



MAKE THE FUTURE JAVA

Java Mission Control 6.0 Tutorial

Marcus Hirt

Consulting Member of Technical Staff

ORACLE®

Index

This document describes a series of hands on exercises designed to familiarize you with some of the key concepts in Java Mission Control. The material covers several hours' worth of exercises, so some of the exercises have been marked as bonus exercises. The bonus exercises can be skipped in the interest of seeing as many different parts of Mission Control as possible. You can always go back and attempt them when you have completed the standard exercises.

Index	1
Introduction.....	4
Installing Mission Control	5
Installing JMC in Eclipse (Optional)	5
Starting Java Mission Control.....	8
Exercise 1.a – Starting the Stand-Alone Version of JMC	8
Exercise 1.b – Starting JMC in Eclipse	10
The Java Flight Recorder	13
Exercise 2.a – Starting a JFR Recording	13
Exercise 2.b – Hot Methods.....	20
Exercise 3 – Latencies	25
Exercise 4 (Bonus) – Garbage Collection Behavior	29
Exercise 5 (Bonus) – WebLogic Server Integration	31
Exercise 6 (Bonus) – JavaFX.....	36
Exercise 7 (Bonus) – Exceptions	38
Exercise 8 – Custom Events in JDK 9 (Bonus)	43
Exercise 9 – Custom Rules (Bonus)	45
Running Rules.....	45
Creating Rules.....	46
Exporting the rule	54
Exercise 9b – Custom Pages	56
Filters	56
Grouping	59
Boolean Filter Operations	63
The Management Console (Bonus)	67
Exercise 10.a – The Overview	67
Exercise 10.b – The MBean Browser	70
Exercise 10.c – The Threads View	73
Exercise 10.d (Bonus) – Triggers	75
Heap Waste Analysis (Bonus)	77
Object Selection	77
Referrer Tree-table.....	78
Class Histogram	78
Ancestor referrer	79
Exercise 11a – Reducing Memory Usage.....	79
JCMD (Java CoMmanD) (Bonus)	80
More Resources	81

Introduction

Oracle Java Mission Control is a suite of tools for monitoring, profiling and diagnosing applications running in production on the HotSpot JVM. There is also a sibling product named JRockit Mission Control for the JRockit JVM.

Java Mission Control mainly consists of two tools at the time of writing:

- The JMX Console – a JVM and application monitoring solution based on JMX.
- The Java Flight Recorder – a very low overhead profiling and diagnostics tool.

There are also plug-ins available that extend the functionality of Java Mission Control to, for example, perform heap waste analysis on heap dumps.

This tutorial will focus on the Java Flight Recorder part of Java Mission Control, with bonus exercises for the heap dump analysis tool (JOverflow) and the JMX Console towards the end.

Java Mission Control can be run both as a stand-alone application and inside of Eclipse. This tutorial can be used with either way of running Mission Control.

In this document, paths and command prompt commands will be displayed using a bold fixed font. For example:

C:\Tutorial

Graphics user interface strings will be shown as a non-serif font, and menu alternatives will be shown using | as a delimiter to separate sub-menus. For example:

File | Open File...

Installing Mission Control

If you have downloaded the full tutorial for windows, and unpacked it to **C:\Tutorial**, you already have everything that is required for this tutorial and can skip this section. If you are on a different platform, you will need to install Mission Control, and optionally Eclipse Oxygen.

The easiest way of installing Java Mission Control is to download and install the latest Oracle Java SE JDK (Java Development Kit). You will need the latest Oracle JDK even if you want to run this Tutorial from within Eclipse.

Here is how to get the latest JDK:

1. Go to <http://java.oracle.com>.
2. Click on **Java SE** under **Top Downloads**.
3. Download the latest Java SE JDK for your platform. At the time of the writing of this tutorial, the latest Oracle JDK is 9.
4. Also download the latest Oracle JDK 8.

With the JDK, you now have everything you need to do this Tutorial. That said, the collateral for this tutorial is provided as a set of Eclipse projects. It is not necessary to access them through Eclipse, but it may make playing around with the examples a bit easier.

***Note:** For some versions of JMC the installation of experimental plug-ins will work better if you install the JDK somewhere where you have write permission, e.g. not under Program Files on Windows.*

Installing JMC in Eclipse (Optional)

This section describes how to prepare for running this tutorial from within Eclipse. This optional part is a bit more demanding than just running the stand-alone version of JMC, but will on the other hand make it easier to experiment with the material later on.

Here is how to get the latest Eclipse:

1. Go to <http://eclipse.org>.
2. Click on **Download**.
3. Download the **Eclipse IDE for Eclipse Committers** for your platform.
4. To install simply unpack the Eclipse zip where you want it.
5. Run Eclipse by running the executable in the root of the unpacked zip.

At the time of writing this tutorial, Eclipse Oxygen (4.7.0) was available. Eclipse Oxygen does not run well on JDK 9. Next step will be to make sure that Eclipse uses your JDK 8.

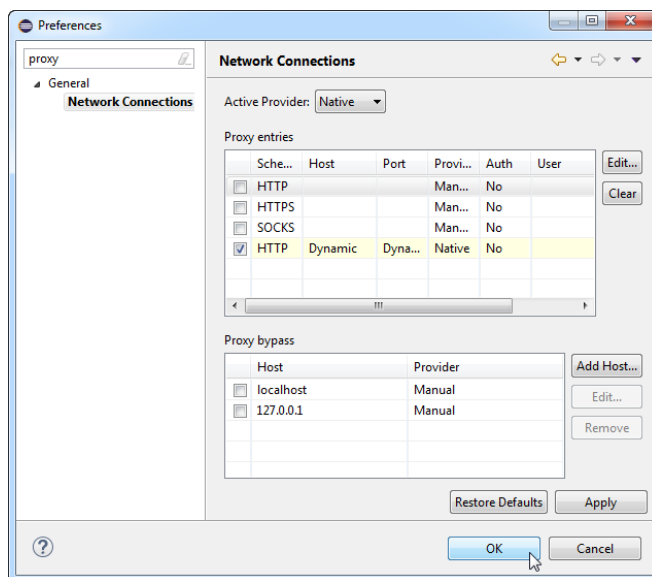
1. Find the `eclipse.ini` file.
 - a. On Windows it will be next to your eclipse launcher.
 - b. On Mac OS X you will need to show the files for the Eclipse.app (**Show Package Contents**). The file is under `Contents/Eclipse`.
2. Edit the `eclipse.ini` file to add your JDK 8 as the JDK to use for launching Eclipse. Add a `-vm` argument before the `-vmargs` argument. Remember that the arguments must be on their own line. For example:

```
-vm
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents
/Home/bin/java
-vmargs
-Xms512m
-Xmx2048m
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200
-XX:+UnlockCommercialFeatures
-XX:+FlightRecorder
...
```

3. Remember to restart Eclipse after updating your `eclipse.ini` file.

Next you need to install the JMC Eclipse plug-ins. First you may need to set up the proxy settings in Eclipse (if you have direct access to the internet, you can skip this step):

1. Start Eclipse, if not already running.
2. Go to the preferences dialog (**Window | Preferences** on Windows, **Eclipse | Preferences** on Mac)
3. Type proxy in the filter box to quickly find the settings.



4. Change the settings to match what you need for your current network.

Next install the Java Mission Control plug-ins:

1. Go to <https://www.oracle.com/missioncontrol>.
2. Click on **Eclipse Update Site**.
3. Click on **Use Update Site** under **Download and Install**.
4. Follow the instructions.

To access the extension and/or experimental plug-ins (such as JOverflow), follow nearly the same procedure as when installing the core plug-ins:

1. Go to <https://www.oracle.com/missioncontrol>.
2. Click on **Eclipse JMC Extension Plug-ins** (the second last bullet under the **Overview** section) and/or **Eclipse JMC Experimental Plug-ins**.
3. Click on **Use Update Site** under **Download and Install**.
4. Follow the instructions. Remember to install the WebLogic Tab Pack, JOverflow, and the Java FX plug-in if you want to do the corresponding parts of the tutorial.

Finally, you need to import the projects from the unpacked tutorial zip:

1. From within Eclipse, select **File | Import...**
2. In the Import dialog, select **General/Existing Projects into Workspace**.
3. Click the **Browse** button to select the root folder, and browse to the root folder for the unpacked tutorial zip.
4. Select all projects and hit **Finish**.

You should now be all set for running this tutorial from within Eclipse.

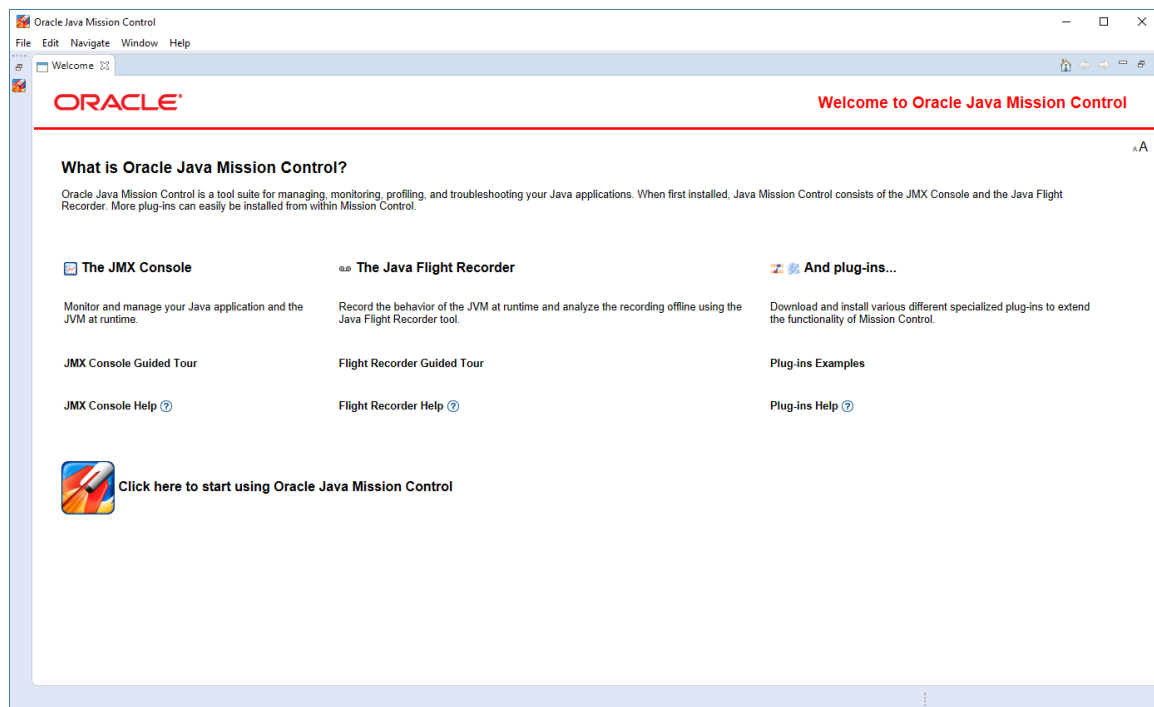
Starting Java Mission Control

There are two separate ways of running Java Mission Control available: as a stand-alone application or from within Eclipse.

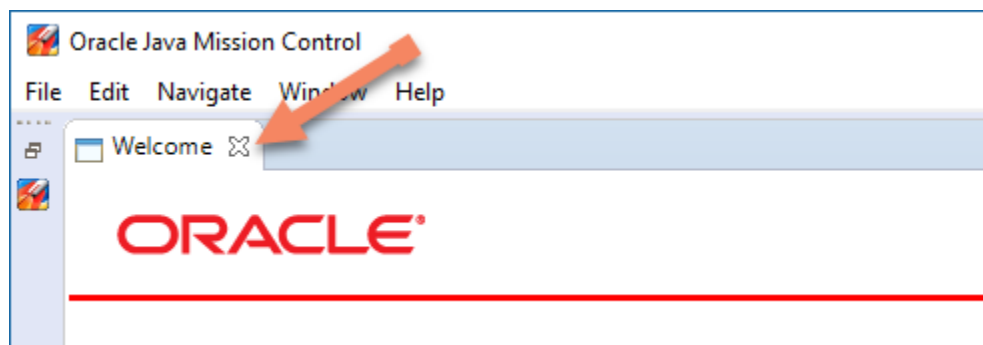
This exercise familiarizes you with the layout of the exercise collateral on disk, and shows you how to start both the stand alone and the Eclipse plug-in versions of Java Mission Control.

Exercise 1.a – Starting the Stand-Alone Version of JMC

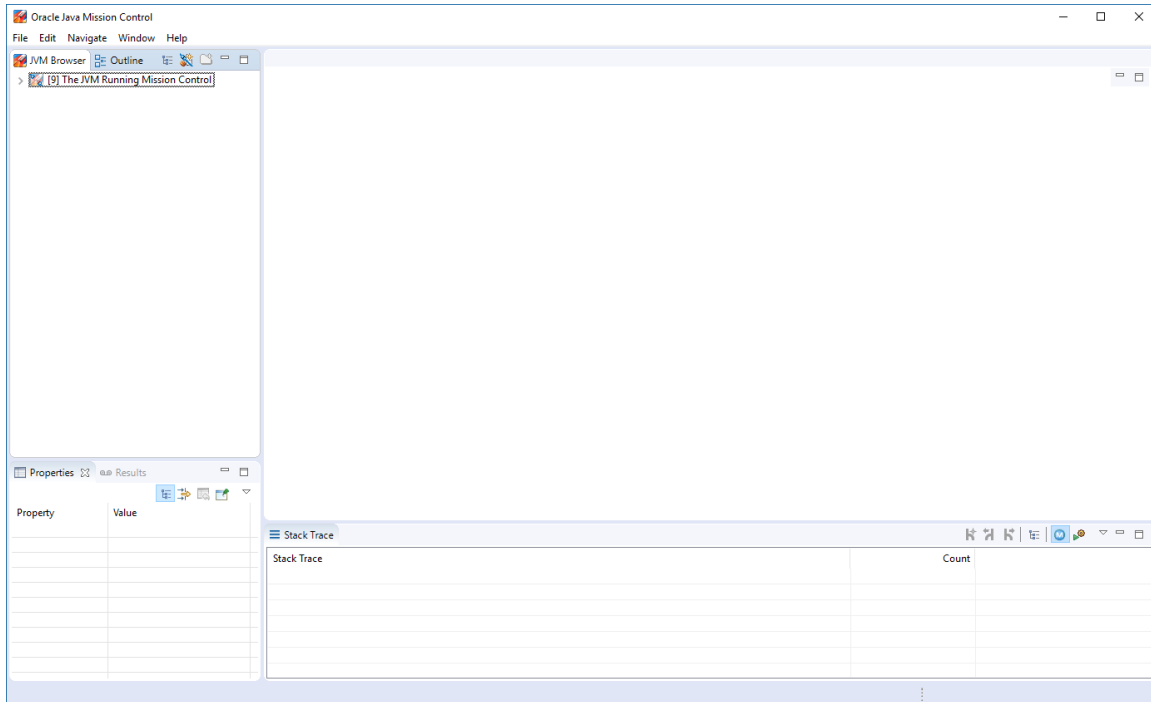
Go to the %JDK9_HOME%\bin directory (C:\Tutorial\jdk9_181\bin, if using the full Windows tutorial) and double click the `jmc` executable. A splash screen should show, and after a little while you should be looking at something like this:



The welcome screen provides guidance and documentation for the different tools in Java Mission Control. Since you have this tutorial, you can safely close the welcome screen.



Closing the welcome screen will show the basic Java Mission Control environment. The view (window) to the left is the **JVM Browser**. It will normally contain the automatically discovered JVMs, such as locally running JVMs and JVMs discovered on the network.



To the bottom left is the Properties view, showing properties for anything selected in the editor.

Behind the Properties view is the Results view, which shows the results from the automated analysis relevant to the currently opened page in the editor.

Below the editor area is the Stack Trace view, which shows the aggregates stack traces for anything selected in the editor.

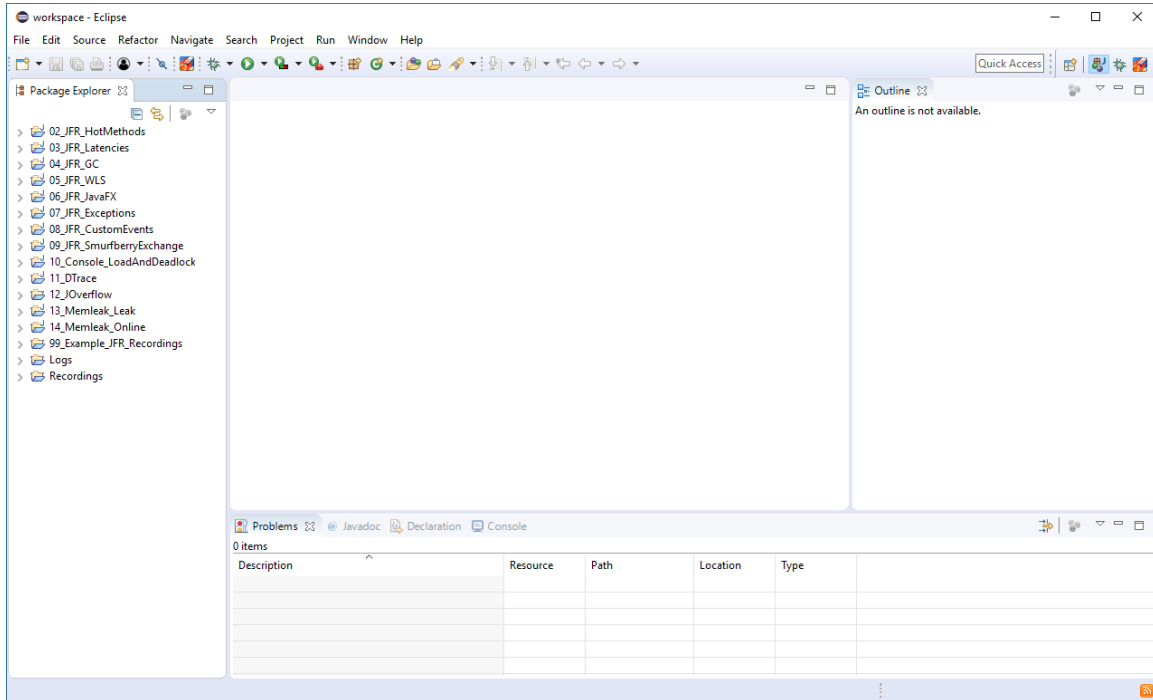
To launch the different tools, simply select a JVM in the JVM Browser and then select the appropriate tool from the context menu. For example, the Management Console can be started on a JVM by selecting the JVM in question in the JVM Browser and selecting Start JMX Console from the context menu.

In the JVM Browser, the JVM running Mission Control will be shown as **The JVM Running Mission Control**.

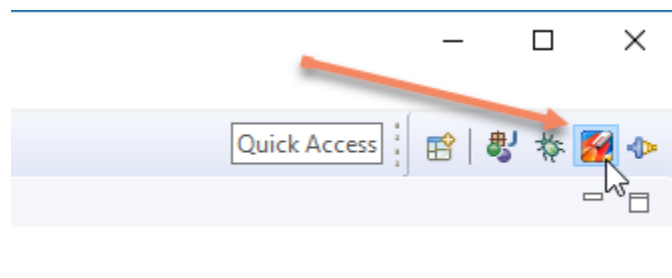
If you will be running the tutorial from within Eclipse, please **shut down** the stand-alone version of Java Mission Control.

Exercise 1.b – Starting JMC in Eclipse

First start Eclipse (run the `C:\Tutorial\startEclipse.bat` script, if using the full Windows tutorial). You should now be presented with the Java perspective, looking somewhat like below:

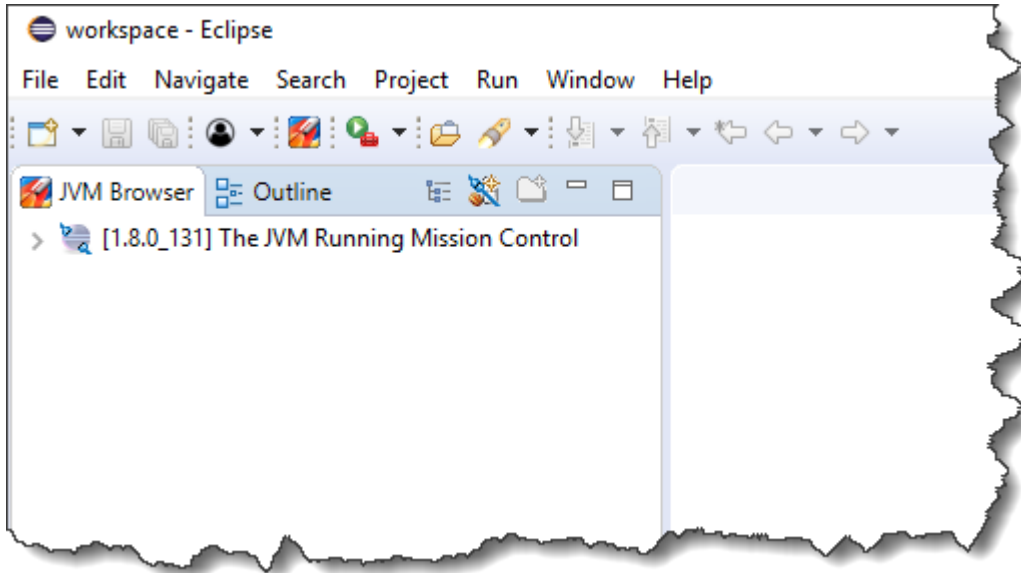


To the left the projects for the different exercises can be seen. In Eclipse a configuration of views is called a perspective. There is a special perspective optimized for working with JMC, called the Java Mission Control perspective. To open the Java Mission Control Perspective, click the Mission Control perspective in the upper right corner of Eclipse.

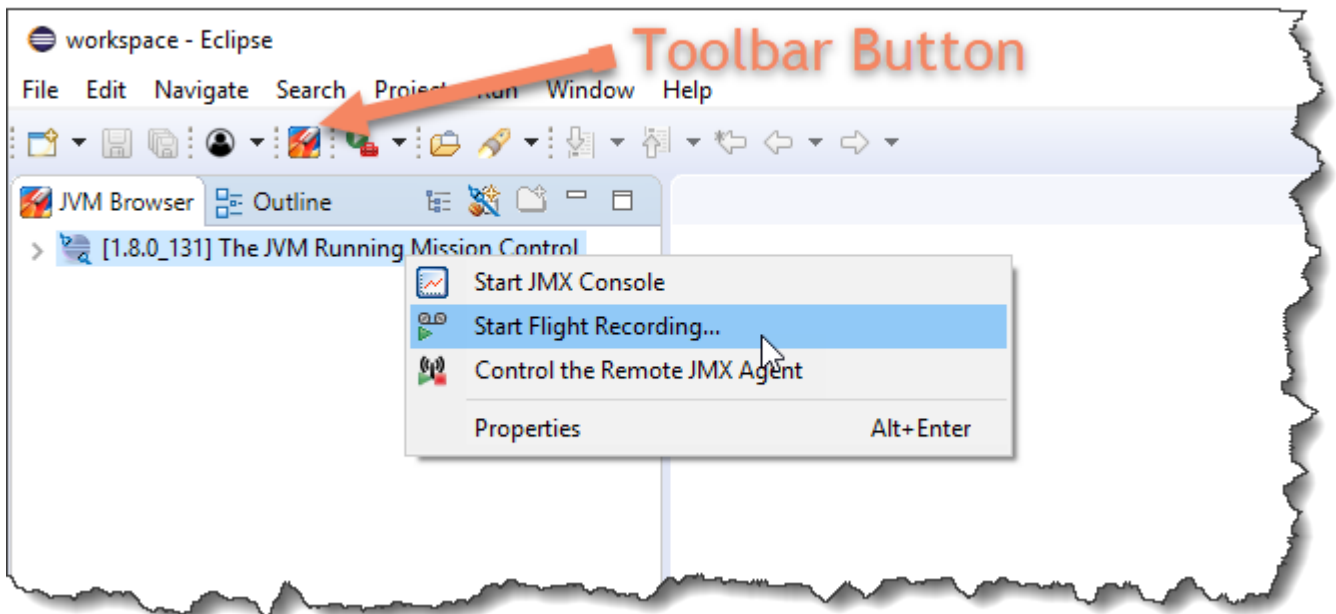



In this tutorial, you will be constantly switching between the Java perspective, to look at code and to open flight recordings, and the Mission Control perspective, to access the JVM browser and optimize the window layout for looking at flight recordings.

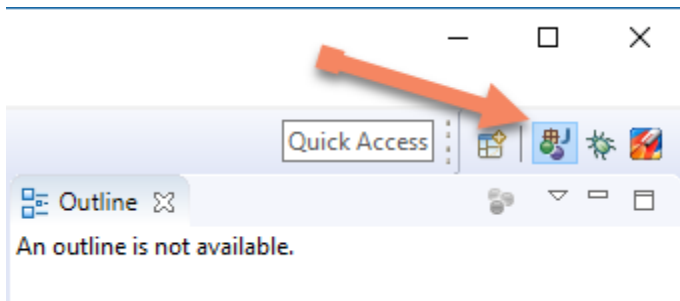
As can be seen from the picture below, the JVM running Eclipse (and thus Java Mission Control) will be named **The JVM Running Mission Control**, just like in the stand-alone version of Java Mission Control.



