.lang.String p0	HeapDumpOnOutOfMemoryError	null
java.lang.String p1	true	🗌 null
	ОК	Cancel

MBean snapshots

In addition to viewing the live values of MBeans, you can also take snapshots of their current state. Each new snapshot will be added as a separate view in the MBean view section and can be assigned a custom label. When the snapshot is taken, only those MBeans that are displayed according to the current filter are included. In this way, you can focus on specific MBeans and reduce the overhead of querying MBeans that are not relevant for your purpose.

CPU Views	Q- java.lang ⊋ × ▼	Q . Filter	
Threads	 com.ejt.demo com.jprofiler.api.agent.mbean com.sun.management 	Name AllThreadIds CurrentThreadA	
Monitors & Locks		CurrentThreadC CurrentThreadC CurrentThreadU	true
Databases	MemoryPool [type]	DaemonThread ObjectMonitor	true
HTTP, RPC & JEE	 Compilation [type] Memory [type] OperatingSystem [type] 	ObjectName PeakThreadCou SynchronizerUs	
JVM & Custom Probes	 Runtime [type] Threading [type] 	ThreadAllocate ThreadAllocate ThreadContenti	true
MBeans	 java.nio java.util.logging jdk.management.jfr 	ThreadContenti ThreadCount	39
Live MBeans	, <u> </u>	ThreadCpuTime ThreadCpuTime	true
at 0:10.812.124		TotalStartedThr TotalThreadAllo	
at 0.15.122.490			

When saving a snapshot in the JProfiler UI, all MBean snapshots are saved as well, while the live MBean view will not be saved. For offline profiling [p. 129], you can use the Controller API or the "Save MBean snapshot" trigger action to take MBean snapshots programmatically.

Both the controller API and the trigger action support an optional label that is shown in the view selector as well as an optional regular expression to filter the included MBeans.

7 Trigger Wizard - CPU load th	nreshold	×
1. Trigger type	Configure actions for this trigger	
2. Threshold 3. Actions	Configured actions:	
4. Description 5. Group ID	🛞 Trigger MBean snapshot	+
6. Finished	Label:	×
	Regex filter for object names: (?	
	▲ Back Next ► Finish Can	cel