Launching the tools work exactly the same as in the stand-alone version. Either use the context menu of the JVM that you wish to launch the tool on, or click the Mission Control button on the toolbar to launch a Wizard.



The Java perspective is the perspective with a little J on the icon ( Java):




Go back to the Java perspective, so that you can see the projects in the **Package Explorer view** again.

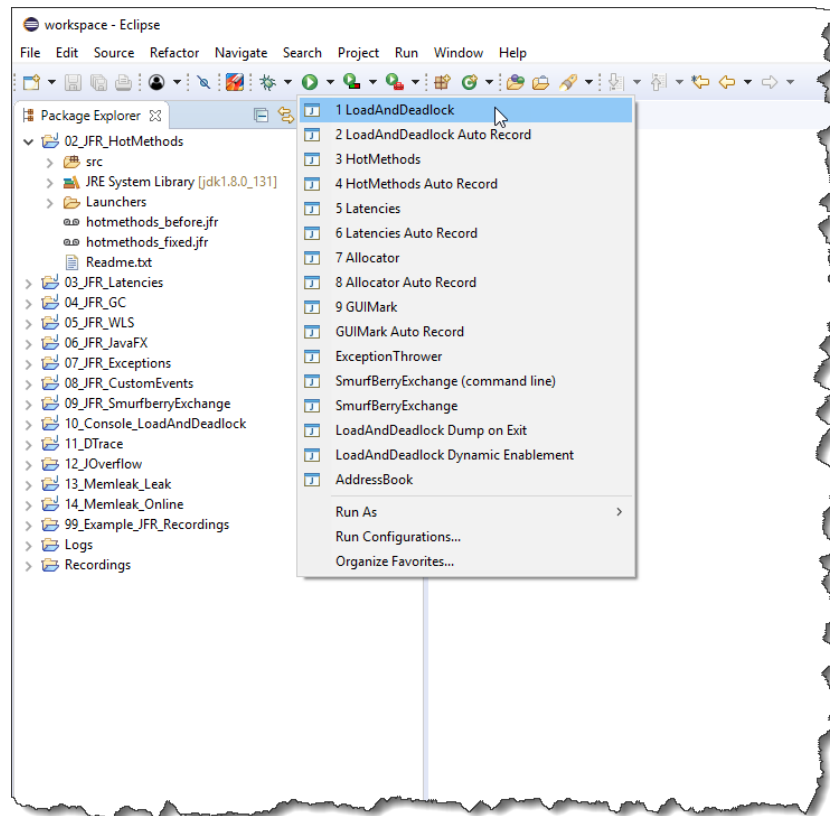
The Java Flight Recorder

The Java Flight Recorder (JFR) is the main profiling and diagnostics tool in Java Mission Control. Think of it as analogous to the “black box” used in aircraft (FDR, or Flight Data Recorder), but for the JVM. The recorder part is built into the HotSpot JVM and gathers data about both the HotSpot runtime and the application running in the HotSpot JVM. The recorder can both be run in a continuous fashion, like the “black box” of an airplane, as well as for a predefined period of time. For more information about recordings and ways of creating them, see <http://hirt.se/blog/?p=370>.

Exercise 2.a – Starting a JFR Recording

There are various ways to start a flight recording. For this exercise, we will use the Flight Recording Wizard built into Java Mission Control.

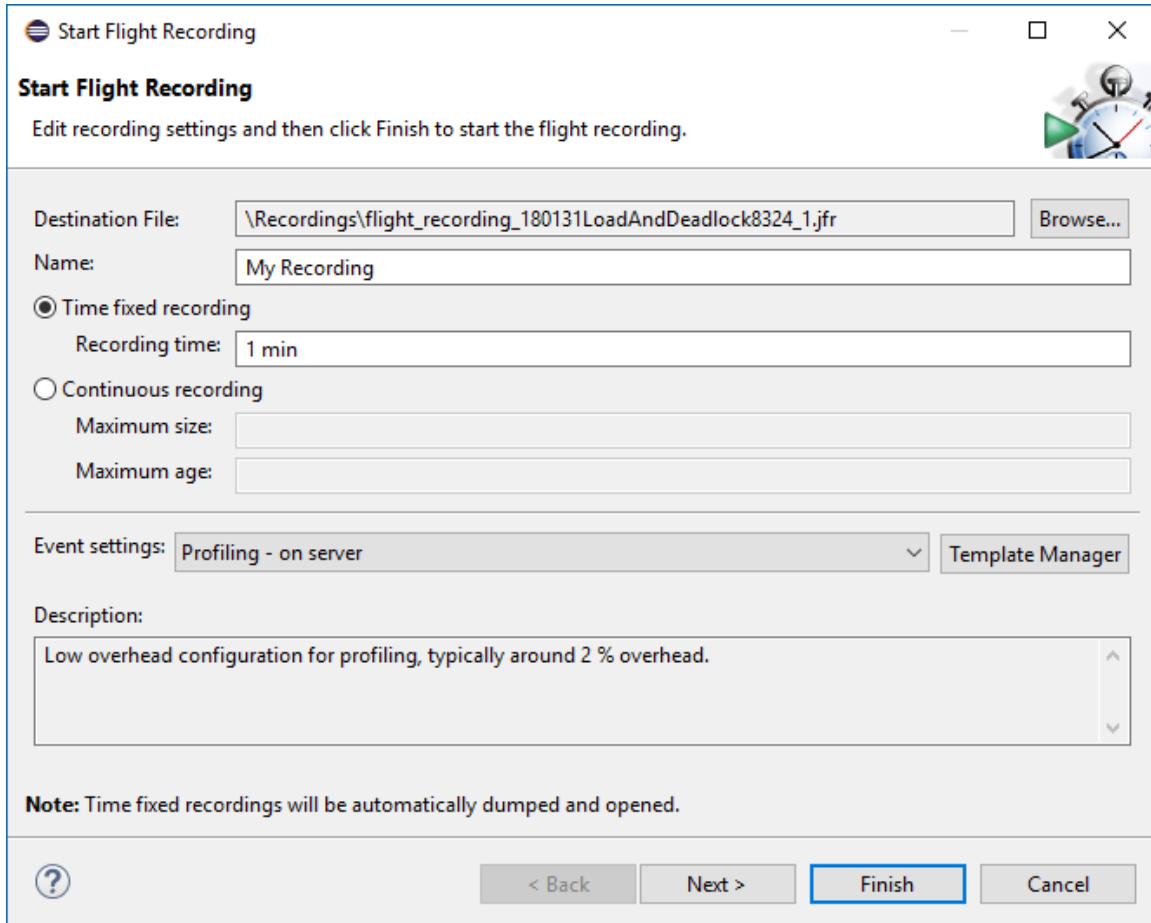
First switch to the **Java** perspective. Note that there is an empty project named **Recordings**. Next start the **LoadAndDeadlock** program by selecting it from the drop-down menu next to the run icon () as show below.



***Note:** There are “Auto Record” versions of most launchers, which will launch the application and automatically create a recording in the C:\Tutorial root folder. This is since, in certain environments (slow machines running a virtualized Windows for example), the highly loaded JVM that JMC is trying to communicate with will take so long to getting around to communicate with JMC that it simply is no fun to wait. If that is the case, simply run the “Auto” launchers. Or if you’re lazy. I will not judge. Just try doing this first exercise without Auto, as it is about doing recordings from JMC. If running the tutorial on anything other than the full Windows tutorial located in C:\Tutorial, the paths in the launchers will need to be updated.*

Switch to the **Mission Control** perspective and select the newly discovered JVM running the LoadAndDeadlock class in the **JVM Browser**. Select **Start Flight Recording...** from the context menu. The Flight Recording Wizard will open. Click **Browse** to find a suitable location (e.g. the **Recordings** project) and filename for storing the recording. Don’t forget to name the recording so that it can be recognized by others connecting to the JVM, and so that the purpose of the recording can be better remembered. The name will be used when listing the ongoing recordings for a JVM, and will also be recorded into the recording itself.

Next select the template you want to use when recording. The template describes the configuration to use for the different event types. Select the **Profiling – on Server** template, and hit **Finish** to start the recording.



The image shows a Windows-style dialog box titled "Start Flight Recording". It has a standard title bar with minimize, maximize, and close buttons. Below the title bar, the text "Start Flight Recording" is followed by a subtitle "Edit recording settings and then click Finish to start the flight recording." and a small icon of a clock and a play button. The main area contains several input fields and options: "Destination File:" with a text box containing "\Recordings\flight_recording_180131LoadAndDeadlock8324_1.jfr" and a "Browse..." button; "Name:" with a text box containing "My Recording"; two radio buttons for "Time fixed recording" (selected) and "Continuous recording"; a "Recording time:" text box with "1 min" for the fixed recording; "Maximum size:" and "Maximum age:" text boxes for continuous recording; an "Event settings:" dropdown menu showing "Profiling - on server" and a "Template Manager" button; and a "Description:" text box with the text "Low overhead configuration for profiling, typically around 2 % overhead." At the bottom, there is a "Note:" stating "Time fixed recordings will be automatically dumped and opened." and a row of buttons: a help button (question mark in a circle), "< Back", "Next >", "Finish" (highlighted with a blue border), and "Cancel".

Start Flight Recording

Edit recording settings and then click Finish to start the flight recording.

Destination File: \Recordings\flight_recording_180131LoadAndDeadlock8324_1.jfr Browse...

Name: My Recording

☒ Time fixed recording

Recording time: 1 min

☐ Continuous recording

Maximum size:

Maximum age:

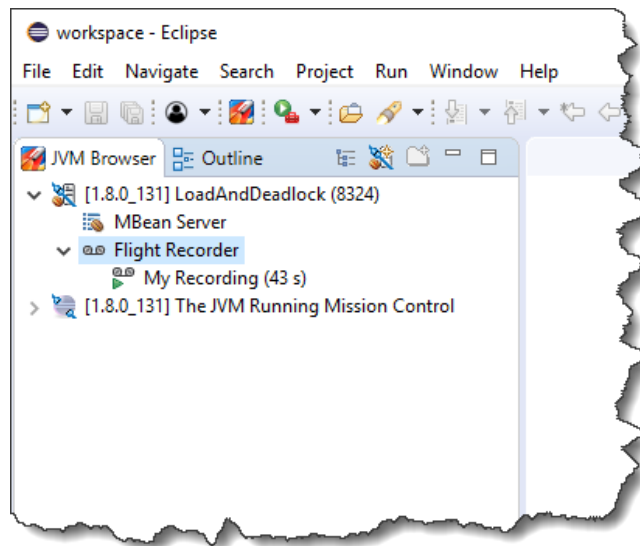
Event settings: Profiling - on server Template Manager

Description: Low overhead configuration for profiling, typically around 2 % overhead.

Note: Time fixed recordings will be automatically dumped and opened.

? < Back Next > Finish Cancel

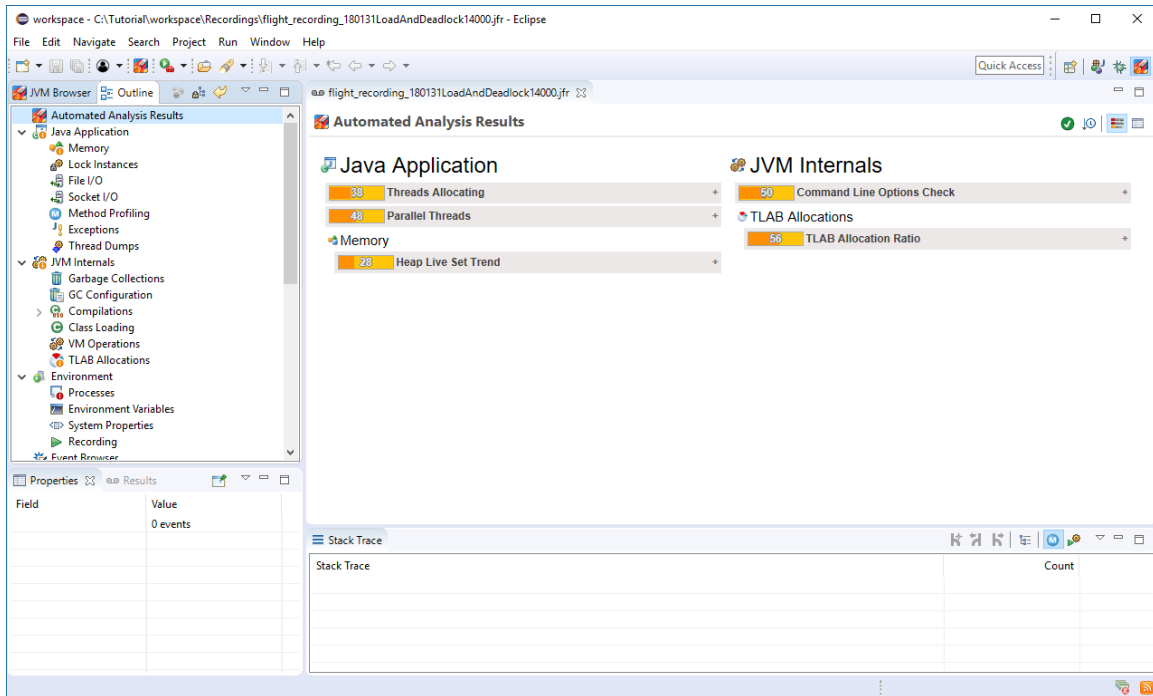
The progress of the recording can be seen in the JVM Browser, when the Flight Recorder node is expanded. It can also be seen in the status bar.



Use the minute to contemplate intriguing suggestions for how to improve Mission Control (don't forget to e-mail them to marcus.hirt@oracle.com), get a coffee, or read ahead in the tutorial.

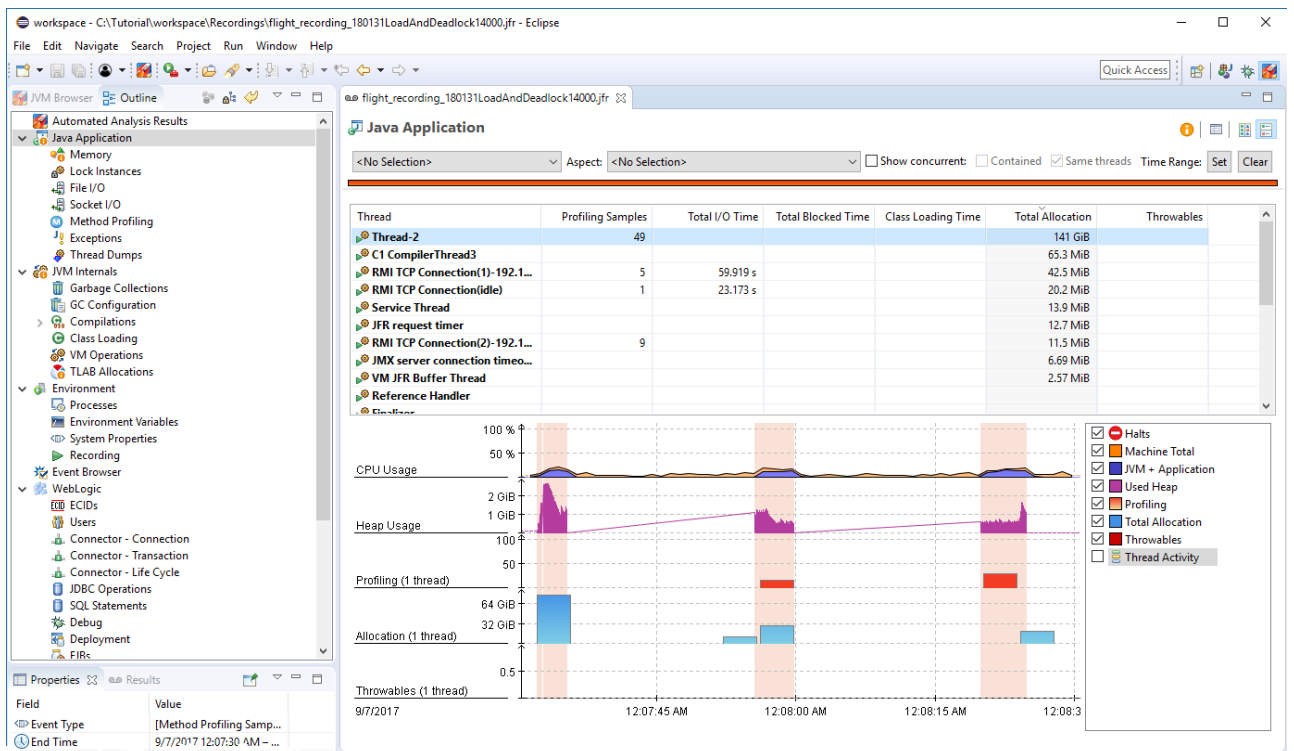
Once the recording is done, it will be downloaded to your Mission Control client, and opened. Switch to the Java Mission Control perspective.

You should be looking at the automated analysis of the recording.



This exercise is just to familiarize you with one of the ways to create a flight recording. This will be a rather boring recording, in terms of results from the automated analysis, so don't mind the results of this analysis.

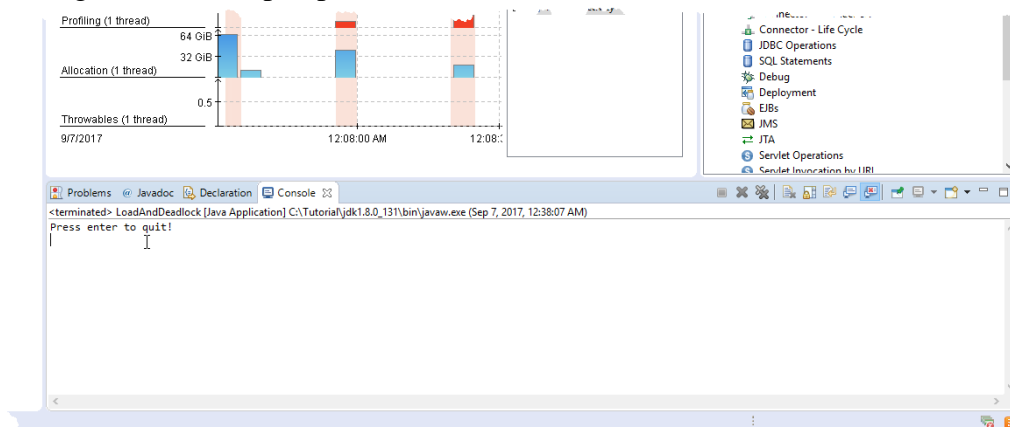
The **Outline** view shows the various *pages* in the Java Flight Recorder user interface. Select **Java Application**.



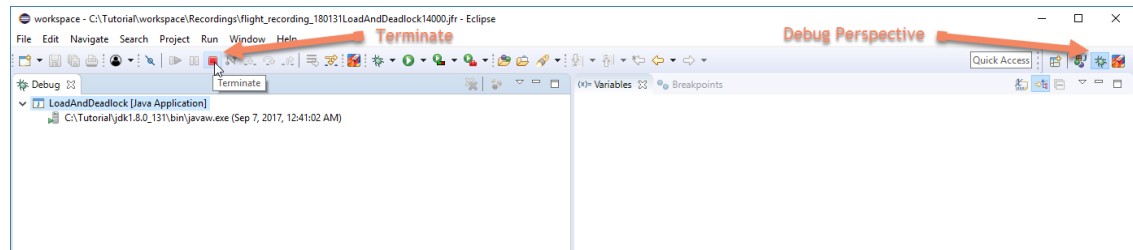
Here we get an overview of commonly interesting properties of the recording. We can, for example, see that the application does bursts of allocations, in a cycle. We can also see that it was mostly one thread being responsible for the allocations.

This exercise was mostly to describe how to make a recording, and basic navigation in the user interface. Once you are done with the recording, remember to shut down the LoadAndDeadlock application.

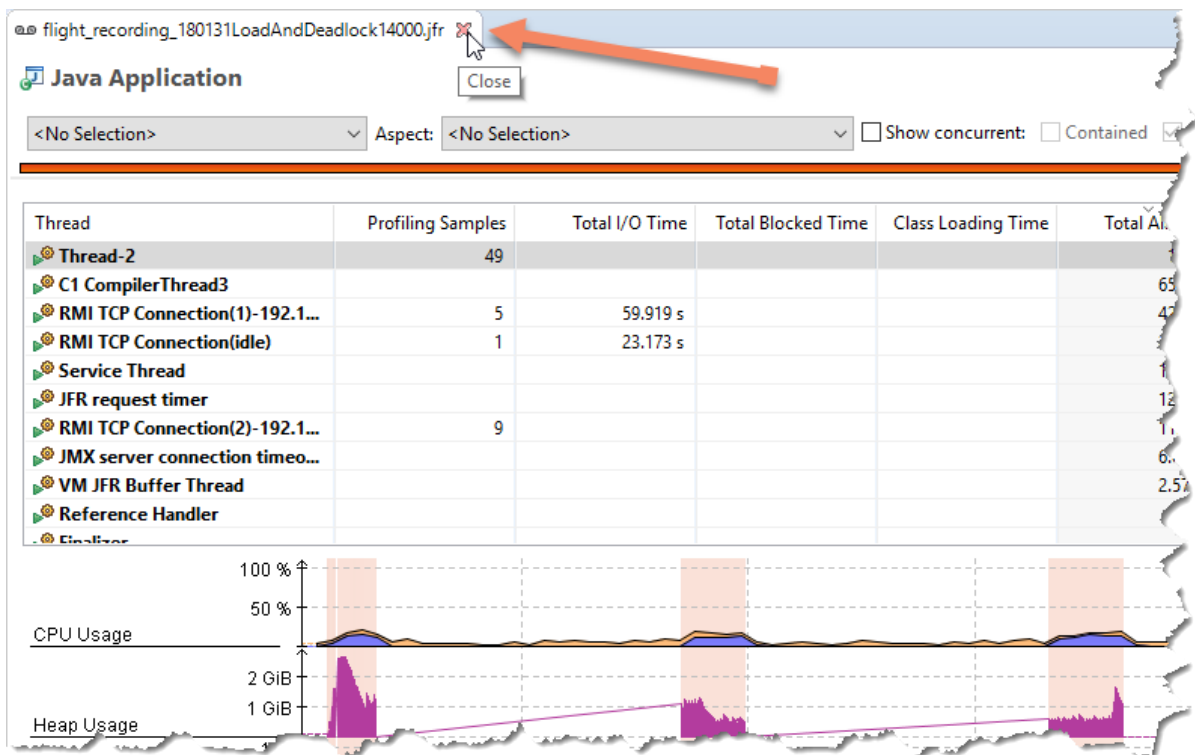
- Either go to the **Java** perspective and hit enter in the **Console** view,



- ... or go the **Debug** perspective, find the **LoadAndDeadlock** process and click the **Terminate** button



***Note:** Also, remember to close the recording editor window when you are done with a recording. Recordings contain a lot of information, and can consequently use a lot of memory.*

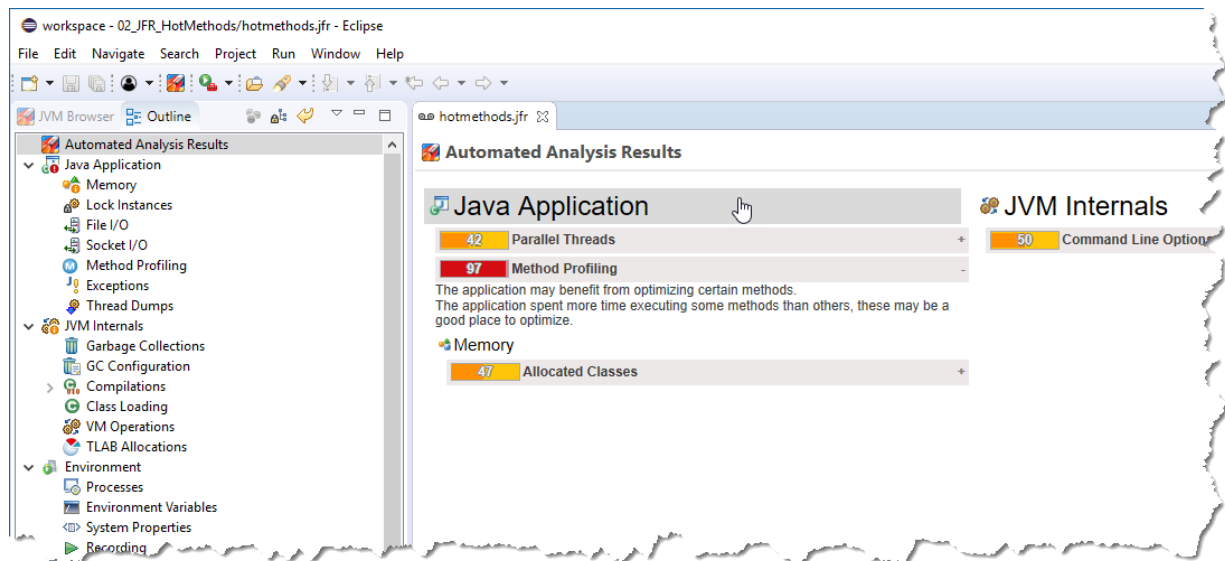


Exercise 2.b – Hot Methods

One class of profiling problems deals with finding out where the application is spending the most time executing. Such a “hot spot” is usually a very good place to start optimizing your application, as any effort bringing down the computational overhead in such a method will affect the overall execution of the application a lot.