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>
```

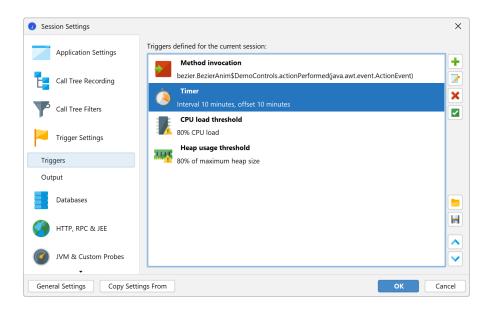
from the Maven Central repository.

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. 27], 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.

1. Trigger type 2. Specify methods	Configure actions for this trigger	
3. Actions	Configured actions:	
4. Description 5. Group ID	Print method invocation	Ŀ
6. Finished	Run interceptor script	
	On method entry:	
	On method exit:	
	On exception exit:	

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.

1. Select local session 2. Offline profiling	Check required actions
8. Locate output directory 4. Check actions 5. Finished	The conversion wizard has finished collecting all necessary information and is now about to execute the required actions. Please check the summary below.
	Conversion type: Convert local session to offline session Application session: Animated Bezier Curve Demo Output directory: C:\Users\ingo For offline profiling, a start script named start_session_offline_101.bat will be created in the output directory. Use this start script to start offline profiling.

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.

Integration Wizard	×
1. Choose wizard 2. Local or remote	Choose whether to wait for the JProfiler GUI
3. Profiled JVM 4. Startup mode	Please choose whether you would like your profiled JVM to wait for a connection from the JProfiler GUI frontend before starting up:
	O Wait for a connection from the JProfiler GUI
	[Easy] Profiling settings are transmitted directly by the JProfiler GUI at startup. With this option you can profile the startup phase of your application.
	Startup immediately, connect later with the JProfiler GUI
	[Easy] Profiling settings are transmitted directly by the JProfiler GUI once you connect.
	O Profile offline, JProfiler GUI cannot connect
	[Advanced] You have to configure triggers that record data and save snapshots that can be opened with the JProfiler GUI later on.
	Back Next Finish Cancel

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 .jprofiler15 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. 251].

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 8 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.

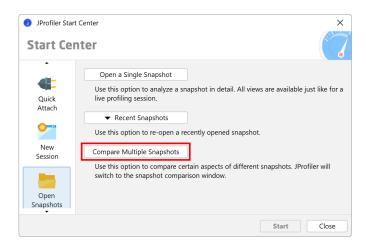
If the profiling agent is only loaded, but no profiling settings have been applied, no recording actions can be switched on and so <code>jpcontroller</code> will not be able to connect. This includes the

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

More information on the <code>jpenable</code> and <code>jpcontroller</code> executables is available in the command line reference [p. 251].

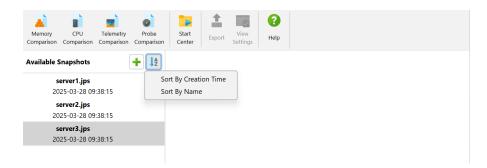
Comparing Snapshots

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 of 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 snapshots, 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.

1. Choose comparison type	Choose the threads that should be compared
2. Select snapshots 3. Thread selection	Please choose the thread for the comparison:
4. View parameters	First snapshot: Servlet request simulator 1 [main]
	Second snapshot: 🔿 Same selection as for first snapshot
	 Different thread
	Servlet request simulator 2 [main]
	A Dark Night N
	● Back Next ▶ Finish Cancel

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.

📭 File View Window Help	Snapshot Comparison - JProfiler —	
Memory CPU Telemetry Probe Comparison Comparison Comparison Comparison	Start Center Export View Export Settings Help	
Available Snapshots	Objects comparison	
server1.jps 2025-03-28 09:38:15	Aggregation: Classes Liveness mode: Live objects	
1 server2.jps 2025-03-28 09:38:15	Objects: Recorded objects Recording rate: All allocations	
server3.jps	Name Instance Count	Size 🕜
2 2025-03-28 09:38:15	char[] +15,764 (+72 %)	+2,786 kB
	java.lang.String +6,947 (+66 %)	+166 kB
	java.util.HashMap\$Node +5,267 (+47 %)	+168 kB
	com.sun.org.apache.xerces.internal.xni.QName +4,140 (+95 %)	+132 kB
	java.lang.StringBuilder +1,762 (+115 %)	+42,288 bytes
	java.lang.Object[] +1,673 (+47 %)	+125 kB
	com.sun.org.apache.xerces.internal.util.Symbol +1,305 (+91 %)	+41,760 bytes
	com.sun.org.apache.xerces.internal.util.XMLStri + 1,080 (+95 %)	+25,920 bytes
	byte[] +1,002 (+60 %)	+2,682 kB
	java.lang.String[] +994 (+48 %)	+118 kB
	com.sun.org.apache.xerces.internal.xni.XMLStri +990 (+95 %)	+23,760 bytes
	java.util.HashMap +797 (+25 %)	+38,256 bytes
	java.util.HashMap\$Node[] +786 (+25 %)	+88,360 bytes
	Total from 289 rows: +68,538 (+63 %)	+7,506 kB
	Q• Class View Filters	•
	Comparison 1 📥 Comparison 2	

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.

Dbjects Comparison View Settings	×
Size Scale	
● Automatic 😢 ─ Mixed units ─ MB ─ kB	Obytes
Primary Measure	0
O Instance count	
Shallow size	
Differences of Primary Measure	0
Sort by values	
Sort by percentages	
Show rows with a difference of zero 😮	
Only show classes that appear in both snapshots	
ок	Cancel

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.

Tile View Window Help	Snapshot Com	parison - JProfiler	– 🗆 X
Memory Comparison Comparison	Start Center Export Settings (2)		
Available Snapshots	Objects comparison		
server1.jps	Aggregation: Classes		
2025-03-28 09:38:15	Liveness mode: Live objects		
server2.jps	Objects: Recorded objects		
1 2025-03-28 09:38:15	Recording rate: All allocations		
server3.jps	Name	Size	Instance C 🚱
2 2025-03-28 09:38:15	char[]	+2,786 kB (+83 %)	+15,764
	byte[]	+2,682 kB (+80 %)	+1,002
	iava util HachMan®Node	169 LR (±17 %)	+5 267

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.

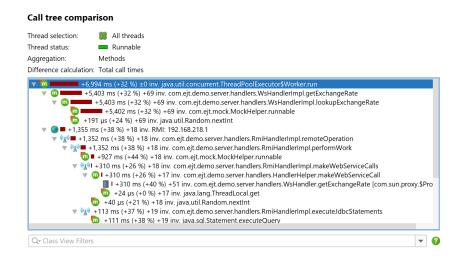
Objects comparison

Aggregation: Classe	es					
Liveness mode: Live o	bjects					
Objects: Recor	ded of	ojects				
Recording rate: All all	ocatio	ns				
	Name			Instance Count	~	Size 🕜
char[]				+15,764	(+72 %)	+2,786 kB
java.lang.String				+6,947 (+66 %)		+166 kB
java.util.HashMap\$No	ode 🗕			7 (+47 %)		+168 kB
com.sun.org.apache.x	erc	Create Allocation (Call Tree Comparison	95 %)		+132 kB
java.lang.StringBuilde	r	Create Allocation H	Hot Spot Comparison			+42,288 bytes
java.lang.Object[]				1		+125 kB
com.sun.org.apache.x	erc 🛓	Show Source	F4			+41,760 bytes
com.sun.org.apache.x	erc	Show Bytecode				+25,920 bytes
byte[]						+2,682 kB
java.lang.String[]		Sort Classes		Sort By Name		+118 kB
com.sun.org.apache.x	erc	Find	Ctrl+F	Sort By Instance Count		+23,760 bytes
java.util.HashMap	-	r mu	Culti			+38,256 bytes
java.util.HashMap\$No	ode 🔒	Export View	Ctrl+R	Sort By Size		+88,360 bytes
Total from 289 rows	: _	View Settings	Ctrl+T			+7,506 kB
Q- Class View Filters	_	· · · · · · · · · · · · · · · · · · ·	Currie Currie	J		•

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.

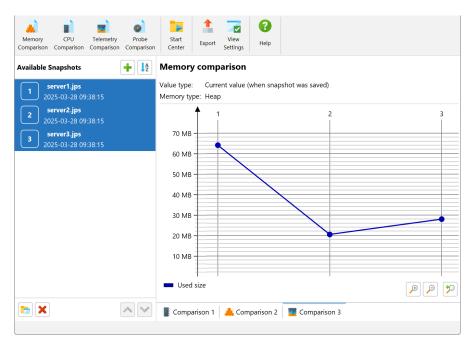
Tee Comparison View Settings
Time Scale
O Automatic 2 O Mixed units O s O ms O μs
Node Description
✓ Show percentage bar
🗹 Always show fully qualified names 🛛 😮
Always show signature
Shorten packages
Time Differences
 Sort by values
Sort by percentages
Show rows with a difference of zero 😮
Only show call stacks that appear in both snapshots
OK Cancel
OK Calleer

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.

1. Choose comparison type 2. Select snapshots	Select view parameters	
 Select snapshots Thread selection View parameters 	Please specify the following parameters that are necessary in order to comparison:	calculate the snapsho
	Thread status: Runnable	•
	Aggregation level: 0 Methods	•
	Difference calculation: Total call times	

Telemetry comparisons

For telemetry comparisons you can compare more than two 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.

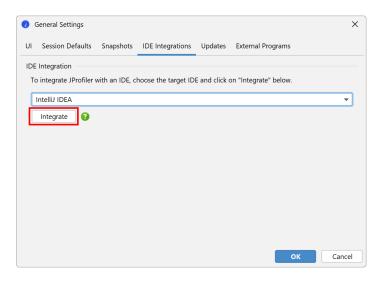
1. Choose comparison type	Choose the comparison type				
2. Select snapshots 3. Memory type 4. Comparison type	One value is extracted from each snapshot for the comparison graph. Please select what kind of value should be compared:				
5. Compared measurements	 Current value (when snapshot was saved) 				
	O Maximum value				
	◯ Value at bookmark				
	[Select]				
	Only bookmark names that exist in all snapshots are shown.				

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.

Telemetries	All objects dumps:	× 4	Aggregatic	on level:	Classes			•
	at 0:04.339.976	[Nar	ne	Instance Co	ount 🔻	Size	
Live Memory			byte[]		16	,184 (13 %)	904	kB
			java.lang.s	String	14,6	71 (12 %)	352	kB
I All Objects			java.util. 		E 710 // 0/1		182	kB
All Objects			java.lang	💼 Shov	w Selection In H	eap Walker	240	kB
Recorded Objects			java.util.o	741	_		116	kB
,			java.lang	🔮 Shov	w Source	F4	408	kB
Allocation Call Tree			java.secu	Shov	w Bytecode		131	kB
			java.awt.				214	
Allocation Hot Spots			java.awt.	Sort	Classes)	So	rt By N
Class Tracker			double[]	🔎 Find		Ctrl+F	💙 So	rt By In
Class Hacker			int[]	·			So	rt By Si
			sun.java2	🚺 Ехро	ort View	Ctrl+R		- 1
Heap Walker			java.lang		e:	C 1 7	,784 by	
					/ Settings	Ctrl+T	,240 by	
					1,598 (1 %)		51,136 by	_
CPU Views					1,588 (1 %)		50,816 by	_
			, ,		1,565 (1 %)		15,952 by	
Threads					1,490 (1 %)		35,760 by	
1 meaus					116,763 (100 %)		13,425	
Monitors & Locks		[Q . Class	View Filt	ers		•] 🕜

If the session was not started from the IDE, the built-in source code viewer is shown that uses 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

Ø Session Settings		×					
Application Settings	Session name: Animated Bezie	r Curve Demo Id: 101 💡					
Profiled JVM		dy running HotSpot/OpenJ9 JVM and profile it					
Code Editor) Select from all local JVMs 🔿 Attach to remote JVM 🔿 Kubernete:					
Call Tree Recording	10 1						
Call Tree Filters	Application Settings						
Trigger Settings	Java VM:	21 [C:\Users\ingo\jdks\jbrsdk-21.0.5-wi Configure JREs					
	Working directory:	[startup directory]					
Databases	VM options:	(
	Main class or executable JAR:	bezier.BezierAnim					
HTTP, RPC & JEE	Program arguments:	block					
JVM & Custom Probes	Open browser with URL						
Advanced Settings	Java File Path	emo\bezier\src					
	Source path ??						
	🔵 Library path 🕜						
General Settings Copy Settin	ngs From	OK Cancel					

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

Viewer Window bezier.BezierAnim	(C:\Users\ingo\projects	\jprofiler\dist\	demo\bezier\src) —	×
Show: 📀 BezierAnim				•
0 0				
General Information Constant Pool Interfaces Fields Methods Attributes	Minor version: Major version: Constant pool count: Access flags: This class: Super class: Interfaces count: Fields count: Methods count: Attributes count:	0x0021 [publ cp_info #25	ic] dezier/BezierAnim> <javax japplet="" swing=""></javax>	
Source Bytecode				

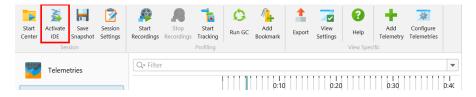
If the session is launched from the IDE, the integrated source code viewer is not used and the *Show Source* action defers to the IDE plugin. The IDE integrations support launched profiling sessions, opening saved snapshots as well as attaching to running JVMs.

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.

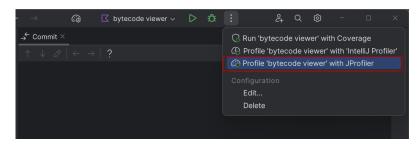
For snapshots, the IDE integration is set up by opening a snapshot file from within the IDE with the *File->Open* action or by double-clicking on it in the project window. Source code navigation from JProfiler will then be directed into the current project. Finally, the IDE plugin adds an *Attach to 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, or click on the drop-down menu next to the run or debug actions in the main tool bar to choose the "Profile with JProfiler" action. JProfiler can profile most run IDEA configuration types, including application servers.



The JProfiler plugin adds additional settings to run configurations which are not immediately visible. To access these settings, select the "Profile" option in the "Modify options" dropdown. All other profiling settings can be configured in the startup dialog of the JProfiler window.

Build and full	
java 21	Add Run Options
-cp install4j.install4j.ide_run.main @dist/install4j/bin/options/base.options @dist/install4j/bin/opt	Operating System ✓ Allow multiple instances Alt+U ✓ Environment variables Alt+E Redirect input
com.install4j.Install4JGUI Program arguments Press Alt for field hints	Java Do not build before run ✓ Use classpath of module Alt+O Modify classpath
Working directory: C:\Users\ingo\projects\install4j Environment variables: Environment variables or .env files	Add dependencies with "provided" scope to classpath Implicitly declared class run configuration Shorten command line ✓ Add VM options Alt+V
Separate variables with semicolon: VAR=value; VAR1=value1 Profile Use profiling settings: Project-specific Skip se Debug parameters for profiling agent:	Logs Specify logs to be shown in the console Save console output to file Show console when a message is printed to stdout Show console when a message is printed to stderr
Open run/debug tool window when started \times Allow multiple instances \times	Code Coverage Specify classes and packages Exclude classes and packages
	Startup/Connection ✓ Profile Alt+P
Ø JProfiler	Before Launch Add before launch task ✓ Open run/debug tool window when started Focus run/debug tool window when started Show the run/debug configuration settings before start

Once the profiling session is started, the output appears in a separate JProfiler tool window. That tool window displays the console output like the regular run tool window, along with a "JProfiler" tab that can be used after you connect with the JProfiler UI:

			tignore ild.grad	
	JPr	ofiler	🖒 byt	tecode viewer ×
	$\mathcal{C}_{\mathcal{O}}$			Console 😰 JProfiler
2 2				·Listening locally on port: 34558. ·Enabling native methods instrumentation. ·Can retransform classes.
\triangleright				· Can retransform any class. • Native library initialized
⊳	<u>=+</u>			· VM initialized
	ت ا			• Retransforming 7 base class files. • Base classes instrumented. • Waiting for a connection from the JProfiler GUI
() 약				· Using instrumentation · Time measurement: elapsed time · CPU profiling enabled
// В				sfully in 38 sec, 532 ms (a minute ago)

The JProfiler tool window is also shown when you open a JProfiler snapshot in IntelliJ IDEA or when you attach to a running JVM with the "Attach to JVM with JProfiler" action.

The "JProfiler" tab contains actions to start and stop data recording for CPU data, allocation data and probe events. Additionally, an action allows you to switch to the JProfiler window. The JProfiler window includes a similar action for switching back to the IDEA window so that it becomes convenient to work with the two separate windows. Precise source code navigation from JProfiler into IntelliJ IDEA is implemented for Java and Kotlin.

Profiling information is typically displayed in the JProfiler window, but the CPU graph data is also integrated in the IntelliJ IDEA UI because it makes sense to show this data directly in the source code. Use the "Apply graph" action in IntelliJ IDEA or generate a CPU graph in JProfiler to display CPU data within IntelliJ IDEA. 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 should generate the graph in the JProfiler window.

		tignore ild.gradle.kts		
	JProfiler	\Box bytecode viewer \times		
	C 🔲	: Console 🖄 JProfiler		
	🗀 ዺ Apply	Graph 🗙 Recordings: 🔘 🕮 🌡		
Т	Method		Total Time ${}^{\checkmark}$	
\triangleright				
⊘				
Þ.			Apply Graph	
!				
വ				
	Build completed	successfully in 38 sec, 532 ms (2 minutes ago)		

Once the CPU data has been applied, the "JProfiler" tab displays a list of recorded methods. Double-clicking on a method will take you to the source code. In the gutter of the source code editor, arrows for incoming and outgoing calls are added.

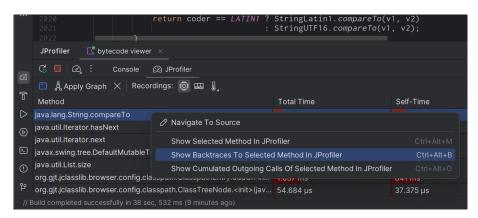
	\equiv \blacksquare jclas \checkmark $?$ master \checkmark \checkmark \odot \bigcirc \checkmark \leftarrow	\rightarrow	🚱 🖸 bytecode v	iewer ~	\triangleright	æ 🗖	:	ĉţ	Q E	ja −	· 0	×
	Project \checkmark \rightarrow	Commit	R ClasspathBrowser.k	⊑ c	lassPatl	hContaine	er.kt ×					Ļ
-0-	Cajclasslib C:\Users\ingo\projects.git\jclasslib		abstract class C override fun								Selectio	n: @
ଟ୍ଟ			return c	reateJr	eEntr	y()?.f	indCla	ss(cla	assNam	ie, moo	dulePath	Se 🗎
ያያ	> 🗋 gradle 30									1		_
80	> □ lib-compile 3 > □ modules 3		override fun for (ent if (lassp	ath) {						ψ
	> D web 34 O .gitignore 33			entry.m nergedE				ee(cla	issPat	hMode]	L, modul	eР
	ି build.gradle.kts		<u>}</u>									
	JProfiler bytecode viewer ×											
C2	🖒 🔲 🍳 : Console 🙆 JProfiler											
	🗀 🖧 Apply Graph 🗙 Recordings: 🧰 💷 🜡											
Т	Method	Total Time		Self-Tim	ne			Invoc	ations			
\triangleright	java.awt.EventDispatchThread.run	4.398 ms		2.753 m	s							
	java.util.concurrent.ThreadPoolExecutor\$Worker.run	1.556 ms		59 µs								
	org.gjt.jclasslib.browser.config.classpath.ClasspathBrowser.sync\$l			12 µs								
	org.gjt.jclasslib.browser.config.ClassPathContainer.mergeClassesl	1.556 ms		19 µs								
\odot	org.gjt.jclasslib.browser.config.classpath.ClasspathJrtEntry.merge			2.238 µs								
0-	org.gjt.jclasslib.io.JimageKt.forEachClassNameInJrt	1.548 ms		212 µs								
	org.gjt.jclasslib.io.JimageKt.forEachClassInJrt	1.548 ms		497 µs				1				
_ // В	uild completed successfully in 38 sec, 532 ms (3 minutes ago)					章	31:18 🤇) @	CRLF	UTF-8	4 spaces	ď

Clicking on a gutter icon displays the incoming or outgoing methods in a popup window, along with a bar chart showing the recorded times. Clicking on rows in the popup will navigate to the corresponding methods.

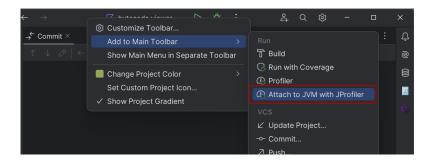
Also, the total recorded time and the invocation count for the target method will be shown at the bottom of the popup. The "Show in JProfiler" drop down in the bottom-right corner of the popup provides context-dependent navigation actions into the JProfiler UI. You can show the selected node or the corresponding call tree analysis in the method graph. For outgoing calls, the "Cumulated outgoing calls" analysis and for the incoming calls the "Backtraces" analysis is offered.

		$\blacksquare \ jclas \lor \ \mathfrak{P} \ master \lor \ \checkmark \ \mathfrak{S} \ \bigcirc \ \backsim \ \mathfrak{S} \ \leftarrow \ \rightarrow \qquad \mathfrak{Ca} \ \ \mathfrak{S} \ bytecode \ viewer \lor \ \mathfrak{S}$	> 🅸 🚺		ĉ+	Q ĝ				
	🕞 Clas	sspathBrowser.kt © DefaultTreeModel.java 🕼 ClasspathEntry.kt × 🖾 ClassPathContainer.kt © Even	ntDispatchT	hread.java		Stream.ja	va ⓒ		Ļ	
-0-		<pre>abstract class ClasspathEntry : ClasspathComponent { protected fun addEntry(path: String, moduleName: String?, classPathMo</pre>	del: Def	aultTre	eMode	l, mod	ulePat	hModel:		
Ê										
ያያ 	60 = 61 62	Drivate_fun_addEntrv(model: DefaultTreeModelpathComponents: List <st Choose outgoing calls</st 	rinơ> r	reset· P	oolea	n) {				
80		1.057 ms (293.519 inv.) org.gjt.jclasslib.browser.config.classpath.Classpath	Entry.ad	ddOrFind	Node -	1, mo	del, r	eset)		
		98.723 μs (53.301 inv.) <self-time></self-time>								
	⁶⁵ 20.643 μs (346.820 inv.) java.util.Iterator.hasNext									
		17.750 μs (293.519 inv.) java.util.Iterator.next								
		17.435 μs (293.519 inv.) java.util.List.size								
		3.689 μs (53.301 inv.) java.lang.Iterable.iterator								
œ۵		3.346 μs (53.301 inv.) javax.swing.tree.DefaultTreeModel.getRoot								
		This method: 1.219 ms total time, 53.301 invocations	🖄 Sho	w in JProfi	ler ~					
T		II (ireset) {								
\triangleright		<pre>model.nodesWereInserted(parentNode, intArrayOf(insertionIndex }</pre>	.))							
⊘										
2		<pre>fun addToClassPath(classpath: MutableList<classpathentry>): Boolean { return if (!classpath.contains(this)) {</classpathentry></pre>								
()		classpath.add(this)								
		true								
억		<pre>} else { false</pre>								
// Bu		14100								

The same navigation actions are also available in the context menu of the method table in the "JProfiler" tab:



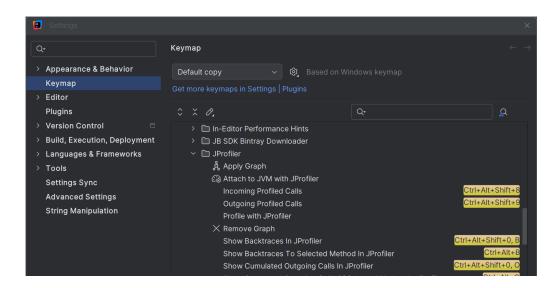
The JProfiler plugin provides a toolbar quick action for the "Attach to JVM with JProfiler" action that you can add to the main toolbar. With that action you can attach to a process that is already running and still get source code navigation from the JProfiler UI into IntelliJ IDEA as well as inline CPU graph data in source code editors:



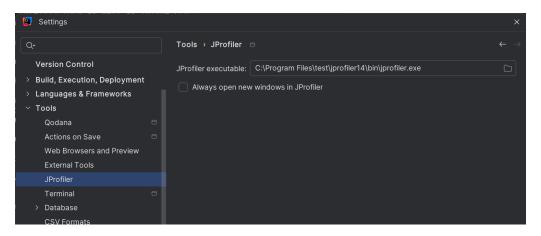
This is how the action button looks like once it has been added:

- →	6	☐ bytecode view	wer 🗸 🕞	æ	:	ĉ;	Q	ŝ	-	O	×
_→ Commit ×											Ģ
$\uparrow \downarrow \mathscr{O}$											Q
											ψ

The key bindings for all actions in JProfiler can be customized in the "Keymap" settings in Intellij IDEA. Given the limited availability of non-conflicting keyboard shortcuts, the navigation actions from the source code editor to the JProfiler UI are chained shortcuts where you first hit Ctrl-Alt-Shift-O and then another key to select the navigation action. If you frequently use this functionality, you may want to assign simpler keyboard shortcuts.



On the *Tools->JProfiler* page 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.

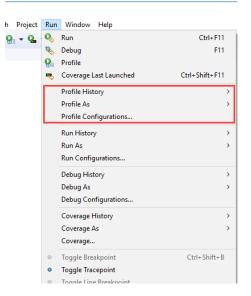


Eclipse integration

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.

٢	eclipse	-workspa	ce - Eclipse	:												
File	Edit	Source	Refactor	Navigate	Search	Project	Run	Window	Help							
1	• 🛛	6	• ی	x 💠 🕶	0 - 9	- 🖣	- 9	- 😰 🤆	9 - 🤔	🖨 🤞	- 1	• 🕅	• *>	~ -	·	-

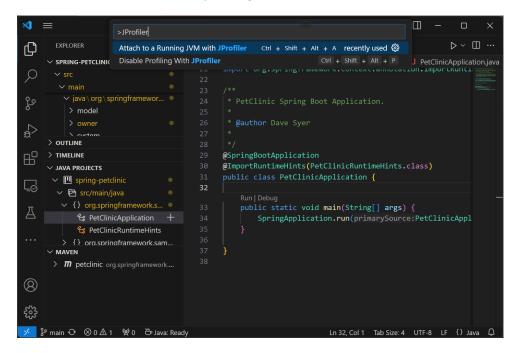


If the menu item *Run->Profile* ... does not exist in the Java perspective, enable the "Profile" actions for this perspective under *Window->Perspective->Customize Perspective* by bringing the *Action Set Availability* tab to front and selecting the *Profile* checkbox.

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

VS Code integration

The VS Code extension adds an action Enable Profiling with JProfiler. When invoked, the debug and run actions will start profiling for Java launch configurations. JProfiler will be started, and the session startup dialog where you can configure profiling settings. The application will be started after the session startup dialog is confirmed.



With the Disable Profiling with JProfiler action, the default behavior for run and debug actions is restored. Notifications about the changes in the profiling mode are shown in VS Code as toast messages in the bottom-right corner of the editor. Both Enable Profiling with JProfiler and Disable Profiling with JProfiler have the same default keybinding, so it can be used to toggle the profiling mode.

	 ✓ JAVA PROJECTS ✓ I spring-petclinic ✓ Src/main/java ✓ () org.springframework.s ✓ () org.springframework.sam ✓ () org.springframework.sam ✓ () org.springframework.sam ✓ MAVEN ✓ m petclinic org.springframework 	30 31 32 33 34 35 36 37 38	<pre>@ImportRuntimeHints(PetClinicRuntimeHints.class) public class PetClinicApplication { Run Debug public static void main(String[] args) { SpringApplication.run(primarySource:PetClinicAppl } }</pre>
833 100			① Profiling with JProfiler has been enabled
՝ չ՝ ֆ	⁹ main 😌 🛞 0 🛆 1 👾 0 😁 Java: Ready		Ln 32, Col 1 Tab Size: 4 UTF-8 LF {} Java 🗘

To profile an already running JVM, use the Attach to a Running JVM with JProfiler action.

Source navigation actions in JProfiler will show the corresponding source code in VS Code. To get source navigation into VS Code for a JProfiler snapshot, open the snapshot from within VS Code with *File->Open*.

NetBeans integration

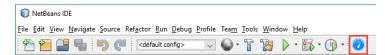
In NetBeans, you can profile standard, free form and Maven projects that use the exec Maven plugin. To profile your application from NetBeans, choose one of the profiling commands in the *Run* menu or click on the corresponding toolbar button. For Maven projects that start an application in another way 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 filenbproject/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.



	Profile Main Project With JProfiler	Ctrl+Shift+F
	Profile File With JProfiler	
	Profile Project (C:\Users\ingo\Documents\NetBeansProjects\maven)	Ctrl+F2
	Profile File	
	Profile Test File	
• <u>··</u> •	Attach to Project	
1	Attach to External Process	
	Finish Profiler Session	Shift+F2
	Take Thread Dump	
	Take Heap Dump	
Î	Run GC	
	Take Snapshot of Collected Results	Alt+F2
Þ	Reset Collected Results	Alt+Shift+F
4	Insert Profiling Point	

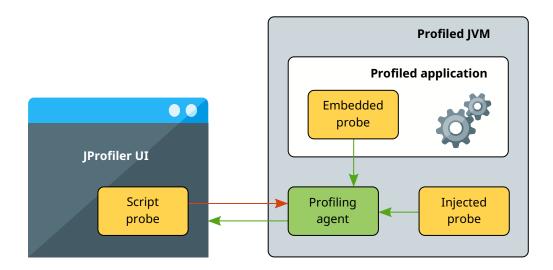
A Custom Probes

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 three different ways to do that:

- A **script probe** [p. 159] 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. 168] 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. 163] 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.

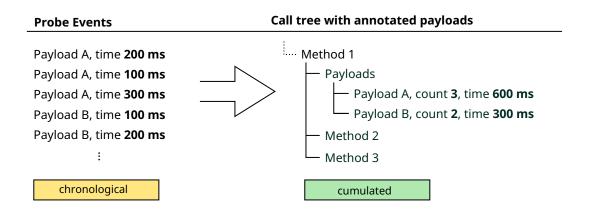


The next question is: what should JProfiler do with the string that you have created? There are two different strategies available: payload creation or call tree splitting.

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.

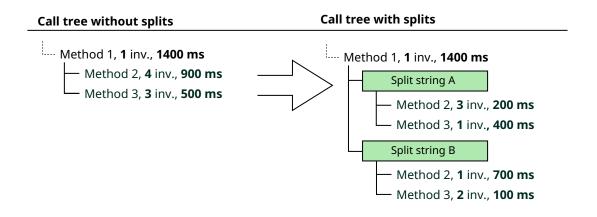
Script probes create payloads automatically from the return value of the configured script. Injected probes are similar, they return the payload description from an interception handler method annotated with PayloadInterception either as a string or as a Payload object for advanced functionality. Embedded 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

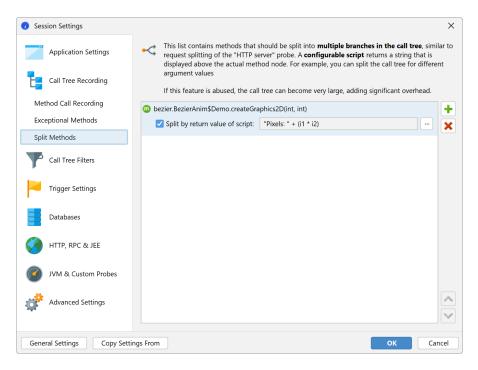
The probe can also take the perspective of the execution site. In that case, it is not important how the intercepted method is called, but rather what method calls are executed after it. A typical example is a probe for a servlet container where the extracted string is a URL.

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 it 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. 186]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.

 Configure Telemetries 	×
Several types of telemetries are shown in the Telemetries section:	
The set of standard JVM telemetries that is always shown	
Probes telemetries that are only shown in the probe views by default, but car	n be added here as well
Custom telemetries from MBeans or scripts	
Probe telemetries and custom telemetries can be added below.	
TataBus connections [plain]	+
DataBus connections [Script line ""DataBus.getInstance().getActiveConnection	Add Probe Telemetry
	Add MBean Telemetry Line
[Add Script Telemetry Line
	Add Empty Custom Telemetry View
	
Help	OK Cancel

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 API and the injected probe API which generate 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.

art Detach Snapshot Settings Session	Window Help factor Start Stop Start Recordings Recordings Profiling	Run GC Add Bookmark	Export View Settings	Help Record View Sp View Sp	be View Ti
Databases	Siz Time Line	Control Objects	📥 Call Tree 🔸	F Records request to t	actorial cache de factorial cache
HTTP, RPC & JEE	Both open and closed 💌	Filter in all text colu	imns 💌 🔍 🖓	lter	-
	ID Name 🔺	Start Time	End Time	Event Count	Event Duration
	1 Factorial cache c	0:00.506 [Mar 3,		864	10,450 m
JVM & Custom Probes	4 Factorial cache c			796	9,744 m
	3 Factorial cache c	0:00.506 [Mar 3,		770	9,928 m
Class Loaders	2 Factorial cache c	0:00.506 [Mar 3,		736	9,107 m
Exceptions	5 Factorial cache c	0:00.506 [Mar 3,		834	10,619 m
Sockets Files Processes Garbage Collector					
Factorial server					
Factorial cache					
MBeans	Total from 5 rows:			4,000	49,850 m

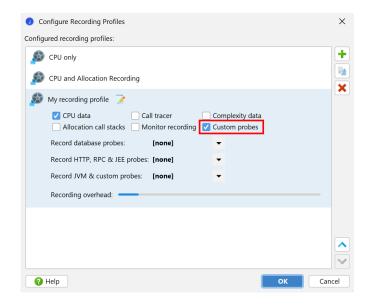
Control objects have display names that have to be specified when they are opened. If a new control object is used when creating a probe event, the probe has to provide a name resolver in its configuration.

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 filter for single event types. More importantly, the timeline 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".

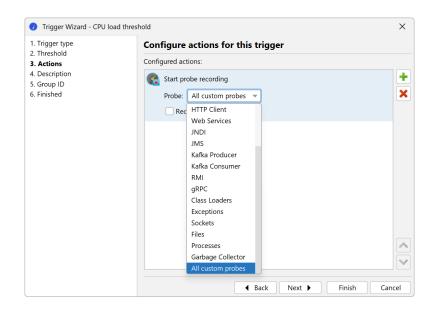
Session View Profiling N	Window Help factorial.jps - J	IProfiler			_					
Start enter Detach Save Session Session Session	Start Recordings Recordings Tracking Profiling	Add Bookmark	Export Settings	Help R w Specific	ecord Probe Probe					
Databases	Sine Line	bjects	📥 Call Tree 🕠	Records reque	Factorial st to the factori	(C-				
HTTP, RPC & JEE	Both open and closed 💌 Sort by start time 💌 🔍 Filter									
	Control Objects		0:10	0:20	0	:30				
JVM & Custom Probes	Factorial cache connection #4 [ID 2]									
	Factorial cache connection #1 [ID 1]									
Class Loaders	Factorial cache connection #3 [ID 3]									
Exceptions	Factorial cache connection #2 [ID 4]									
	Factorial cache connection #5 [ID 5]									
Sockets										
Files										
Processes										
Processes										
Garbage Collector										
Factorial server										
Factorial cache										
	🔲 Idle 💻 Read 💻 Write									
MBeans	idie 📥 kead 📕 Write				/	P				
	3 recordings		VA	A #1						

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.

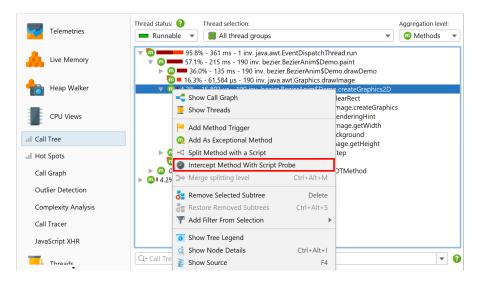
Ø Session Settings		×
Application Settings	Script probes defined for the current session:	
Application settings	mage buffers	•
Call Tree Recording	Measures the areas of image buffers	
Call Tree Filters		×
Trigger Settings		
Databases		
HTTP, RPC & JEE		
JVM & Custom Probes		
Built-In Probes		
Script Probes		
Custom Probes		H
Advanced Settings		
General Settings Copy Setti	ings From ОК Сап	cel

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.

Oreate Script Probe	X
1. Name and description 2. Payload interceptions	Specify the payload interceptions
Advanced options:	Intercepted methods:
Call tree annotations	bezier.BezierAnim\$Demo.createGraphics2D(int, int)
3. Finished	▼ Payload creation script: "Pixels:" + Search in Other JAR or Class Files Search in Profiled Classes Enter Manually (Advanced)
	Quick Help Use the current object (null for static methods) as well as the method parameters to construct and return a payload string. Probe events, probe call tree, and probe hot spots will be shown for these payloads. ▼ Advanced Options ▲ Back Next ▶ Finish Cancel

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.

🥑 Settings Edit Search Code Help I	idit ×
ち 더 🔰 🐰 🚦 📓 🌶	≥ 💋 🛓 🛦 (2)
Undo Redo Copy Cut Paste Show F	nd Replace Modify Test Classpath Compile Help
Please enter an expression (no trailing semic consists of regular Java code. The following - <u>com.jprofiler.api.agent.ScriptContext</u> scr - <u>java.lang.Class<? ></u> c - bezier.BezierAnim.Demo currentObject - int il	
- int i2	
The expected return type is java.lang.String	
Script:	
1"Pixels: " + (i1 * i2)	
	OK Cancel

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.

1)	Intercepting method A : • object c • name n scriptContext.putObject(c,n); return null;	
2	Timed intercepting method B: • object c Image: scriptContext.getObject(c);	

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. 152]. 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. 152]

A.3 Injected Probes

Similarly to script probes, injected probes define interception handlers for selected methods. However, injected probes are developed outside 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 to 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 15.0.2.

Jar, source and javadoc artifacts are published in the Maven Central 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 probe'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 <code>isReentrant()</code> to return true. Split probes can also create a probe view and publish the split strings as payloads if you override <code>isPayloads()</code> 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 <code>@Telemetry</code> and return a numeric value. See the chapter on probe concepts [p. 152] for more information. The <code>@Telemetry</code> 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 methods will be far too 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 invocation counts, only shows longer method calls, and several views do not have their full functionality when sampling is used.

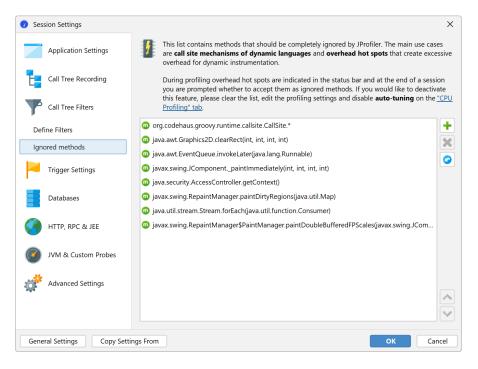
To alleviate the problem with instrumentation, JProfiler has a mechanism called auto-tuning. From time to time, the profiling agent checks for methods with high instrumentation overhead and transmits them to the JProfiler GUI. In the status bar, an entry alerting to the presence of overhead hot spots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

Complexity Analysis	
Call Tracer	
JavaScript XHR	
Threads	Q+ Call Tree View Filters
	🚦 7 overhead hot spots 🔰 🕆 💡 🏓 🕒 1 active recording 🔹 Auto-update 5 s VM #

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.

Vernead H	lot Spots Detected	×
	ethods with excessive instrumentation overhead have been detected. They are ly, their execution times are very short, and the time required for measuring th rtional.	
	ey distort the overall picture, JProfiler recommends that you add these metho ed methods.	ds to the list
You can	edit the list of ignored methods in the filter settings section of the session sett	lings.
🄟 java.awt.Gra	phics2D.clearRect(int, int, int, int)	X
🎯 java.awt.Eve	ntQueue.invokeLater(java.lang.Runnable)	
🌀 javax.swing.	JComponentpaintImmediately(int, int, int, int)	
回 java.security	AccessController.getContext()	
🌀 javax.swing.	RepaintManager.paintDirtyRegions(java.util.Map)	
🔟 java.util.stre	am.Stream.forEach(java.util.function.Consumer)	
🌀 javax.swing.	${\tt RepaintManager} \\ {\tt PaintManager.paintDoubleBufferedFPScales} \\ (javax.swing.JContextCo$	mpo
		
Disable auto	-tuning	
🕜 Help	ок	Cancel

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.

Thread status: 2 Thread selection:	,	Aggregation level:
30 314 ms - 1 inv. java.awt.EventDis 94.5% - 314 ms - 1 inv. java.awt.EventDis 56% - 10-010 ms - 1 inv. java.awt.EventDis 56% - 56\% - 5		
 Add Method Trigger Add As Exceptional Method Split Method with a Script Intercept Method With Script Probe Merge splitting level Ctrl+Alt+M 		
Remove Selected Subtree Delete Restore Removed Subtrees Ctrl+Alt+S		
Add Filter From Selection Show Tree Legend Show Node Details Show Node Details Show Source F4 Show Bytecode	O lanore bezier	
Qr 🛃 Evpand Multiple Levels		▼ €

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

Ø Session Settings	×
Application Settings	C Enable CPU profiling
Application settings	Auto-Tuning For Instrumentation
Call Tree Recording	C Enable auto-tuning 😮
	A method is an overhead hot spot and will be suggested for inclusion into the list of ignored methods , if both of the following conditions are true:
Call Tree Filters	1. The total time of the method is more than 10 = per mille of the entire total time
Trigger Settings	2. The average time of the method is less than 100 🗢 µs
Databases	Auto-tuning is only performed if the method call recording type is set to "Instrumentation" on the <u>method call recording tab</u> .
HTTP, RPC & JEE	Call Tree Recording Options CPU times for instrumentation: • Elapsed times • Elapsed times • CPU times • CPU times
JVM & Custom Probes	 Instrument native methods Thread resolution for async sampling
Advanced Settings	Exceptional Method Run Recording Maximum number of separately recorded method runs: 5 \$ 2
CPU Profiling	Time type for determining exceptional method runs:
Probes & JEE	
Memory Profiling	Call Tree Splitting
Thread Drofiling	Maximum number of splits: 128 🗢 🕜
General Settings Copy Setti	ngs From OK Cancel

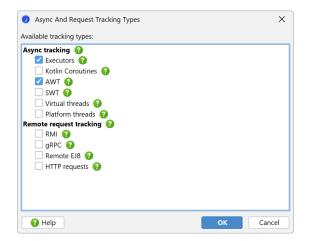
B.2 Async And Remote Request Tracking

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. 190] 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. 53] 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

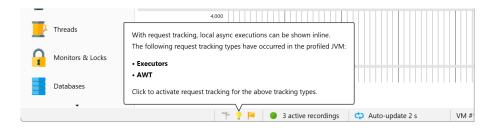
Async mechanisms can be implemented in various ways, and the semantics of starting tasks on a separate thread or in a different JVM cannot be detected in a generic way. JProfiler explicitly supports several common asynchronous and remote request technologies. You can enable or disable them in the request 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 \mathbf{F} "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 checkbox at the top of the analysis allows you to do that where appropriate.

Thread status: 🕜	Thread selection:		Aggregation leve	l:
🔲 Runnable 🔻	Search and a search and a search and a search a	•	🔟 Methods	▼
Async tracking: Execu	ators and AWT Inline Async Executions			
🕨 💿 💻 24.8% - 283	773 ms - 1 inv. jdbc.JdbcDemo.main 3 ms - 5 inv. java.util.concurrent.ThreadPoolExecutor\$Worker.run 2 μs - 1 inv. java.awt.EventDispatchThread.run			

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

⁽¹⁾ https://kotlinlang.org/docs/reference/coroutines.html

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.

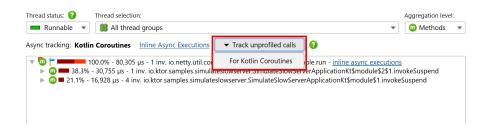
18 execution	n sites were were inlined 😧	× 🖉 🤤
Thread status:	Thread selection:	Aggregation level:
💻 Runnable	🕷 All thread groups	Methods
Add async e	execution time to tree 😮	
Show susper	nded time 😗	
	 100.0% - 80,305 μs - 1 inv. io.netty.util.concurrent.FastThreadLocalRunnable.run 8 ms async execution (79,567 μs was already inline) 96.2% - 2,008 ms suspended time 1.5% - 30,755 μs - 1 semantic inv. io.ktor.samples.simulateslowserver.SimulateSlowServerApplication 1.5% - 30,725 μs - 1 inv. io.ktor.response.ApplicationResponseFunctionsKtrespondText\$default 0.0% - 4 μs - 1 inv. io.ktor.response.ApplicationResponseFunctionsKtrespondText\$default 0.0% - 16,928 μs - 2 semantic inv. io.ktor.samples.simulateslowserver.SimulateSlowServerApplication 0.0% - 16,928 μs - 2 semantic inv. io.ktor.samples.simulateslowserver.SimulateSlowServerApplication 0.0% - 4,296 μs - 2 inv. kotlinx.coroutines.time.TimeKt.delay 0.2% - 4,296 μs - 2 inv. java.time.Duration.ofSeconds 	