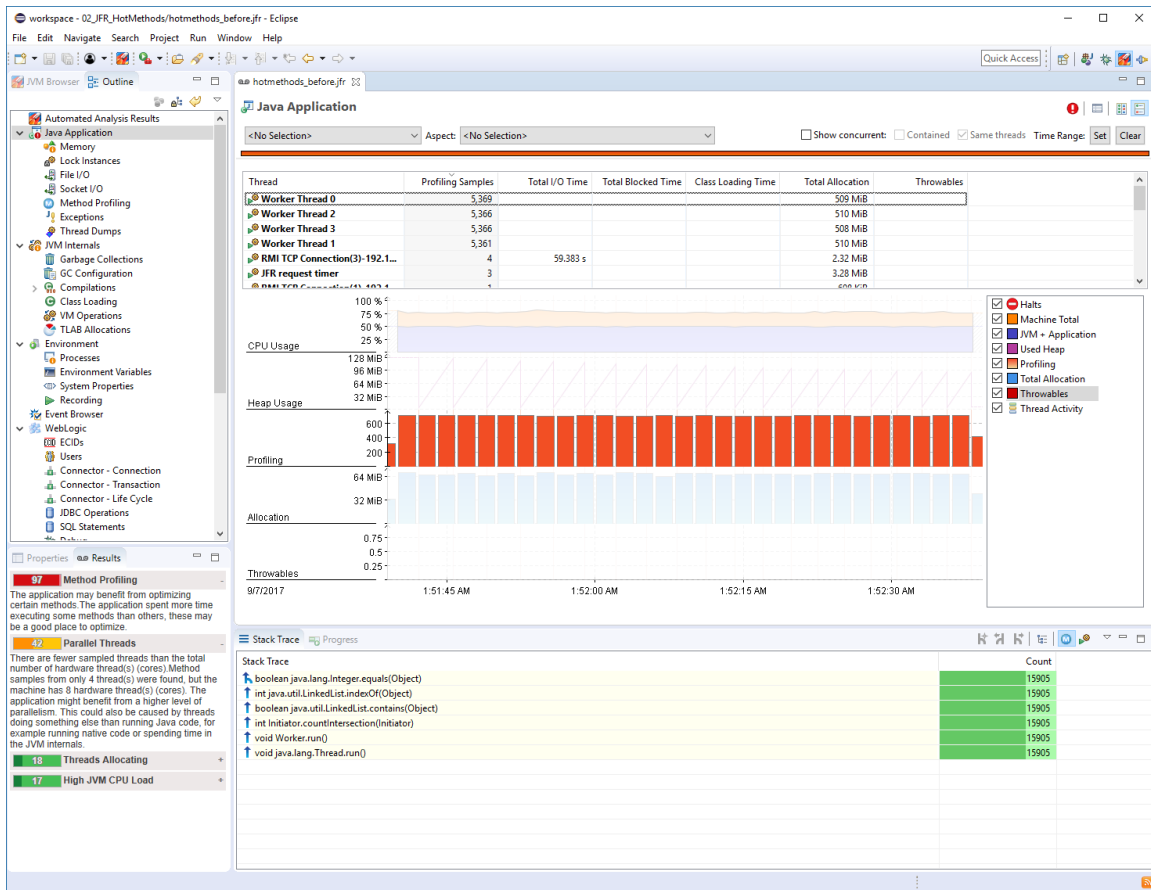
Open the `hotmethods_before.jfr` recording.

Switch to the **Java** perspective and open (double click) the recording in the `02_JFR_HotMethods` project named `hotmethods_before.jfr`.



Switch back to the **Java Mission Control** perspective once the recording is open. The automated analysis indicates that there is great value in optimizing certain methods. Do not worry about the non-descript textual information – in JMC 6.1.0 the prime candidate methods are listed, plus the first few frames of the stack trace aggregate.

Since there is apparently interesting information in the Java Application tab, click on the **Java Application** header in the **Automated Analysis Results** page.



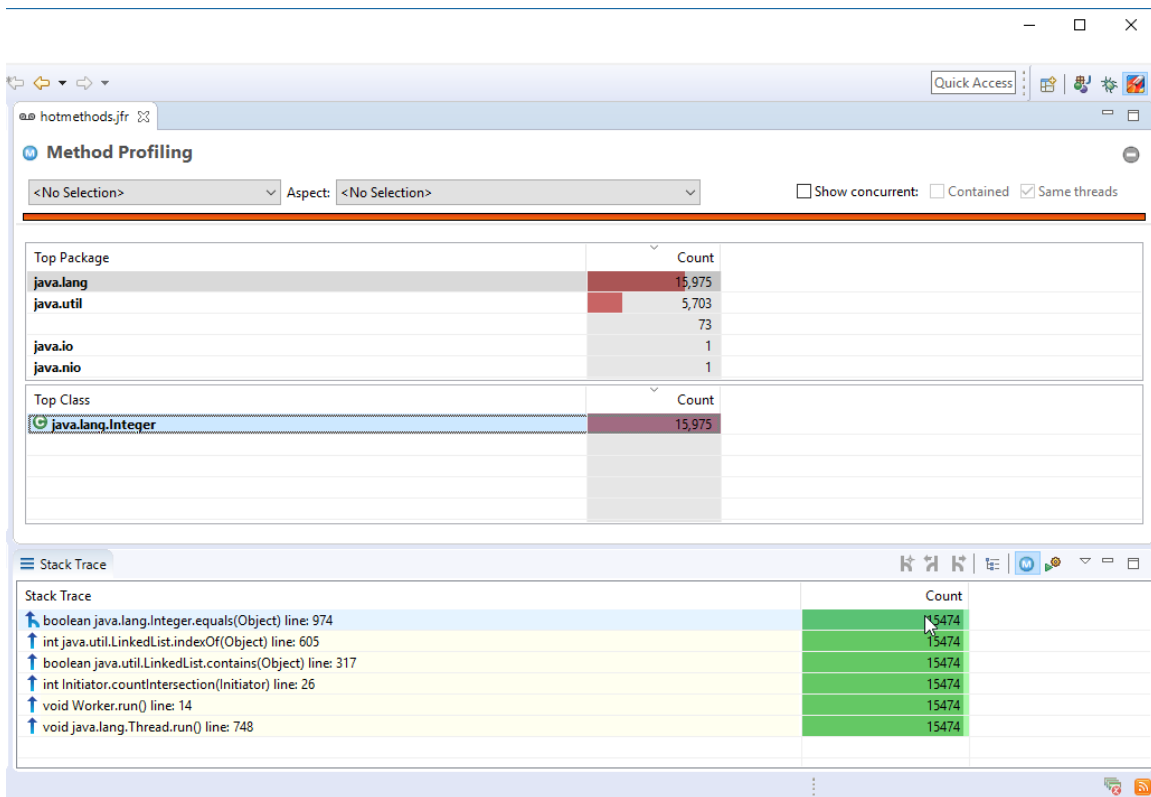
Next click on the Profiling lane. The stack trace view will show the aggregated stack traces of any selection in the editor, and will now show you the stack traces for the profiling samples.

In the recording, one of these methods has a lot more samples than the others. This means that the JVM has spent more time executing that method relative to the other methods. Which method is the hottest one? From where do the calls to that method originate?

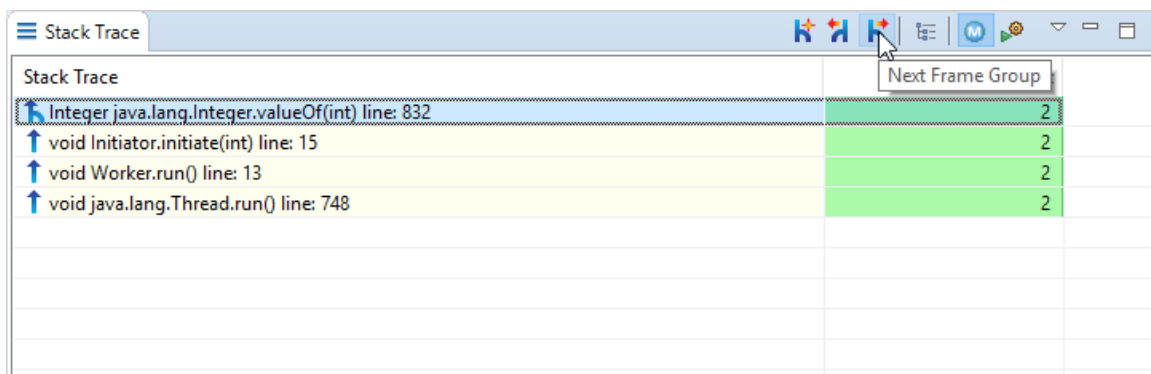
Which method do you think would be the best one to optimize to improve the performance of this application?

***Note:** Often the hotspot is in a method beyond your control. Look for a predecessor that you can affect.*

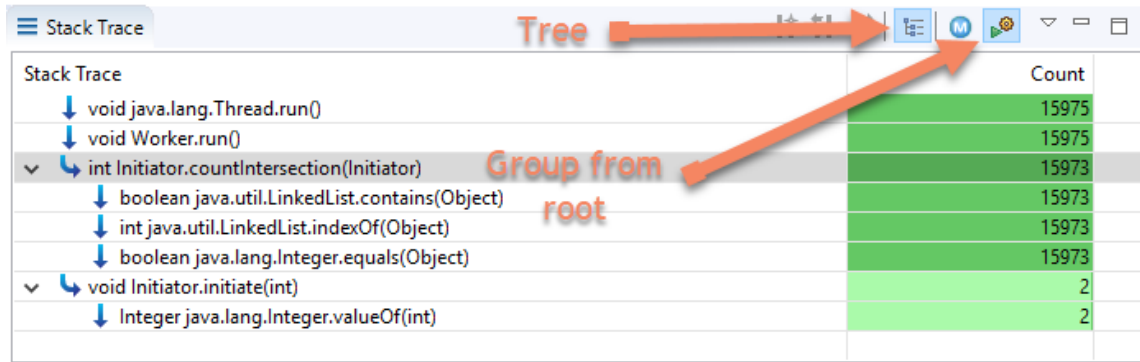
There is another page, **Method Profiling**, that makes it easy to break down the method profiling samples per package and class of where the sample was captured.



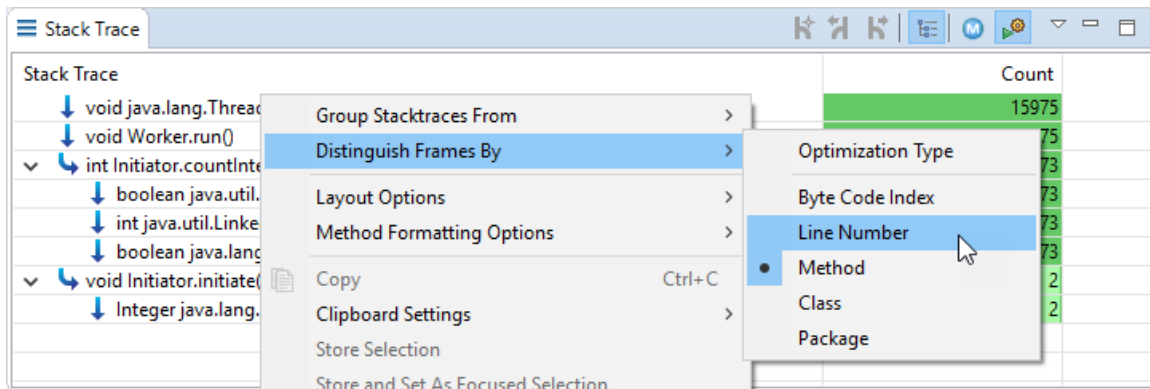
In the stack trace view, the most commonly traveled path is shown by default, effectively giving you the most common stack trace directly. Wherever the path branches, there is a branch icon. You can use the right and left arrow keys to select between the different branches (Frame Groups), or use the toolbar buttons:



If you would rather use a tree representation, and see the aggregation done from the thread roots, this can also be done:



Note that there are no line numbers in the last screenshot. You can select at what granularity to distinguish the frames from each other, in effect grouping frames together. This is controlled from the context menu:



Using **Method** will often be a helpful tool to declutter the view.

Deep Dive Exercises:

1. Can you, by changing one line of code, make the program much more effective (more than a factor 10)?

Note: If you get stuck, help can be found in the Readme.txt file in the projects.

*Note: To save resources, **remember to close the flight recordings you no longer need.***

2. Is it possible to do another recording to figure out how much faster the program became after the change?

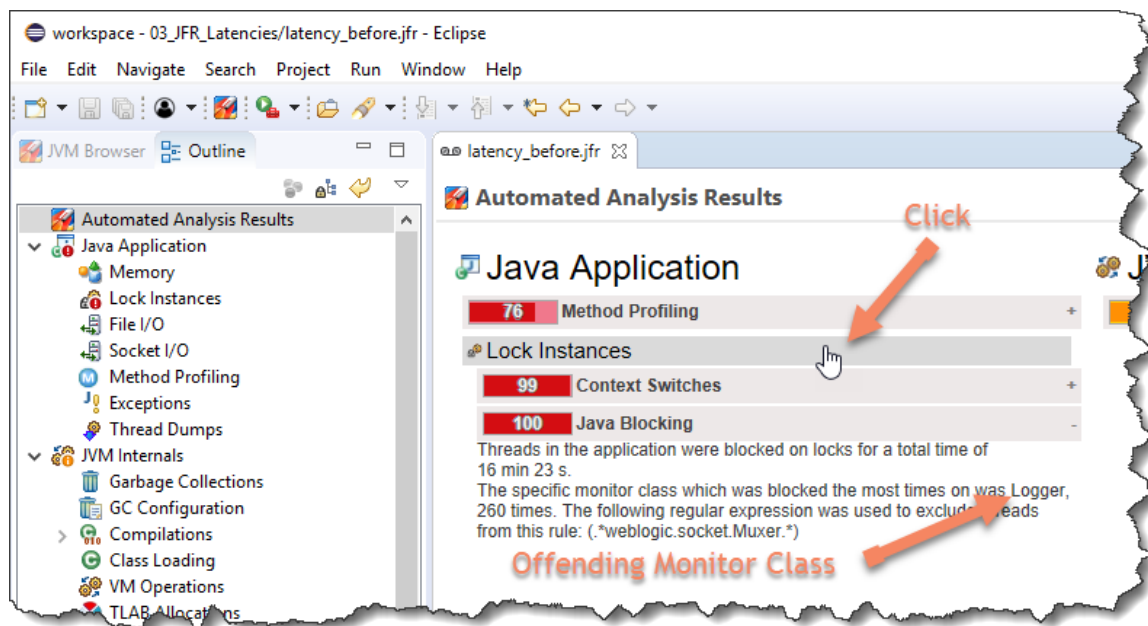
***Note:** The application generates custom events for each unit of “work” done. This makes it easy to compare the time it takes to complete a unit of work before and after the code change. Would it be possible to decide how faster the program became without these custom events?*

The moral of the exercise is that no matter how fast the JVM is, it can never save you from poor choices in algorithms and data structures.

Exercise 3 – Latencies

Another class of problems deals with latencies. A symptom of a latency related problem can be lower than expected throughput in your application, without the CPU being saturated. This is usually due to your threads of execution stalling, for example due to bad synchronization behavior in your application. The Flight Recorder is a good place to start investigating this category of problems.

Like any good cooking show, we've provided you with a pre-recorded recording to save you from having to wait another few minutes for the recording to finish. Open the **03_JFR_Latencies/latency_before.jfr** recording (same procedure as when opening the hotmethod recording in the previous exercise). Then switch back to the **Mission Control** perspective.



From the rule result our threads seem to be waiting a lot to enter a java monitor. We can already see what monitor class seems responsible. Click the suggested Lock Instances page to take a closer look.

The screenshot shows the Eclipse IDE with the 'JVM Browser' and 'Lock Instances' views. The 'Lock Instances' view displays a table of lock data:

Monitor Class	Total Blocked Time	Distinct Threads	Count
Logger	16 min 23 s	20	260

Monitor Address	Total Blocked Time	Distinct Threads	Count
0x258DF9E0	16 min 23 s	20	260

Thread	Total Blocked Time	Count
Worker Thread 11	52.100 s	7
Worker Thread 19	51.899 s	7
Worker Thread 9	50.598 s	9
Worker Thread 17	50.580 s	15
Worker Thread 15	50.343 s	11

The 'Stack Trace' view shows the following stack:

```

Stack Trace
void WorkerThread.run()
void Logger.log(String)

```

The 'Properties' view shows '100 Java Blocking' with a description: 'Threads in the application were blocked on locks for a total time of 16 min 23 s. The specific monitor class which was blocked the most times on was Logger, 260 times. The following regular expression was used to exclude threads from this rule: (*weblogic.socket.Muxer.*)'

Of what class is the lock we're blocking on? From where in the code is that event originating?

***Note:** In this case it is a very shallow trace. In a more complex scenario it would, of course, have been deeper.*

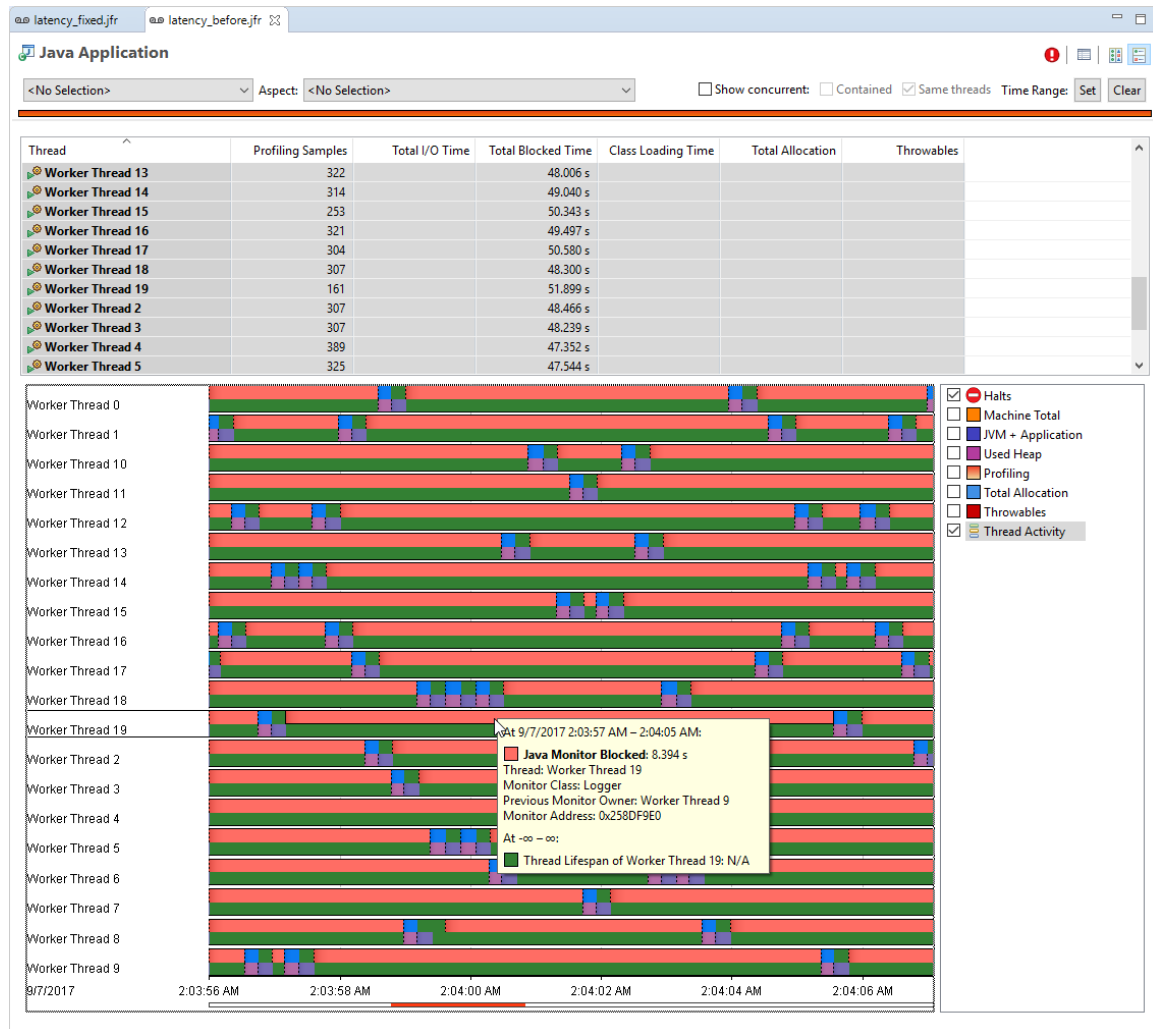
It seems most of these blocking events come from the same source.

Let's take a step back and consider the information we've gathered. Most of our worker threads seem to be waiting on each other attempting to get the Logger lock. All calls to that logger seem to be coming from the `WorkerThread.run()`.

Can you think of a few ways to fix this?

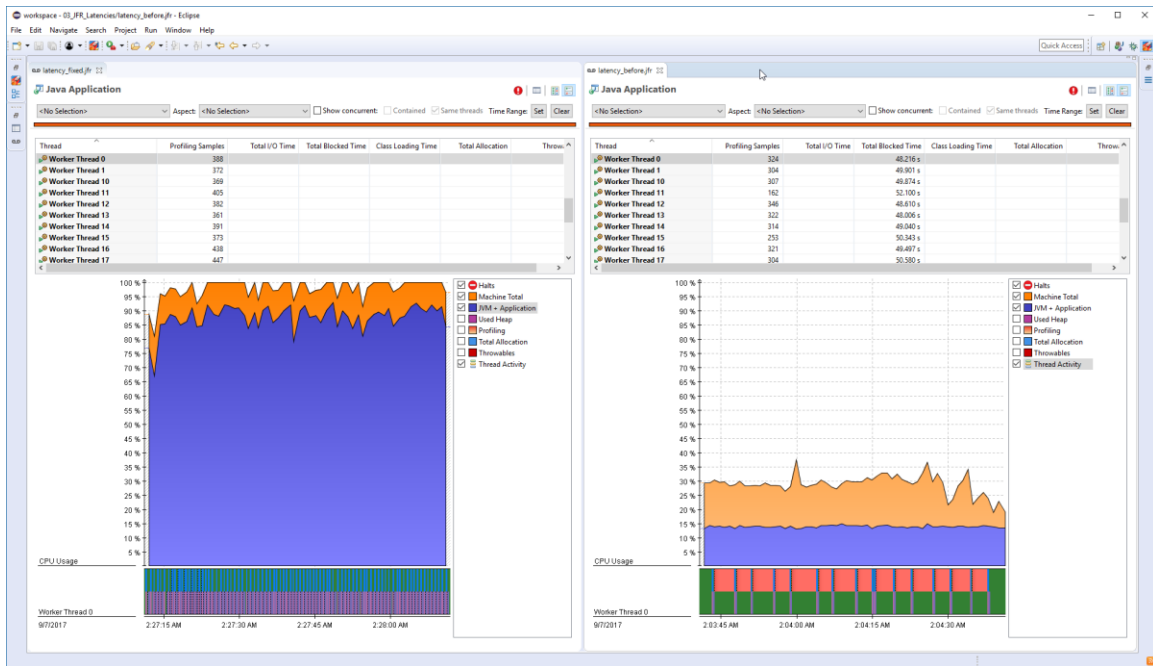
Note: Right click on the `Logger.log(String)` method and select *Open Method* if running JMC in Eclipse. If not running in Eclipse open the source file and take a look at it. We get several matches; select the one in `03_JFR_Latencies`. The method is *synchronized*.

Note: The events can also be visualized directly in the Java Application view. Select all worker threads.



Note: More hints in the *Readme.txt* document in the project.

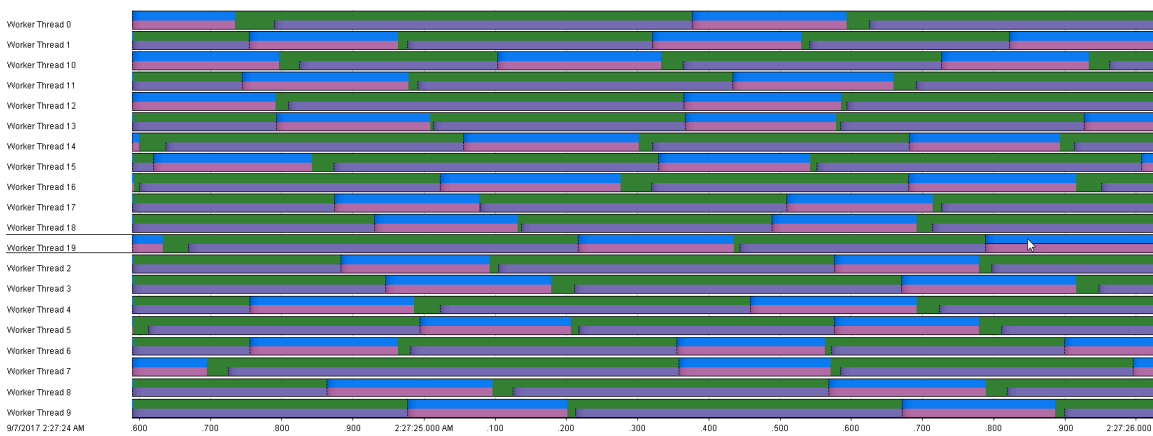
In the `latency_fixed.jfr` recording we simply removed the `synchronized` keyword from the `Logger.log(String)` method. Can you see any difference to the other recording? Are the threads getting to run more or less than before? Are we getting better throughput now? How many threads are stalling now?