Q• Call Tree Vi	iew Filters	~ 2

Suspending methods can interrupt the execution which is then possibly resumed on different threads. Methods where suspension was detected have an additional \subseteq "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 \bigcirc "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.



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. 11] 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
- gRPC
- 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.

rread status: 😮 Thread selection:		Aggregation lev	el:
🔲 Runnable 🔻 🛛 🚼 All thread groups	•	Methods	
100.0% - 4,278 ms - 5 inv. c.e.d.server.DemoServer\$1.run 99.9% - 4,273 ms - 5 inv. c.e.d.s.handlers.RequestHandler.run 98.9% - 4,230 ms - 5 inv. c.e.d.s.handlers.RequestHandler.performWork 87.3% - 3,736 ms - 5 inv. c.e.d.s.handlers.RequestHandler.makeJpaCall 1 8.1% - 346 ms - 5 inv. c.e.d.s.handlers.RequestHandler.makeJpaCall 1 2.3% - 99,190 µs - 5 inv. c.e.d.s.handlers.RequestHandler.makeZmathere.makeJpaCall			
 Q 1 2.3% - 97,763 µs - 5 inv. c.e.d.s.handlers.HandlerHelper.makeRmiCall 1.9% - 80,484 µs - 5 inv. jr.registry.Registry.lookup 0.3% - 10,731 µs - 5 inv. jr.registry.LocateRegistry.getRegistry 0.3% - 10,731 µs - 5 inv. c.e.d.s.handlers.RmiHandler.renoteOperation (j.proxy2.\$Proxy3) - jump 	o to exect	<u>ution site</u> [call site	e: #
 0 1.1% - 48,322 µs - 5 inv. c.e.d.s.handlers.RequestHandler.makeHttpCall 0.0% - 6 µs - 5 inv. j.util.Random.nextInt 1.0% - 42,792 µs - 5 inv. c.e.d.s.handlers.RequestHandler.workWithGlobalResource 0.0% - 19 µs - 1 inv. c.e.d.s.handlers.RmiHandler.<clinit></clinit> 0.0% - 4 µs - 1 inv. c.e.d.s.handlers.RequestHandler.clinit> 			
y 0.0% - 4 μs - τ inv. c.e.u.s.nanuers.κequestranger. <clinit></clinit>			
Q- Call Tree View Filters			

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.

ead status: 😢 Thread selection:		Aggregation leve	el:
All states All thread groups	•	Methods	
72.3% - 143 s - 9 inv. j.u.concurrent.ThreadPoolExecutor\$Worker.run			_
9.1% - 18,122 ms - 15 inv. called from <u>call site #1 (remote VM #2)</u>			
▼ 18,118 ms - 15 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation			
🔻 🙀 🛯 9.1% - 18,118 ms - 15 inv. c.e.d.s.handlers.RmiHandlerImpl.performWork			
Image: Second S Second Second Sec			
د 🕼 🕨 🖌 🖉 🕒 🗩 🖉 🕒 🗩 🕹 🕹 🗩 🕹 🕹 🕹 🕹 🕹 🕹 🕹 🕹 🕹 🕹 🕹 🕹 🕹			
🔟 0.3% - 631 ms - 15 inv. c.e.mock.MockHelper.runnable			
🔟 0.0% - 28 μs - 15 inv. j.util.Random.nextInt			
0 3.1% - 6,211 ms - 90 inv. c.e.d.s.handlers.DemoHttpServer.lambda\$start\$0			
🔟 💻 27.7% - 54,909 ms - 1 inv. c.e.d.s.test.RemoteDemoServer.main			

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. 195] feature to split the call tree further, as shown in the screen shot below.

Thread status: 😮 Thread selection:		Aggregation leve	el:
All states 🔻 🦉 All thread groups	▼	Methods	•
 w 100.0% - 141 s - 8 inv. j.u.concurrent.ThreadPoolExecutor\$Worker.run w 11.6% - 16.344 ms - 15 inv. called from call site #1 (remote VM #3) w 10% - 1.6% - 1.477 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [merged exceptional runs] w 10% - 1.0% - 1.477 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.performWork w 0.5% - 708 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.performWork w 0.5% - 684 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.makeHttpCalls w 0.1% - 644 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.makeHttpCalls w 0.1% - 1.369 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 0.5% - 108 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 0.5% - 1343 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 0.5% - 1,343 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.0% - 1,343 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.0% - 1,343 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.0% - 0.9% - 1,275 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 1 inv. c.e.d.s.handlers.RmiHandlerImpl.remoteOperation [exceptional run] w 1.399 ms - 180 ms			
Q+ Call Tree View Filters		-	

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.

Q- Call Tree View Filters					
	🕇 🎽 🔵 3	active recordings	Auto-update 5 s	VM #1	Profiling

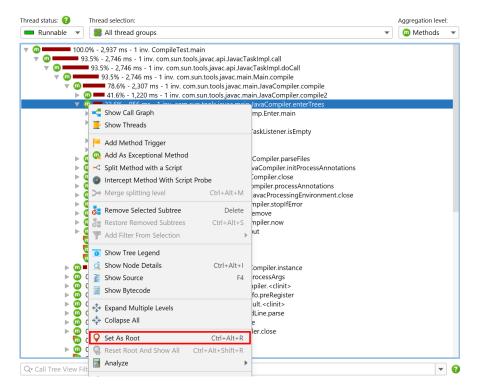
B.3 Viewing Parts Of The Call Tree

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 subtree, 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 subtree can be analyzed separately. Once you have found the entry point to such a subtask, the surrounding call tree is only a distraction and the timing percentages in the subtree inconveniently refer to the root of the entire call tree.

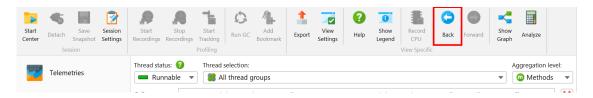
To focus on a particular subtree, JProfiler offers the *Set As Root* context action in the call tree and the allocation call tree views.



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.

ead status: 🕜	Thread selection:		Aggregation level
Runnable	🝔 All thread groups	•	Methods
□ □	0.0% - 956 ms - 1 inv. com.sun.tools.javac.comp.Ente 100.0% - 956 ms - 1 inv. com.sun.tools.javac.comp.Ente 168.1% - 651 ms - 1 inv. com.sun.tools.javac.comp.Ente 9% - 267 ms - 1 inv. com.sun.tools.javac.comp.Ente - 37 543 us - 1 inv. com sun tools.javac.comp.Ente		× tring[], java.lang.S m.sun.tools.javac.
			ОК
- Call Tree View F	19.		

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.

hrea	d status:	? Thread selection:	Aggregation level:	Hot spot options:	
	Runnabl	e 🔻 😻 All thread groups	👻 🔟 Methods 📼	Self times	
all t	ree root:	com.sun.tools.javac.main.JavaCompiler.enterTrees	com.sun.tools.javac.main.JavaComp	iler.compile ← cor ►	Show more
		Hot Spot	Self Time 🔍	Average Time	Invocations
► /	👠 com.si	un.tools.javac.util.List.reverse	73,956 µs (7 %)	18 µs	4,02
- 1	👠 com.si	un.tools.javac.util.List.prependList	71,340 µs (7 %)	30 µs	2,35
-	🔥 com.si	un.tools.javac.util.List. <init></init>	67,065 µs (7 %)	0 µs	298,47
-	🔥 com.si	un.tools.javac.file.ZipFileIndex\$ZipDirectory.readEntry	65,179 µs (6 %)	2 µs	26,8
	🔥 com.si	un.tools.javac.util.List.nonEmpty	59,241 µs (6 %)	0 µs	591,3
	🔥 java.ut	il.AbstractCollection. <init></init>	31,504 µs (3 %)	0 µs	298,4
	com.si	un.tools.javac.util.List.setTail	29,018 µs (3 %)	0 µs	291,3
	🔥 com.si	un.tools.javac.file.ZipFileIndex\$Entry.compareTo(java	21,872 µs (2 %)	0 µs	91,5
	🔥 com.si	un.tools.javac.file.ZipFileIndex\$Entry.compareTo(com	21,785 µs (2 %)	0 µs	91,5
- 1	🔥 java.ut	il.Map.get	18,226 µs (1 %)	0 µs	50,0
- 1	💧 java.ut	il.Arrays.sort	16,364 µs (1 %)	818 µs	
. 7	com.si	un.tools.javac.file.ZipFileIndex.get4ByteLittleEndian	14,128 µs (1 %)	0 µs	133,2
		un.tools.javac.file.ZipFileIndex.get2ByteLittleEndian	12,811 µs (1 %)	0 µs	107,7
		ng.String.compareTo	12,328 μs (1 %)	0 us	92.8
- 1		un.tools.javac.file.RelativePath\$RelativeFile. <init>(co</init>		0 µs	11,7
. 7		un.tools.javac.util.Name.getBytes	10,141 µs (1 %)	0 µs	14,8
	-	un.tools.javac.file.ZipFileIndex\$ZipDirectory.buildIndex	=	440 µs	
		un.tools.javac.util.SharedNameTable.fromUtf	8,390 µs (0 %)	0 µs	11.6
. 1		na Strina zinit>	7 700 µc (0 %)	0.00	28 54

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 subtrees by selecting them and hitting the Delete key or by choosing *Remove Selected Subtree* from the context menu. Times in ancestor nodes will be corrected accordingly as if the hidden nodes did not exist.

Thread status: 🕜	Thre	ad selection:			Aggregation level:
Runnable 🔻	8	All thread groups			▼
▼ (1) ▼ (1) ► (1)	m	93.5% - 2,746 ms - 1 inv. com.si 78.6% - 2,307 ms - 1 inv. com 41.6% - 1,220 ms - 1 inv. com	.sun.tools.javac.m .sun.tools.javac.m	ain.JavaCompiler.compile	
· · · ·		Show Call Graph		mp.Enter.main	
		Show Threads		askListener.isEmpty	
		Add Method Trigger		asklistener.isempty	
	0	Add As Exceptional Method		Commilian more films	
		Split Method with a Script		Compiler.parseFiles vaCompiler.initProcessAnnotations	
▶	ا 🖉	Intercept Method With Script	Probe	Compiler.close	
		Merge splitting level	Ctrl+Alt+M	ompiler.processAnnotations avacProcessingEnvironment.close	
	ð	Remove Selected Subtree	Delete	mpiler.stopIfError emove	
▶ (0 a	Restore Removed Subtrees	Ctrl+Alt+S	mpiler.now	
►	97	Add Filter From Selection	•	ut	
	j o	Show Tree Legend			
▶ @		Show Node Details	Ctrl+Alt+I	iompiler.instance	
	N.	Show Source	F4		
Q- Call Tree View Fil	IT	Show Putocodo			· · · · · · · · · · · · · · · · · · ·

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 subtrees. The *Remove subtree only* option only removes the selected subtree. 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.

📀 Remove Mode	×	
There are several ways to ren	nove the selected node:	
O Remove all invocations	of the selected method	• 0
Remove subtree only		
O Set self-time to zero		
	ОК	Cancel

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 Subtrees* button. Clicking on that button will bring up a dialog where you can select removed elements that should be shown again.

nread status: 🕜	Thread selec				
Runnable 🔻	🍔 All thr	ead groups			 Methods
emoved nodes: 1	I removed not	le			
v 🔟 🗖	90.3% -	1,789 ms - 1 inv	. com.sun.tools.ja	vac.main.Main.compile	
▼ 🚥	68.2%	- 1,351 ms - 1 in	iv. com.sun.tools.j	avac.main.JavaCompiler.compile	
▶			1 inv. com.sun.toc m.sun.tools.javac	Ø Select Removed Nodes to be Restored	×
▶			om.sun.tools.java .sun.tools.javac.r	Currently removed nodes:	
	0.0% - 27	µs - 1 inv. com.s	sun.tools.javac.m	🛛 🔟 com sun tools javac main JavaCompiler er	nterTrees(com sun tools ia
A (com.sun.tools.javac.main.JavaCompiler.er	nterTrees(com.sun.tools.ja
A A A	0.0% - 10	μs - 1 inv. com.s	sun.tools.javac.pr	com.sun.tools.javac.main.JavaCompiler.er	nterTrees(com.sun.tools.ja
* * *	0.0% - 10 0.0% - 7	us - 1 inv. com.s us - 1 inv. com.su	sun.tools.javac.pr un.tools.javac.ma	com.sun.tools.javac.main.JavaCompiler.er	nterTrees(com.sun.tools.ja
* * * *	0.0% - 10 0.0% - 7 0.0% - 7	µs - 1 inv. com.s µs - 1 inv. com.su µs - 1 inv. com.su	sun.tools.javac.pr un.tools.javac.ma un.tools.javac.util	u com.sun.tools.javac.main.JavaCompiler.ei	nterTrees(com.sun.tools.ja
* * * * *	 0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 4 	µs - 1 inv. com.s us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su	sun.tools.javac.pr un.tools.javac.ma un.tools.javac.util un.tools.javac.ma	com.sun.tools.javac.main.JavaCompiler.et	nterTrees(com.sun.tools.ja
	0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 4 0.0% - 3	μs - 1 inv. com.s us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su	sun.tools.javac.pr un.tools.javac.ma un.tools.javac.util un.tools.javac.ma un.tools.javac.util	com.sun.tools.javac.main.JavaCompiler.er	nterTrees(com.sun.tools.ja
	0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 4 0.0% - 3 0.0% - 1	μs - 1 inv. com.s us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 2 inv. java.lau	sun.tools.javac.pr un.tools.javac.ma un.tools.javac.util un.tools.javac.ma un.tools.javac.util ng.StringBuilder.	com.sun.tools.javac.main.JavaCompiler.et	nterTrees(com.sun.tools.ja
	 0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 4 0.0% - 3 0.0% - 1 0.0% - 1 	us - 1 inv. com.s us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 2 inv. java.lai us - 5 inv. java.lai	sun.tools.javac.pr un.tools.javac.ma un.tools.javac.util un.tools.javac.util un.tools.javac.util ng.StringBuilder. ng.StringBuilder.a	com.sun.tools.javac.main.JavaCompiler.et	nterTrees(com.sun.tools.ja
	0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 4 0.0% - 3 0.0% - 1 0.0% - 1 0.0% - 1	us - 1 inv. com.s us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 2 inv. java.la us - 5 inv. java.la us - 2 inv. java.la	sun.tools.javac.pr un.tools.javac.ma un.tools.javac.util un.tools.javac.util un.tools.javac.util ng.StringBuilder. ng.StringBuilder.a ng.StringBuilder.t	com.sun.tools.javac.main.JavaCompiler.et	terTrees(com.sun.tools.ja
	0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 3 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1	us - 1 inv. com.s us - 2 inv. java.la us - 2 inv. java.la 5 ms - 1 inv. com	sun tools javac.pr un tools javac.ma un tools javac.util un tools javac.util ng.stringBuilder. ng.StringBuilder. ng.StringBuilder. n.s.Untools.javac.u	com.sun.tools.javac.main.JavaCompiler.et	ter Trees(com.sun.tools.ja
▶ 0	0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 4 0.0% - 3 0.0% - 1	μs - 1 inv. com.st μs - 2 inv. java.lat μs - 2 inv. java.lat 5 ms - 1 inv. com.st μs - 1 inv. com.st μs - 1 inv. com.st	sun tools javac.pr un tools javac.ma un tools javac.util un tools javac.util ng.StringBuilder. ng.StringBuilder. ng.StringBuilder. n.sun.tools.javac.ru un.tools.javac.ru		tter Trees (com. sun. tools ja
▶ 0	Π 0.0% - 10 Π 0.0% - 7 Π 0.0% - 7 Π 0.0% - 4 Π 0.0% - 3 Π 0.0% - 1	us - 1 inv. com.s us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 1 inv. com.su us - 2 inv. java.lau us - 5 inv. java.lau us - 2 inv. java.lau 5 ms - 1 inv. com.su s - 1 inv. com.su	sun tools javac.pr un tools javac.uti un tools javac.uti un tools javac.uti ng.StringBuilder. ng.StringBuilder. ng.StringBuilder. ns.un.tools javac.ru un tools javac.mai	com.sun.tools.javac.main.JavaCompiler.er Type into the list to start search	terTrees(com.sun.tools.ja
	0.0% - 10 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 7 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 1 0.0% - 610 0.1% - 2,493 0.0% - 670 µ 0.0% - 427 µ	$\mu_{s} = 1$ inv. com.st $\mu_{s} = 2$ inv. java.lai $\mu_{s} = 2$ inv. java.lai $\mu_{s} = 2$ inv. java.lai $\mu_{s} = 1$ inv. com $\mu_{s} = 1$ inv. com.st $\mu_{s} = 1$ inv. com.st $\mu_{s} = 1$ inv. com.st $\mu_{s} = 1$ inv. com.st	sun tools javac.pr un tools javac.ma un tools javac.util un tools javac.util ng.StringBuilder. ng.StringBuilder. ng.StringBuilder. n.sun.tools.javac.ru un.tools.javac.ru		OK Cancel

Call tree view filters

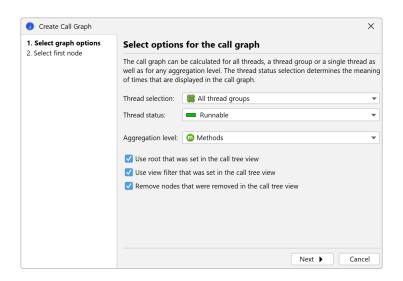
The third feature in the call that has an influence on the displayed data in the hot spots view is the view filter. When you change your call tree filters, it has a large effect on the calculated hot spots [p. 53]. To emphasize this interdependence with the call tree view, the hot spots view shows the call tree view filter in a line above the view together with a button to remove the additional filters.

hread status: 🕜	Thread selection:	Aggregation level:	Hot spot options:			
💻 Runnable 🔻	All thread groups	▼ Methods ▼	Self times			
all tree root:	com.sun.tools.javac.main.JavaCompiler.generate ←	com.sun.tools.javac.main.JavaC	ompiler.compile2 🔸	Show more		
emoved nodes:	3 removed nodes					
all tree view filters:	com.sun.tools					
	Hot Spot	Self Time 💌	Average Time	Invocations		
java.io.Outpu	itStream.close	10,909 µs (6 %)	909 µs	12		
com.sun.tool	s.javac.jvm.Gen.genMethod	7,740 µs (4 %)	151 µs	5		
🕨 🛕 java.io.FileΟι	itputStream. <init></init>	6,427 µs (4 %)	535 µs	1		
🕨 🛕 com.sun.tool	s.javac.jvm.Gen.initCode	4,767 µs (3 %)	93 µs	5		
🕨 🛕 com.sun.tool	s.javac.jvm.Code. <init></init>	3,377 µs (2 %)	66 µs	5		
A com.sun.tool	s.javac.jvm.Code.emitStackMapFrame	3,067 µs (1 %)	105 µs	2		
A com.sun.tool	s.javac.jvm.Pool.makePoolValue	2,597 µs (1 %)	1 µs	1,69		
🕨 🛕 com.sun.tool	s.javac.jvm.ClassWriter\$CWSignatureGenerato 🔳 🛾	1,907 µs (1 %)	3 µs	58		
🕨 🛕 com.sun.tool	s.javac.code.Types\$DescriptorCache.findDesc 📕 1	1,662 µs (1 %)	207 µs			
🕨 🛕 com.sun.tool	s.javac.jvm.ClassWriter.writePool 🛛 🚺 1	1,655 µs (1 %)	137 µs	1		
A com.sun.tool	s.javac.code.Kinds.kindName 🛛 🚺 1	1,636 µs (1 %)	818 µs			
🕨 🛕 com.sun.tool	s.javac.jvm.Gen.setTypeAnnotationPositions 🛛 📕 1	1,602 µs (1 %)	8 µs	19		
A com.sun.tool	s.javac.jvm.Pool.put 📃 1	1,570 µs (0 %)	0 µs	1,69		
A com.sun.tool	s.javac.code.Type.hasTag 🛛 📕 1	,351 µs (0 %)	0 µs	3,92		
🔺 🔺 com sun tool	s javac code Tunes\$26 visitClassTune 📃 1	279 Lie (0 %)	1 110	77		
Q . Hot Spot View F				•		

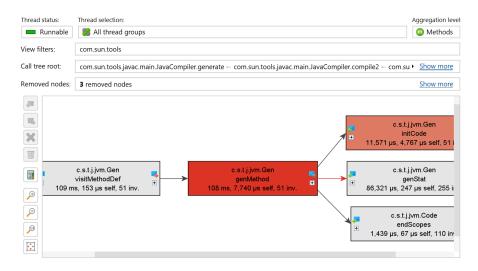
Setting a call tree root, removing parts of the call tree and view filters can be used together, with the limitation that view filters have to be set last. As soon as view filters are configured in the call tree, the *Set As Root* and *>Remove Selected Subtree* actions do not work anymore.

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.



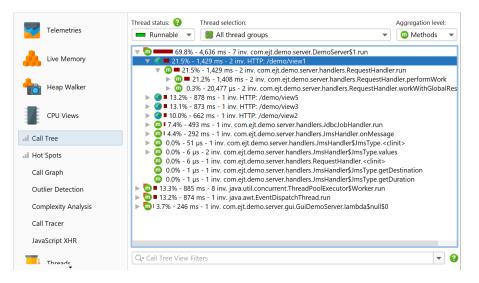
B.4 Splitting The Call Tree

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 subtrees 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. 104] 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. 168] and the injected [p. 163] custom probe systems.

Splitting methods with scripts

The same splitting functionality that is available for 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.

Thread status: 😮	Thread selection:			Aggregation level:
💻 Runnable 🔻	All thread groups			 Methods
	- 4,636 ms - 7 inv. com.ejt.demo.server.DemoServer\$1.run 1,429 ms - 2 inv. HTTP: /demo/view1 % - 1,429 ms - 2 inv. com.ejt.demo.server.handlers.RequestH 1.2% - 1,408 ms - 2 inv. com.ejt.demo.server.handlers.Request % - 20,477 us - 2 inv. com.ejt.demo.server.handlers.Request % - 20,477 us - 2 inv. com.ejt.demo.server.handlers. % - 20,477 us - 2 inv. com.ejt.demo.view3 % ms - 1 inv. HTTP: /demo/view3 % ms - 1 inv. HTTP: /demo.view2 % ms - 1 inv. HTTP: /demo.server.handlers.JdbcJobHandler.r	stHandle landler.v	er.performWork	
	2 ms - 1 inv. com.ejt.demo.server.handlers.JdbCJobHandler.h			
▼ 14.4% ▼ 14.4%	291 ms - 1 inv. com.ejt.demo.server.handlers.JmsHandler.ha % - 244 ms - 1 inv. com.ejt.demo.server.handlers.JmsHandler % - 47,602 µs - 1 inv. com.ejt.demo.server.handlers.JmsHand	ndle 🛁 .per 🍟		
▶ 🔟 0.0% - 51 ▶ 🔟 0.0% - 6	ys - 1 inv. com.ejt.demo.server.handlers.JmsHandler\$JmsTyp s - 2 inv. com.ejt.demo.server.handlers.JmsHandler\$JmsTyp s - 1 inv. com.ejt.demo.server.handlers.RequestHandler.cclir	e. <c 📙<br="">.valı</c>	Add Method Trigger Add As Exceptional Method	
	s - 1 inv. com.ejt.demo.server.handlers.JmsHandler\$JmsType		Split Method with a Script	
	s - 1 inv. com.ejt.demo.server.handlers.JmsHandler\$JmsType		Intercept Method With Script I	Probe
▶ 💿 = 13.2% - 874	ms - 8 inv. java.util.concurrent.ThreadPoolExecutor\$Worker. ms - 1 inv. java.awt.EventDispatchThread.run		 Merge splitting level 	Ctrl+Alt+M
▶ 🔟 । 3.7% - 246 m	s - 1 inv. com.ejt.demo.server.gui.GuiDemoServer.lambda\$nu	II\$0	Remove Selected Subtree	Delete
Q• Call Tree View F	ters		Restore Removed Subtrees	Ctrl+Alt+S

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.

Settings Edit Search Code Help Edit	×
· · · ·)》 🐰 🚦 🚠 🔎 💋 🔹 🐞 🔞	
Undo Redo Copy Cut Paste Show Find Replace Modify Test Classpath Compile	
Please enter an expression (no trailing semicolon) or a script (ends with a return statement) that consist	s of
Please enter an expression (no trailing semicolon) or a script (ends with a return statement) that consist regular Java code. The following parameters are available:	
- <u>com jprofiler.api.agent.ScriptContext</u> scriptContext	
 - java.lang.Class<? > c - com.ejt.demo.server.handlers.JmsHandler currentObject 	
- <u>javax.jms.Message</u> message	
The expected return type is java.lang.String	
Script:	
1message.getJMSDestination().toString()	
ок	Cancel

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 <code>ScriptContext</code> 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 <code>getObject/putObject</code> and <code>getLong/putLong</code> 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

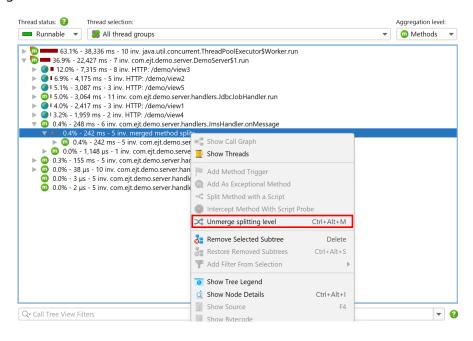
```
if (scriptContext.getObject(text) != null) {
    scriptContext.putObject(text);
    return text;
} else {
    return null;
}
```

as part of your splitting script.

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



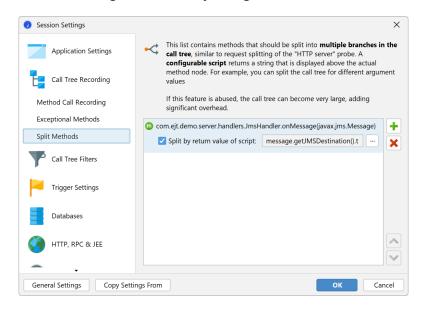
The splitting location is bound to a method, not to the selected call stack. If the same method is present somewhere else in the call tree, it will be split as well. If you use the *Merge splitting level* action, all splits will be merged into a single node. That node gives you a chance to unmerge the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

Runnable 🔻	S All thread groups	•	Methods
66.5%	- 29,354 ms - 11 inv. java.util.concurrent.ThreadPoolExecutor\$Worker.run		
🔻 🍈 💳 58.9	1% - 25,966 ms - 396 inv. com.ejt.demo.server.handlers.DemoHttpServer.lambda\$start\$0		
v 💿 💳	58.9% - 25,959 ms - 395 inv. com.ejt.mock.servlet.MockServlet.service		
🔻 🄇 💳	58.8% - 25,951 ms - 395 inv. HTTP: /exchangeRate		
▼ 🚥	58.8% - 25,949 ms - 395 inv. com.ejt.demo.server.handlers.DemoHttpServer.lambda\$createServ	let\$1	1
▼	👼 💶 58.7% - 25,881 ms - 395 inv. com.ejt.demo.server.handlers.DemoHttpServer.lookupExchange	eRat	e
	🔻 < 💳 57.8% - 25,498 ms - 390 inv. capped method splits reset splitting cap counter 😮		
	🛅 🛑 57.8% - 25,491 ms - 390 inv. com.ejt.mock.MockHelper.runnable		
	🔟 0.0% - 482 μs - 390 inv. java.util.concurrent.ThreadLocalRandom.nextInt		
	🗓 0.0% - 446 μs - 390 inv. java.util.concurrent.ThreadLocalRandom.current		
	0.0% - 445 µs - 390 inv. java.lang.String.hashCode		
	0.0% - 349 µs - 390 inv. java.lang.String.equals		
	•< 0.2% - 101 ms - 1 inv. Time stamp: 1743681483988		
	•< 0.2% - 86,667 μs - 1 inv. Time stamp: 1743681484034		
	•< 0.1% - 65,596 μs - 1 inv. Time stamp: 1743681483920		
	▶ < 0.1% - 59,453 μs - 1 inv. Time stamp: 1743681484018		
	▶ < 0.1% - 51,954 μs - 1 inv. Time stamp: 1743681483982		
	0.1% - 31,192 µs - 392 inv. com.sun.net.httpserver.HttpExchange.sendResponseHeaders		
	0.0% - 17,983 µs - 392 inv. java.io.OutputStream.close		
►	0.0% - 6,232 µs - 395 inv. com.ejt.demo.server.handlers.DemoHttpServer.toParameterMap		
	0.0% - 2,629 µs - 392 inv. java.lang.String.valueOf		
	0.0% - 2,062 µs - 392 inv. java.lang.String.getBytes		

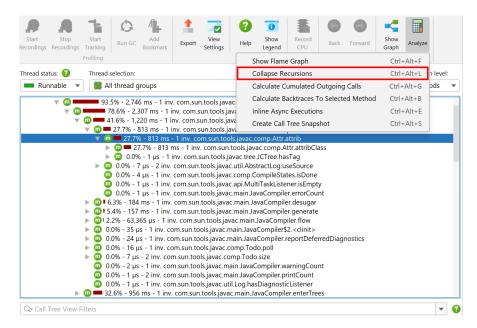
To edit split methods after you have created them, go to the session settings dialog. If you don't need a particular split method anymore, but want to keep it for future use, you can disable it with the checkbox in front of the script configuration. This is better than just merging it in the call tree,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



B.5 Call Tree Analyses

The call tree [p. 53] 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. For pinned views, a rename button is shown at the top of the view that can be used to change the name that is displayed in the view selector on the left.

6,890 recursi	ons were collapsed in the selected call tree fragment 💡	× 🖉 🔾
Thread status:	Thread selection:	Aggregation level
🔲 Runnable	All thread groups	Methods
Call tree root: c	om.sun.tools.javac.main.JavaCompiler.attribute ← com.sun.tools.javac.ma	ir • <u>Show more</u>
V 00 0 V 00 0	100.0% - 813 ms - 12 inv. com.sun.tools.javac.comp.Attr.attribClass 100.0% - 813 ms - 31 inv. com.sun.tools.javac.comp.Attr.attribC	

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.

Call tree snapshots

The "Create Call Tree Snapshot" analysis simply creates a static copy of the current call tree. This is useful for comparing different use cases without saving and opening JProfiler snapshots. Also, it offers a way to work with a frozen copy of the call tree while it is still being recorded.

The "Create Call Tree Snapshot" analysis is only available in the "Call tree" view in the "CPU Views" section. If you pin the call tree snapshot view, you can have multiple call tree snapshots

at the same time. Unlike the other analyses, call tree snapshots are saved in a JProfiler snapshot because they constitute separate sets of data.

In addition to the call tree analyses that are available for the call tree view, call tree snapshots also have a "Calculate Hot Spots" action that calculates the hot spots of the parent view, similar to the "Hot spots" view in the "CPU views" section. All analyses that are accessible from views nested below a call tree snapshot view, work with data from their parent call tree snapshot, rather than with data from the top level call tree view.

Start Stop Snapshot Settings	Start Stop Stop Stort Tracking Perofiling	Image: Second	Show Analyze
Telemetries	A snapshot of the call tree was taken ? Thread status: Thread selection:	Calculate Hot Spots Show Flame Graph Collapse Recursions Calculate Cumulated Outgoing Calls	Ctrl+Alt+H Ctrl+Alt+F Ctrl+Alt+L Ctrl+Alt+G Aggregation level:
Live Memory	Runnable Runnable All thread groups Call tree root: bezier.BezierAnim\$Demo.paint - j	Calculate Backtraces To Selected Method	Ctrl+Alt+B Methods Ctrl+Alt+E Show more
CPU Views	 □ □ □ 05.2% - 197 ms - 257 inv. bezier. Bezi □ □ 27.5% - 83,418 µs - 257 inv. java.awt.Gra □ 0.6% - 20,110 µs - 257 inv. bezier.BezierAn □ 0.6% - 1,774 µs - 257 inv. bezier.BezierAni □ 0.0% - 130 µs - 257 inv. java.awt.Graphics2 	ohics.drawlmage m\$Demo.createGraphics2D \$Demo.step	
Call Tree Snapshot			
Call Graph Outlier Detection Complexity Analysis			
Call Tracer JavaScript XHR	Q- Call Tree View Filters		
·	📫 💡 🎽 🌒 0 a	ctive recordings	VM #1 🛛 🐠 Profiling

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. 181] for the analysis. To analyze the entire call tree, select one of the top-level nodes.

Telemetries	(2) 6,890 recursions were collapsed in the selected call tree fragment (2)	
	Thread status: Thread selection: Aggregat	tion level:
Live Memory	Runnable 🔋 All thread groups	thods
	Call tree root: com.sun.tools.javac.main.JavaCompiler.attribute com.sun.tools.javac.mair • Show	more
Heap Walker	V 0 0 100.0% - 813 ms - 12 inv. com.sun.tools.javac.comp.Attr.attribClass	
CPU Views	 ■ 100.0% - 813 ms - 31 inv. com.sun.tools.javac.comp.Attr.attribClass ■ 0 0 ■ 97.% - 811 ms - 12 inv. com.sun.tools.javac.comp.Attr.attribClassBody ■ 0 0.2% - 1,613 µs - 12 inv. com.sun.tools.javac.comp.Attr.isSerializable ■ 0 0.0% - 175 µs - 12 inv. com.sun.tools.javac.code.Lint.augment 	1
▲ Call Tree	 0.0% - 175 µs - 24 inv. com.sun.tools.javac.util AbstractLog.useSource 0.0% - 162 µs - 12 inv. com.sun.tools.javac.comp.Check.checkClassOverrideEquation 	alsAnd
Collapsed Recursions	🕨 ወ 🧿 0.0% - 125 μs - 31 inv. com.sun.tools.javac.code.Types.supertype	
Hot Spots	 	
Call Graph	 ⁽¹⁾ 0.0% - 35 μs - 29 mV. com.sun.tools.javac.comp.check.check.vonCyclic ⁽¹⁾ 0 0.0% - 26 μs - 12 inv. com.sun.tools.javac.code.DeferredLintHandler.flush 	
Outlier Detection	 0 0 0.0% - 24 µs - 12 inv. com.sun.tools.javac.comp.Check.checkDeprecatedAnnotat 0 0 0.0% - 12 µs - 11 inv. com.sun.tools.javac.comp.TypeEnvs.get 	tion
Complexity Analysis	 0.0% - 11 μs - 22 inv. com.sun.tools.javac.tree.JCTree.pos 0.0% - 9 μs - 20 inv. com.sun.tools.javac.comp.Check.setLint 	
Call Tracer	o moved 166 µs - 19 inv. com.sun.tools.javac.comp.Attr.attribClass	_
JavaScript XHR	Q+ Call Tree View Filters	• 😯

A recursion is detected when the same method was already called higher up in the call stack. In that case, the subtree 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.

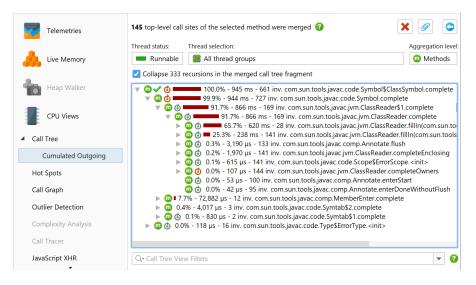
v 🔟 🚯 💶 100.0% - 813 ms - 12 inv. com.sun.tools.javac.comp.Attr.attribClass
🔻 🧰 👌 💶 100.0% - 813 ms - 31 inv. com.sun.tools.javac.comp.Attr.attribClass
m is 99.7% - 811 ms - 12 inv. com.sun.tools.iavac.comp.Attr.attribClassBody
16 recursive calls have been stitched back to this node. Attr.isSerializable
🕨 🗿 🗿 0.0% - 175 μs - 12 inv. com.sun.tools.javac.code.Lint.augment
▶ 💷 🙆 0.0% - 175 µs - 24 inv. com.sun.tools.javac.util.AbstractLog.useSource
🕨 💿 🍈 0.0% - 162 μs - 12 inv. com.sun.tools.javac.comp.Check.checkClassOverrideEqualsAnc
0.0% - 125 µs - 31 inv. com.sun.tools.javac.code.Types.supertype
🕨 💿 🎯 0.0% - 72 μs - 12 inv. com.sun.tools.javac.comp.Check.checkFunctionalInterface

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 subtree 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.

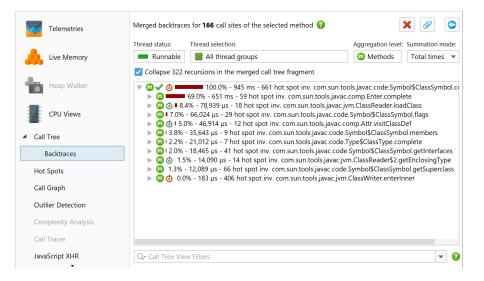


For the selected method, JProfiler collects all its top-level invocations without considering recursive calls and cumulates them in the result tree. The header shows how many such top-level call sites were summed in that process.

At the top of the view, there is a checkbox 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.



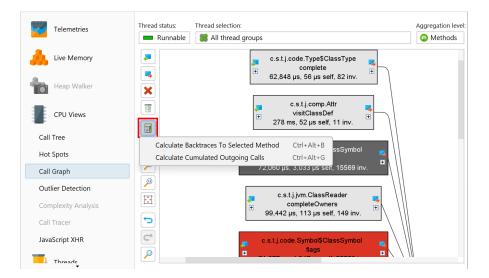
This analysis is similar to the hot spots view, only that by default it sums total times instead of self-times for the selected method, and the hot spots view only shows methods whose self-time is a significant fraction of the total time. At the top of the view there is a radio button group labeled *Summation 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. 141], 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. 71], but restricted for a particular allocation spot.

Telemetries	selected call stack		🗘 Reload analy	SIS	
	Recorded allocations:	Live objects at , 1/10 alloca	ations, All classes		
Live Memory	Aggregation level:	Methods			
All Objects	Allocation spot:	java.awt.Graphics2D.fill ← b	ezier.BezierAnim\$Demo.drawl	Demo ← I	b • Show more
Recorded Objects		Name	Instance Count	T	Size
	java.util.HashMap\$No	de	172 (2	0 %)	5,504 bytes
Allocation Call Tree	double[]		118 (13 %)		6,024 bytes
AU	sun.java2d.loops.Rend	lerCache\$Key	78 (9 %)		1,872 bytes
Allocation Classes	java.awt.geom.Path2D	\$Float\$CopyIterator	56 (6 %)		1,792 bytes
Allocation Hot Spots	java.awt.geom.Point2I	D\$Double	56 (6 %)		1,792 bytes
Allocation not spots	java.awt.geom.AffineT	ransform	48 (5 %)		3,456 bytes
Class Tracker	java.awt.GradientPain	tContext	35 (4 %)		2,240 bytes
	java.awt.RenderingHir	nts	35 (4 %)		560 bytes
Heap Walker	java.awt.geom.Point2I	D\$Float	35 (4 %)		840 bytes
	sun.java2d.pipe.Alpha	PaintPipe\$TileContext	34 (3 %)		1,632 bytes
	java.awt.geom.Rectan	gle2D\$Double	32 (3 %)		1,536 bytes
CPU Views	java.lang.ref.WeakRef	erence	32 (3 %)		1,024 bytes
CFO VIEWS	java.util.HashMap\$No	de[]	32 (3 %)		1,536 bytes
	iava lang Integer		26 (3 %)		416 hvte
Threads	Total from 17 rows:		860 (100 %)		55,864 bytes
	Q+ Class View Filters				•

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.

Telemetries	Thread status: 😮 💷 All states					<u>Change</u>
_	Method	Total Time	Inv.	Avg. Time	Max. Time	Outlier Coeff. 🕜 🛛 🍸
Live Memory	bezier.BezierAnim\$Demo.block(boolean)	801 ms	427	1,875 µs	200 ms	106.05
Live Memory	bezier.BezierAnim\$Demo.step(int, int)	803 ms	423	1,900 µs	200 ms	104.66
	bezier.BezierAnim\$Demo.paint(java.awt.Grap	1,367 ms	423	3,233 µs	204 ms	62.1
Heap Walker	java.lang.Thread.sleep(long)	4,522 ms	423	10,692 µs	12,096 µs	0.131
CPU Views						
Call Tree						
Hot Spots						
Call Graph						
Outlier Detection						
Complexity Analysis						
Call Tracer						
JavaScript XHR						
Threads	Q- Class View Filters					•

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.

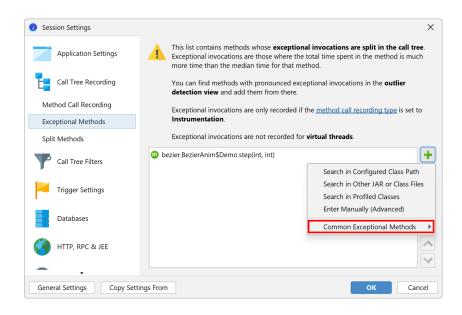
Met	hod	Total Time	Inv.	Avg. Time	Max. Time	Outlier Coeff. 🕜 🛛 🔻
bezier.BezierAnim\$Demo.block(boolean)		801 ms	427	1,875 µs	200 ms	106.0
pezier.BezierAnim\$Dem		°03 ms	423	1,900 µs	200 ms	104.6
bezier.BezierAnim\$Den	Add As Exceptional N	1ethod 57 ms	423	3,233 µs	204 ms	62.1
	Show SourceShow Bytecode	2 ms F4	423	10,692 µs	12,096 µs	0.13
	Sort Outlier Statistics	► Ctrl+F				
	1 Export View	Ctrl+R				
	📷 View Settings	Ctrl+T				

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.

Ø Session Settings	×
Application Settings	✓ Enable CPU profiling
Application settings	Auto-Tuning For Instrumentation
Call Tree Recording	Enable auto-tuning 😮
	A method is an overhead hot spot and will be suggested for inclusion into the list of ignored methods, if both of the following conditions are true:
Call Tree Filters	ignored methods, in boar of the following conditions are due.
•	1. The total time of the method is more than 10 🔷 per mille of the entire total time
Trigger Settings	2. The average time of the method is less than $100 \Leftrightarrow \mu s$
<u> </u>	Auto-tuning is only performed if the method call recording type is set to "Instrumentation" on
Databases	the <u>method call recording tab</u> .
	Call Tree Recording Options
HTTP, RPC & JEE	CPU times for instrumentation: 💿 Elapsed times 😮 🔵 Estimated CPU times 😮
	Instrument native methods
JVM & Custom Probes	Thread resolution for async sampling ?
Advanced Settings	Exceptional Method Run Recording
	Maximum number of separately recorded method runs: 5 💠 😯
CPU Profiling	Time type for determining exceptional method runs:
Probes & JEE	
Memory Profiling	Call Tree Splitting
Thread Profiling	Maximum number of splits: 128 🗢 😯
Miscellaneous	Async And Remote Request Tracking
	☑ Enable asvnc and remote request tracking
General Settings Copy	Settings From OK Cancel

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.

Telemetries	Thread status: ⁽²⁾	
Live Memory	 0 === 50.0% - 8,280 ms - 1 inv. bezier.BezierAnim\$Demo.run 0 === 50.0% - 8,280 ms - 1 inv. java.awt.EventDispatchThread.run 0 = 11.7% - 1,937 ms - 653 inv. bezier.BezierAnim\$Demo.paint 	l
Heap Walker	▶ @12.7% - 453 ms - 653 inv. bezier.BezierAnim5Demo.drawDemo	
CPU Views	♥ 00, 1.2% - 200 ms - 1 inv. bezier.BezierAnim5Demostep [exceptional run] ▶ 00, 1.2% - 200 ms - 1 inv. bezier.BezierAnim5Demostlock ▶ 00, 0.0% - 1 µs - 12 inv. bezier.BezierAnim5Demostlock	
.1 Call Tree	1.2% - 200 ms - 1 inv. bezier.BezierAnim\$Demo.step [exceptional run] 1.2% - 200 ms - 1 inv. bezier.BezierAnim\$Demo.step [exceptional run]	
Hot Spots	 1.2% - 200 ms - 1 inv. bezier.BezierAnim\$Demo.step [exceptional run] 1.2% - 200 ms - 1 inv. bezier.BezierAnim\$Demo.step [exceptional run] 0.3% - 49,828 µs - 653 inv. bezier.BezierAnim\$Demo.createGraphics2D 	
Outlier Detection	 0.0% - 427 μs - 653 inv. java.awt.Graphics2D.dispose 0.0% - 3,975 μs - 653 inv. bezier.BezierAnim\$Demo.onEDTMethod 	
Complexity Analysis		
Call Tracer		
JavaScript XHR		
Threads	Q• Call Tree View Filters	▼ ?

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

Q [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.

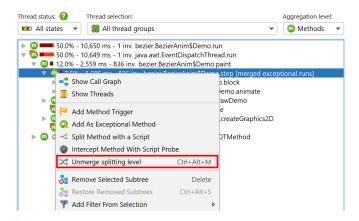
@ [merged exceptional runs]

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.

Qurrent 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. 104] and split methods [p. 186], 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.

Onfigure Complexity Recordings	×
Configured methods:	
sort.Comparison.executeBubbleSort(int[], int)	+
Script returning the complexity as an integer: i	
sort.Comparison.executeSelectionSort(int[], int)	
sort.Comparison.executeInsertionSort(int[], int)	
sort.Comparison.executeQuickSort(int[], int)	
О К Са	ancel

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().

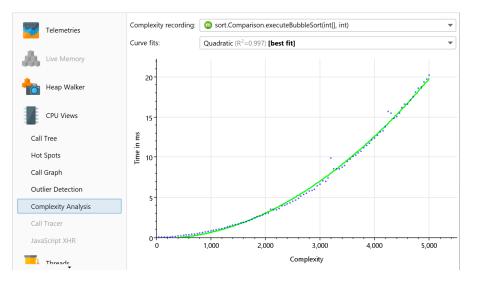
Settings Edit Search Code Help Edit	×
Image: Comparison of the comparison	
Please enter an expression (no trailing semicolon) or a script (ends with a return statement) that consists of regular Java code. The following parameters are available: - comjprofiler.api.agent.ScriptContext - java.lang.Class c - sort.Comparison currentObject - int[] int i The expected return type is long	
Script:	
lintArray.length	cel

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. 27]. 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.

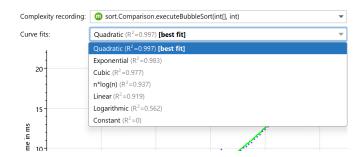
⁽¹⁾ https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

⁽²⁾ https://bigocheatsheet.com/

The graph is a bubble chart, where the size of each data point is proportional to the number 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 R² 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 R², 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. 254] 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. 27] or with the profiling API [p. 129]. 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.

Call Tracer View Settings	×
Trace Recording	
Maximum number of recorded call traces: 100,000	\$
Record calls into unprofiled classes (
Time Display	
 Relative to first trace 	
 Relative to previous node 	
Relative to previous node of the same type	
Method Display	
🗹 Show signature	
Show class names in method nodes	
OK Car	cel

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 \mathbb{P} threads, \mathbb{Q} packages and \mathbb{C} classes. Each time the current value for any of these groups changes, a new grouping node is created.

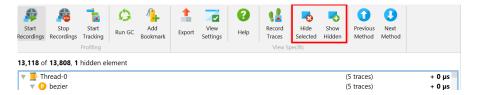
At the lowest level there are ⁽ⁱ⁾ method entry and ⁽ⁱ⁾ method exit nodes. Below the table with the call traces, the stack trace of the currently selected method trace is shown. If call traces into other methods have been recorded from the current method or if another thread interrupts the current method, 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.

Start Stop Save Se	Run GC	Add okmark Export Settings Q Constraints	Hide Sho Selected Hidd
Telemetries	13,808 traces, 0 hidden element ▼ ↓ Thread-0 ▼ ○ bezier	(7 traces) (7 traces)	+ 0 µs + 0 µs
Live Memory	♥ ③ bezier.BezierAnim\$Demo	(7 traces)	+ 0 µs + 0 µs + 0 µs
Heap Walker	scheduleBlockingActivity() scheduleRepaint() scheduleRepaint() scheduleRepaint()	Hide Selected Delete	+ 71 μs + 73 μs + 91 μs
CPU Views	block(boolean) block(boolean) block(boolean) AWT-EventQueue-0	Show Hidden Ctrl+Alt+S Show Source F4	+ 92 μs + 92 μs + 242 μs
Il Call Tree Il Hot Spots	AWT-EventQueue-0		11 ms 922 µs + 12 ms 48 µs 22 ms 306 µs
Call Graph	AWT-EventQueue-0	Skip To Next Method Trace Alt+Down	22 ms 407 µs
Outlier Detection Complexity Analysis		Find Ctrl+F Export View Ctrl+R	-
.il Call Tracer JavaScript XHR		View Settings Ctrl+T	J
Threade			
-	💡 📁 🛛 🔵 1 active recording	VM #1	🐠 Profiling

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.

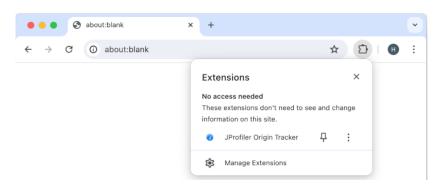


C.4 JavaScript XHR Origin Tracking

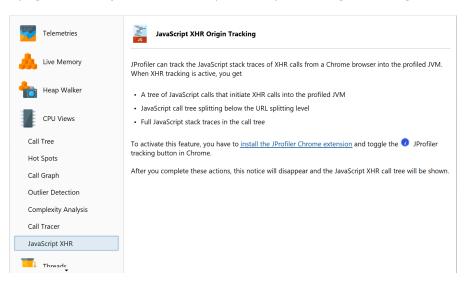