Note: You can compare recordings side by side by dragging and docking the editors that contain them in the standard Eclipse way.

Note: The CPU load can be seen in the **Java Application** tab.

Note: Green means the thread is happily running along (also purple, for our own custom Work events). In the `latency_before.jfr` recording only one thread is running at any given time, the rest are waiting. In the `latency_fixed.jfr` recording, they are happily running in parallel. Also, no salmon-colored Java Monitor Blocked events can be seen at all.



The moral of this exercise is that bad synchronization can and will kill the performance and responsiveness of your application.

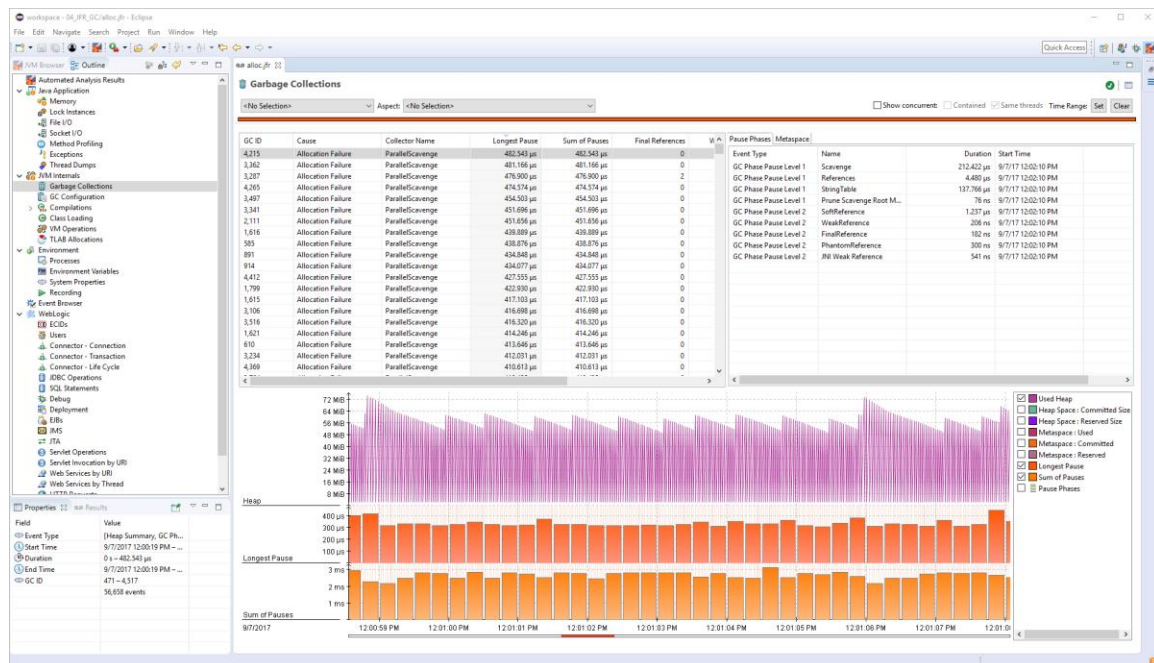
Exercise 4 (Bonus) – Garbage Collection Behavior

While JVM tuning is out of the scope for this set of exercises, this exercise will show how to get detailed information about the Garbage Collections that happened during the recording, and how to look at allocation profiling information.

Open the `allocator_before.jfr` recording in the `04_JFR_GC` project. Switch to the Mission Control perspective (if in Eclipse). Note that in JMC 6.0, the **Automated Analysis** doesn't really show a clear signal here (it will in JMC 6.1). Instead, got to the **Garbage Collections** page.

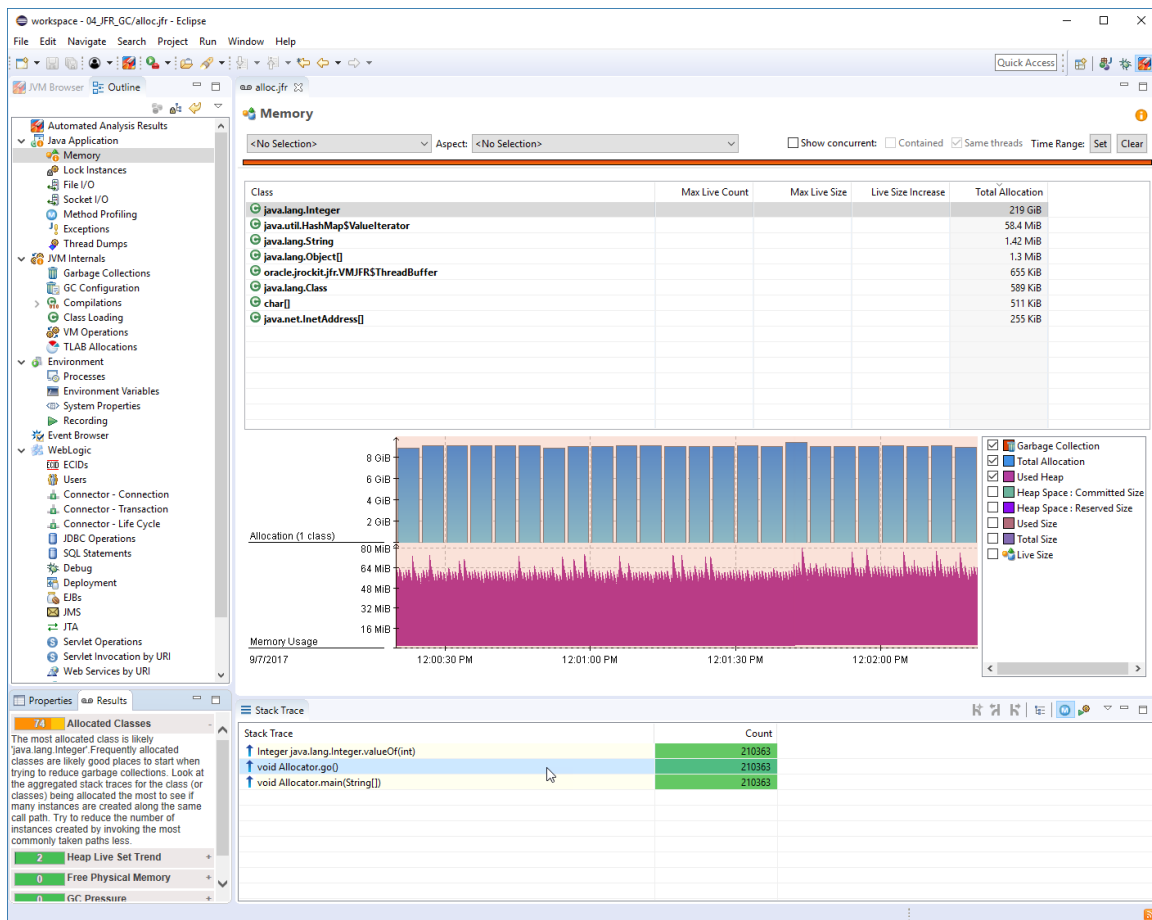
In this page you can see many important aspects about each and every garbage collection that happened during the recording. As can be seen from the graph, garbage collections occur quite frequently.

Note: the charts can be zoomed with the mouse scroll wheel or with the key buttons (left/right to pan, up/down to zoom). If you want to use the keys to zoom/pan, make sure the chart panel is selected, for example by clicking on the x-axis underneath the charts.



It does not seem like there is anything special, like the handling of special reference types, causing garbage collections to take an unreasonably long time, not to mention that the garbage collections are pretty short. We are simply creating quite large amounts of garbage.

Go to the **Memory** page. What kind of allocations (what class of objects) seems to be causing the most pressure on the memory system? From where are they allocated?



Note: Jump to the first method in the trace that you think you can easily alter.

Deep Dive Exercises:

- Can you, with a very simple rewrite of the inner `MyAlloc` class only, cause almost all object allocations to cease and almost no garbage collections to happen, while keeping the general idea of the program intact? You only need a minor change in two lines of the code. To see the difference, look at the [allocator_after.jfr](#) recording. How many garbage collections are there after the fix?

Note: Hints in the *Readme.txt*

- You can see even more detail if you go to the TLAB Allocations page. Does the TLAB size seem aptly sized for this application?

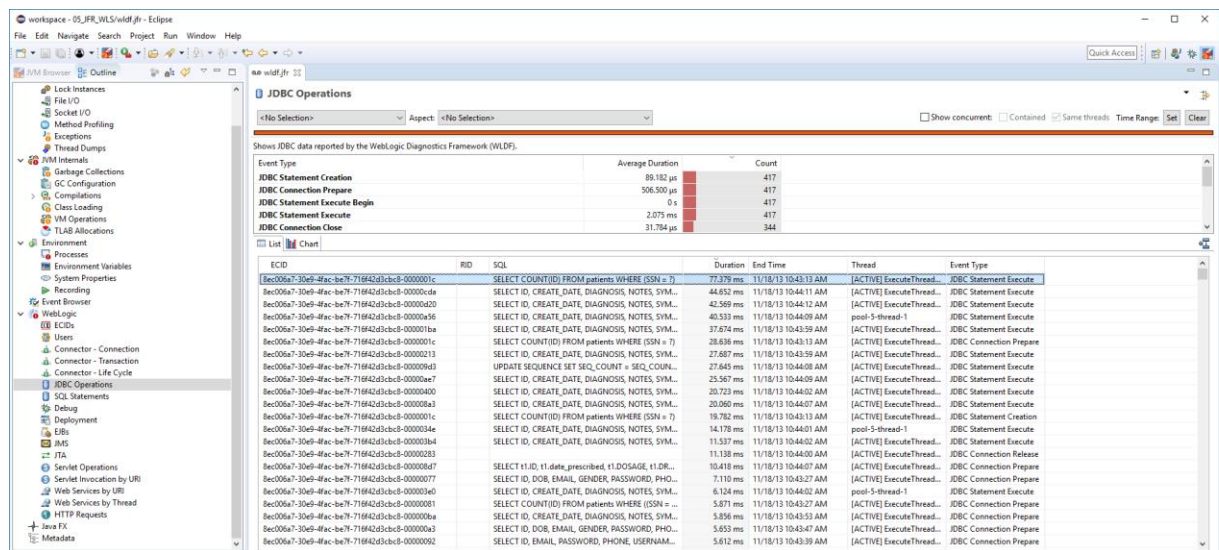
The moral of this exercise is that whilst the runtime will happily take care of any and all garbage that is thrown at it, a great deal of performance can be gained by not throwing unnecessary garbage at the poor unsuspecting runtime.

Exercise 5 (Bonus) – WebLogic Server Integration

This exercise will familiarize you with various elements of the new user interface. It also shows that it is possible to create integration with JFR and JMC from a party outside of the JDK, in this case the WebLogic Diagnostics Framework (WLDF). Even if you are not using WLS, this is a good exercise, as it walks through some powerful features in the JMC JFR user interface.

First open the file named `05_JFR_WLS/wldf.jfr`. This recording contains, aside from the standard flight recorder events, events contributed by WLDF.

Open the **WebLogic / JDBC Operations** page. Can you tell which JDBC query took the longest time? How long did it take?



The screenshot displays the WebLogic JMC interface with the 'JDBC Operations' page selected. The top summary table shows the following data:

Event Type	Average Duration	Count
JDBC Statement Creation	89.182 µs	417
JDBC Connection Prepare	506.500 µs	417
JDBC Statement Execute Begin	0 s	417
JDBC Statement Execute	2.075 ms	417
JDBC Connection Close	31.794 µs	344

The bottom table lists individual JDBC operations with the following columns: ECID, RID, SQL, Duration, End Time, Thread, and Event Type. The longest duration shown is 77.759 ms for the first row.

ECID	RID	SQL	Duration	End Time	Thread	Event Type
bec006a7-30e8-4fac-be7f-71642d3c8c0-0000007c		SELECT COUNT(*) FROM patients WHERE (SSN = ?)	77.759 ms	11/18/13 10:44:11 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000de		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	44.652 ms	11/18/13 10:44:11 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000020		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	42.569 ms	11/18/13 10:44:12 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000056		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	40.533 ms	11/18/13 10:44:09 AM	pool-5-thread-1	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-000001ba		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	37.674 ms	11/18/13 10:43:59 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-0000001c		SELECT COUNT(*) FROM patients WHERE (SSN = ?)	28.626 ms	11/18/13 10:43:13 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000013		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	27.687 ms	11/18/13 10:43:59 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000d3		UPDATE SEQUENCE SET SEQ_COUNT = SEQ_COUNT...	27.645 ms	11/18/13 10:44:08 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000e7		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	25.567 ms	11/18/13 10:44:08 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000040		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	20.723 ms	11/18/13 10:44:02 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000a3		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	20.960 ms	11/18/13 10:44:07 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-0000001c		SELECT COUNT(*) FROM patients WHERE (SSN = ?)	18.782 ms	11/18/13 10:43:13 AM	[ACTIVE] ExecuteThread...	JDBC Statement Creation
bec006a7-30e8-4fac-be7f-71642d3c8c0-0000004e		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	14.178 ms	11/18/13 10:44:01 AM	pool-5-thread-1	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000b4		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	11.537 ms	11/18/13 10:44:02 AM	[ACTIVE] ExecuteThread...	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000033		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	11.138 ms	11/18/13 10:44:00 AM	[ACTIVE] ExecuteThread...	JDBC Connection Release
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000d7		SELECT ID, DOB, EMAIL, GENDER, PASSWORD, PHO...	10.418 ms	11/18/13 10:44:07 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000077		SELECT ID, DOB, EMAIL, GENDER, PASSWORD, PHO...	7.110 ms	11/18/13 10:43:27 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000a0		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	6.124 ms	11/18/13 10:44:02 AM	pool-5-thread-1	JDBC Statement Execute
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000081		SELECT COUNT(*) FROM patients WHERE (SSN = ...	5.871 ms	11/18/13 10:43:27 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare
bec006a7-30e8-4fac-be7f-71642d3c8c0-0000009a		SELECT ID, CREATE_DATE, DIAGNOSIS, NOTES, SYM...	5.856 ms	11/18/13 10:43:55 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare
bec006a7-30e8-4fac-be7f-71642d3c8c0-000000a3		SELECT ID, DOB, EMAIL, GENDER, PASSWORD, PHO...	5.653 ms	11/18/13 10:43:47 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare
bec006a7-30e8-4fac-be7f-71642d3c8c0-00000092		SELECT ID, EMAIL, PASSWORD, PHONE, USERNAM...	5.612 ms	11/18/13 10:43:39 AM	[ACTIVE] ExecuteThread...	JDBC Connection Prepare

Open the **WebLogic / Servlet Invocation by URI** page. Can you tell which invocation of the **viewPatients** servlet took the longest time? How long did it take?

The screenshot shows the 'Servlet Invocation by URI' page in the WebLogic IDE. The table below summarizes the data presented in the main view:

URI	Count	Average Duration	Total Duration
/physician-web/physician/viewRecordSummary.action	92	426.285 ms	39.218 s
/physician-web/physician/viewPatients.action	54	719.293 ms	38.936 s
/physician-web/physician/action	101	209.232 ms	21.132 s
/physician-web/physician/createRecord.action	90	186.443 ms	16.780 s
/console/console.portal	5	1.093 s	5.463 s

The detailed view of the longest invocation (ECID: bec006a7-306f-4fac-bc7f-716a2d3cb8-00000b7) shows a duration of 6.267 s, starting at 11/18/13 10:43:47 AM and ending at 11/18/13 10:43:53 AM. The thread is [ACTIVE] ExecuteThread... and the event type is Servlet Request Run. The stack trace at the bottom shows the execution flow from the servlet request to the database layer.

Let's take a look at everything that was going on during that request. Select the longest lasting viewPatients servlet, and select the ECID (Execution Context ID) in the Properties view. An ECID is an identifier which follows a request through the system across process and thread boundaries. A little bit like an Open Tracing Span ID.

Select **Store and set as focused selection** from the context menu.

The screenshot displays the Eclipse IDE interface with the WebLogic Diagnostics Framework (WDF) data. The 'Servlet Invocation by URI' table is the primary focus, showing a list of URIs and their associated performance metrics. The 'Properties' window on the left provides detailed information about the selected event, including its event type, start and end times, thread, user ID, return value, method name, class name, transaction ID, RCD, and RID. The 'Stack Trace' window on the right shows the call stack for the selected event. The 'Store and Set As Focused Selection' option is highlighted in the context menu.

URI	Count	Average Duration	Total Duration
/physician-web/physician/viewRecordSummary.action	92	426.285 ms	39.218 s
/physician-web/physician/viewPatients.action	54	770.293 ms	38.356 s
/physician-web/physician/viewPatients.action	101	209.232 ms	21.132 s
/physician-web/physician/createRecord.action	90	186.445 ms	16.780 s
/console/console.portal	5	1.085 s	5.463 s

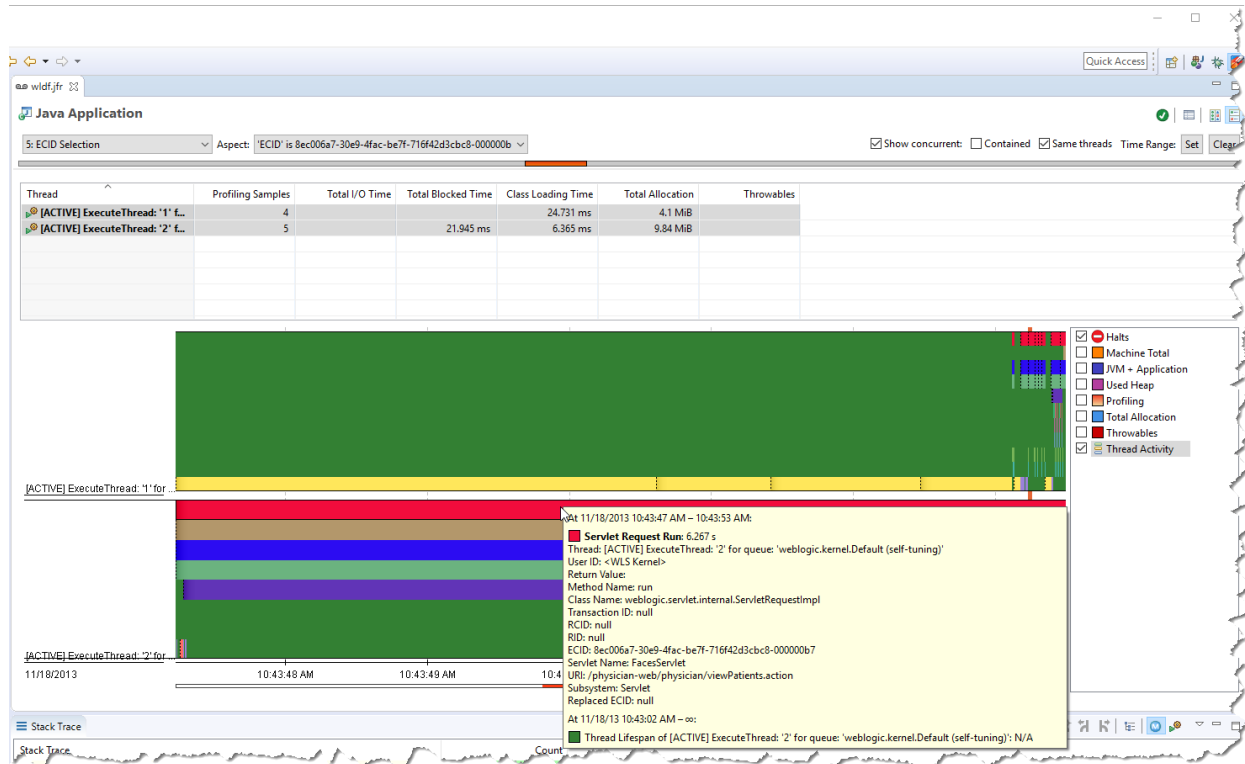
URI	ECID	RID	Duration	Start Time	End Time	Thread	Event Type
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000067		6.267 s	11/18/13 10:43:47 AM	11/18/13 10:43:53 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000068		6.238 s	11/18/13 10:43:47 AM	11/18/13 10:43:53 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000069		6.216 s	11/18/13 10:43:47 AM	11/18/13 10:43:53 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-000006a		6.215 s	11/18/13 10:43:47 AM	11/18/13 10:43:53 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-000006b		6.214 s	11/18/13 10:43:47 AM	11/18/13 10:43:53 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-000006c		940.148 ms	11/18/13 10:43:57 AM	11/18/13 10:43:58 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-000006d		925.701 ms	11/18/13 10:43:57 AM	11/18/13 10:43:58 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-000006e		886.324 ms	11/18/13 10:43:57 AM	11/18/13 10:43:58 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-000006f		182.682 ms	11/18/13 10:44:12 AM	11/18/13 10:44:12 AM	pool-5-thread-1	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000070		157.159 ms	11/18/13 10:44:09 AM	11/18/13 10:44:09 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000071		142.934 ms	11/18/13 10:43:59 AM	11/18/13 10:43:59 AM	pool-5-thread-1	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000072		138.304 ms	11/18/13 10:44:09 AM	11/18/13 10:44:09 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000073		129.887 ms	11/18/13 10:44:09 AM	11/18/13 10:44:09 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000074		126.755 ms	11/18/13 10:43:59 AM	11/18/13 10:43:59 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000075		122.044 ms	11/18/13 10:44:02 AM	11/18/13 10:44:02 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000076		117.967 ms	11/18/13 10:44:07 AM	11/18/13 10:44:07 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000077		106.411 ms	11/18/13 10:43:58 AM	11/18/13 10:43:58 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000078		106.156 ms	11/18/13 10:44:03 AM	11/18/13 10:44:03 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000079		105.638 ms	11/18/13 10:44:08 AM	11/18/13 10:44:08 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000080		103.848 ms	11/18/13 10:44:03 AM	11/18/13 10:44:03 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000081		103.454 ms	11/18/13 10:44:13 AM	11/18/13 10:44:13 AM	[ACTIVE] ExecuteThread...	Servlet Request Run
/physician-web/physician/viewPatients.action	8ec006a7-30e9-4fac-be79-716a2d3cb8-0000082		103.224 ms	11/18/13 10:44:00 AM	11/18/13 10:44:00 AM	[ACTIVE] ExecuteThread...	Servlet Request Run

Stack Trace

Stack Trace	Count
void weblogic.diagnostics.instrumentation.gathering.FlightRecorderEventHelper.recordT...	1
void weblogic.diagnostics.instrumentation.action.FlightRecorderRoundAction.FlightRe...	1
void weblogic.diagnostics.instrumentation.action.FlightRecorderRoundAction.FlightRe...	1
void weblogic.diagnostics.instrumentation.action.FlightRecorderRoundAction.postProc...	1
umentation.support.instrumentation.supportimpl.postProc...	1
umentation.instrumentation.support.postprocess.LocalHol...	1
ServletRequestImpl.run()	1
ContainerSupportProviderImpl\$WebRequestExecutor.run()	1
read.execute(Runnable)	1
read.run()	1

This will focus the user interface on events with the property value of that ECID. Open the **Java Application** page.

The selection box at the top shows that we are now looking at events matching our selection.

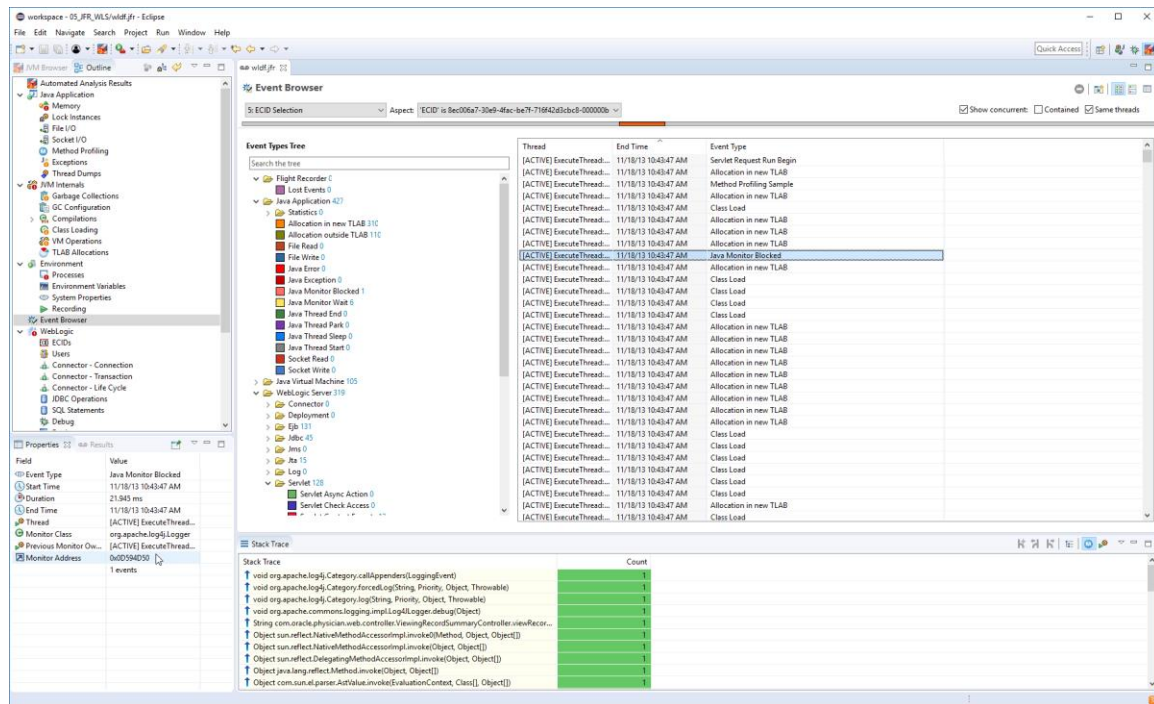


Also note that we can select other aspects of the selection to determine what the UI shows. Some pages may only be able to act on certain aspects of a selection. If the **Show concurrent** box is checked, events concurrent (happening during the same time interval) to the selection will also be showed. If **Contained** is checked, only events that is fully contained within the time range will be shown. If **Same threads** is selected, only the threads in the selection will be shown.