With JavaScript XHR origin tracking, you can split servlet invocations for different stack traces in the browser during XMLHttpRequest⁽¹⁾ or Fetch⁽²⁾ requests, so you can better correlate the activity in the profiled JVM with actions in the browser. in the following, "XHR" designates both the XMLHttpRequest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



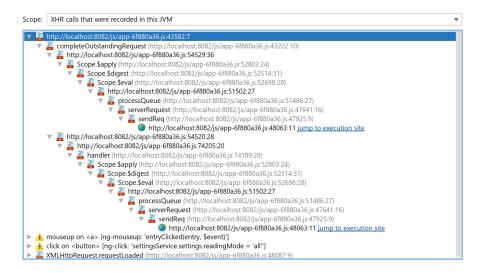
When tracking is activated, the JProfiler extension will ask you to reload the page. This is necessary for adding instrumentation. If you choose to not reload the page, event detection may not work.

- ⁽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

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.

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 $\stackrel{2}{=}$ call stack nodes include information on the source file and the line number. The function where the XHR call is made has a \bigcirc special icon 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. 53] 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 \clubsuit 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.

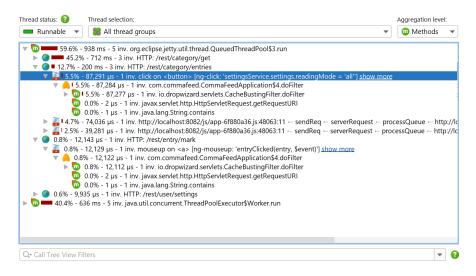
In some cases, the automatic detection of interesting attributes may not be suitable and you may prefer a different call tree splitting. For example, some frameworks assign automatic IDs, but it would be more readable to group all elements together with a semantic description of the action. To achieve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

http:	//localhost:8082	2/js/app-6f88	30a36.js:48063:11	
sendR	eq (http://local	lhost:8082/js	s/app-6f880a36.js:47925:9)	
serve	rRequest (http:/	//localhost:8	8082/js/app-6f880a36.js:47641:16)	
proce	ssQueue (http://	/localhost:80	082/js/app-6f880a36.js:51486:27)	
http:	//localhost:808	2/js/app-6f88	30a36.js:51502:27	
Scope	.\$eval (http://	localhost:808	32/js/app-6f880a36.js:52698:28)	
Scope	.\$digest (http:/	//localhost:8	8082/js/app-6f880a36.js:52514:31)	
Scope	.\$apply (http://	/localhost:80	082/js/app-6f880a36.js:52803:24)	
	when Theman to day	onumours (ht	tp://localhost:8082/js/app-6f880a3	6 30.672
HIMLB	outconfiguent. <a< th=""><th>nonymous/ (ne</th><th>.cp.//iocainosc.cocz/js/app-ciccocas</th><th>0. js. 0/2</th></a<>	nonymous/ (ne	.cp.//iocainosc.cocz/js/app-ciccocas	0. js. 0/2
			ispatch (http://localhost:8082/js/a	2
HTMLB	SuttonElement.jQ	uery.event.di		2
HTMLB elemD	SuttonElement.jQu Data.handle (http	uery.event.di p://localhost	ispatch (http://localhost:8082/js/a	
HTMLB elemD HTMLB	ButtonElement.jQu Data.handle (http ButtonElement. <a)< th=""><th>uery.event.di p://localhost nonymous> (<a< th=""><th>ispatch (http://localhost:8082/js/a :8082/js/app-6f880a36.js:15674:28)</th><th>pp-6f880</th></a<></th></a)<>	uery.event.di p://localhost nonymous> (<a< th=""><th>ispatch (http://localhost:8082/js/a :8082/js/app-6f880a36.js:15674:28)</th><th>pp-6f880</th></a<>	ispatch (http://localhost:8082/js/a :8082/js/app-6f880a36.js:15674:28)	pp-6f880
HTMLB elemD HTMLB	ButtonElement.jQu Data.handle (http ButtonElement. <a)< th=""><th>uery.event.di p://localhost nonymous> (<a< th=""><th><pre>ispatch (http://localhost:8082/js/a :8082/js/app-6f880a36.js:15674:28) anonymous>:4:186)</pre></th><th>pp-6f880</th></a<></th></a)<>	uery.event.di p://localhost nonymous> (<a< th=""><th><pre>ispatch (http://localhost:8082/js/a :8082/js/app-6f880a36.js:15674:28) anonymous>:4:186)</pre></th><th>pp-6f880</th></a<>	<pre>ispatch (http://localhost:8082/js/a :8082/js/app-6f880a36.js:15674:28) anonymous>:4:186)</pre>	pp-6f880
HTMLB elemD HTMLB	ButtonElement.jQ Data.handle (http ButtonElement. <au : on <button> [ng</button></au 	uery.event.di p://localhost nonymous> (<a g-click: 'set</a 	<pre>ispatch (http://localhost:8082/js/a ::8082/js/app-6f880a36.js:15674:28) anonymous>:4:186) :tingsService.settings.readingMode</pre>	pp-6f880
HTMLB elemD HTMLB click	ButtonElement.jQ Data.handle (http buttonElement. <au : on <button> [ne This Invocation</button></au 	uery.event.di p://localhost nonymous> (<a g-click: 'set Subtree ?</a 	<pre>Lispatch (http://localhost:8082/js/a ::8082/js/app-6f880a36.js:15674:28) anonymous>:4:186) ttingsService.settings.readingMode All Invocations ?</pre>	pp-6f880

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. 129], 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.

🕖 Trigger Wizard - Out of mem	ory exception	\times
1. Trigger type 2. Actions	Configure actions for this trigger	
3. Description	Configured actions:	
4. Group ID 5. Finished	Create an HPROF/PHD heap dump	+ ×
	▲ Back Next ► Finish Car	ncel

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>

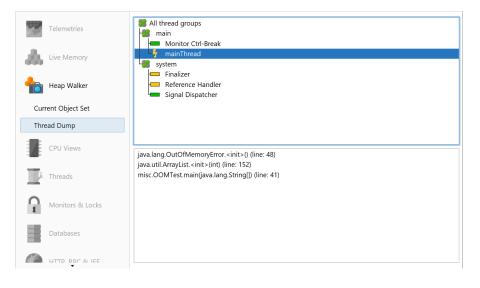
⁽¹⁾ http://docs.oracle.com/javase/9/troubleshoot/command-line-options1.htm#JSTGD592

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.

🧿 JProfiler Star	rt Center	×
Start Ce	nter	
	• On this computer On another computer On a Kubernetes cluster	
Open Session	Container: 📗 None, showing top level processes	Select Container
Session	Status: All detected HotSpot/OpenJ9 JVMs 🔻	💛 Show Services
Quick Attach New Session Open Snapshots	PID Process Name 10620 org.gradle.launcher.daemon.bootstrap.GradleDaemon 8.13 11196 org.gradle.wrapper.GradleWrapperMain screenshotsLightEn 14724 C:\Users\ingo\AppData\Local\UeBrains\Toolbox\bin\jre 17936 bezier.BezierAnim block 21516 org.jetbrains.kotlin.daemon.KotlinCompileDaemondaemon-runFilesPath C:\Users\ingo	NAppData\Local\k
	Legend: Profiling agent loaded JProfiler GUI connected Offline mode JFR runni	ng
	Start Heap Dump Only ? Start JFR	Close

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.



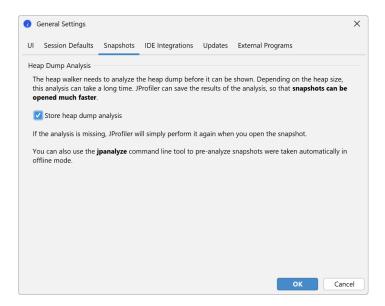
D.2 Minimizing Overhead In The Heap Walker

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 <code>.analysis</code> next to the snapshot file. When you open the snapshot file, the heap walker will be available very quickly. If you delete the <code>.analysis</code> 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. 129] 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. 254] 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.

V Heap Snapshot Options	×
Select recorded objects	
Initially, the heap walker will show only those objects that have been recorded in the dynamic memory view section.	
🗹 Perform full GC in heap snapshot (
Retain objects held by references of type soft	
Hide Overhead Options •	
✓ Calculate retained sizes	
Record primitive data	
OK Cancel	

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.

Classes	2 Allocations	Biggest Objects	🔀 References	🚫 Time	🌼 Inspections
Current object set:		es of java.util.Hasl 42 kB shallow size, <u>Calc</u>	•	o sizes <u>Use retair</u>	ned objects
Outgoing references	▼ Use	. 🔻 Apply filter	▼ Show In C	Graph	.
	Object	By restrictin	ig the selected value	/ Size	Allocation Time (h:m:s)
🔻 😑 java.util.HashMap	\$Node (0x8825)	By Running	a Script On Live Object	ts 32 by	tes n/a
hash = 202854941 ▼ key 💙 java.lang.St		By Running	a Script On Dumped D	ata	
coder (declared	d by java.lang.Abstra	ctStringBuilder) = 0			
mavbeLatin1	declared by iava land	a AbstractStringBuilder)	= false		

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.

c Classes	P Allocations	Biggest Objects	🔀 References	🚫 Time	🄯 Inspections	,
Current object se	-	es of java.util.Hash	•	deep sizes Use retained	objects	
Outgoing references	▼ Use		► Show	In Graph 🙀 👹		
	Object	Retair	ned Size 🔍	Shallow Size	Allocation Time (h:m:s)	
🔻 🛑 java.util.HashM	ap\$Node (0x8825)		1,508 kB	32 bytes	n/	/a
coder (decl maybeLatin count (decl ▶ value (decla	9414 g.StringBuffer (0x8826) ared by java.lang.Abstr. n1 (declared by java.lar lared by java.lang.Abstr red by java.lang.Abstra il.HashMap\$Node (0x8	ng.Abst Select all of Select a	object has an outg	urrent object set for whi oing reference chain jus value satisfies the followi	like the selected one	
4,465 instances of jav	a.util.HashMap\$Node ra.util.HashMap\$Node objects, after full GC, re		quals 🔻 123	45 🔻	OK Cancel	

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.

Classes	Parallocations	2 2 2	Biggest Objects	🔀 References	🕑 Time	🌼 Inspections
Current object set:	4,465 instanc 2 selection steps, 1		Filter Value Select all o	bbjects from the curre	nt object set for which	imes the following is true:
Outgoing references	▼ Use	-			g reference chain just lik	
	Object		🍼 The	selected reference sat	isfies the following con	dition:
coder (declare maybeLatin1 count (declare value (declared		g.Abst actStri ctStrin	 The reference The reference The instance java.lang.Str Also mate 	is null is of the type:		
Selection step 2: Class java.ut 4,465 instances of java.u Selection step 1: All obj		taining	String value	the toString() method s	atisfies the condition:	Ignore case OK Cancel

The most powerful filter types are those using code snippets. There are two fundamentally different types of ways to filter objects:

In live sessions, JProfiler can run the filtering script in the profiled JVM and pass the actual instances to your script. In the script editor that is shown by the *Apply filter by running a script on live objects*, you can write an expression or a script that accesses properties directly and whose boolean return value decides whether an instance should be retained in the current object set or not.

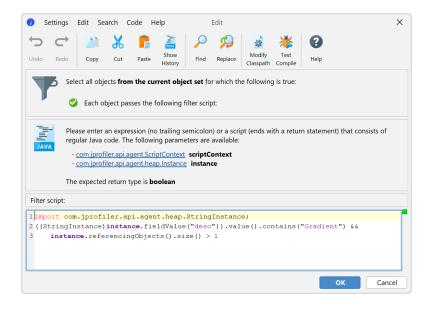
🥥 Settir	ngs Edit Se	arch Code	Help	Edit				×	
Undo R	edo Copy	Cut Paste	Show	d Replace	Modify Classpath	Test Compile	? Help		
T	 Select all objects from the current object set for which the following is true: The object has an outgoing reference chain just like the selected one The selected reference passes the following filter script: 								
JAVA	Please enter an expression (no trailing semicolon) or a script (ends with a return statement) that consists of regular Java code. The following parameters are available: - <u>com.jprofiler.api.agent.ScriptContext</u> scriptContext - java.lang.Object object								
		d return type is	boolean						
Filter scrip		+7] = h = Me = h	() c 255 21						
1 pixelo	onverter.ge	-CAIDUAMASK	() & 255 == 2						
							ок	Cancel	

Naturally, this feature can only work for live sessions, because JProfiler needs access to the live objects. Another factor to consider is that an object may have been garbage collected after 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.

The second option that also works for snapshots, including HPROF and PDH snapshots, is the *Apply filter by running a script on dumped data* action. Each instance will be passed to your script as an instance of com.jprofiler.api.agent.heap.HeapObject. There are several sub-interfaces that you can down-cast the parameter to, if applicable. Consult the Javadoc for more information. For example, the com.jprofiler.api.agent.heap.Instance interface

is available if the object is an object instance and offers access to field values. If the script operates on top-level objects and all objects in the current object set are of the same type, the script parameter will automatically have a suitable subtype.

In these filter scripts, you also have access to all incoming and outgoing references through the methods of the HeapObject parameter.

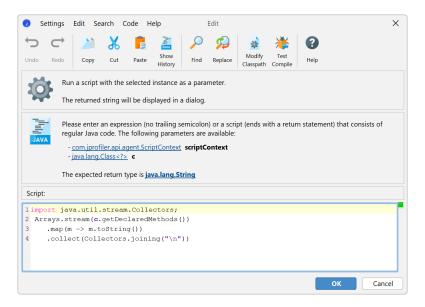


Note that PHD snapshots do not contain field information, so all instances are passed as com. jprofiler.api.agent.heap.HeapObject Or com.jprofiler.api.agent.heap. ClassObject for such snapshots and field values can only be accessed through the referencedObjects() method.

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.

 Classes 	Parallocations	Biggest Obje	ects	🔀 Reference	s 🚫 Time	🔅 Inspections
Current object set: Outgoing references		36 bytes shallow siz	ze, <u>C</u>	alculate retained a	nd deep sizes Use reta	ined objects
	Object		Retair	ned Size 🔍	Shallow Size	Allocation Time (h:m:s)
Selection step 2: Class	e ava.lang.String (0x890	x8ce4) ["[sun.font.Fo			mily (0x8ce3) ["Font fam	X nily: Monospaced ic=null initialized=false"]
7 instances of sun.font.f	FontFamily	taining soft		This dialog is nor	ı-modal	

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

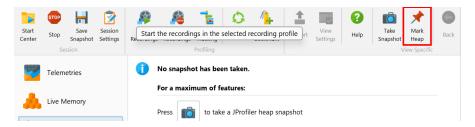
Distinguishing regular memory usage from a memory leak is often not quite simple. However, both excessive memory usage and memory leaks have the same symptoms and so they can be analyzed in the same way. The analysis proceeds in two steps: Locating suspicious objects and finding out why those objects 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. 71] 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 call histogram 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. 80] 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.

 Classes 	P Allocations	Biggest Objects	🔀 References	🚫 Time	🄯 Insp	pections
Current object se	et: 107,534 obje	cts in 1,436 classes.				
	1 selection step, 8,	788 kB shallow size				
	39 095 new instan	ces (36.4%) since the last h	ean dumn Use new U	se old		
	55,055 Hew Instant	co (oo.+/o) since the last i				
Classes 🔹		oup By Class Loaders			ulate estimate	d retained siz
Classes					ulate estimate	ed retained siz
	Use 🔻 🛛 🧕 Gro			Calc	-	
byte[]	Use 🔻 🛛 🧕 Gro			Calc	-	Size
byte[] java.lang.String	Use ▼ Gro Name		Inst	ance Count 22,293 (20	-	Size 1,025 kB
Classes byte[] java.lang.String java.util.HashMap\$Noo java.lang.Long	Use ▼ Gro Name		Inst	Calc ance Count 22,293 (20 15,634 (14 %) ,060 (10 %)	-	Size 1,025 kE 375 kE

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 be practicable. With the built-in facility in the JVM to automatically save an HPROF snapshot [p. 206] 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.

Classes	P Allocations	Biggest Objects	🔀 References	🚫 Time	🌼 Inspections
Current object se		ts in 1,390 classes. ,315 kB shallow size			
No grouping	▼ F T		Use 🔻 🔣	v In Graph 🔯	1
	Object			Retained Size	•
sun.awt.AppCo	ntext (0x6988)				1,568 kB (24 %)
I	esignMetrics (0x5d1)			?7 kB (8 %)	
🔻 🤎 bezier.BezierAn			475	kB (7 %)	
	B (99.8%) leakMap 🌍				
		java.util.HashMap\$Node		de ale construction d'ale con	(
		th a total retained size of tained size of 448 bytes a			
	im\$Demo (0x7417)	tailled size of 440 bytes a	368 kB		les
I	File\$Source (0x3d10)		110 kB (1 %)	(0.70)	
i com.jprofiler.ac			64,864 bytes (1 %	6)	
	.MethodType (0x9f5)		64,120 bytes (1 %	6)	
sun.security.uti	KnownOIDs (0x8ef)		53,392 bytes (0 %)	
sun.security.pro	vider.Sun (0x2f17)		53,264 bytes (0 %)	
sun.java2d.loop	s.GraphicsPrimitiveMg	r (0xad1)	38,104 bytes (0 %)	
java.util.concur	rent.ConcurrentHashN	lap (0x52a7)	35,088 bytes (0 %)	

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 be 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.

Current object set: 1,534 instances of java.awt.geom.GeneralPath 2 selection steps, 49 kB shallow size, Calculate retained and deep sizes Use retained objects Incoming references Use < Show In Graph Image: Show Paths To GC Root Object Object Retained Size Show Paths To GC Root Object Retained Size Shallow Size Allocation Time (h:m:s) ava.awt.geom.GeneralPath (0x747) V Go table of java.util.HashMapSNodel (0x747c) V Go table of java.util.HashMap (0x747c) V Go table of		Classes	P Allocations	Biggest Objects	🔀 References	🚫 Time	🕸 Inspections 🔹
Incoming references Use • Show In Graph Image: Comparison of the state of the sta	Cur	rent object set:	-	, ,			
Object Retained Size Shallow Size Allocation Time (h:m:s) Image: Size of			2 selection steps, 4	19 kB shallow size, Cale	culate retained and dee	ep sizes Use retained of	objects
▼ ● java.awt.geom.GeneralPath (0x747f) 248 bytes 32 bytes 0:0:13.0 ▼ ● value of java.util.HashMap\$Node (0x747d) ▼ ● element of java.util.HashMap\$Node[(0x763) ▼ ● element of java.util.HashMap\$Node] (0x763) ▼ ● table of java.util.HashMap\$Node] (0x747c) ▼ ● table of java.util.HashMap\$Node] (0x763) ▼ ● table of java.util.HashMap\$Node] (0x747c) ▼ ● table of java.util.HashMap\$Node] (0x747c) ▼ ● table of java.util.HashMap\$Node] ▼ ● table of java.util.HashMap\$Node] ▼ ● table of java.util.HashMap\$Node] ○x071.0 ▼ ● table of java.util.HashMap\$Node] ▼ ● table of java.util.HashMap\$Node] ○x071.0 ▼ ● table of java.util.HashMap\$Node] ○x071.0 ● ↓ Target of sun.avt.windows.WPanelPeer (declared by sun.avt.windows.WObjectPeer) (0x6c12) ● ↓ ● ↓ java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:06.9 ● ↓ java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 ● ↓ java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 ● ↓ java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 ● ↓ java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 ● ↓ java.avt.geom.GeneralPath (0x7491) 248 bytes <	Inc	oming references	▼ Use	🔻 🛛 <	Graph	💿 🐧 Show Par	ths To GC Root
▼ Solution value of java.util.HashMap\$Node (0x747d) ▼ Solution element of java.util.HashMap\$Node[] (0xd763) ▼ Solution table of java.util.HashMap\$Node[] (0xd763) ▼ Solution table of java.util.HashMap (0xd747c) ▼ Solution table of java.util.HashMap (0xd747c) ▼ Solution Solution ● JeakMap of bezier.BezierAnim (0x6c11) ▼ Solution Solution ● Java.avt.geom.GeneralPath (0x7485) 248 bytes 32 bytes ● Java.avt.geom.GeneralPath (0x7482) 248 bytes 32 bytes 0:0:04.8 ● Java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 ● Java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 ● Java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:06.9 ● Java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 ● Java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 ● Java.avt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 ● Java.avt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:06.9 ● Java.avt.geom.GeneralPath (0x74			Object	Reta	ained Size 🔍	Shallow Size	Allocation Time (h:m:s)
▼	▼	👂 java.awt.geom.Ge	neralPath (0x747f)		248 bytes	32 bytes	0:0:13.0
 table of java.util.HashMap (0x747) tagtet of java.util.HashMap (0x6c11) target of sun.awt.windows.WPanelPeer (declared by sun.awt.windows.WObjectPeer) (0x6c12) JNI global reference java.awt.geom.GeneralPath (0x7485) 248 bytes 32 bytes 0:0:06.9 java.awt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:08.9 		🔻 🔇 value of java.u	util.HashMap\$Node	(0x747d)			
 leakMap of bezier.BezierAnim (0x6c11) target of sun.awt.windows.WPanelPeer (declared by sun.awt.windows.WObjectPeer) (0x6c12) JNI global reference java.awt.geom.GeneralPath (0x7485) 248 bytes 32 bytes 0:0:06.9 java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:08.9 		🔻 🔇 element of	java.util.HashMap\$1	Node[] (0xd763)			
 G target of sun.awt.windows.WPanelPeer (declared by sun.awt.windows.WObjectPeer) (0x6c12) I'NI global reference java.awt.geom.GeneralPath (0x7485) 248 bytes 32 bytes 0:0:06.9 java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748c) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:08.9 		🔻 🔇 table o	f java.util.HashMap (0x747c)			
INI global reference ● java.awt.geom.GeneralPath (0x7485) 248 bytes 32 bytes 0:0:11.0 ● java.awt.geom.GeneralPath (0x7482) 248 bytes 32 bytes 0:0:06.9 ● java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 ● java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 ● java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes n/a ● java.awt.geom.GeneralPath (0x748c) 248 bytes 32 bytes 0:0:08.9 ● java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 ● java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:08.9		🔻 🔇 leak	Map of bezier.Bezie	rAnim (0x6c11)			
• java.awt.geom.GeneralPath (0x7485) 248 bytes 32 bytes 0:0:11.0 • java.awt.geom.GeneralPath (0x7482) 248 bytes 32 bytes 0:0:06.9 • java.awt.geom.GeneralPath (0x7482) 248 bytes 32 bytes 0:0:06.9 • java.awt.geom.GeneralPath (0x7480) 248 bytes 32 bytes 0:0:04.8 • java.awt.geom.GeneralPath (0x7488) 248 bytes 32 bytes n/a • java.awt.geom.GeneralPath (0x7480) 248 bytes 32 bytes 0:0:08.9 • java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 • java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 • java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:06.9		-			ared by sun.awt.windov	ws.WObjectPeer) (0x6c	12)
java.awt.geom.GeneralPath (0x7482) 248 bytes 32 bytes 0:0:06.9 java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748c) 248 bytes 32 bytes n/a java.awt.geom.GeneralPath (0x748c) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:08.9			🕽 🖞 JNI global refe	rence			
java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes 0:0:04.8 java.awt.geom.GeneralPath (0x748b) 248 bytes 32 bytes n/a java.awt.geom.GeneralPath (0x748c) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:06.9	▶	🛑 java.awt.geom.Ge	neralPath (0x7485)		248 bytes	32 bytes	0:0:11.0
java.awt.geom.GeneralPath (0x7488) 248 bytes 32 bytes n/a java.awt.geom.GeneralPath (0x748e) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:06.9	▶	🟓 java.awt.geom.Ge	neralPath (0x7482)		248 bytes	32 bytes	0:0:06.9
Image: system Image: s	▶	🛑 java.awt.geom.Ge	neralPath (0x748b)		248 bytes	32 bytes	0:0:04.8
java.awt.geom.GeneralPath (0x7491) 248 bytes 32 bytes 0:0:08.9 java.awt.geom.GeneralPath (0x7494) 248 bytes 32 bytes 0:0:06.9	▶	🛑 java.awt.geom.Ge	neralPath (0x7488)		248 bytes	32 bytes	n/a
Image: spin-sector of the spin sector of the sp	▶	🛑 java.awt.geom.Ge	neralPath (0x748e)		248 bytes	32 bytes	0:0:08.9
	▶	🛑 java.awt.geom.Ge	neralPath (0x7491)		248 bytes	32 bytes	0:0:08.9
• • java.awt.geom.GeneralPath (0x7497) 248 bytes 32 bytes 0:0:11.0	▶	🛑 java.awt.geom.Ge	neralPath (0x7494)		248 bytes	32 bytes	0:0:06.9
	►	👂 java.awt.geom.Ge	neralPath (0x7497)		248 bytes	32 bytes	0:0:11.0

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

 Classes 	2 Allocations	🕂 Biggest Objects	🔀 References	🚫 Time	🕸 Inspections 🔹			
Current object set: 1,534 instances of java.awt.geom.GeneralPath 2 selection steps, 49 kB shallow size, <u>Calculate retained and deep sizes</u> Use retained objects Incoming references Use Use Show In Graph Show Paths To GC Root								
	Object	Retai	ned Size 🔻	Shallow Size	Allocation Time (h:m:s)			
🔻 🛑 java.awt.geom.Gei	neralPath (0x747f)		248 bytes	32 bytes	0:0:13.0			
value of java.util.HashMap\$Node (0x747d)								
Velement of java.util.HashMap\$Node[] (0xd763)								
🔻 🔇 table of	java.util.HashMap (0x747c)						
🔹 🔍 🖉 leak	Map of bezier Bezier	Anim (0x6c11)						

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.

	Instance in current object set
0	Class object in current object set
0	Incoming reference
0	Reference cycle
0	Incoming reference in path to GC root
0	Class reference
Ĵ	GC root
X	Cutoff node
Terms And Abbre	eviations
JNI global	Global reference from native JNI code
JNI local	Local reference from native JNI code
array content	Reference from an array
class loader	Reference from a class loader
collection	Reference from a Java collection
constant pool	Reference from the constant pool of a class
field	Reference from an instance field of an object
instance of class	Reference from an instance to its class
interface	Reference through implementing an interface
java stack	Reference from the Java stack
map key	Reference from a key in a Java map
map value	Reference from a value in a Java map
monitor used	The monitor of an object is being used
and a secol	Defense of form the method starts
tack frames can be	ed in an active stack frame cannot be garbage collecte active permanently if a method calls never returns. method name of the stack frame are specified.

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 <code>java.lang.Class</code> objects are referenced except in the context of the classloader

Classes	P Allocations	Biggest Objects	🔀 References	🚫 Time	🕸 Inspections 🔸
Current object set:		•	•	nd deep sizes Use reta	ined objects
Incoming references	▼ Use				hs To GC Root
	Object	Ret	ained Size 🔹	Shallow Size	Allocation Time (h:m:s)
🔻 🛑 java.util.HashMap	\$Node (0x8c82)		5,344 bytes	32 bytes	n/a
▼	hent of javax.swing.U ables of javax.swing value of java.util. element of jav table of jav table of jav table of jav table of jav	ng.UIDefaults (0x88c8)	4) (0xd681) 3) 1x6988)	netovt (Av105)	
		sticky class	of class sun.awt.AppCo	ontext (UX105)	
		constant of class cor	n.sun.java.swing.Swing	Utilities3 (0x893)	
				ndows.AnimationContr ndows.WindowsButton	

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.

Path To GC Root Options	×
Select options for the path to root analysis:	
Calculating a single path to a garbage collector root is faster and often sufficient for memory leak detection.	
💿 Single root 🔞	
◯ Up to 2 ♦ roots	
All roots ?	
This search follows strong , as per the initial retention setting for the heap dump.	
Also follow soft, weak, phantom and finalizer for this search	2
✓ Stop search at classes	2
OK Cance	
Cance	<u> </u>

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.

Heap Snapshot Options	Х
Select recorded objects Initially, the heap walker will show only those objects that have been recorded in the dynam memory view section.	ic
Perform full GC in heap snapshot Perform full GC in heap snapshot Retain objects held by references of type strong	
Retain objects held by references of type strong	
Show Overhead Options - Soft weak phantom finalizer	cel

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.

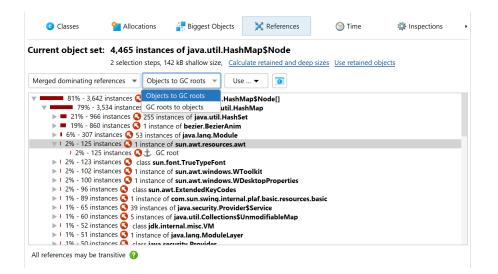
Classes	P Allocations	🕂 Bigge	st Objects	🔀 Refe	rences	🚫 Time	🕸 Inspections
Current object set Available Inspections: Duplicate object: Collections & An Reference & field	1 selection step, 6, s		Description Remove This will taking th	objects that only work fo e heap snap	r weak refere		oft or phantom reference. I have not removed when
 Weak references Select weakly refer Remove objects re Stack references Thread locals Classes & Class I 	enced objects tained by weak refere	nces	Status —	ion ierence type calculated		•	d create a new object set

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.

Path To GC Root Options	\sim
V Paul to de Root Options	
Select options for the path to root analysis:	
Calculating a single path to a garbage collector root is faster and often sufficient for memory leak detection.	
Single root 🕜	
◯ Up to 2 🜲 roots	
All roots 🕜	
This search follows strong , as per the initial retention setting for the heap dump.	
Also follow soft, weak, phantom and finalizer for this search	8
✓ Stop search at classes	0
OK Cance	el

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 JDK Flight Recorder (JFR)

E.1 Support For JDK Flight Recorder (JFR)

JDK Flight Recorder (JFR)⁽¹⁾ is a structured logging tool that records a broad range of system-level events. Similar to the black box of an aircraft that continuously records flight data for use in incident investigations, JFR continuously records a stream of events in the JVM for use in diagnosing problems. The advantage of this approach is that it captures chronologically detailed information about the system leading up to an incident. JFR is designed to have a minimal impact on performance, and to be safe to run in production environments over extended periods of time.

Starting with Java 17, JFR is also one of JProfiler's data sources. In addition to the native agent that uses the profiling interface of the JVM, there are high-level systems in the JVM that are of interest in a profiling context. One is the MBean system that provides data for some telemetries in JProfiler, and the other is JFR that is used for the garbage collector probe [p. 118]. For that purpose, you do not interact with JFR, but JProfiler handles JFR event streaming transparently.

JFR integration in JProfiler

JProfiler fully integrates JFR recording [p. 223], so you can easily capture data from running JVMs on the local machine or on remote machines where JFR recording was not configured.

When you open a JFR snapshot in the JProfiler UI, the available views and sections are different from a regular profiling session. The centerpiece of the UI is the event browser [p. 227]. All other views that are available for JFR views are explained in a separate chapter [p. 234].

As you work with event types, while setting filters and viewing analyses, JProfiler will occasionally have to rescan the JFR snapshot file. JFR snapshot files are potentially huge, and it is not viable to hold all data in memory or to calculate all analyses upfront. Because of this, it is not recommended to open JFR snapshots from network drives.

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 categories that are not required for your analysis. The available event categories cover single probes and view sections. Event types for CPU views, memory views and for the telemetry views are not optional and have to be loaded.

For example, if you are only interested in CPU data, you can exclude all probes and the event browser. JProfiler aims to be the fastest JFR viewer and opens typical JFR snapshots quickly, but JFR recordings are potentially unbounded and you could be confronted with a snapshot that is tens of gigabytes in size where the opening speed may become an issue.

Stack traces in JFR snapshots

One important feature of JFR is the ability to log the entire stack trace for a certain event type in an efficient way. For such events types, you can toggle stack trace recording in the JFR settings. Many JVM application event types, especially the ones that are concerned with threads, have stack trace recording enabled by default.

JFR only collects stack traces up to a fixed depth, so long stack traces 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>

⁽¹⁾ https://en.wikipedia.org/wiki/JDK_Flight_Recorder

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

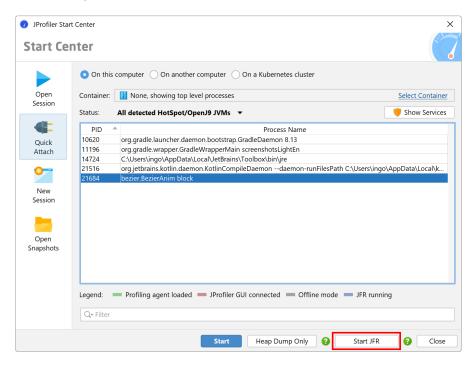
E.2 Recording JFR Snapshots With JProfiler

Due to the benefits of running JFR in production JVMs with a minimal overhead and no requirement to enable the profiling interface, JProfiler supports JFR recording directly in the UI. While you can start JFR programmatically or by adding the -XX:StartFlightRecording VM parameter on the command line, JProfiler helps you to start and stop recordings for JVMs that are already running.

When you attach to a JVM with JProfiler, you can choose to start and stop JFR recordings instead of loading the native profiling agent. With JProfiler's extensive remote connection capabilities, you can, for example, start JFR recordings in JVMs that run in Docker or Kubernetes containers without the need to modify a container.

Starting and stopping JFR recordings

On the "Quick attach" tab of the start center, select a JVM and click on the *Start JFR* button at the bottom of the dialog. Locally running JVMs are shown in the screenshot, but the same button is also available when you attach to a remote JVM.



In the JFR settings wizard, you can then select one of the **event settings templates** that are transmitted from thelib/jfr directory of the JRE that is used by the selected process. By default, there are two such templates, "default" and "profile", where "profile" records more data and adds more overhead. If you create other files in that directory, you will be able to select the corresponding templates in the wizard.

These template files contain the available events as well as configuration directives for important **high-level settings**. Each of the high-level settings can be coupled to a number of different events. This UI is dynamically generated based on the contents of the template file. Switching between the different profiles will show you the different default values. There are many more event types that are not included in this UI and which are only configurable in the next step.

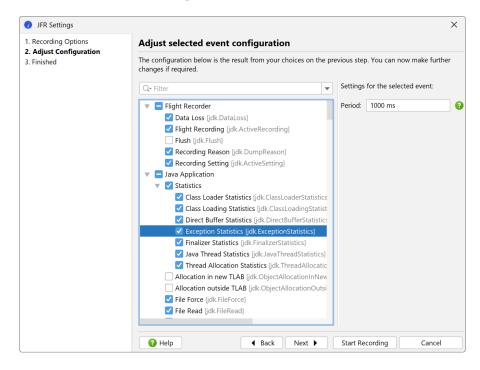
If you have already started a JFR recording for a JVM with the same set of event types, JProfiler will offer you the option to use the **last settings**.

JFR Settings			X
1. Recording Options 2. Adjust Configuration	JFR Recording opt	ions for the selected JVM	
3. Finished	Maximum snapshot size:	1,024 🗢 MB 😧	
	Available templates:	profile	-
Available templates: profile Settings for the selected t default Low overhead configuratii Last used for this type of JVM	default		
	settings for the selected t	profile	
	Low overhead configuration	Last used for this type of JVM	
	Garbage Collector:	Detailed	•
	Allocation Profiling:	Medium	•
	Compiler:	Detailed	•

If you select that option, the high-level recording settings will not be available and you can proceed to the next step to see the entire configuration and make further changes.

Another important setting on this step of the wizard is the **maximum snapshot size**. Due to the nature of JFR recordings, the size of a snapshot can increase very quickly and might fill up your entire hard disk. To avoid that, the maximum snapshot size constraint prevents excessive storage utilization. When the maximum size is reached, older events will be discarded while new events will continue to be recorded. This process is an automatic mechanism of JFR.

In the next step of the wizard you can see a categorized tree of all event types with further configuration for each event on the right side.



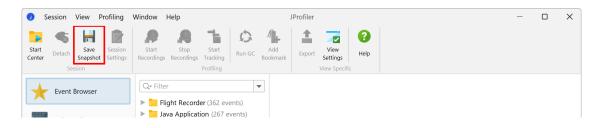
Events may have a setting for a **period**, a **threshold** and a flag whether to record a **stack trace** for each event or not. Both periods and thresholds are settings with time units and you can press the down key to get an auto-completion popup for the available units. Periods also support the special values "everyChunk", "beginChunk" and "endChunk" that are also available from the auto-completion popup. A "chunk" refers to a part of a JFR recording which holds a contiguous set of event data and metadata and functions as the basic unit of storage and data transport in a recording.

The more events are selected in the tree, the more data is being recorded. Some event types generate huge amounts of data while some generate only few events.

Unlike the full profiling mode or the "Heap dump only" mode where you immediately see some data in the UI, starting a JFR snapshot only modifies the background color of the JVM in the table when it is not selected so you can see that JProfiler has started a recording. When the JVM is selected, the text of the JFR button at the bottom now shows you that recording will be stopped.

tart Ce	On this com	puter 🔿 On another computer 🔿 On a Kubernetes cluster	
Open Session	Container:	None, showing top level processes detected HotSpot/OpenJ9 JVMs	Select Container
Quick Attach New Session Open napshots	11196 or 14724 C. ¹ 21516 or	Process Name g.gradle.launcher.daemon.bootstrap.GradleDaemon 8.13 g.gradle.wrapper.GradleWrapperMain screenshotsLightEn Users\ngo.VappDat\Local/UetBrainsToolbox\bin\jre g.jetbrains.kotlin.daemon.KotlinCompileDaemondaemon-runFilesPat zier.BezierAnim block	h C:\Users\ingo\AppData\Local\k
	Legend: F	Profiling agent loaded 📁 JProfiler GUI connected 📁 Offline mode	JFR running

When you stop a JFR recording that was started by JProfiler, a JFR snapshot will be transferred and opened in JProfiler. The **snapshot is temporary** and will be deleted when you close the window. To save the snapshot to a permanent location, use the "Save snapshot" action in the toolbar.



Terminated JVMs with JFR recordings

One mentioned use of JFR is to investigate the moments before a crash. In that case, the JVM will not be available in the JVM table anymore to stop JFR recording and open the JFR snapshot. If a JFR recording has been started in JProfiler and the JVM terminates before you stop the recording, a special entry prefixed with "Terminated JFR:" will be added to the JVM table. By double-clicking on that entry or using the "JFR" button, you can open the JFR snapshot.

Vame Vaemon 8.13 shotsLightEn x\bin\jre nondaemon-runFilesPath C:\Users\ing
baemon 8.13 shotsLightEn xx\bin\jre
shotsLightEn x\bin\jre
x\bin\jre
nondaemon-runFilesPath C:\Users\ing
nected 🔲 Offline mode 💻 JFR runnir

Once you open such an entry, it will be removed from the list. Just like for recordings that are stopped manually, the opened JFR snapshot will be temporary and you have to save it if you want to keep it for later analysis.

Showing externally started JFR recordings

In the example above, the JFR recording has been started and stopped in JProfiler. JFR recordings that are started outside JProfiler can also be shown. Continuous JFR recordings can easily be started with a VM parameter like

"-XX:StartFlightRecording=maxsize=500m=filename=\$TEMP/myapp.jfr,name=Continuous recording"

The indication via the special background color in the JVM table that a JFR recording is running only refers to JFR recordings that were started in JProfiler. If you connect to a JVM where a JFR recording has been started by other means, another dialog will be shown.

Select JFR recording	×
Existing JFR recordings have been found in the selected JVM. You can start a new recording from JProfiler or dump an existing recording and open the resulting JFR snapshot in JProfiler.	
Start a new JFR recording from JProfiler	
Open a JFR snapshot for the selected recording:	
Continuous recording [running]	
Type into the list to start search	
OK	cel

You can now choose to start a new recording in JProfiler or to dump an existing recording and show the resulting JFR snapshot in JProfiler. Externally started JFR recordings have a separate life cycle and will not be stopped by JProfiler.

E.3 The JFR Event Browser

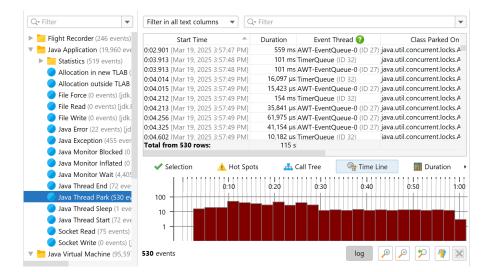
The event browser shows all data that has been recorded in a JFR snapshot.

A	Q - Filter	•	
Event Browser	Flight Recorder (246 events)		
	V by Java Application (19,960 events)		
Telemetries	Statistics (519 events)		
	Allocation in new TLAB (11,096 ev	vents) <mark>(</mark> jdk.Objec	
Memory	Allocation outside TLAB (2,713 ev	ents) [jdk.Objec	
•	File Force (0 events) [jdk.FileForce	1	
CPU Views	🥏 File Read (0 events) [jdk.FileRead]		