Check the **Show concurrent** and **Same threads** checkboxes. Next click the **Set** button to set the time range for the page to the time range of the active selection. Can you find any low-level events that do not have an ECID?

Note: There is, for example, a tiny bit of contention on a log4j logger in the beginning. The Blocking event does not have an ECID; it is shown due to “Show concurrent” being enabled.

Open the **Event Browser** page. Here you can look at the events grouped by Event Type. With our selection settings, we will now see the same events listed. Selecting an event will show the properties for the selected event in the **Properties** view. Selecting multiple events will show the common properties for the selection. As shown before, the properties can be used to establish new selections.



Deep Dive Exercises:

- Can you find the aggregated stack traces for where the SQL statement taking the longest time to execute (on average) originated?

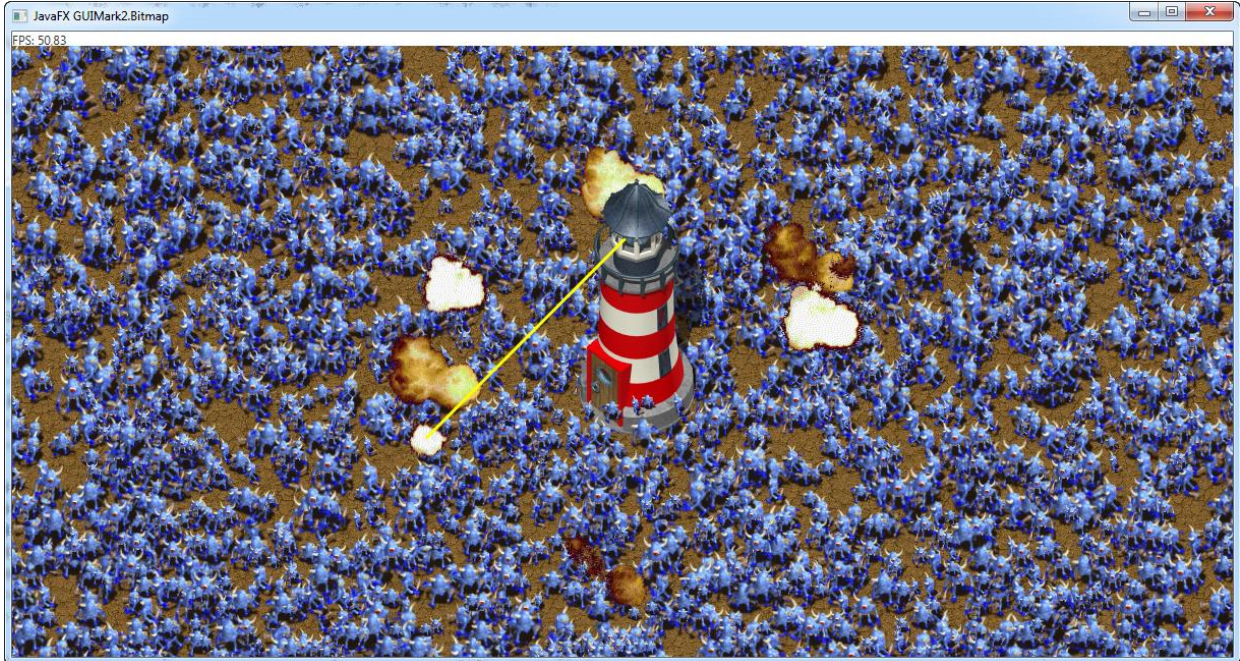
*Hint: Go to the **SQL Statements** page, find the line in the table representing the events with the longest average duration. Look at the stack traces from the roots down.*

- Can you find out which EJB the application seems to be spending the most time in on average?
- Which user seems to be starting the most transactions?

The moral of this exercise is that there are tools available that extend flight recorder and that can be quite useful/powerful. Also, using selections and aspects of a selection can be a useful way to focus the user interface. The Properties view can also be a source of selections.

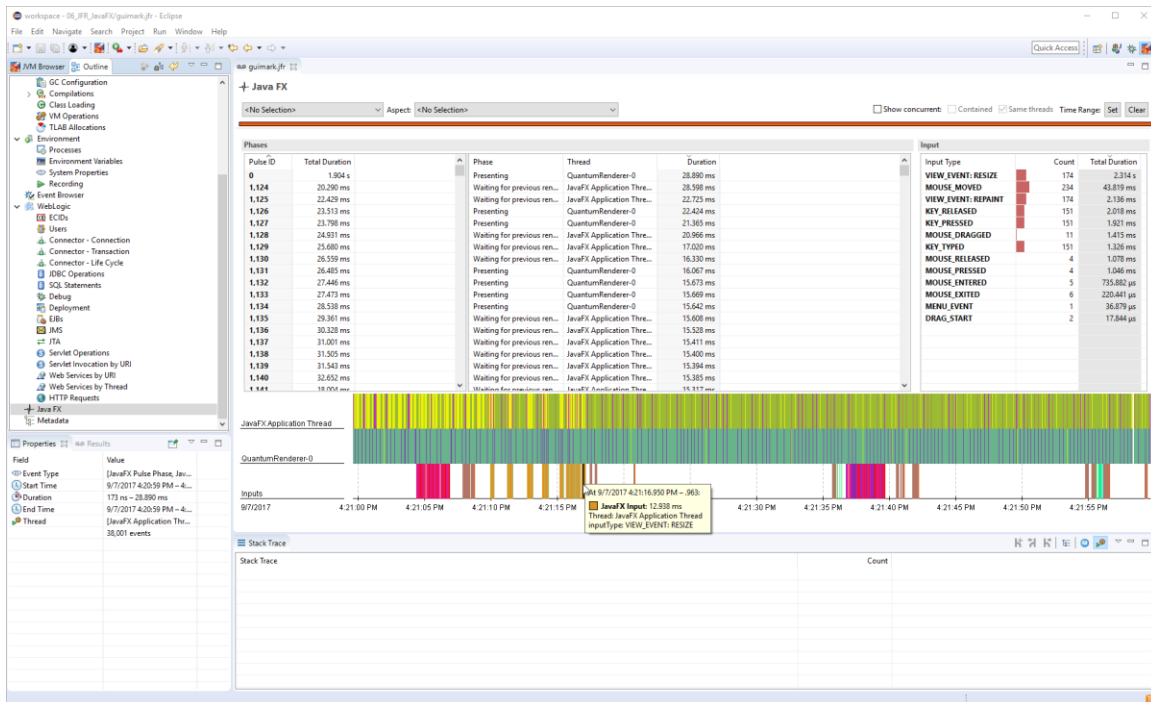
Exercise 6 (Bonus) – JavaFX

In this exercise, we will explore the Java FX integration with Java Flight Recorder. Use the GUIMark launcher to launch the GUIMark Bitmap benchmark application. You should see a tower shooting a laser at monsters.

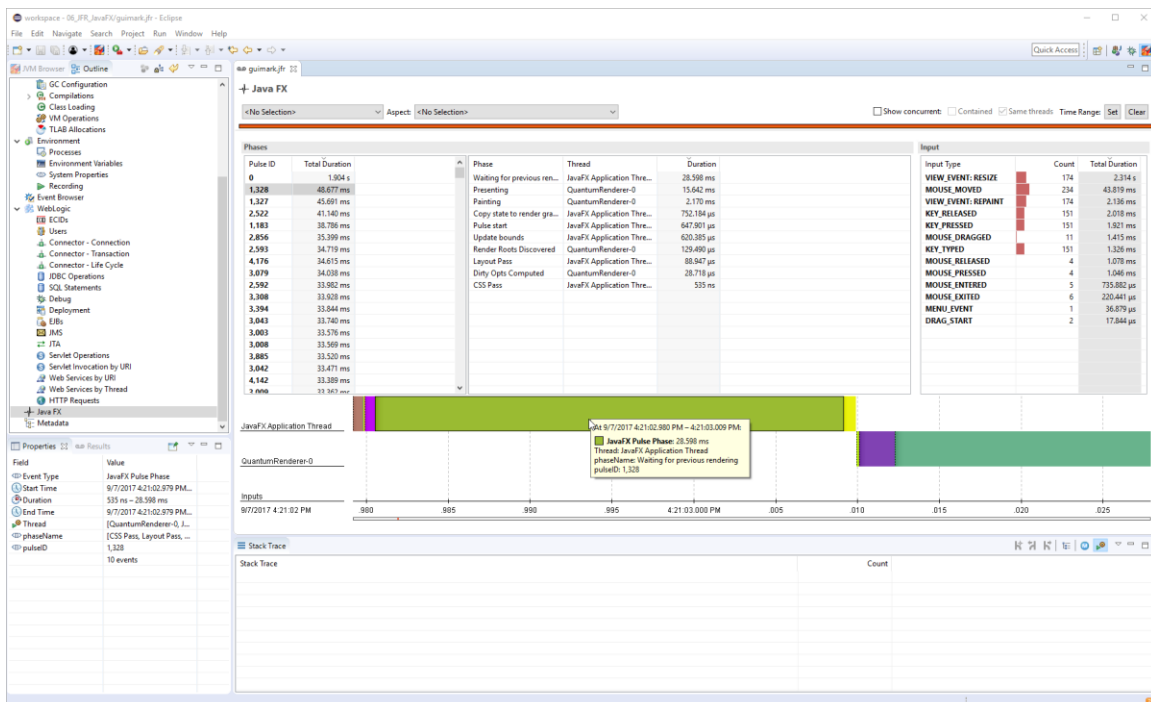


We will use a pre-recorded recording for this exercise, **06_JFR_JavaFX/guimark.jfr**, so you starting GUIMark was mostly pointless. But, come on – lasers? Monsters? It had to be seen.

***Note:** If you insist on having a locally produced recording, it is better to run the **GUIMark Auto Record** launcher. If you insist on not using the auto recording launcher, make sure to enable the JavaFX events in the recording wizard, or import the template in the project folder.*



Try looking at the recording using the special Java FX page. Can you tell which pulse took the longest time (disregard the weird pulse 0)? What phase was the one that took the longest for that particular pulse? Which was the input event that took the longest?

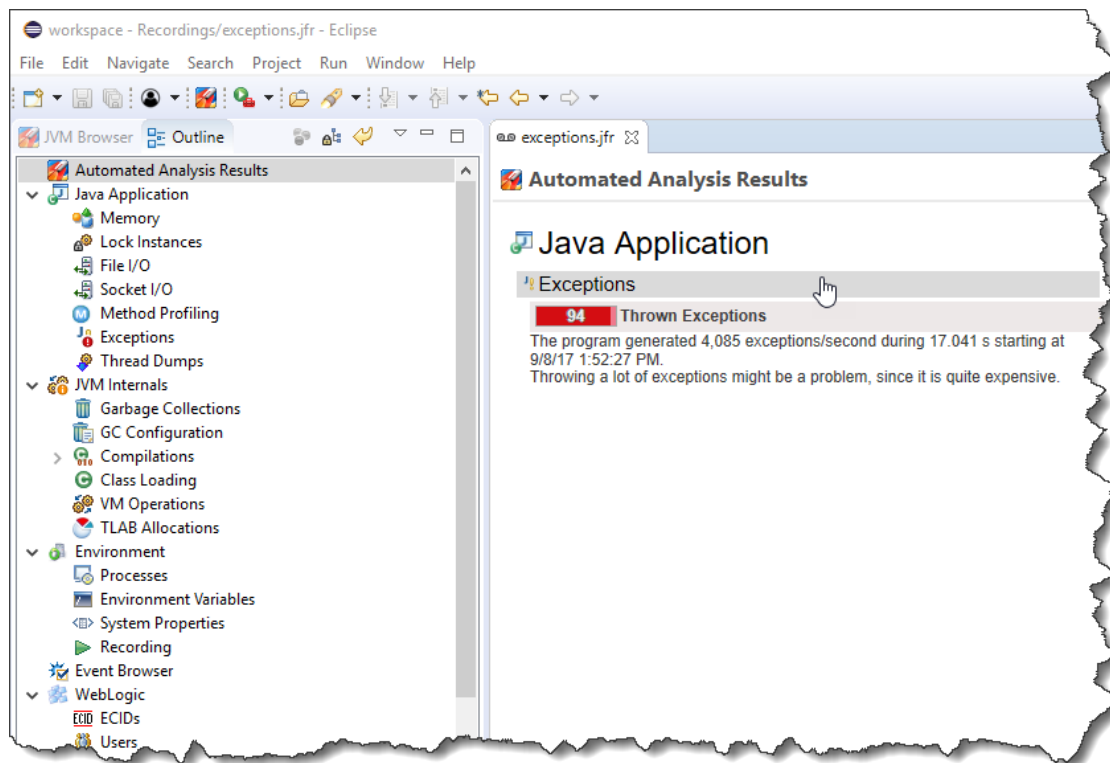


Exercise 7 (Bonus) – Exceptions

Some applications are throwing an excessive amount of exceptions. Most exceptions are caught and logged. Handling these exceptions can be quite expensive for the JVM, and can cause severe performance degradation. Fortunately, finding out where exceptions are thrown for a specific time interval is quite easy using the Flight Recorder, even for a system running in production.

Open up the flight recording named `exceptions.jfr` in the **07_JFR_Exceptions** project.

The automated analysis should indicate that things could be better:



Click on the **Exceptions** header above the rule result (or the **Exceptions** page in the **Outline**) to go to the Exceptions page.

The screenshot shows the IntelliJ IDEA IDE with the 'Exception Analysis' tool open. The left sidebar displays the 'Automated Analysis Results' tree, with 'Exceptions' selected. The main window shows the 'Exceptions' tab, which includes a table of counts for 'NullPointerException' and 'StackOverflowError'. Below the table is a message 'Just your average exception message.' and a timeline graph showing the distribution of exceptions over time. A statistics bar chart is also present, followed by a stack trace for the selected exception.

Class	Count
NullPointerException	8,197
StackOverflowError	78,755

Message: Just your average exception message.

Timeline: Event Log

Statistics

Exceptions

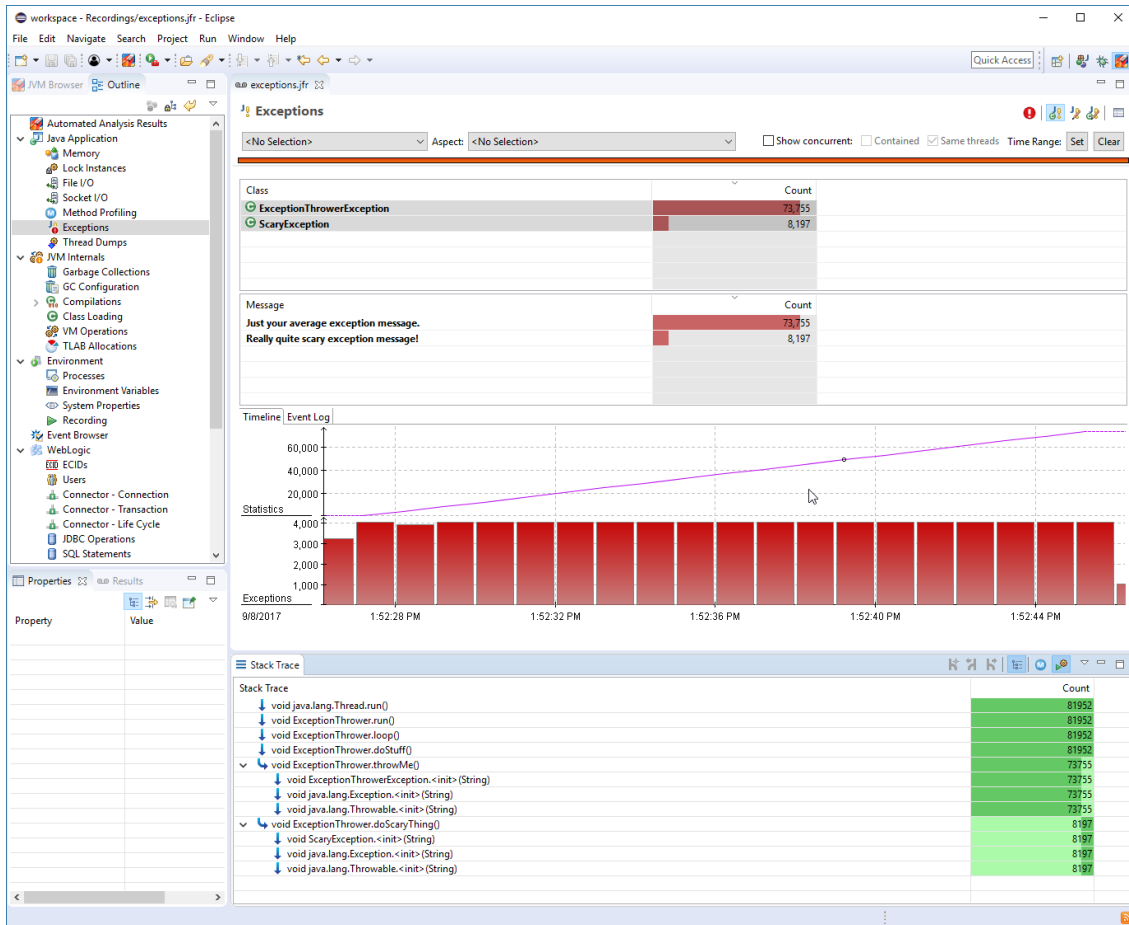
Stack Trace

```

Stack Trace
void java.lang.Thread.run()
void ExceptionThrower.run()
void ExceptionThrower.log()
void ExceptionThrower.doRun()
void ExceptionThrower.throwMe()
void ExceptionThrowerException.<init>(String)
void java.lang.Exception.<init>(String)
void java.lang.Throwable.<init>(String)

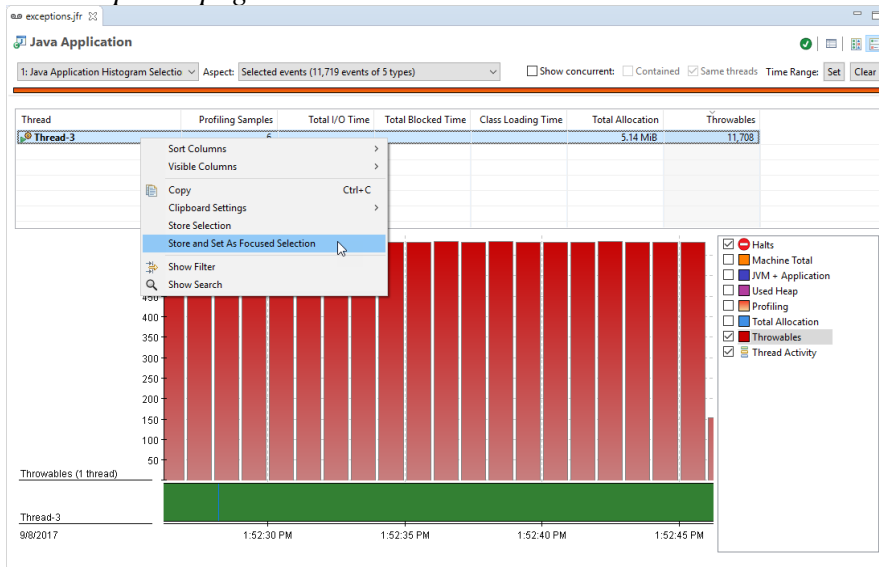
```

Note: Click the exception class you want stack traces for. You can also select multiple classes to see the aggregate traces for all classes in the selection.

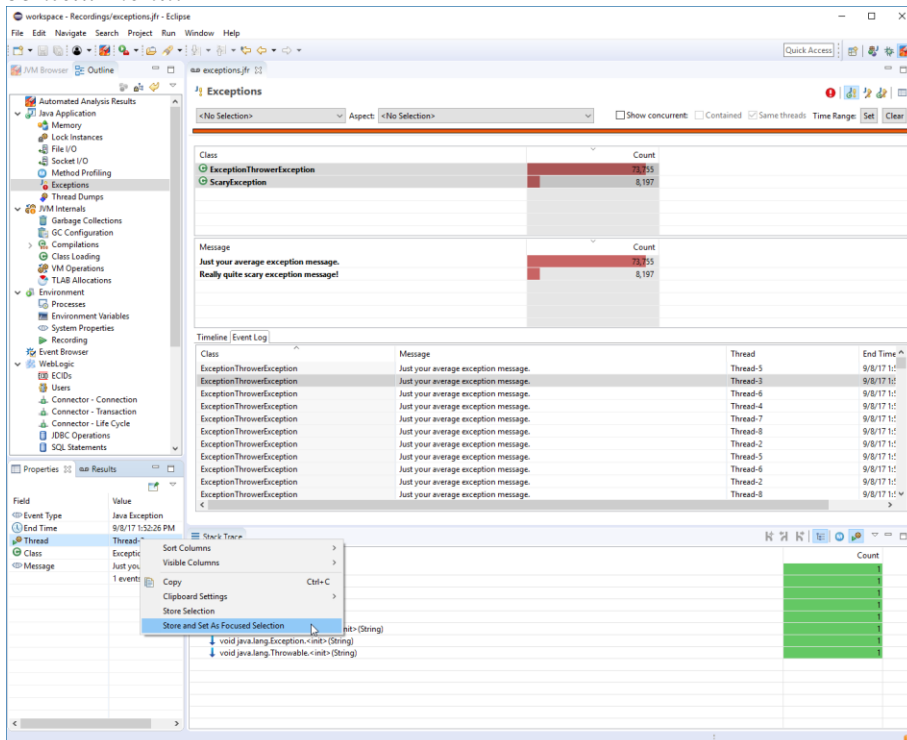


Can you tell exactly where the Scary exceptions are originating? In what threads are they originating? Can you study the time line for the exceptions in just one of the threads?

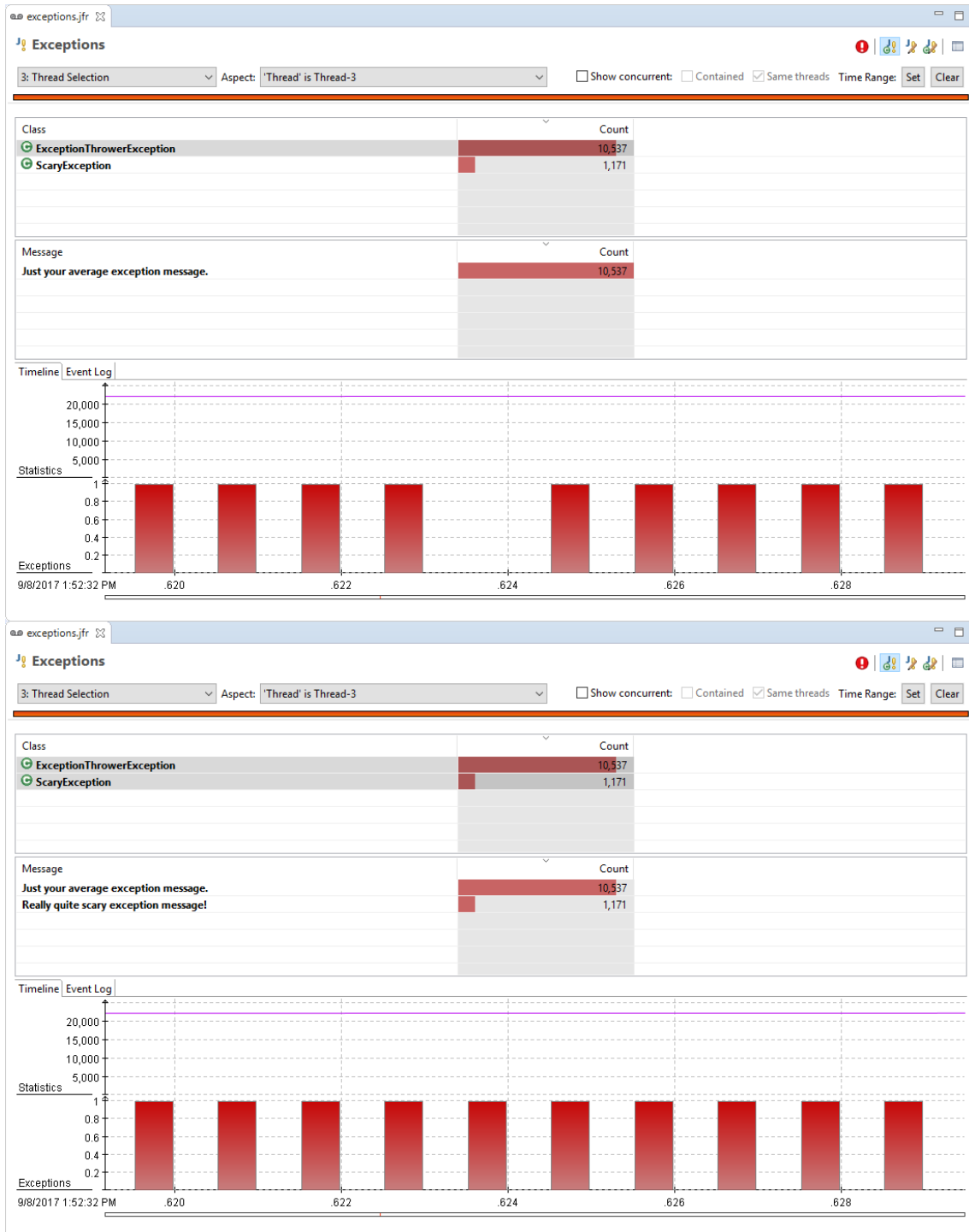
Note: Either go back to the **Java Application** page class, sort on **Throwables** in the thread table, and pick a Thread and select **Store and Set As Focused Selection** then go back to the **Exceptions** page:



...or pick an event in the **Event Log** directly on the **Exceptions** page, then select the **Thread** in the **Properties** view and use **Store and Set As Focused Selection** from the context menu:



Once a single thread has been selected, try zooming in the graph and switch between the two types of events, one each and both simultaneously. Can you see a pattern?



Note: Zoom until you can distinguish individual event buckets (the y-axis is showing 1).

Exercise 8 – Custom Events in JDK 9 (Bonus)

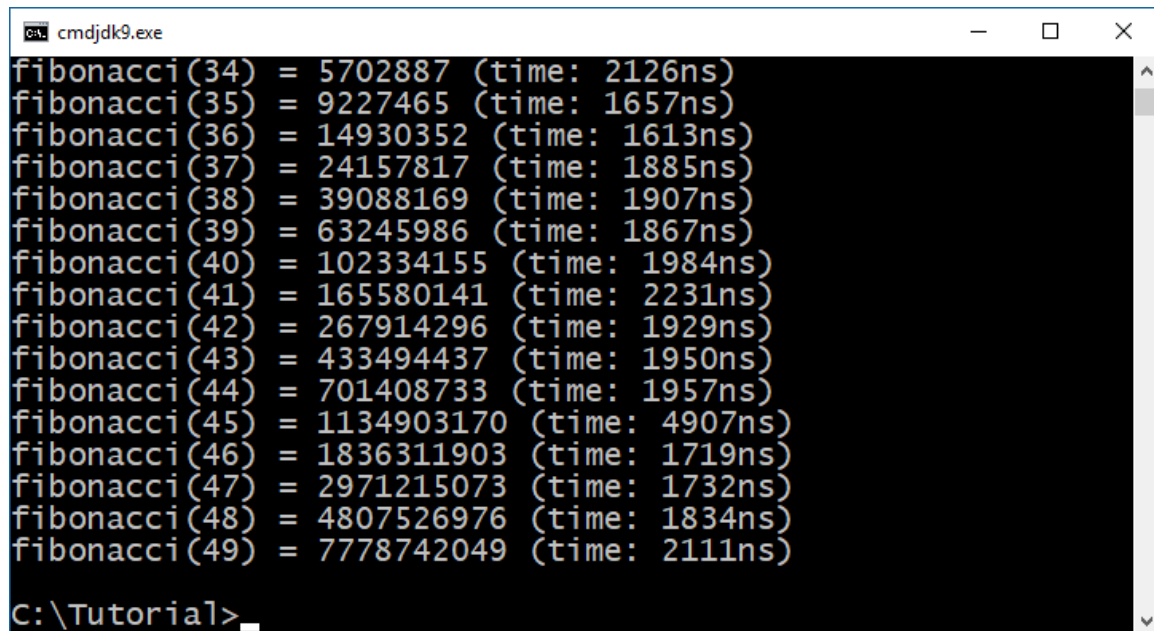
In JDK 9 the API for controlling the Java Flight Recorder, contributing JFR events, and parsing Flight Recordings is now supported by Oracle. Since we are running Eclipse on a JDK 8, we will be using the command line to run the examples. We will also cheat a little by compiling against a JDK 8 built jar with the JDK 9 API, so that the code can be edited from within Eclipse.

Go to the Java perspective and open the `08_JFR_CustomEvents` project. In the `com.oracle.example.jdk9jfr.control` package, there is a class named `RecordAndConsume`. Study the code and see if you can figure out what it does. To see how to create a custom event, study the `com.oracle.example.jdk9jfr.fib.FibonacciEvent` class.

Let's run the code. On Windows you can start a command line by double clicking on `cmdjdk9.exe`. This will start a terminal session with the appropriate environment variables set. On other platforms, make sure that you use JDK9. Next run the following command:

```
call recordAndConsume
```

This will run the `RecordAndConsume` class. You should see something similar to this:



```
cmdjdk9.exe
fibonacci(34) = 5702887 (time: 2126ns)
fibonacci(35) = 9227465 (time: 1657ns)
fibonacci(36) = 14930352 (time: 1613ns)
fibonacci(37) = 24157817 (time: 1885ns)
fibonacci(38) = 39088169 (time: 1907ns)
fibonacci(39) = 63245986 (time: 1867ns)
fibonacci(40) = 102334155 (time: 1984ns)
fibonacci(41) = 165580141 (time: 2231ns)
fibonacci(42) = 267914296 (time: 1929ns)
fibonacci(43) = 433494437 (time: 1950ns)
fibonacci(44) = 701408733 (time: 1957ns)
fibonacci(45) = 1134903170 (time: 4907ns)
fibonacci(46) = 1836311903 (time: 1719ns)
fibonacci(47) = 2971215073 (time: 1732ns)
fibonacci(48) = 4807526976 (time: 1834ns)
fibonacci(49) = 7778742049 (time: 2111ns)
C:\Tutorial>
```

How long did it take to calculate the longest Fibonacci number? How long did it take to calculate the shortest one?

Deep Dive Exercises:

8. If the first Fibonacci number took unexpectedly long, why do you think that is?

9. Is there any way you can use the flight recorder to prove your hypothesis?
10. The algorithm used is an iterative one. There is also a recursive version available. Try changing the code to use the recursive version and see how long it takes. Is there any difference?

Note: Change `Fibonacci.fibonacciIterative(n)` to `Fibonacci.fibonacciRecursive(n)`.

Note: You may want to lower the number of Fibonacci numbers to calculate from 50 (in the for loop) to maybe half before starting the program.

11. (**Very deep dive**) There is a program (`EnableDisableTesterFibonacci`) and a command line batch script (`runEnableDisableFib`) to run the Fibonacci example continuously, printing out the disassembled compiled `calculateFibonacci` method. Run the script (call `runEnableDisableFib`), and wait until the method has been compiled two times. Can you see any difference between the two versions? What do you think the overhead of having a disabled event in your code would be (for an optimized method)?

Note: To see the disassembled output, you need to get the appropriate library (`hsdis-amd64.dll`) and put it next to the java launcher in bin.

Note: Look for anything related to the `FibonacciEvent` class.

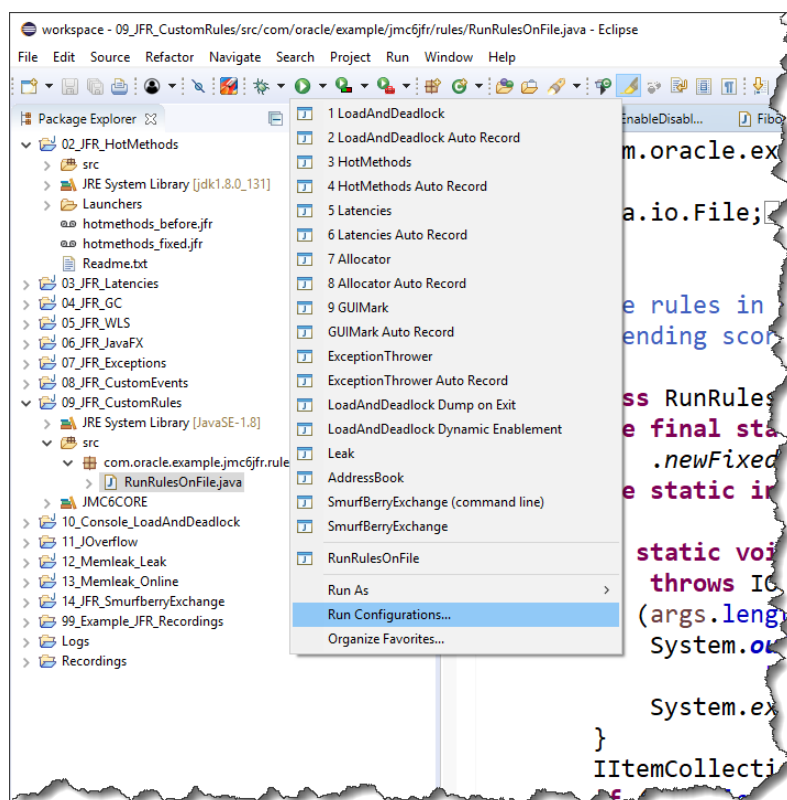
Exercise 9 – Custom Rules (Bonus)

There is one more Parser for JFR recordings. One that can run on JDK 7, and which also allows the parsing of JDK 7, 8 and 9 recordings transparently. It is the parser used in Java Mission Control. This parser supports internal iteration of events, and provides statistical aggregators. It is also the parser used when evaluating rules. In this exercise, we will evaluate the Java Mission Control rules headless by using a small program calling into the Java Mission Control core libraries. A custom rule will also be created, using the Java Mission Control Eclipse PDE support.

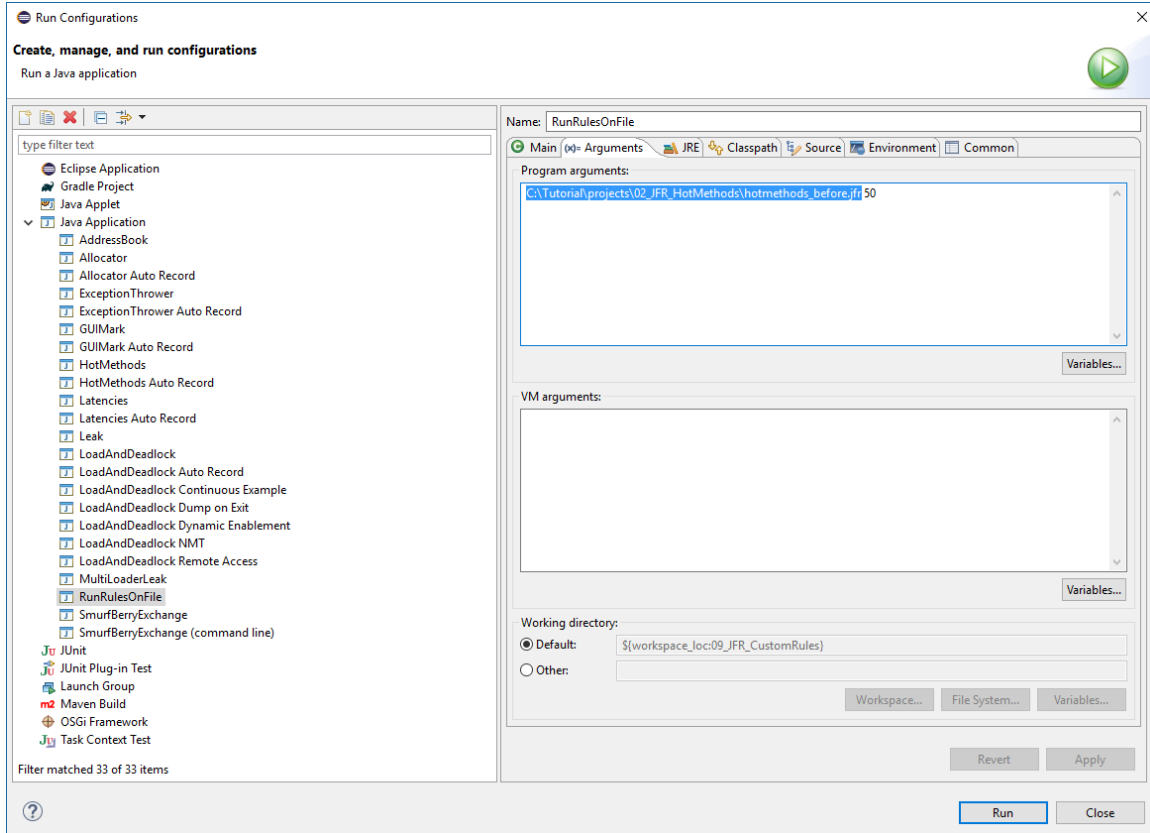
Running Rules

Take a look at the `RunRulesOnFile` class in [09_JFR_CustomRules](#). This class uses some JDK 8 features, but can easily be re-written to work on JDK 7, if required. Try running the class on different recordings by changing the `RunRulesOnFiles` launcher.

Open the [Run Configurations...](#) as shown below.



Then change the first parameter to the program to run it on various loads:



Try it on some of the recordings you have analyzed so far.

In Java Mission Control 6.1.0 there will be a built-in class which will output HTML looking very similar to the **Automated Analysis Results** page in JMC. For now, there is a build in class (`JfrRulesReport`) which will output in xml and html, and which can be configured with a custom xslt.

Deep Dive Exercises:

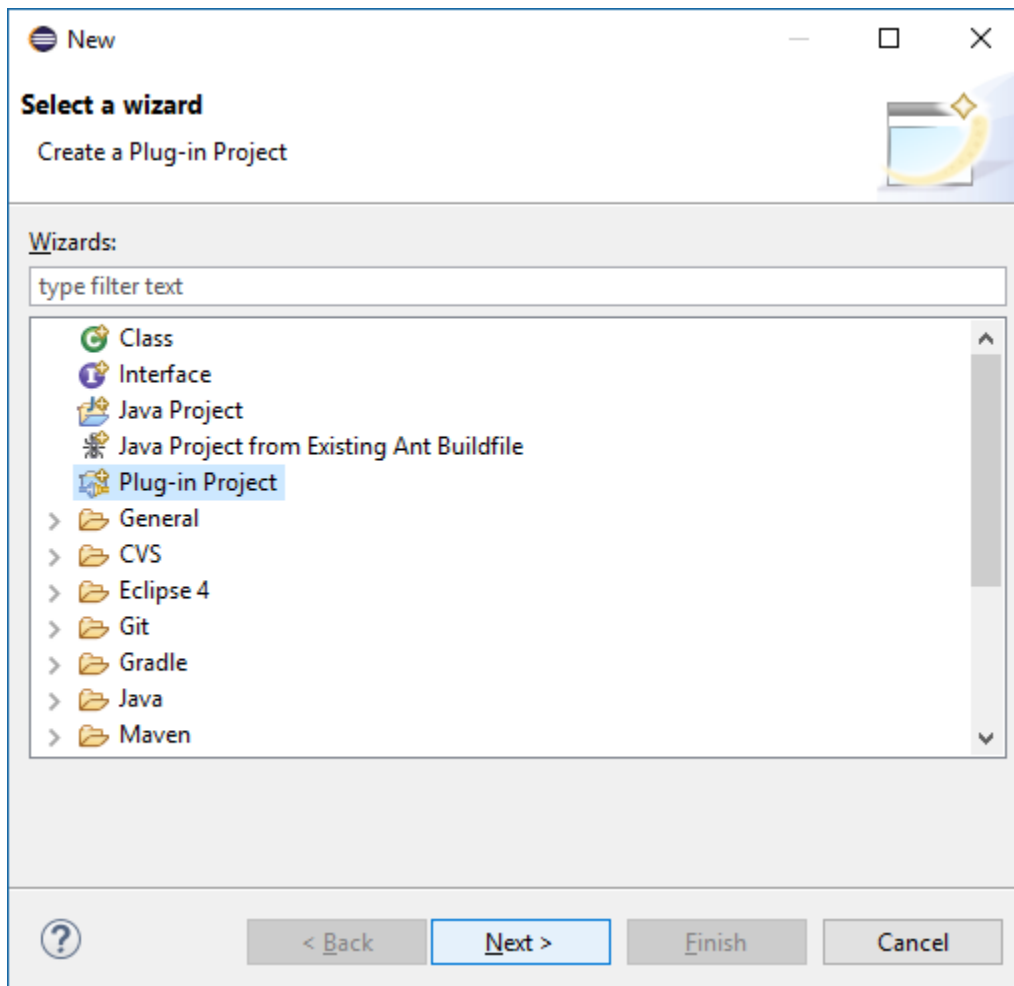
12. There is a launcher for running the built in `JfrRulesReport`. Try running it. There are no arguments set. Can you set the appropriate arguments to get xml, html and text reports for the `latencies_before.jfr` recording?

Creating Rules

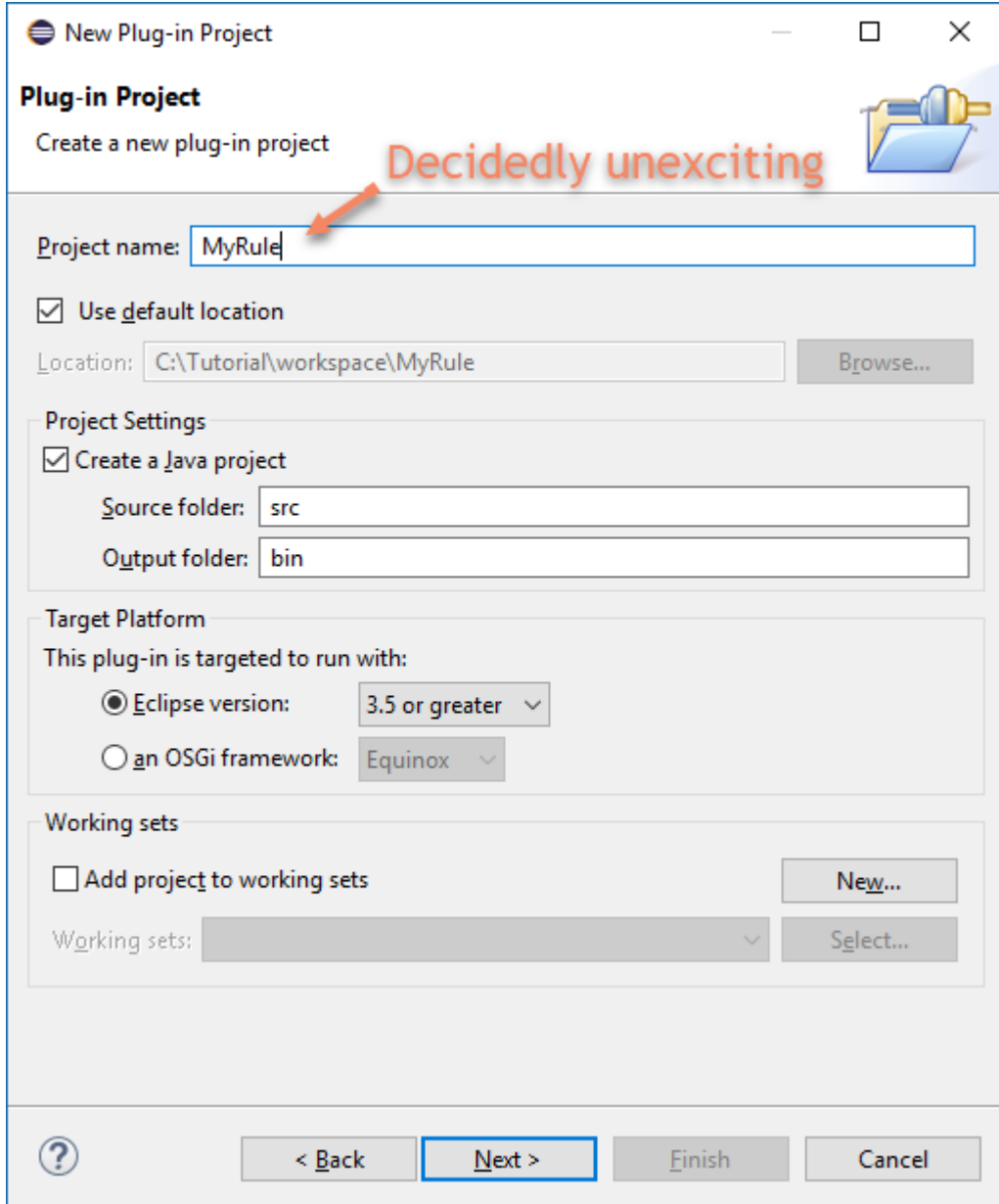
The easiest way to get started creating rules is to use the PDE plug-in for Eclipse.

Note: The JMC PDE plug-in should already be installed in your lab environment. That said, if you are running this Tutorial in your own Eclipse environment, it can be found at the experimental update site (look for it at <http://oracle.com/missioncontrol>).

Press ctrl-n (or click the **File | New | Other...** menu) to bring up the **New** wizard.



Select **Plug-in Project** and hit **Next**. Name your rule project something exciting.



New Plug-in Project

Plug-in Project
Create a new plug-in project

Project name:

☒ Use default location
Location:

Project Settings
☒ Create a Java project
Source folder:
Output folder:

Target Platform
This plug-in is targeted to run with:
☒ Eclipse version:
☐ an OSGi framework:

Working sets
☐ Add project to working sets
Working sets:

Unclick that this plug-in will make contributions to the UI and hit **Next**.

New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID: MyRule
Version: 1.0.0.qualifier
Name: MyRule
Vendor:
Execution Environment: JavaSE-1.8 **Environments...**

Options

☐ Generate an activator, a Java class that controls the plug-in's life cycle
Activator: myrule.Activator

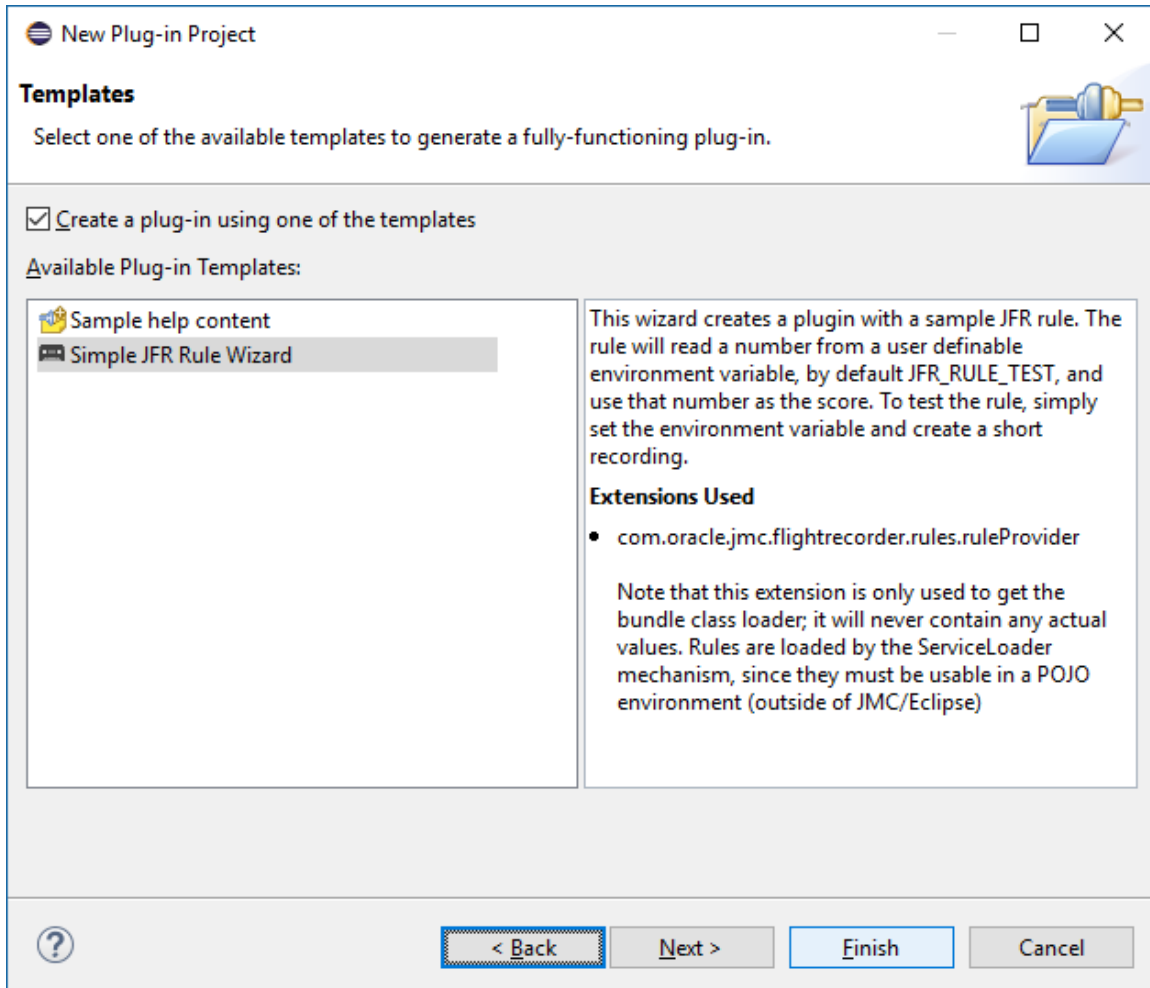
☒ This plug-in will make contributions to the UI
☐ Enable API analysis

Rich Client Application
Would you like to create a rich client application? ☐ Yes ☒ No