CFO VIEWS	🥏 File Write (0 events) [jdk.FileWrite	1	
_	🥏 Java Error (22 events) [jdk.JavaErr	orThrow]	
🦻 Threads	Java Exception (455 events) [jdk.J	avaExceptionTh	Select an event type to view data
-	Java Monitor Blocked (0 events) [jdk.JavaMonitor	
Monitors & Locks	🥏 Java Monitor Inflated (0 events) [j	jdk.JavaMonitor	
	Java Monitor Wait (4,405 events)	[jdk.JavaMonitc	
	Java Thread End (72 events) [jdk.]	[hreadEnd]	
Probes	🥏 Java Thread Park (530 events) [jdl	k.ThreadPark]	
	🥏 Java Thread Sleep (1 event) [jdk.T	'hreadSleep]	
	🥏 Java Thread Start (72 events) [jdk.	ThreadStart]	
	🔵 Socket Read (75 events) [jdk.Sock	etRead]	
	🔵 Socket Write (0 events) [jdk.Socke	etWrite]	
	🔻 😓 Java Virtual Machine (95,597 events)		

JFR organizes event types into hierarchical categories that make up the tree on the left side of the event browser. You can select a single event type to show the recorded events. By default, JProfiler shows all registered event types, even if no events were recorded for them. Alternatively, you can choose to hide empty event categories in the view settings dialog.

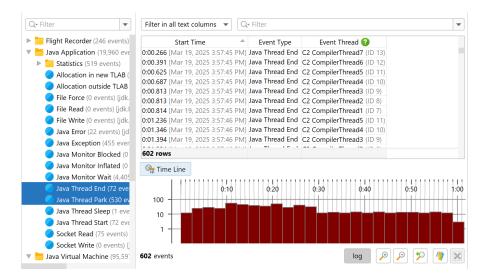
JFR events

Events are shown as rows in the main table with the columns depending on the selection in the tree of event types.



The events in the table are sorted chronologically by default. To avoid overloading the UI, only the first 10000 events are shown in the table. The **analyses at the bottom** are always calculated from all events. If you set a filter, it will also check all events, not just the first 10000. This means that when setting a filter, events may show up in the table that were previously not displayed.

You can also **select multiple event types** or entire categories. In that case, the union of all selected events is shown in the table. Because each event type has its own set of columns, only those columns that are common to all selected event types will be included.

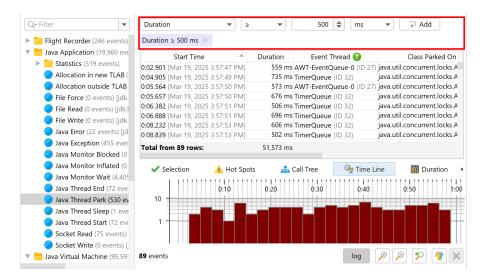


The number of available analyses may also be reduced because analysis views are added based on the available columns.

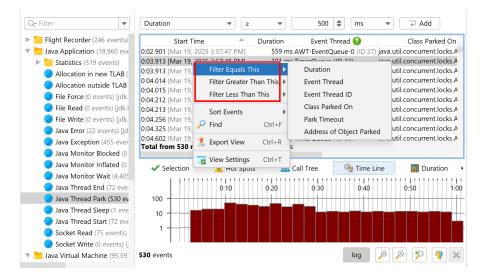
Column widths are adjusted automatically based on their actual content until you resize a column. Then, the width of columns with the same content type will be fixed to your selection and will not change automatically anymore until you **clear the column widths** in the view settings dialog. Scales in columns with units like time or memory are also calculated automatically for each cell. If you prefer to fix the scale of a column for better comparability, the view settings dialog offers an option for each such column. In this case, the setting is persisted separately for each selected event type.

🥑 JFR Eve	ents View Settings			X
General	Hot Spots and Call Tree	Time Line	Histograms	
Time Axis				
🗹 Scal	e to fit window 😯			
Show Boo	kmarks			
O Nor	ie 🔵 In time scale 💿 Ii	n entire view		
Grid Lines	for Time Axis			
O Nor	ie 🔵 At major ticks 💿	At major and	minor ticks	
Grid Lines	for Vertical Axis			
O Nor	ie 🔵 At major ticks 💽	At major and	minor ticks	
Event Cou	unts			
🔽 Logi	arithmic display for event c	ounts		
		_		
		ОК	Cano	el

There are several ways to filter events. At the top of the table, there is a filter selector that allows you to filter in all text columns or to select a single column and configure a filter that matches the column type.



Another way to filter is to select a row of interest and use the context menu to select a specific filter based on the values in the selected row. The filter selector at the top will be adjusted, so that it displays your selection. You can now choose another value and add the filter again, it will then replace the previous filter for the same column. In general, **each filter type can only be present once** and setting the same filter again will replace the previous filter.



Stack traces

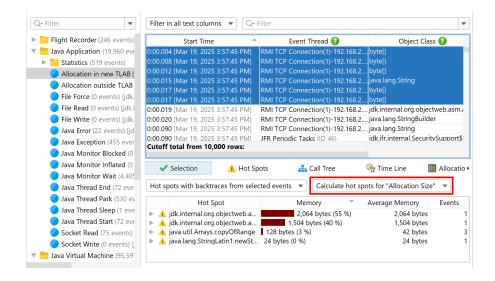
In JProfiler, the stack trace of a selected event is visible in the "Selection" tab of the split pane below the event table.

Flight Recorder (246 events)	Start Time	A	Duration	Event Thr	ead 🕜		Class Parl
Java Application (19,960 eve 🛛 🕻	02.901 [Mar 19, 2025	5 3:57:47 PM]	559 ms	AWT-EventQue	ue-0 (ID 27)	java.util.co	ncurrent.locks.
Statistics (519 events)	03.913 [Mar 19, 2025	5 3:57:48 PM]	101 ms	TimerQueue (II	0 32)	java.util.co	ncurrent.locks.
Allocation in new TLAD	0:03.913 [Mar 19, 2025			AWT-EventQue		-	ncurrent.locks.
Allocation outside LLAB	04.014 [Mar 19, 2025			TimerQueue (II		-	ncurrent.locks.
File Force (0 events) fidk	0:04.015 [Mar 19, 2025			AWT-EventQue			
	0:04.212 [Mar 19, 2025			TimerQueue (II		-	ncurrent.locks.
	0:04.213 [Mar 19, 2025	4		AWT-EventQue		·	ncurrent.locks.
	:04.256 [Mar 19, 2025			AWT-EventQue			
	04.325 [Mar 19, 2025			AWT-EventQue			
	04.602 [Mar 19. 2025] Total from 530 rows:		10, 182 µs 115 s	TimerOueue (IE) 32)	lava.util.co	ncurrent.locks.
Java Monitor Blocked (0	otal nom 550 rows.	•	112.3				
Java Monitor Inflated (0	Selection	🔥 Hot Spots	s 🚣 (Call Tree	🎭 Time Li	ne	🔢 Duration
Java Monitor Wait (4,405							
Java Thread End (72 eve	Stack trace:						
🥏 Java Thread Park (530 ev	jdk.internal.misc.Unsa	afe.park(boolear	n, long)				
Java Thread Sleep (1 eve	java.util.concurrent.lc	ocks.LockSuppor	t.park(java.laı	ng.Object)			
Java Thread Start (72 eve	java.util.concurrent.lc		euedSynchro	nizer\$Condition	Object.await	0	
Socket Read (75 events)	java.awt.EventQueue	.getNextEvent()					
	java.awt.EventDispate	chThread.pump(OneEventForF	-ilters(int)			
Socket Write (0 events) [java.awt.EventDispate	chThroad purport	Twonts For Filto	vr(int_inva_awt C	onditional is	ave and Eve	ntEiltor)

If you select multiple events, the selection tab changes to a view that shows you either the hot spots or the cumulated call tree calculated from the stack traces of the selected events.

Java Application (19,960 evc 0.02.901 [Mar 19, 2025 3:57.47 PM] Statistics (519 events) 0.03.913 [Mar 19, 2025 3:57.48 PM] Allocation in new TLAB (0.04.013 [Mar 19, 2025 3:57.48 PM] Allocation outside TLAB 0.04.015 [Mar 19, 2025 3:57.49 PM] File Force (0 events) [jdk. 0.04.015 [Mar 19, 2025 3:57.49 PM] File Read (0 events) [jdk. 0.04.212 [Mar 19, 2025 3:57.49 PM] O:04.212 [Mar 19, 2025 3:57.49 PM] 0.04.213 [Mar 19, 2025 3:57.49 PM] O:04.225 [Mar 19, 2025 3:57.49 PM] 0.04.225 [Mar 19, 2025 3:57.49 PM] O:04.225 [Mar 19, 2025 3:57.49 PM] 0.04.325 [Mar 19, 2025 3:57.49 PM]	101 ms TimerC 101 ms AWT-E 16,097 µs TimerC 15,423 µs AWT-E 154 ms TimerC 35,841 µs AWT-E	Queue (ID 32) EventQueue-0 (ID 27) Queue (ID 32) EventQueue-0 (ID 27)	java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks
 Allocation in new TLAB (Allocation outside TLAB Allocation outside TLAB File Force (0 events) [idk.] File Read (0 events) [idk.] File Write (0 events) [idk.] Cot4.212 [Mar 19, 2025 3:57:49 PM] 	101 ms AWT-E 16,097 μs TimerC 15,423 μs AWT-E 154 ms TimerC 35,841 μs AWT-E	EventQueue-0 (ID 27) Queue (ID 32) EventQueue-0 (ID 27) Queue (ID 32)	java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks
 Allocation outside TLAB Allocation outside TLAB File Force (0 events) [idk.] File Read (0 events) [idk.] File Write (0 events) [idk.] Oct4.213 [Mar 19, 2025 3:57:49 PM] Oct4.213 [Mar 19, 2025 3:57:49 PM] Oct4.213 [Mar 19, 2025 3:57:49 PM] 	16,097 µs TimerC 15,423 µs AWT-E 154 ms TimerC 35,841 µs AWT-E	Queue (ID 32) EventQueue-0 (ID 27) Queue (ID 32)	java.util.concurrent.locks java.util.concurrent.locks java.util.concurrent.locks
 Allocation outside TLAB File Force (0 events) (jdk.) File Read (0 events) (jdk.) File Write (0 events) (jdk.) Cod4.212 [Mar 19, 2025 3:57:49 PM] Ocd4.213 [Mar 19, 2025 3:57:49 PM] Ocd4.213 [Mar 19, 2025 3:57:49 PM] Ocd4.214 [Mar 19, 2025 3:57:49 PM] Ocd4.215 [Mar 19, 2025 3:57:49 PM] 	15,423 µs AWT-E 154 ms TimerC 35,841 µs AWT-E	EventQueue-0 (ID 27) Queue (ID 32)	java.util.concurrent.locks. java.util.concurrent.locks.
 File Force (0 events) [idk.] File Read (0 events) [idk.] File Read (0 events) [idk.] File Write (0 events) [idk.] Otd.213 [Mar 19, 2025 3:57:49 PM] Otd.213 [Mar 19, 2025 3:57:49 PM] Otd.2256 [Mar 19, 2025 3:57:49 PM] 	154 ms TimerC 35,841 µs AWT-E	Queue (ID 32)	java.util.concurrent.locks.
 File Read (0 events) [jdk.1 File Write (0 events) [jdk.4 File Write (0 events) [jdk.4 C:04.213 [Mar 19, 2025 3:57:49 PM] C:04.255 [Mar 19, 2025 3:57:49 PM] C:04.255 [Mar 19, 2025 3:57:49 PM] 	35,841 µs AWT-E	• • •	
File Write (0 events) [jdk. 0:04.256 [Mar 19, 2025 3:57:49 PM]		EventQueue-0 (ID 27)	
	61,975 µs AWT-E		
Java Error (22 events) [id 0:04.325 [Mar 19, 2025 3:57:49 PM]			java.util.concurrent.locks.
			java.util.concurrent.locks.
Java Exception (455 ever Java Exception (455 ever Total from 530 rows:	10,182 µs TimerC 115 s	Dueue (ID 32)	iava.util.concurrent.locks.
Java Monitor Blocked (0	115.5		
Java Monitor Inflated (0		(Q) . T	
Java Monitor Wait (4.405	s 📥 Call Tre	ee 🛛 🙀 Time Lii	ne 🌃 Duration
Java Thread End (72 eve Call tree from selected events	▼ C	Calculate hot spots for	event counts 🔻
Invo Throad Dark (520 or			
60.7% - 2 evt. java.awt.		i.run	
Java Thread Sleep (1 eve	read.run		
Java Thread Start (72 eve			
Socket Read (75 events)			
Socket Write (0 events) [

By default, event counts determine the percentages on the nodes in the call tree and hot spots views. Some event types include other measurements that are suitable for this purpose, such as a duration or allocated memory. If such measurements are available, you can select them as the **hot spot type** from the second drop down in the selection tab.



The "Hot spots" and the "Call tree" views in the lower split pane contain the same views, however, they are **calculated for all events** in the snapshot. Similar to the selection tab, they also have a "hot spot type" drop down. In addition to showing all events, you can also **select a filter** from these views. In the call tree view, selecting a particular call stack and clicking on the *Filter selected* button will only show events with that call stack in the table above. For the hot spots view, you can either select the hot spot at the top level or any node in the back trace, so that only events will be shown whose stack trace ends with the inverted call stack fragment to the selected node.

Start Time	-	Eve	nt Thread 🕜	Object Class 🕜	Allocation Size	TLAB Size	
0:05.084 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	56 bytes	40,000 bytes	
0:05.108 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	208 bytes		
0:05.126 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	112 bytes	40,056 bytes	
0:05.129 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	192 bytes	40,136 bytes	
0:05.132 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	192 bytes	40,136 bytes	
0:05.139 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	184 bytes	40,128 bytes	
0:05.140 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	200 bytes	40,144 bytes	
0:05.141 [Mar 19, 2025 3:57	':50 PM] A	WT-EventQ	ueue-0 (ID 27)	byte[]	184 bytes	40,128 bytes	
0:05.142 [Mar 19, 2025 3:57	:50 PM1 A	WT-EventQ	ueue-0 (ID 27)	byte[]	288 bytes	40,232 bytes	
Total from 400 rows:					65,928 bytes	10,004 kB	
Selection	👠 Hot Sp	oots	📥 Call Tree	🎭 Time Line	🚻 Allocation Size	🚻 TLAB S	ize
Calculate hot spots for "A	llocation S	ize" 🔻	Filter Selected	Remove Filter	Cal	culated from all 1	1,096 ever
	Hot S	pot		Memory	 Avera 	ge Memory	Events
 A java.util.Arrays.copy 0 = 31.6% - 635 k 			jdk.internal.loader.f		3 kB (35 %)	1,163 bytes	613
				stractStringBuilder.ensu	reCapacityInternal		

In the screenshot above, you can see that a node in the backtrace was selected as the filter node. In addition to the regular call tree icon, it also includes a check mark. You can remove filters with the tag label at the top or via the *Remove filter* button. The event count in the table is equal to the number on the selected node. The hot spot tree still shows all events without the filter that was set in the hot spot view.

This is a general feature of filters that are set from the analysis views: The analysis view itself is calculated from all filtered events, **but excluding the filter that was set in the analysis view**. This makes the analysis view more useful because you can see what part of the total event set you have selected there.

Time line view

All JFR events have associated times, so every event type or set of event types has a time line view that shows the chronological distribution of events.

elected hot spot with b	oacktrace ×	Event start time 10 s to 20 s	×			
Start Time		Event Thread 🕜	Object Class 😱	Allocation Size	TLAB Size	
10.154 [Mar 19, 2025 3	3:57:55 PM] AW	-EventQueue-0 (ID 27)	byte[]	192 bytes	70,152 bytes	
		-EventQueue-0 (ID 27)	byte[]	104 bytes	70,064 bytes	
10.160 [Mar 19, 2025 3	3:57:55 PM] AW1	-EventQueue-0 (ID 27)	byte[]	200 bytes	70,160 bytes	
10.161 [Mar 19, 2025 3	3:57:55 PM] AW1	-EventQueue-0 (ID 27)	byte[]	192 bytes	70,152 bytes	
10.163 [Mar 19, 2025 3	8:57:55 PM] AW1	-EventQueue-0 (ID 27)	byte[]	104 bytes	70,064 bytes	
10.163 [Mar 19, 2025 3	3:57:55 PM] AW1	-EventQueue-0 (ID 27)	byte[]	200 bytes	70,160 bytes	
10.418 [Mar 19, 2025 3	8:57:55 PM] AW1	-EventQueue-0 (ID 27)	byte[]	56 bytes	37,472 bytes	
10.454 [Mar 19, 2025 3	8:57:55 PM] AW1	-EventQueue-0 (ID 27)	byte[]	104 bytes	37,520 bytes	
	8:57:55 PM1 AW1	-EventQueue-0 (ID 27)	byte[]	104 bytes	37,520 bytes	
otal from 198 rows:				31,464 bytes	3,466 kB	
 Selection 	႔ Hot Spots	🚠 Call Tree	💁 Time Line	Allocation Size	🔢 TLAB Size	
100 -	0:10	0:20	0:30	0:40	0:50	1:0
10						

To focus on a particular time range, you can **drag along the time axis**. In the above example, we now have two filters: A filter from the backtrace of a hot spot and the filter from the timeline view. Again, the time line view continues to show the entire time range while other analysis views will now only show events from the selected time range.

The default display mode is **logarithmic**, so that regions of low event counts are still visible against regions of high event counts. You can switch to linear mode by deselecting the *log* button below the time line. By default, the entire time range is shown in the available width, but you can switch to a variable time range and zoom and scroll just like in the other telemetries in JProfiler. Also available are **bookmarks** where you can add a vertical marker at selected time ranges. In that way, you can compare moments in time across different event types.

Histogram views

All measurements that can be summed for multiple events, such as durations and allocation sizes, are treated in a special way: First, the columns of these measurements in the event table have a total value at the bottom. Second, the call tree and hot spot analysis views offer a "hot spot type" drop-down to calculate their trees with these measurements instead of event counts. Finally, for each such measurement, a histogram analysis is added to the lower split panel.

Filter in all text colum	nns 🔻 🔾	- Filter									
Duration 905 ms to 1,	805 ms $ imes$										
Start Time		Duration	Event Th	nread 🕜		Cla	iss Parke	d On		P	ark Timeo
0:09.342 [Mar 19, 2025	5 3:57:54 PM]	1,097 ms Ti	imerQueue ((ID 32)	java.util	concurren	t.locks.Al	ostractQue	euedSyn.		
:13.976 [Mar 19, 2025	3:57:58 PM]	1,358 ms Ti	imerQueue ((ID 32)	java.util	concurren	t.locks.Al	ostractQue	euedSyn.		
22.375 [Mar 19, 2025]	5 3:58:07 PM]	1,647 ms Ti	imerQueue ((ID 32)	java.util	concurren	t.locks.Al	ostractQue	euedSyn.		
otal from 3 rows:		4,103 ms									
Total from 3 rows:	🔥 Hot Spo		II Tree	强 Time	Line	📊 Dur	ation	Mari Parl	k Timeou	ıt	
✓ Selection	A Hot Spo	ts 🚠 Ca	II Tree	Mar Time	Line	Dur 2 s	ation	Parl	k Timeou	ıt 3 s	
	Hot Spo	ts 🚠 Ca	1 1 1	Real Provide American Science (Construction) and the second science (C	Line		ation	Parl	k Timeou	1	

Histograms show event counts on their vertical axis while the horizontal axis shows the selected measurement and is divided into a number of bins, so that a distribution can be calculated. Bin sizes and event counts are available from the tooltip.

The screenshot above shows how a filter has been set in the histogram. Just like for other analysis views, the filter only applies to other analysis views, and the entire histogram is still shown. As for the timeline view, the histogram has a logarithmic vertical axis by default. Here, the selected events in the screenshot would not be visible with a linear axis.

E.4 Views In JFR Snapshots

Apart from the JFR event browser [p. 227], JProfiler uses some of the views that are available for full profiling sessions and fills them with JFR data. This is possible because JFR collects data for memory allocations and method executions. The main limitation is that the recording rates are low, so getting enough data to see problematic hot spots can take a long time.

Telemetries

With the exception of the "Recorded objects telemetry", all telemetries in full profiling sessions are also available in JFR snapshots with some limitations in the displayed data. The memory telemetry does not show GC-specific pools, the threads telemetry does not show thread counts by thread state and the recorded throughput telemetry shows sizes instead of object counts and does not show the objects that being freed.



The table below shows the event types that are used by the various telemetries and whether they are enabled in both the "default" and the "profile" template.

Telemetry	Event types	Enabled in profile
Memory	jdk.GCHeapSummary, jdk.MetaspaceSummary	all
Recorded throughput	jdk.ObjectAllocationSample, jdk.ObjectAllocationInNewTLAB, jdk.ObjectAllocationOutsideTLAB	profile only
GC activity	jdk.GarbageCollection	all
Classes	jdk.ClassLoadingStatistics	all
Threads	jdk.JavaThreadStatistics	all
CPU load	jdk.CPULoad	all

Memory views

In the "Memory" section, two different event types are used to populate the views with data. The "Live objects" view shows you a statistical representation of all classes and instance counts that remain on the heap after a full garbage collection. This data is only available if the jdk. ObjectCount event is enabled, which is not the case for either of the default JFR templates,

because it comes with a significant overhead. You can also toggle this setting in the high-level JFR configuration with the "Garbage collector" drop-down. Prior to Java 17, this drop-down is labeled as "Memory profiling".

If the jdk.ObjectCount event was recorded more than once in the snapshot, the view will show you the **difference between the first and the last occurrences** of the jdk.ObjectCount event. In that way, you get a sense of how the numbers changed during the recording time and may provide some indication of a memory leak. 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 included.

For any serious investigation consider using a full profiling session [p. 71] or taking an HPROF snapshot [p. 206].

Event Browser	All objects dumps:	🔺 🗾 🗙	Aggregation level:	Classes	
	at 1:41.196.857		Name	Instance Count	Size
Telemetries	at 0:37.290.151		byte[]	178,105 (23 %)	17,099 k
-	at 0:36.471.646		java.lang.String	172,729 (23 %)	4,145 k
	at 0:36.317.685		java.util.HashM	66,292 (8 %)	2,121 k
Memory			java.util.concurr	34,363 (4 %)	1,099 k
•	at 0:35.953.029		long[]	29,888 (4 %)	2,403 k
ive Objects	at 0:35.742.638		jdk.internal.jima	27,209 (3 %)	653 k
ive objects	at 0:35.600.776		jdk.internal.jima	27,209 (3 %)	870 k
ecorded Objects	at 0:35.245.674		java.lang.Object[]	25,838 (3 %)	1,492 k
·	at 0:35.109.264		java.util.Hashta	24,457 (3 %)	782 k
llocation Call Tree	at 0:34.252.359		java.lang.Object	19,661 (2 %)	314 k
	at 0:19.664.393		java.util.Linked	19,430 (2 %)	777 k
location Hot Spots	at 0:18.573.544		java.lang.Class[]	17,595 (2 %)	375 k
	at 0:17.059.798		java.util.HashM	15,243 (2 %)	1,148 k
CPU Views			java.lang.reflect	14,949 (2 %)	1,315 k
P	at 0:16.335.596		java.lang.Class	14,538 (1 %)	1,757 k
	at 0:15.683.881		java.util.ArrayList		308 k
A Threads	at 0:14.691.350		java.lang.ref.We	12,215 (1 %)	390 k
1	at 0:13.936.178			11,335 (1 %)	634 k
	at 0:05.238.510		Total from 22 r	744 845 (100 %)	48,390 k
Monitors & Locks	at 0:00.117.003		1010111221	144,045 (100 70)	40,330 K
-			Q+ Class View Filt	ers	-

The "Recorded objects" view as well as the allocation views show you data from the jdk. ObjectAllocationSampleevent since Java 16 and the jdk.ObjectAllocationInNewTLAB and jdk.ObjectAllocationOutsideTLAB events in earlier Java versions. The "Allocation Profiling" drop-down in the high-level UI also provides a way to enable these event types.

Contrary to the "Live objects" view, they only show objects that were allocated while recording was active. Allocations are sampled by JFR but the size is reported as an **estimate for the total allocated size**. Because of this discrepancy, the sizes reported by these views do not correspond to the sample count multiplied by the average instance size. Otherwise, these views have similar functionality to the memory views in full profiling session [p. 71].

CPU views

The "CPU views" include the call tree, the hot spots view as well as the call graph. Data in the "Runnable" thread state is based on the jdk.ExecutionSample events that are recorded by default in both standard JFR templates. However, the sampling rate is set to 20 ms by default, which corresponds to the "Normal" option of the "Method sampling" setting in the JFR high-level UI. Considering that JFR only samples a very small number of random threads, getting sufficient data so that hot spots stand out sufficiently can take a very long time. Consider lowering the period for the jdk.ExecutionSample if necessary. Keep in mind that this can lead to very large snapshot sizes because JFR does not cumulate data.

Event Browser	Thread status: 😗 Thr		55 5		ot options:	
	Runnable 🔻	🖁 All thread grou 🔻	🔟 Methods 🛛 🔻		Self events	
	Hot Spot	Self Events	~	Average Time	Invocations	
Telemetries	🕨 🔺 jdk.internal.jrtfs.JrtPath.n	37	(11 %)	n/a	n/	
	🕨 🔺 java.util.HashMap\$TreeN	. 25 (7 %)		n/a	n/	
B	jj2000.j2k.entropy.encode	17 (5 %)		n/a	n/	
Memory	🕨 🔺 java.lang.StringLatin1.has	15 (4 %)		n/a	n/	
	jj2000.j2k.entropy.encode	13 (4 %)		n/a	n/	
	jj2000.j2k.entropy.encode	13 (4 %)		n/a	n/	
CPU Views	🕨 🔺 java.io.BufferedInputStrea.	12 (3 %)		n/a	n/	
	🕨 🔺 java.lang.StringLatin1.last	🚺 11 (3 %)		n/a	n/	
Call Tree	🕨 🔺 java.lang.AbstractStringB	. 📕 7 (2 %)		n/a	n/	
	🔄 🕨 🛕 java.nio.file.Files.provider	6 (1 %)		n/a	n/	
Hot Spots	🕨 🔺 java.util.Spliterators\$1Ada.	📕 5 (1 %)		n/a	n/	
	🚽 🕨 🔺 jj2000.j2k.wavelet.analysis.	📕 5 (1 %)		n/a	n/	
Call Graph	🕨 🔺 java.awt.image.Compone	. 📕 4 (1 %)		n/a	n/	
	🕨 🔺 java.util.HashMap.getNod	e 📕 4 (1 %)		n/a	n/	
Threads	🕨 🔺 java.util.HashMap.putVal	4 (1 %)		n/a	n/	
2 mileaus	🕨 🔺 com.ejt.framework.graphi	📕 3 (0 %)		n/a	n/	
	🕨 🔺 java.util.zip.ZipFile\$Sourc	. 📕 3 (0 %)		n/a	n/	
A subset of tests	🕨 🔺 jj2000.j2k.codestream.writ.	📕 3 (0 %)		n/a	n/	
Monitors & Locks	jj2000.j2k.entropy.encode	📕 3 (0 %)		n/a	n/	
_	🕨 🕨 🔥 com.install4j.compiler.gui			n/a	n/	
	🕒 🔺 iava in RufferedInnutStrea	2 (0 %)		n/a	n/	
Probes	Q- Hot Spot View Filters				-	

Due to the fact, that threads are sampled sporadically, it is not possible to estimate actual execution times like in a full profiling session. Rather than times, the **event counts** are shown in the call tree and the hot spots views. This is similar to async sampling [p. 66] which has the same drawback. The other JFR thread states are "Waiting", "Blocking" and "Socket and file I/O" and still measure times. Because of this discrepancy, the "All thread states" mode is not available in the thread status selector.

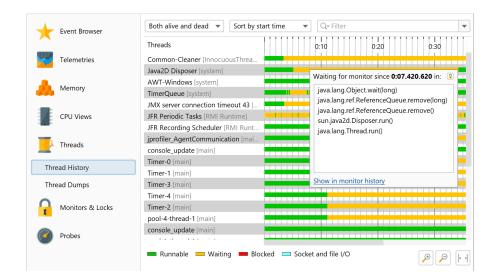
Another consideration is that the non-runnable thread states are calculated from events which have configurable minimum duration thresholds that are shown in the tool tip next to the thread status selector. The actual total time of these thread states may be significantly larger. The table with the event types used for assembling the thread states is shown below:

Thread state	Event types
Runnable	jdk.ExecutionSample
Waiting	jdk.JavaMonitorWait, jdk.ThreadSleep, jdk.ThreadPark
Blocking	jdk.JavaMonitorEnter
Socket and file I/O	jdk.SocketRead, jdk.SocketWrite, jdk.FileRead, jdk.FileWrite

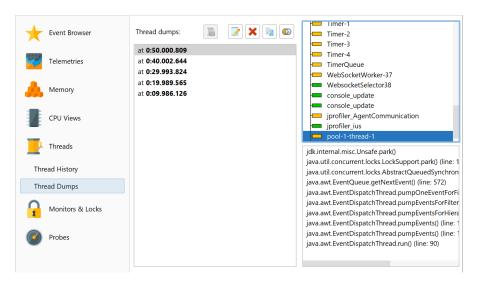
The functionality of the views is explained in the help topic on the CPU views [p. 53]. Note that many features of full profiling sessions are not available in a JFR context.

Thread and monitor views

From the chronological method sampling data, the thread history view can be calculated, including the tool tips that show stack traces for waiting and blocking times.



Thread dumps are a feature in both JFR and JProfiler and are shown in the same view. In this case, the event browser is not a substitute because it has no way of showing the structured content of the thread dump column of the jdk. ThreadDump event. In the thread dumps view you can also compare different thread dumps [p. 97].



From the jdk.JavaMonitorWait, jdk.ThreadSleep and jdk.ThreadPark events, JProfiler calculates a monitor history similar to the one of a full profiling session [p. 97], only without the information on blocking threads. If you require that information for solving your problem, please switch to a full profiling session. This also means that the locking graphs from the full profiling session are not available for JFR snapshots. The monitor usage statistics that shows aggregate information on waiting events is present and shows waiting times only.

	Time 🔺	Duration	Туре	Monitor Address	Monitor Class	Waiting Thread
	0:00.003 [95,996 µs	💻 Wait	0x21a7b273888	com.sun.jmx.remo.	RMI TCP Connectio
Telemetries	0:00.094 [15,444 µs	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
	0:00.099 [3,399 ms	💻 Wait	0x21a7b273888	com.sun.jmx.remo.	RMI TCP Connectio
L. Marian	0:00.109 [15,455 µs	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
Memory	0:00.125 [15,310 µs	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
	0:00.141 [15,444 µs	💻 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
CPU Views	0:00.156 [15,469 µs	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
CPU Views	0:00.172 [15,493 µs 🕯	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
	0:00.187 [15,503 µs	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
A Threads	0:00.203 [15,479 µs 🕻	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
2 micadas	0:00.219 [15,422 µs	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
	0:00.234 [15,460 µs 🛙	🗖 Wait	0x21a7b26e888	java.lang.Object	JFR Periodic Tasks
Monitors & Locks	0.00 250 1		Wait	0v21a7h26e888	iava lang Ohiect	IFR Deriodic Tasks
MONITOR'S & LOCKS	Total fro	692 s				
Monitor History	Filtered stack	trace for waitir	ng thread:	Recording thre	sholds: 10,000 µs bl	locking / 10,000 µs w
Monitor Usage Statistics		ject.wait(long) .remote.intern		tificationBuffer.fetchN	otifications(com.sun	.jmx.remote.internal.N
Probes			-	tificationBuffer\$Share otifForwarder.fetchNot		ons(com.sun.jmx.rem

Probes

Some of the JVM probes in a full profiling session have equivalent data sources in JFR snapshots. Their main advantage compared to the event browser is that they combine multiple related event types. The table below shows the available probes with the event types that are used as their data sources.

Probe	Event types	Enabled in profile
Sockets	jdk.SocketRead, jdk.SocketWrite	all
Files	jdk.FileRead, jdk.FileWrite	all
Classes	jdk.ClassLoad, jdk.ClassUnload, jdk.ClassDefine	none
Exceptions	jdk.JavaErrorThrow, jdk.JavaExceptionThrow	errors in both, exceptions in none
Garbage Collector	jdk.GarbageCollection, jdk.GCPhasePause, jdk.YoungGarbageCollection, jdk.OldGarbageCollection, jdk.GCReferenceStatistics, jdk.GCPhasePauseLevel <n>, jdk.GCHeapSummary, jdk.MetaspaceSummary, jdk.GCHeapConfiguration, jdk.GCConfiguration, jdk.YoungGenerationConfiguration, jdk.GCSurvivorConfiguration, jdk.GCTLABConfiguration</n>	all

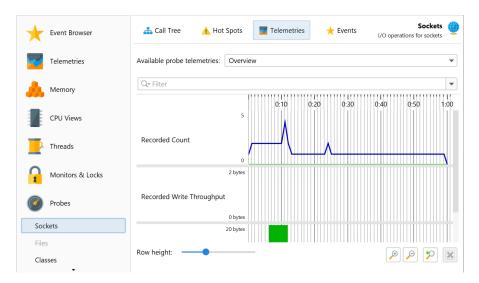
Class loading has a separate check box in the high-level JFR UI that switches on all three class loading events.

Each probe shows a number of views. In contrast to the event browser, the focus is on the aggregated data and not on the single events. This is also how probes in JProfiler differ conceptually from JFR data collection.

Except for the Garbage collector probe, all probes have the following views: The call tree and hot spot views allow you to choose a single thread or a thread group as well as an aggregation level. By default, all threads are shown and the aggregation level is set to "Methods".

Telemetries	📥 Call Tree 🔒 Hot Spots 🖉 Telemetries 🔶 Events	Exceptions
A Memory	Thread selection:	Aggregation level:
CPU Views	Hot Spot	Events 249 (52 %)
Threads	 ▶ ▲ java.lang.NoSuchMethodException ▶ ▲ java.lang.NoSuchMethodError ▶ ▲ java.net.MalformedURLException 	188 (39 %) 22 (4 %) 13 (2 %)
Monitors & Locks	 A java.awt.IllegalComponentStateException A java.io.FileNotFoundException A java.lang.ClassNotFoundException 	2 (0 %) 2 (0 %) 1 (0 %)
Probes		
Sockets		
Files		
Classes		
Exceptions		
Garbage Collector	Qr Payload View Filters	•

The telemetries view displays one or more telemetries from the recorded data with an overview page that shows all of them at once. The full telemetry can be opened by clicking on the telemetry name. By dragging along the time axis, you can select the corresponding events in the events view.



The events view is similar to the one in the JFR browser. However, it shows multiple event types corresponding to the underlying JFR events and offers a type selector. Filtering and stack trace display for single and multiple selection are handled just as in the event browser. Also, there are histogram views for time and memory measurements where you can select ranges by dragging along the horizontal axis.

Telemetries	📥 Call Tree	🛕 Hot Spots	🕎 Telemetries 🛛 🕇 E	Events	Exceptions Created exceptions
Memory	All types 💌 Filt	er in all text colur	nns 🔻 🔍		
	Start Time 🔺	Event Type	Description	Message	Thread
CPU Views	0:04.996 [Mar 1	Exception	java.lang.NoSuchMethodE	com.ejt.framew	AWT-EventQue
	0:04.996 [Mar 1	Exception	java.lang.NoSuchMethodE	com.ejt.framew	AWT-EventQue.
Threads	0:05.000 [Mar 1	Exception	java.lang.NoSuchMethodE	com.jprofiler.fr	AWT-EventQue.
	0:05.083 [Mar 1	Exception	java.lang.Exception		AWT-EventQue
	0:05.084 [Mar 1	Exception	java.lang.Exception		AWT-EventQue.
Monitors & Locks	0:05.084 [Mar 1	Exception	java.lang.Exception		AWT-EventQue.
	0:05.084 [Mar 1	Exception	java.lang.Exception		AWT-EventQue.
	0:05.085 [Mar 1	Exception	java.lang.Exception		AWT-EventQue.
Probes	0:05.085 [Mar 1	Exception	java.lang.Exception		AWT-EventQue.
	0:05.085 [Mar 1	Exception	java.lang.Exception		AWT-EventQue.
Sockets	0:05 085 [Mar 1 477 rows	Excention	iava lang Excention		∆W/T-EventOue
JOCKEIS	Stack trace:				
Files	java.lang.Throwable	<init>(java.lang.9</init>	String)		
Classes	java.lang.Exception.				
			on. <init>(java.lang.String)</init>		
Exceptions			init>(java.lang.String)		
Garbage Collector	java.lang.Class.getD	eclaredMethod(ja	wa.lang.String, java.lang.Class	[])	

The garbage collector view is special, because full profiling sessions can show the exact same information in profiling sessions with Java 17 or higher. When the garbage collector probe in the JVM probe category is recorded, JFR streaming is used to obtain the necessary data. See the chapter on garbage collector analysis [p. 118] for more information.

F Configuration In Detail

F.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 start up.
- 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 for the profiling VM parameter. With the VM parameter -agentpath:path to jprofilerti library>=port=25000, a port of 25000 would be used.
- Do you try to connect to an agent with a direct connection that only listens on the loopback interface? By default, the agent only listens on the loopback interface. You can configure JProfiler to set up an SSH tunnel to the remote machine. If you don't require encryption, you can also use the address=[IP address] option for the -agentpath parameter.
- 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.

Attach problems

When attaching to a running JVM, you sometimes may not see the JVM of interest in the list of all JVMs. To find the cause of this problem, it is important to understand how the attach

mechanism works. When a JVM is started, it writes a PID file into the the hsperfdata_\$USER directory in the temporary directory by which is it discovered. Only the same user or an admin user can then attach to the JVM. JProfiler can help you to connect to a JVM as an admin user.

On Windows, use the *Show Services* button to show all JVM service processes. JProfiler installs a helper service that will run with the system account that can connect to services running with system accounts as well as with a configured user account. The name of that service is "JProfiler helper" and is installed when you click on that button. You have to confirm the UAC prompt to allow the installation of the service. When JProfiler exits, the service is uninstalled again.

On Linux, you can use the user switcher in the attach dialog to attach with the root account. This user switcher is shown when profiling a local JVM as well as when attaching to a remote Linux or macOS machine. For the remote attach case, you can also switch to a different non-root user. If you have the root password, always switch to root rather than to the actual user that runs the service.

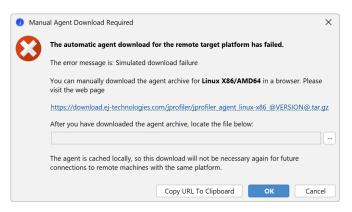
If a JVM is not visible on Linux even though you think it should be, the problem is usually connected with the temporary directory. One possibility is that the access rights for the $/tmp/hsperfdata_{USER}$ directory are wrong. In that case, delete the directory and restart the JVM. The process to be attached to must have write access to /tmp, otherwise attaching is not support.

If you use **systemd**, the process you are interested in may have PrivateTmp=yes set in its systemd service file. Then the pid file is written into a different location. JProfiler will handle this if you change to the root user with the user switcher in the attach dialog or if you use the CLI tools as root.

Automatic agent download for remote attach

For remote-attach, JProfiler requires the agent libraries for the remote target platform. If they are not available locally, it will try to download them. This operation may fail if there is a firewall that either blocks HTTPS connections to https://download.ej-technologies.com or inspects SSL connections with a man-in-the-middle scheme to decrypt traffic. In the latter case, the HTTPS connection will fail because JProfiler does not have the certificate of the firewall.

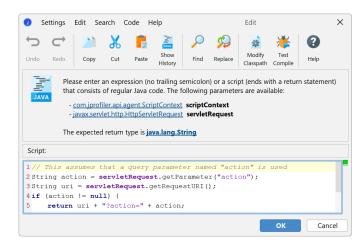
JProfiler offers a manual workaround when an agent download error occurs. A dialog is displayed that shows instructions for downloading the agent archive from the website manually and locating the downloaded archive before continuing with the remote attach operation.



Because agent files are cached, this is a one-time operation for each remote platform. When JProfiler is updated, the agent changes and the downloads will have to be repeated.

F.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.

🥑 General Settings	×
UI Session Defaults Snapshots IDE Integrations Updates External Programs	
Default Session	0
Edit Default Session Settings	
If you have modified the default session settings, you may wish to restore the initial settings.	
Reset Default Session Settings	
JREs For Launching Profiling Sessions	0
Default JRE: 21 [C:\Users\ingo\jdks\jbrsdk-21.0.5-windows-x64-b792.48] Configure	JREs
JDK For Code Editor	0
Currently Used JRE Configure Configure Configure	IDK-
	IDKS
ок	Cancel

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.

String uri = servletRequest. getRe	q 🐽 getAuthType()	String
if (action != null) {	getCharacterEncoding()	String
<pre>5 return uri + "?action=" + actio 6} else {</pre>	(getContentType()	String
7 return uri;	<pre>getContextPath()</pre>	String
3 }	🔲 getHeader(String arg0)	String
Э	🔟 getLocalAddr()	String
	<pre>getLocalName()</pre>	String
	<pre>getMethod()</pre>	String
	getParameter(String arg0)	String
	🔟 getPathInfo()	String

The auto-completion popup can suggest:

- 🔍 variables and script parameters. Script parameters are displayed in bold font.
- 😳 packages, when typing an import statement
- classes
- **1** 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.

🥥 Java Editor Settings 🛛 🗙 🗙
Editor Settings
Tab size: 4
Code Completion Popup Settings
Auto-import classes during code completion
Insert method stubs when completing constructors for abstract classes
Show suggestions as you type
Popup height: 10 🗢 entries
Code Formatting Settings
Use Eclipse formatting profile 😧
✓ Format block when entering a closing brace
Display Code Problems
○ None ○ Errors only 😮 O Errors and Warnings 😮
JDK For Code Editor
The runtime libraries of the configured JDK will be used for code completion and script compilation. You can configure a default JDK in the general settings.
O Default JDK
Override default JDK with [Select] Configure JDKs
ОК Сапсе

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/. jprofiler15/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.3 Custom Help

If you have an internal website that provides additional guidance for users, you can add an extra help button to the toolbar and the "Help" menu. To do that, add the following properties to the .vmoptions file:

```
-Dcustom.help.url=https://www.internal.website.com
-Dcustom.help.toolBarText=Internal#help
-Dcustom.help.actionName=Show internal help
```

All three properties have to be defined to make the action visible in the UI. The custom.help. toolBarText property is the text that is displayed in the toolbar. It should be concise and a second line can be added with a # separator as in the example above.

The location of the .vmoptions file is under <JProfiler installation directory>/bin/ jprofiler.vmoptions on Windows and Linux and /Applications/JProfiler.app/ Contents/vmoptions.txt on macOS. In addition, there are user-writable locations under %USERPROFILE%\.jprofiler15\jprofiler.vmoptions on Windows, \$HOME/.jprofiler15/ jprofiler.vmoptions on Linux and \$HOME/Library/Preferences/jprofiler.vmoptions on macOS.

F.4 Setting Profiling Settings At Startup

Before the profiling agent can start any recordings, the profiling settings have to be set. This happens when you connect with the JProfiler UI. Under some circumstances, it is required that the profiling agent knows the profiling settings at startup. The main use cases are:

Offline profiling

Triggers or the API are used to record data and save snapshots. The JProfiler GUI cannot connect in this mode. See the help topic on offline profiling [p. 129] for more information.

Profiling with jpcontroller on a headless machine

The command line utility jpcontroller [p. 251] can be used instead of the JProfiler GUI to record data and save snapshots interactively or with a non-interactive command file. However, jpcontroller has no facility for configuring profiling settings, so they have to be set in advance.

Remote attach to older OpenJ9 and IBM JVMs

Older OpenJ9 and IBM JVMs before 8u281, 11.0.11 and Java 17 do not have the ability to redefine classes without endangering the stability of the profiled process, so profiling settings have to be set at startup. The "Profiled JVM" step of the remote integration wizard in JProfiler asks you about the type of the JVM, and if you select *Older OpenJ9 and IBM JVMs* there, the wizard will add the options discussed below.

In general, setting profiling settings at startup is the most efficient mode of operation, because the least number of class redefinitions have to be performed. If the reduced convenience is not a problem, it can be used for any kind of profiling session.

Setting profiling settings at startup

If you use an integration wizard, select the *On a remote computer* option on the "Local or Remote" step and then the *Apply configuration at startup* option on the "Config synchronization" step. The wizard will then add the same options as discussed in the following paragraphs.

If you have added an -agentpath VM parameter to your start script to load the profiling agent, the profiling settings can be set by adding