Nope

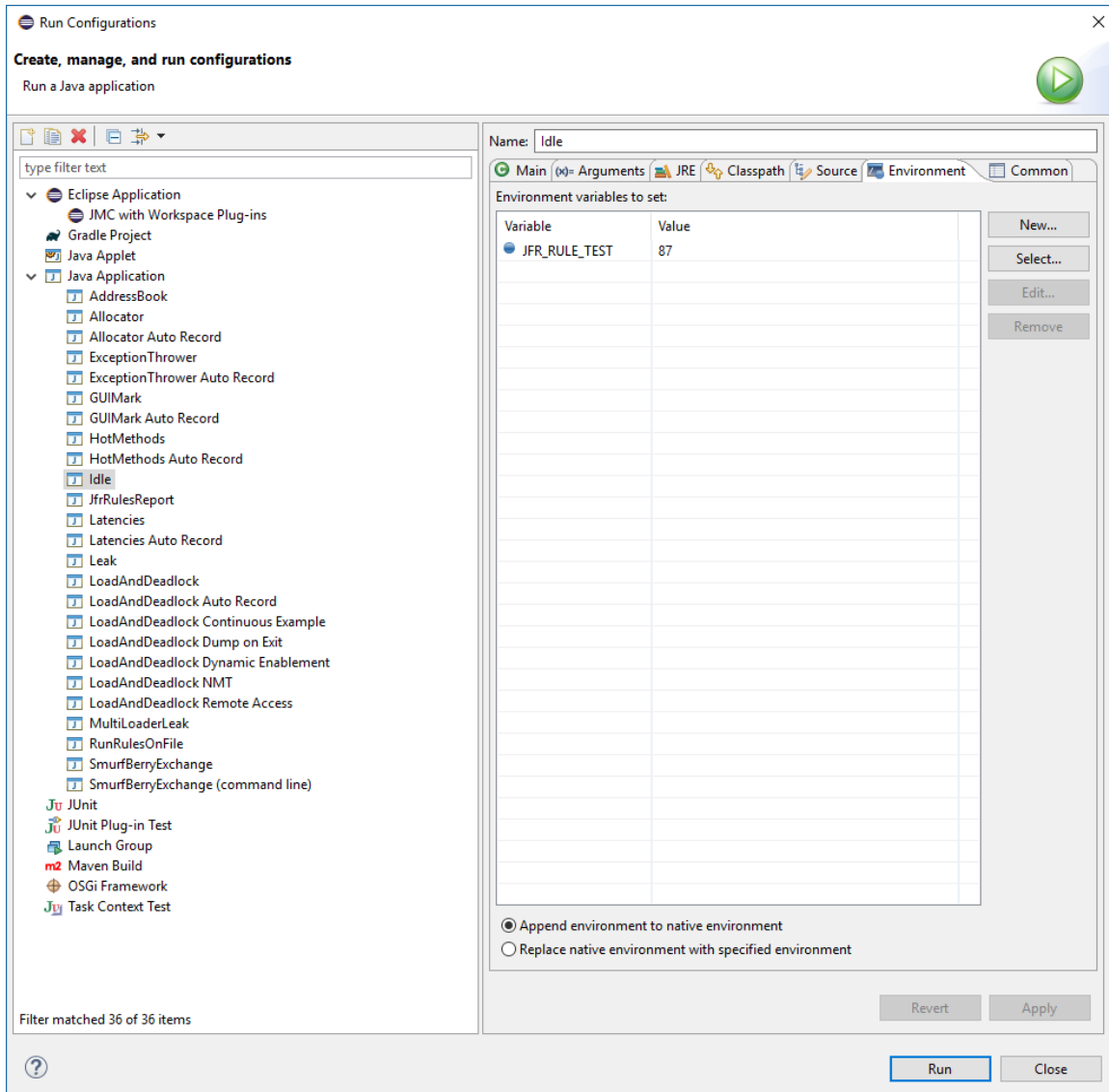
Next >

Next select the **Simple JFR Rule Wizard** and click **Finish** (or **Next**, if you really wish to do some further customizations).

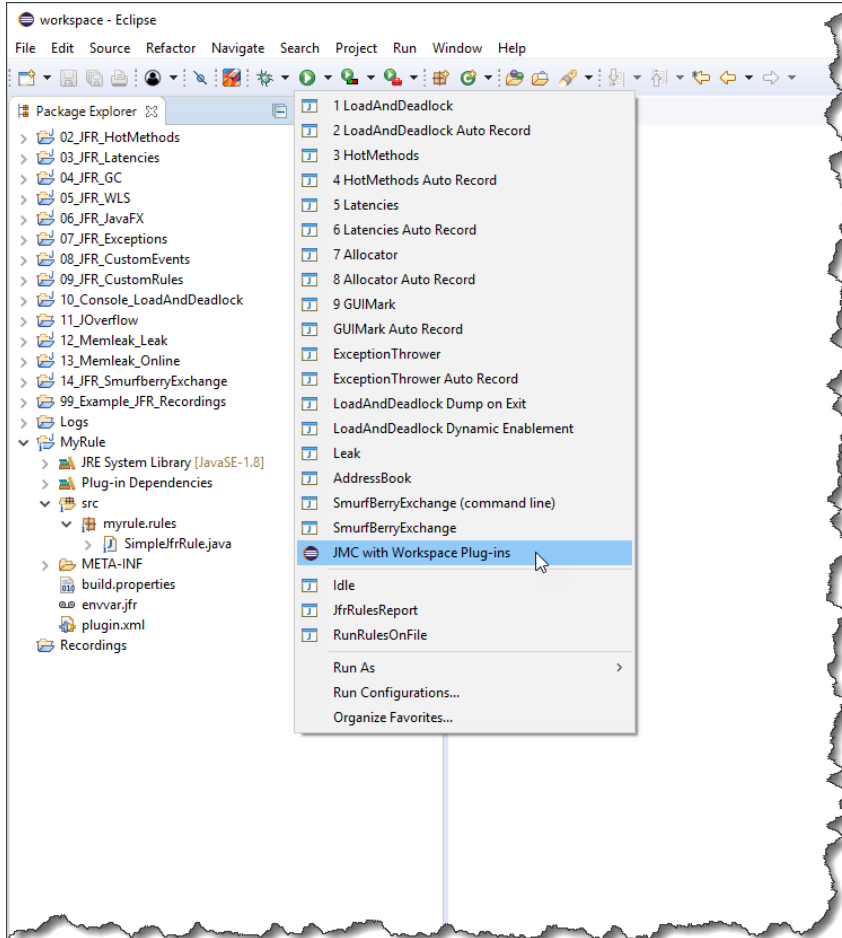


You will now have a new project in your workspace. Open up the project and study the source of the rule. Can you see what it does?

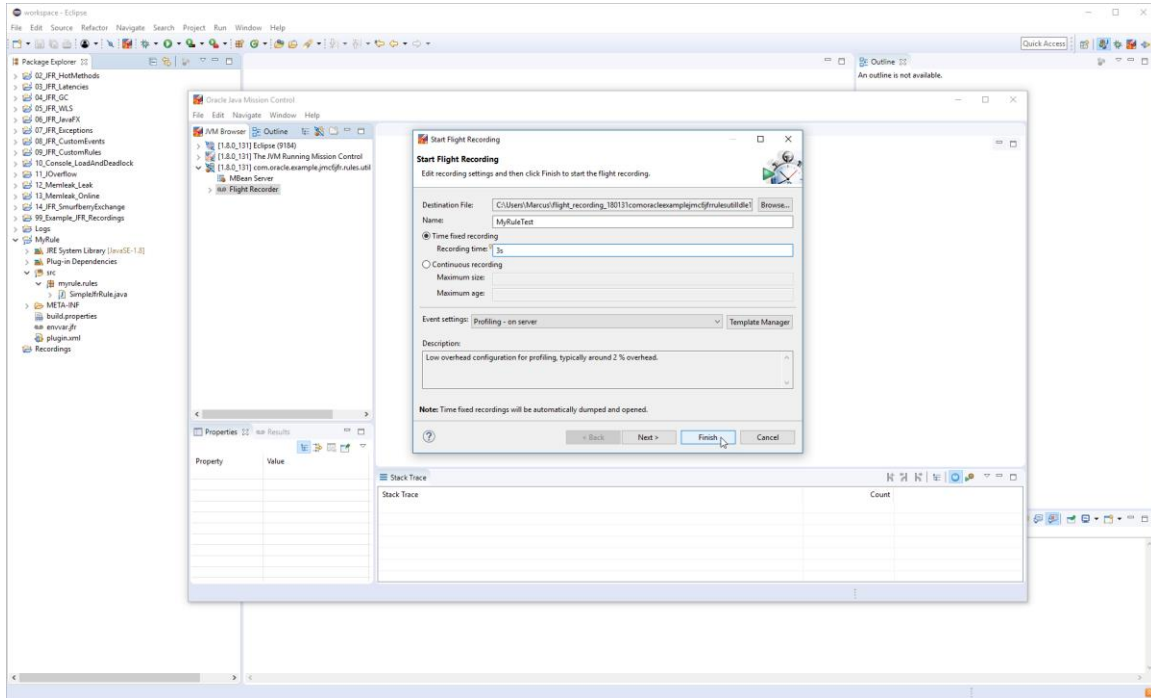
Go to the **Run Configurations** and check out the **Idle** launcher. In the **Environment** tab you can set up an environment variable. Launch the **Idle** launcher by hitting **Run**.



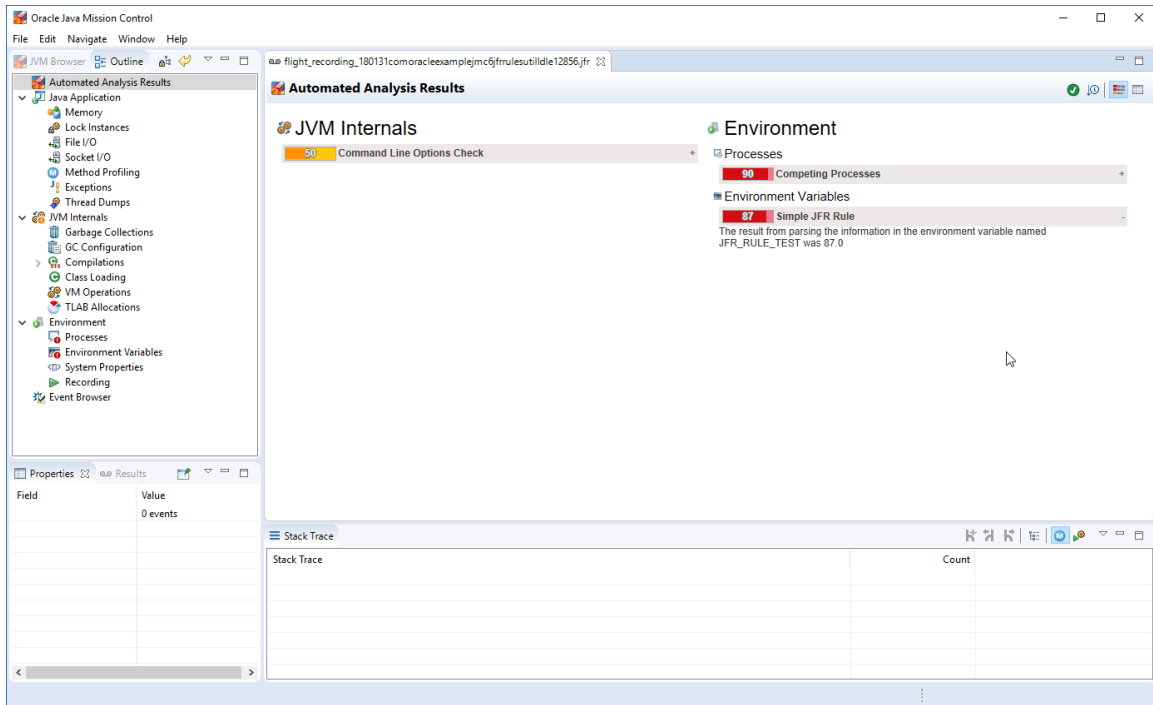
Next launch a JMC with your rule running by launching JMC with Workspace Plug-ins.



If you get a validation complaint, it will be about localization – simply click **Continue**. In the Java Mission Control client that now starts, find your **Idle** program and make a short (3s or so) recording with the default profiling template.

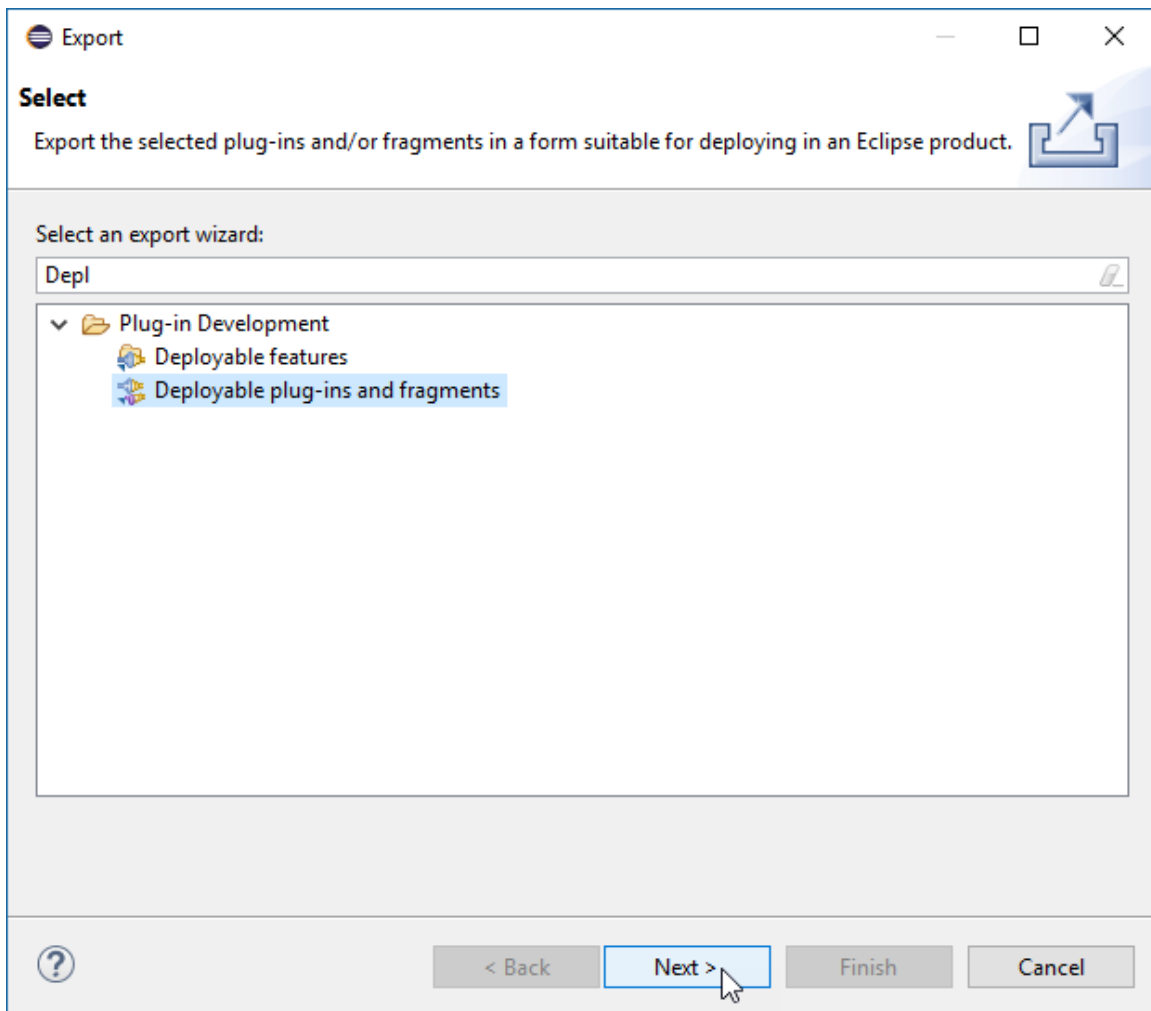


Depending on the value you set the environment variable to, you should now see your rule triggering:



Exporting the rule

The rule can be exported and shared with others. To export it, simply right click on the rule project and select **Export....** In the Export wizard, type Depl in the filter box, and select **Deployable plug-ins and fragments**.



Hit **Next**, select a destination directory and click **Finish**. The resulting plug-in can either be dropped in a JMC dropins folder (**JDK9_HOME/lib/missioncontrol/dropins**), or put on the classpath to a program running the automated analysis headless.

Deep Dive Exercises:

13. Duplicate the RunRulesOnFile launcher by right clicking on it in the **Run Configurations** dialog. In the new launcher, set the arguments to `C:\Tutorial\workspace\09_JFR_CustomRules\ruletest.jfr 51`, and add the exported plug-in to the class path (**Add External Jars**). Run it!
14. Do the same for the JfrRulesReport.

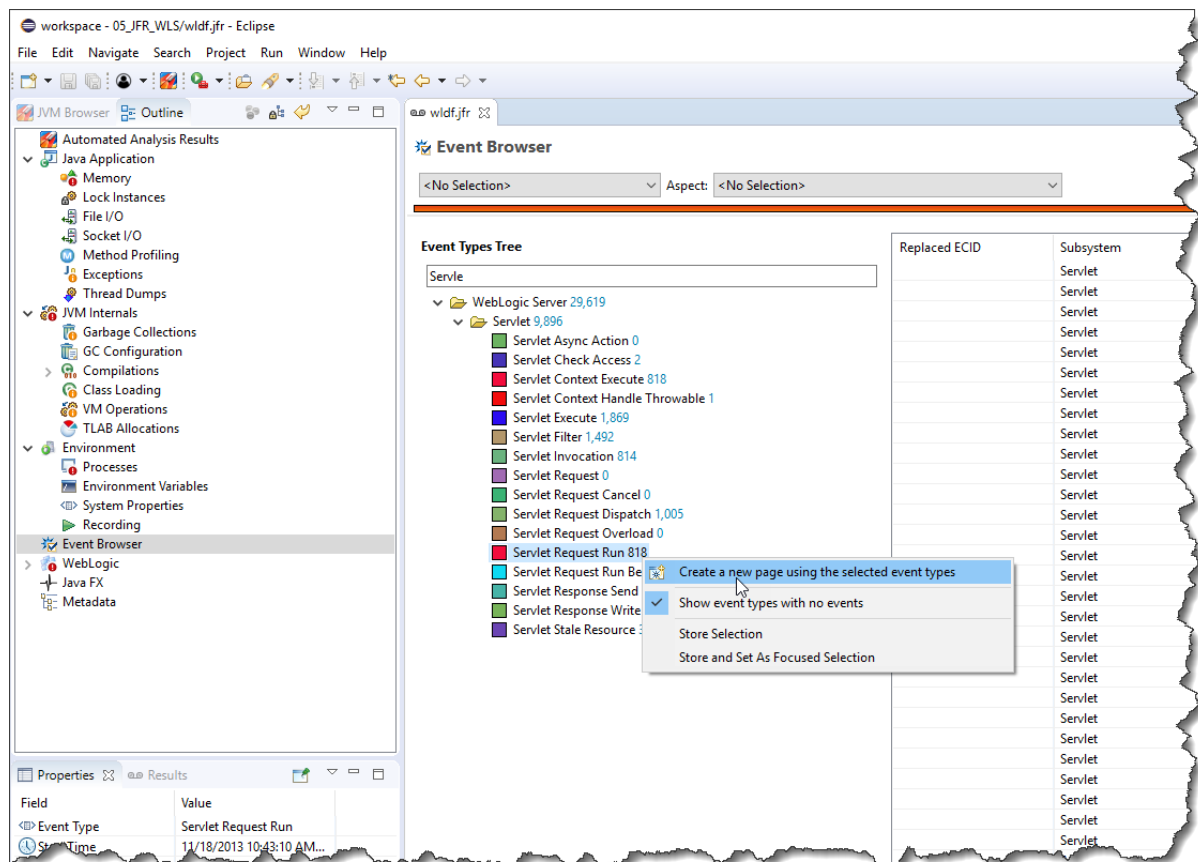
Exercise 9b – Custom Pages

After some time with Java Flight Recorder you may find yourself repeatedly wanting to look at specific pieces of information. JMC 6 provides an easy way to set up custom views.

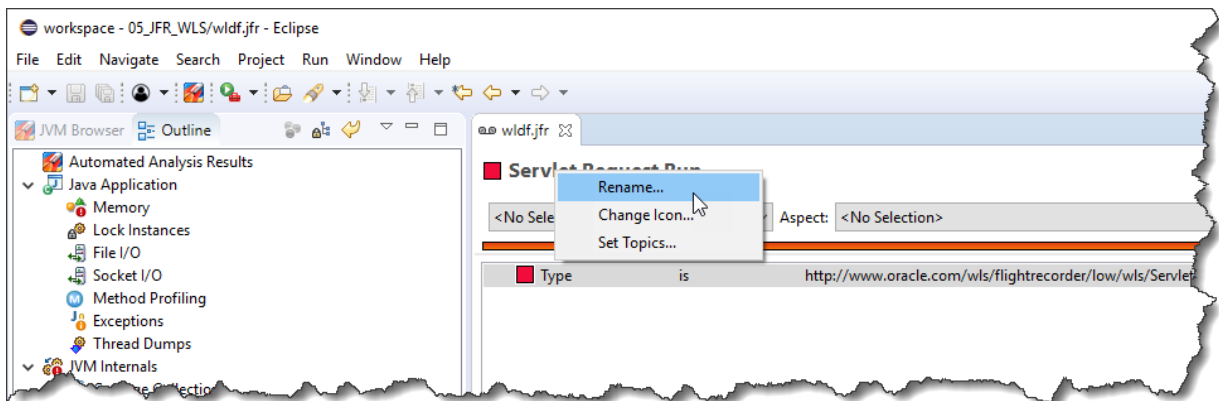
We will use the recording from the **05_JFR_WLS** project for this exercise. Open the **wldf.jfr** recording, and switch to the Java Mission Control perspective.

Filters

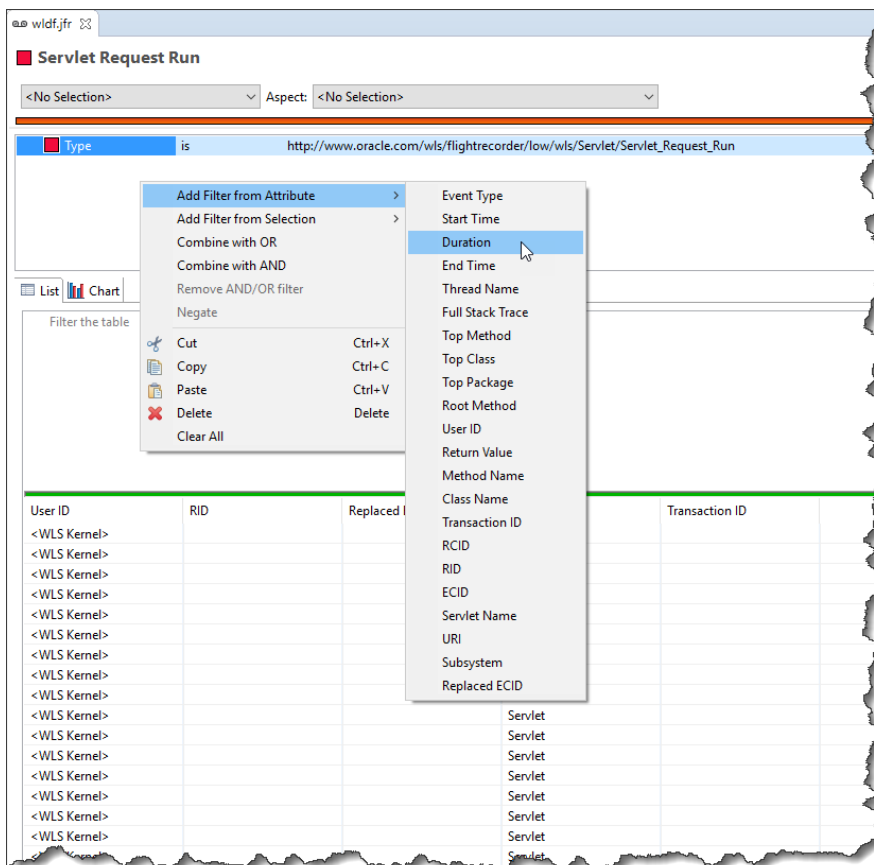
We would like a nice view of the servlet requests taking longer than 2 seconds. Start by going to the **Event Browser** page. Use the filter box to quickly find the **Servlet Request Run** event type, and then use the context menu to create a new page using the selected event type.