```
,config=<path to config file>,id=<session ID>
```

to the -agentpath parameter. A complete parameter will look like this:

-agentpath:/path/to/libjprofilerti.so=port=8849,nowait,config=/path/to/config,id=123

If you use <code>jpenable</code> to load the profiling agent after the process has been started, you can choose *offline mode* in the interactive execution and specify config and ID there. Alternatively, pass the <code>--offline</code>, <code>--config</code> and <code>--id</code> arguments for non-interactive execution.

Preparing the config file

The referenced config file can be the config file of the JProfiler installation on the current machine, in which case the config parameter does not have to be specified at all. The JProfiler config file is located at \$HOME/.jprofiler15/jprofiler_config.xml or \$USERPROFILE%. jprofiler15jprofiler_config.xml and is the default for the config option of the -agentlib VM parameter. Often, automated profiling should be performed on a different machine and referencing the local JProfiler config file is not possible. You can then prepare a session with the profiling settings in the JProfiler UI on your local machine, export it via *Session->Export Session Settings* and transfer it to the machine where JProfiler is running on.

The session ID can be seen in the top-right corner of the "Application settings" tab of the session settings dialog (see the screenshot below). If the exported file only contains one session, the id parameter does not have to be specified.

Synchronizing the config file

After you have completed the initial setup, you may want to adjust profiling settings for future profiling runs. This requires that the config file is copied to the remote machine each time you make a modification.

Remote sessions in JProfiler have a "Config synchronization" feature that can automate this process for you.

Ø Session Settings	>
Application Settings	Session name: Application server on demo Id: 162 2
Profiled JVM	Attach to an already running HotSpot/OpenJ9 JVM and profile it
Code Editor	Attach type: Select from all local JVMs • Attach to remote JVM Kubernetes
Call Tree Recording	Launch a new JVM and profile it Launch type: Application Web Start
Call Tree Filters	Profiled JVM Settings
Trigger Settings	If you have not yet prepared a JVM for profiling, it is recommended to run an <u>integration wizard</u> . It will create the remote session for you.
Databases	SSH tunnel connection Direct SSH to demo:8849 Edit Use SOCKS proxy
HTTP, RPC & JEE	Execute start command
JVM & Custom Probes	Execute stop command Open browser with URL
Advanced Settings	Connection timeout: 60 💠 seconds Config Synchronization Options
	Java File Path Note: the classpath is used for the bytecode viewer only. Class path Source path ?
General Settings Copy Sett	ings From OK Cancel

If the session is started via SSH, you can copy the config file via SSH directly to the remote machine. Otherwise, you can still copy the config file to a local directory which may be mounted on the remote machine. Finally, you can execute an arbitrary command, to copy the config file by other means.

Onfig Synchronization Options						
The synchronization action configured below is performed each time the profiling settings are changed.						
Please choose the desired synchronization action:						
O Manual synchronization	0					
Ocopy with SSH to remote directory:	0	/home/build/cor	nfig			
O Copy config file to directory:	8					
O Execute command:	0					
		_				
			ОК	Cancel		

G Command Line Reference

G.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 the required input on the command
 line.
* with the following arguments, you can partially or completely supply the
 entire 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=<nnnn> The port on which the profiling agent should listen for
                     a connection from the JProfiler GUI
 -a --address=<IP> The address the profiling agent should listen on. Without
                    this parameter, attaching is only possible from localhost.
               Use 0.0.0.0 to listen on all addresses
 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. 206] without loading the profiling agent into the VM:

Usage: jpdump [options] jpdump dumps the heap from a locally running JVM to a file. Hotspot VMs produce HPROF files, OpenJ9 VMs PHD files. HPROF and PHD files 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 the
 entire 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 be 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.

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 (separated by spaces or tabs). 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.

Note that the profiling settings have to be set in the profiling agent. This happens when you connect with the JProfiler UI. If you never connect with the JProfiler UI, they have to set them manually in the startup command or with jpenable, please see the help topic on setting profiling settings at startup [p. 248] for more information.

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: -nnon-interactive Run an automated session where a list of commands is read from stdin. -fcommand-file= <path> Read commands from a file instead of stdin. Only applicable together withnon-interactive.</path>
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 startProbeRecording builtin.JdbcProbe true true sleep 10 stopCPURecording stopProbeRecording builtin.JdbcProbe saveSnapshot /path/to/snapshot.jps
The sleep <seconds> command pauses for the specified number of seconds. Empty lines and lines starting with # are ignored.</seconds>

⁽¹⁾ https://www.ej-technologies.com/resources/jprofiler/help/api/javadoc/com/jprofiler/api/agent/mbean/ RemoteControllerMBean.html

G.2 Command Line Executables For Working With Snapshots

When using offline profiling [p. 129] 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, .jfr
      "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 ','.
    -bitmap=false|true
        Where appropriate, export a bitmap image instead of SVG for the main
        content. The default value is false.
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.
    -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.
        Appending .** to the package name will select packages recursively.
        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
    - \texttt{aggregation} = \texttt{method} \, | \, \texttt{class} \, | \, \texttt{package} \, | \, \texttt{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
    -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
* OutlierDetection
  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
   -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
 like ProbeCallTree, but with additional or changed options:
   -format=html|csv|xml
    -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.
```

Comparing snapshots

The executable bin/jpcompare compares different snapshots [p. 134] 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, .jfr
      "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 ','.
    -bitmap=false|true
        Where appropriate, export a bitmap image instead of SVG for the main
        content. The default value is false.
    -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.
    -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.
    -objects=recorded|heapwalker|all
        Compare recorded objects, objects in the heap walker, or object counts
        from all objects dumps. The default is recorded objects for .jps files
        and heapwalker for HPROF/PHD files.
    -dumpselection=first|last|label
        The all objects dump that is used for calculating the comparison.
        Default is the last value.
    -label
        If dumpselection is set to 'label', the name of the label for which
        the comparison should be calculated.
```

```
* AllocationHotSpots
```

```
Options:
   -format=html|csv
   -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.
   -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".
   -classselection
       Calculate the comparison for a specific class or package. Packages are
       specified by appending '.*' for a single package or '.**' for
       recursive packages. Specify a package with a wildcard, like
        'java.awt.*'.
* AllocationTree
 Options:
   -format=html |xml
   -viewfilters=<comma-separated list>
   -viewfiltermode=startswith|endswith|contains|equals
   -viewfilteroptions=casesensitive
   -aggregation=method|class|package|component
   -liveness=live|gc|all
   -classselection
* 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. 80]. 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, .jfr
       [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.

G.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. 254] 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 pin/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	
nowait	Whether profiling should start immediately or whether the profiled JVM should wait for a connection from the JProfiler GUI.	and nowait cannot both be true.	
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	

Attribute	Description	Required
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 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.	No
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) {
    mainClass = '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. 254]. It supports the following attributes:

Attribute	Description	Required
snapshotFile	The path to the snapshot file. This must be a file with a $.\rm jps$ 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. 254]. 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. 254], 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

Attribute	Description	Required
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
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. 254].

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. 254].

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')
}
```

G.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 thelib 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. 263]. 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. 263]. 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">
<target name="export">
<target name="export">
<target name="export">
<turew name="CallTree" file="calltree.html"/>
<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,

⁽¹⁾ http://ant.apache.org

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.

```
<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>
```