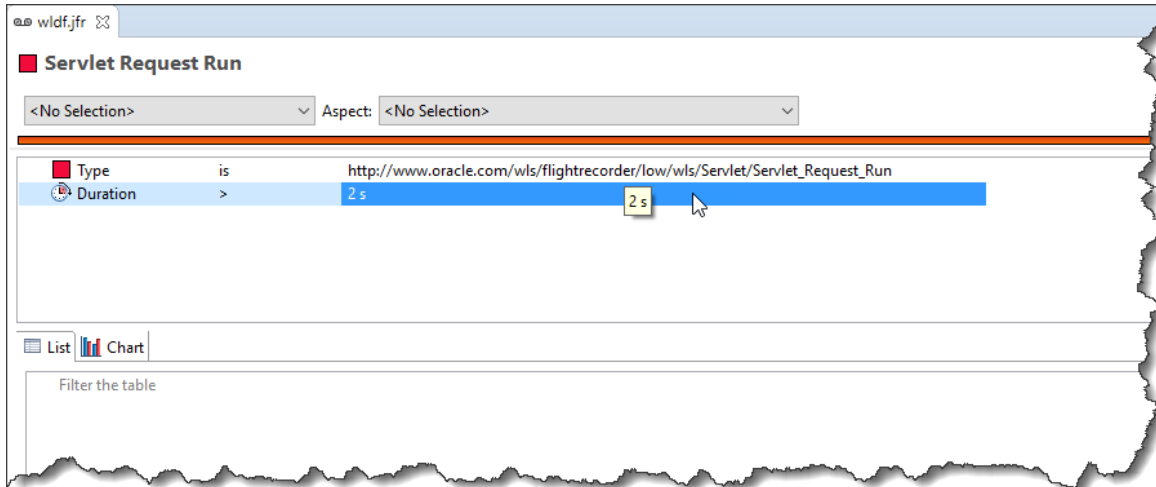
Right click on the name of the new page, and name it “Long Lasting Servlets”:



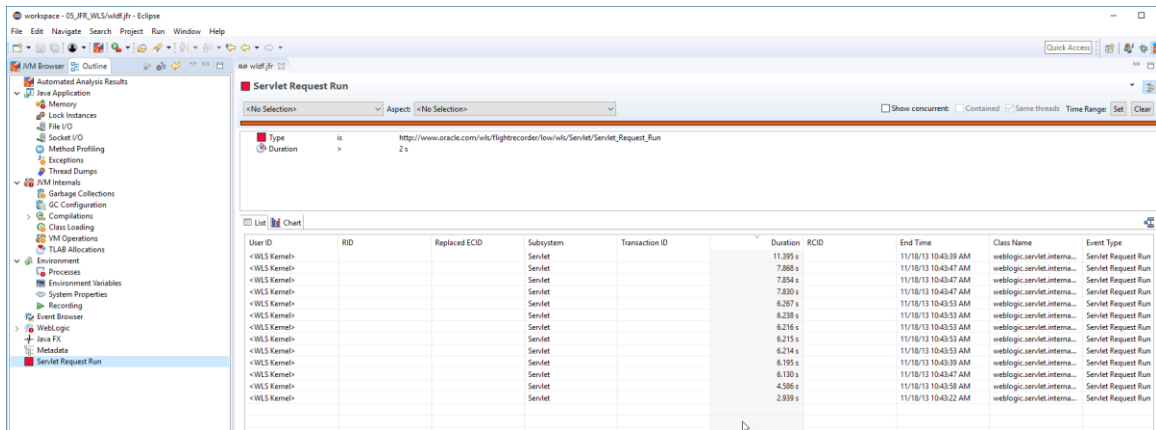
Add a new attribute filter on Duration:



And set it to > 2 seconds:

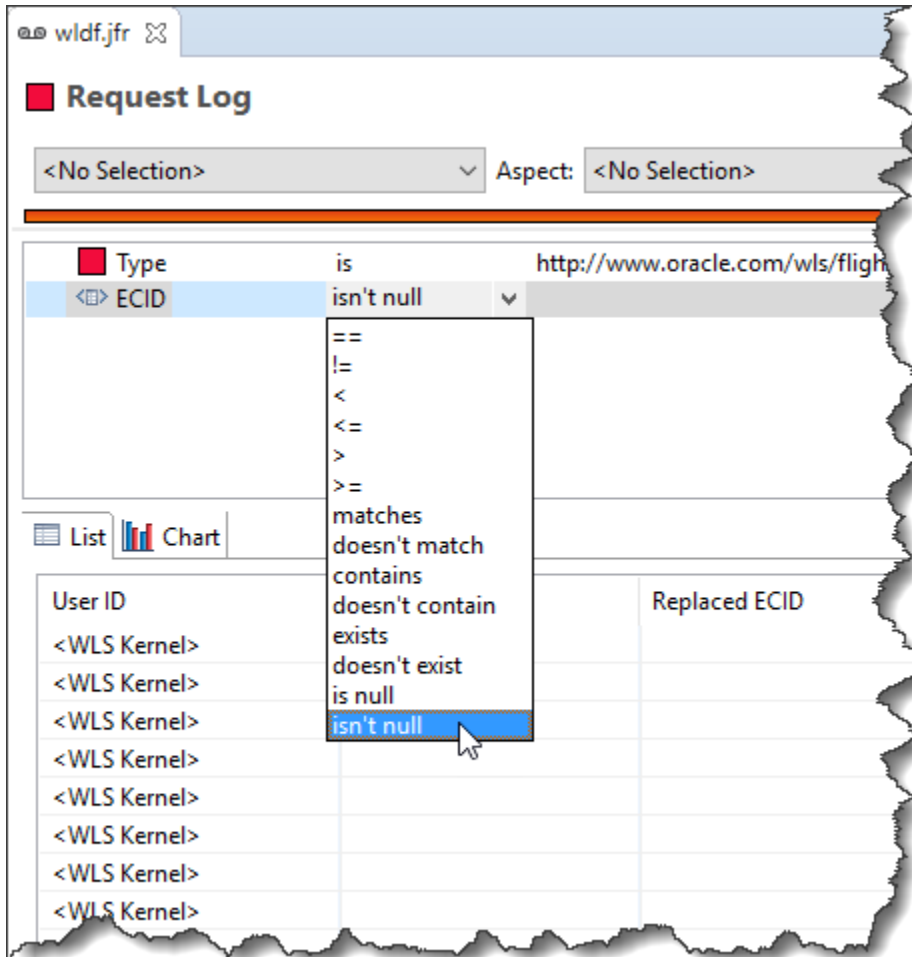


Now we have a custom page showing the longest lasting servlet requests:

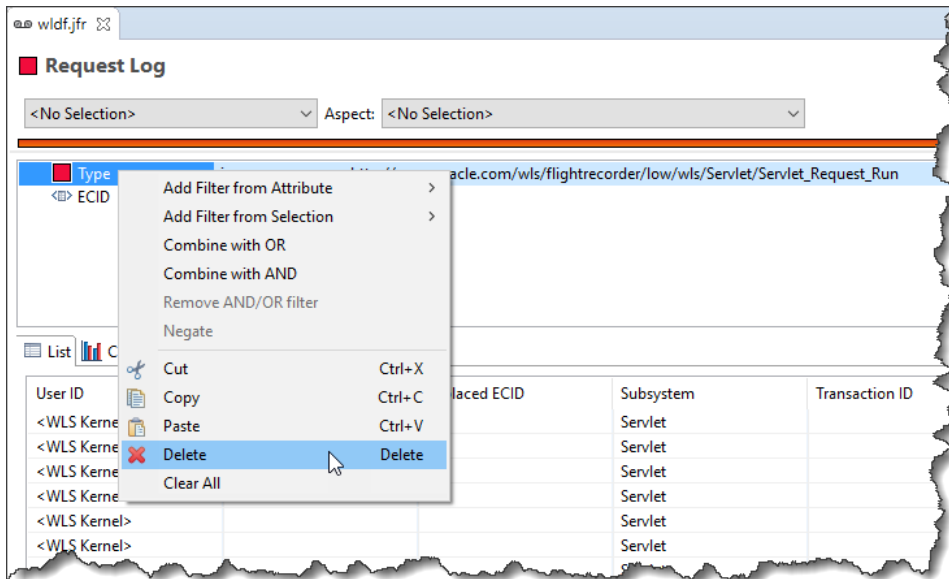


Grouping

In the custom page, events can also be grouped. Create a new page, using the **Servlet Request Run** event again. Name the page Request Log. Add a new filter from attribute, this time **ECID**. Select **isn't null** as the predicate relation.



Now remove the Type filter:



Add a column for the longest duration in the ECID table:

The screenshot shows a web application interface for a 'Request Log'. At the top, there are two dropdown menus: '<No Selection>' and 'Aspect: <No Selection>'. Below these, there's a section for 'ECID' with a filter 'isn't null'. The main part of the interface is a table with columns 'ECID' and 'Count'. The 'ECID' column contains several long hexadecimal strings, and the 'Count' column shows the value '510'. A context menu is open over the 'ECID' column, showing options like 'Sort Columns', 'Visible Columns', 'Copy', 'Clipboard Settings', 'Store Selection', 'Store and Set As Focused Selection', 'Group By', and 'Combine Group By'. The 'Longest Duration' option is highlighted in the 'Sort Columns' submenu. Below the table, there's a 'List' view section with columns 'User ID' and 'RID'. The 'User ID' column contains '<anonymous>' and the 'RID' column contains 'EJB'. The 'List' view is selected, and the 'ECID' column is highlighted in the table header.

Next sort the **ECID** table on the **Longest Duration** column. Select the longest lasting ECID, and next sort the **List**, which will now show all events with the ECID selected in the **ECID** table, on **Start Time**.

After some time rearranging the columns in the List to your liking, you should now have a nice Log of what was happening, in Start time order, for any ECID you select.

Request Log

Aspect: <No Selection> Show concurrent: ☐ Contained ☒ Same threads Time Range: Set Clear

ECID isn't null

ECID	Count	Longest Duration
8ec006a7-30e9-4fac-be7f-716f4243cb8-00000092	98	11.395 s
8ec006a7-30e9-4fac-be7f-716f4243cb8-000000a5	71	7.868 s
8ec006a7-30e9-4fac-be7f-716f4243cb8-000000a3	75	7.854 s
8ec006a7-30e9-4fac-be7f-716f4243cb8-000000a4	68	7.830 s
8ec006a7-30e9-4fac-be7f-716f4243cb8-000000b7	87	6.267 s

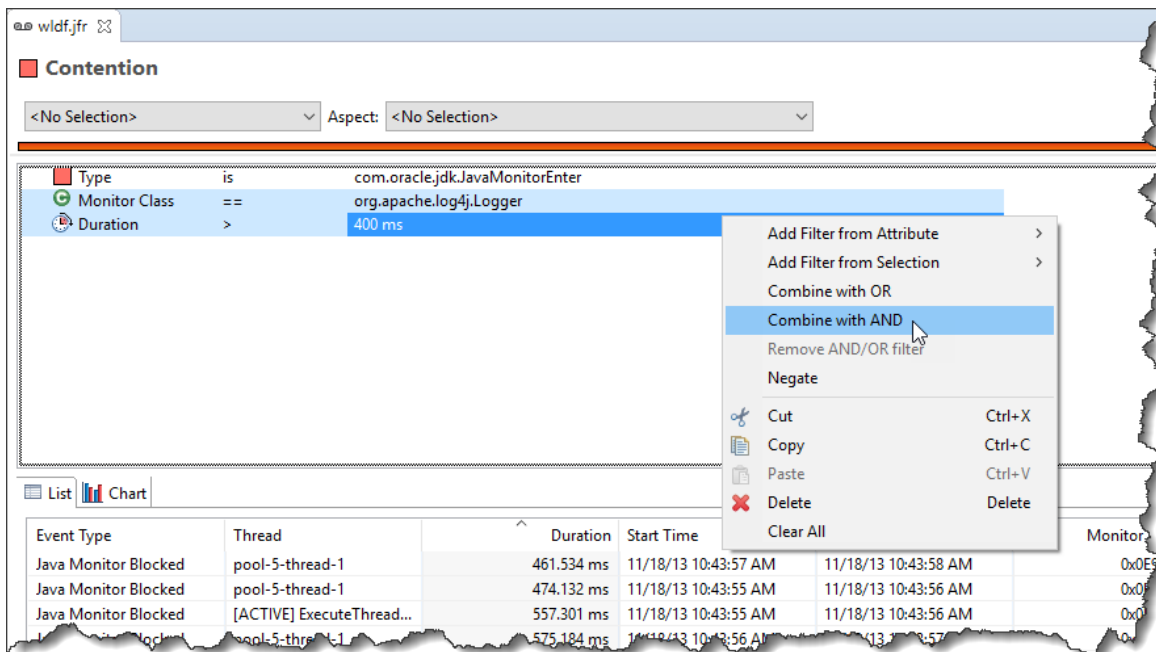
Start Time	Event Type	Duration	Servlet Name	URI	Subsystem	Method Name	User ID	Class Name
11/18/13 10:43:27 AM	ECID Range	11.395 s						
11/18/13 10:43:27 AM	Servlet Request Run Beg...	0 s	FacesServlet	/physician-web/login.action	Servlet	run	<WLS Kernel>	weblogic.servlet.internal.ServletRequest...
11/18/13 10:43:27 AM	Servlet Request Run	11.395 s	FacesServlet	/physician-web/login.action	Servlet	run	<WLS Kernel>	weblogic.servlet.internal.ServletRequest...
11/18/13 10:43:27 AM	Servlet Context Execute	11.393 s	FacesServlet	/physician-web/login.action	Servlet	execute	<WLS Kernel>	weblogic.servlet.internal.WebAppServlet...
11/18/13 10:43:27 AM	Servlet Invocation	11.392 s	FacesServlet	/physician-web/login.action	Servlet	wrapRun	<anonymous>	weblogic.servlet.internal.WebAppServlet...
11/18/13 10:43:27 AM	Servlet Filter	11.392 s	FacesServlet	/physician-web/login.action	Servlet	doFilter	<anonymous>	weblogic.servlet.internal.RequestEventsFi...
11/18/13 10:43:27 AM	Servlet Filter	11.392 s		/physician-web/login.action	Servlet	doFilter	<anonymous>	weblogic.servlet.internal.TailFilter
11/18/13 10:43:27 AM	Servlet Execute	11.392 s		/physician-web/login.action	Servlet	execute	<anonymous>	weblogic.servlet.internal.ServletStubImpl
11/18/13 10:43:27 AM	EJB Business Method In...	11.238 s			EJB	invoke	<anonymous>	weblogic.ejb.container.internal.Session...
11/18/13 10:43:27 AM	EJB PoolManager Create	17.787 ms			EJB	createBean	<anonymous>	weblogic.ejb.container.manager.States...
11/18/13 10:43:33 AM	EJB Pool Manager Pre In...	0 s			EJB	preInvoke	<anonymous>	weblogic.ejb.container.internal.BaseLoca...
11/18/13 10:43:33 AM	EJB Business Method Pr...	0 s			EJB	_WL_preInvoke	<anonymous>	weblogic.ejb.container.internal.Session...
11/18/13 10:43:33 AM	EJB Business Method In...	11.872 ms			EJB	invoke	<anonymous>	weblogic.ejb.container.internal.Session...
11/18/13 10:43:33 AM	EJB PoolManager Create	8.207 ms			EJB	createBean	<anonymous>	weblogic.ejb.container.manager.States...
11/18/13 10:43:33 AM	EJB Pool Manager Pre In...	0 s			EJB	preInvoke	<anonymous>	weblogic.ejb.container.manager.States...
11/18/13 10:43:33 AM	EJB Business Method Pr...	0 s			EJB	_WL_preInvoke	<anonymous>	weblogic.ejb.container.internal.BaseLoca...
11/18/13 10:43:33 AM	EJB Business Method In...	368.330 µs			EJB	invoke	<anonymous>	weblogic.ejb.container.internal.Session...
11/18/13 10:43:33 AM	EJB Pool Manager Pre In...	0 s			EJB	preInvoke	<anonymous>	weblogic.ejb.container.manager.States...
11/18/13 10:43:33 AM	EJB Business Method Pr...	0 s			EJB	_WL_preInvoke	<anonymous>	weblogic.ejb.container.internal.BaseLoca...
11/18/13 10:43:33 AM	EJB Business Method Po...	0 s			EJB	_WL_postInvokeToRetry	<anonymous>	weblogic.ejb.container.internal.BaseLoca...
11/18/13 10:43:33 AM	EJB Pool Manager Post L...	0 s			EJB	postInvoke	<anonymous>	weblogic.ejb.container.manager.States...
11/18/13 10:43:33 AM	EJB Business Method Po...	5.749 µs			EJB	_WL_postInvokeCleanup	<anonymous>	weblogic.ejb.container.internal.BaseLoca...
11/18/13 10:43:33 AM	EJB Business Method Po...	0 s			EJB	postInvoke	<anonymous>	weblogic.ejb.container.manager.States...
11/18/13 10:43:33 AM	EJB Business Method Po...	750.619 µs			EJB	_WL_postInvokeCleanup	<anonymous>	weblogic.ejb.container.internal.BaseLoca...
11/18/13 10:43:39 AM	Servlet Request Run	66.104 ms	PhysicianFacadeService...	/medrec-jaws-services/PhysicianFacade...	Servlet	run	<WLS Kernel>	weblogic.servlet.internal.ServletRequest...
11/18/13 10:43:39 AM	Servlet Context Execute	65.477 ms	PhysicianFacadeService...	/medrec-jaws-services/PhysicianFacade...	Servlet	execute	<WLS Kernel>	weblogic.servlet.internal.WebAppServlet...
11/18/13 10:43:39 AM	Servlet Invocation	64.754 ms	PhysicianFacadeService...	/medrec-jaws-services/PhysicianFacade...	Servlet	wrapRun	<anonymous>	weblogic.servlet.internal.WebAppServlet...
11/18/13 10:43:39 AM	Servlet Execute	64.649 ms	PhysicianFacadeService...	/medrec-jaws-services/PhysicianFacade...	Servlet	execute	<anonymous>	weblogic.servlet.internal.ServletStubImpl
11/18/13 10:43:39 AM	EJB Business Method In...	30.967 ms			EJB	invoke	<anonymous>	weblogic.ejb.container.internal.Session...

Boolean Filter Operations

Lastly we will build a custom view to find any contention on Log4J lasting longer than 400ms, or 200ms if the contention is somewhere else, as this will neatly illustrate the use of Boolean filter operations.

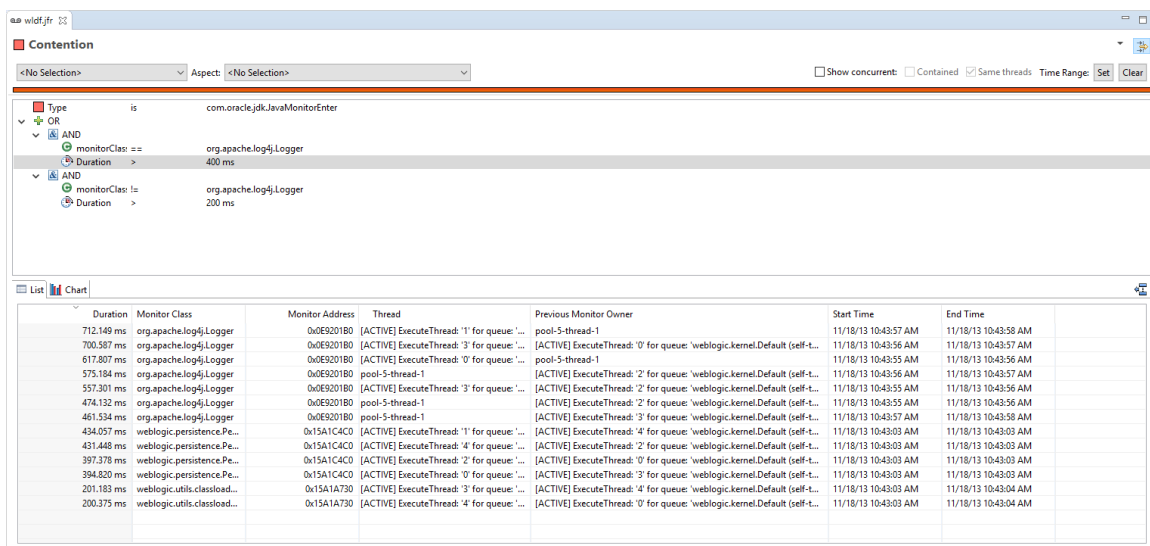
First create a new page on the Java Monitor Blocked event type. Go to the Event Browser. Next use the filter box to quickly find the event type. Use the context menu to create the new page. Name the new page **Log4J Contention**.

Use the context menu in the **List** table to show the search box. (This is available in all tables.)

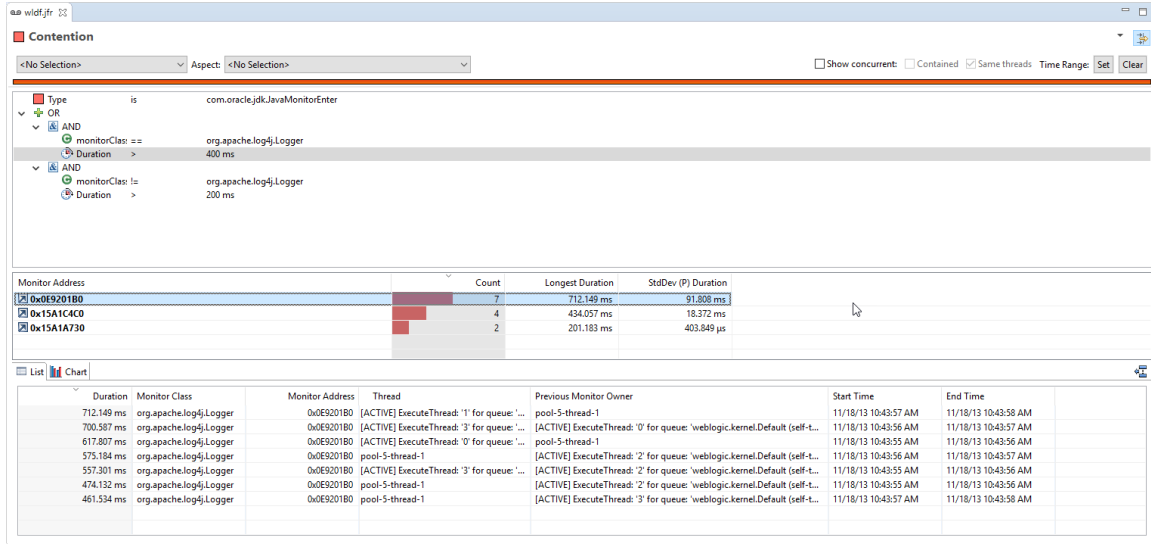


Select the **Monitor Class** filter and the **Duration** filter both, and select **Combine with AND** from the context menu. Next add a filter for Monitor Class != org.apache.log4j.Logger, and Duration > 200 ms. Combine them with an AND filter, and finally combine the both AND filters with an OR filter.

***Note:** There is a bug (fixed in JMC 6.1.0) that makes applying AND/OR filters sometimes not immediately evaluate the expression. If that happens, try clicking in the list or select another page and return.*



Adding grouping on monitor address would yield something like this:



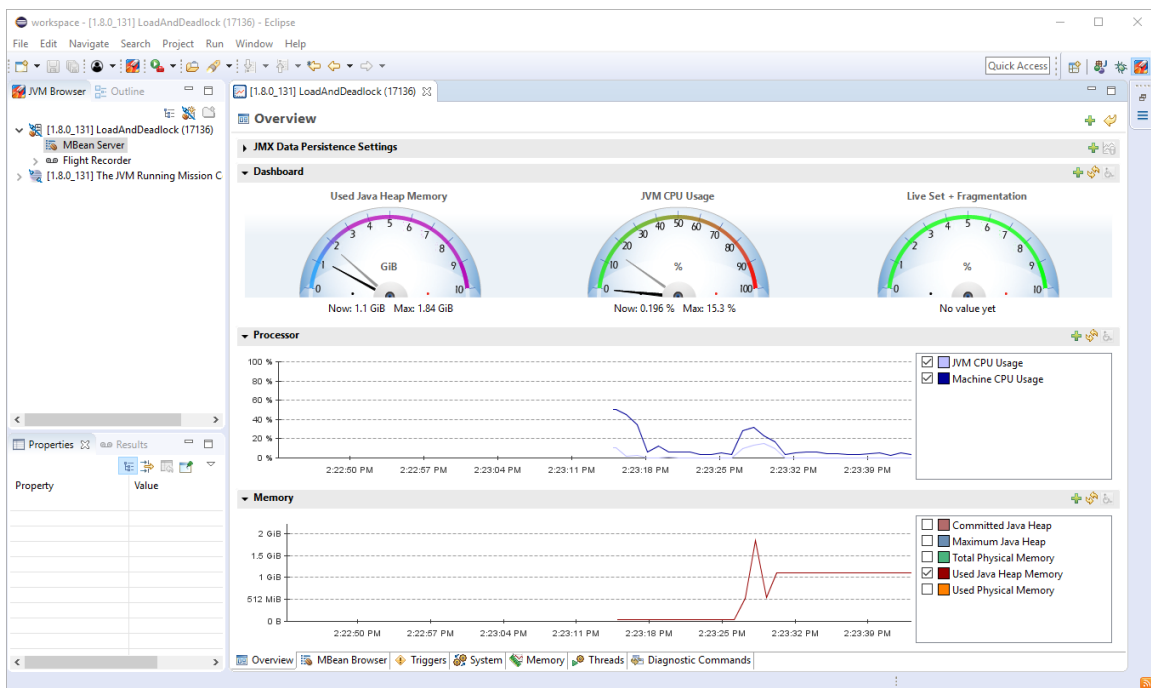
The Management Console (Bonus)

Java Mission Control includes a very handy JMX console. It has been described as a “JConsole on steroids”, and it certainly has some very convenient features. The next few exercises will show some of the more commonly used ones.

Exercise 10.a – The Overview

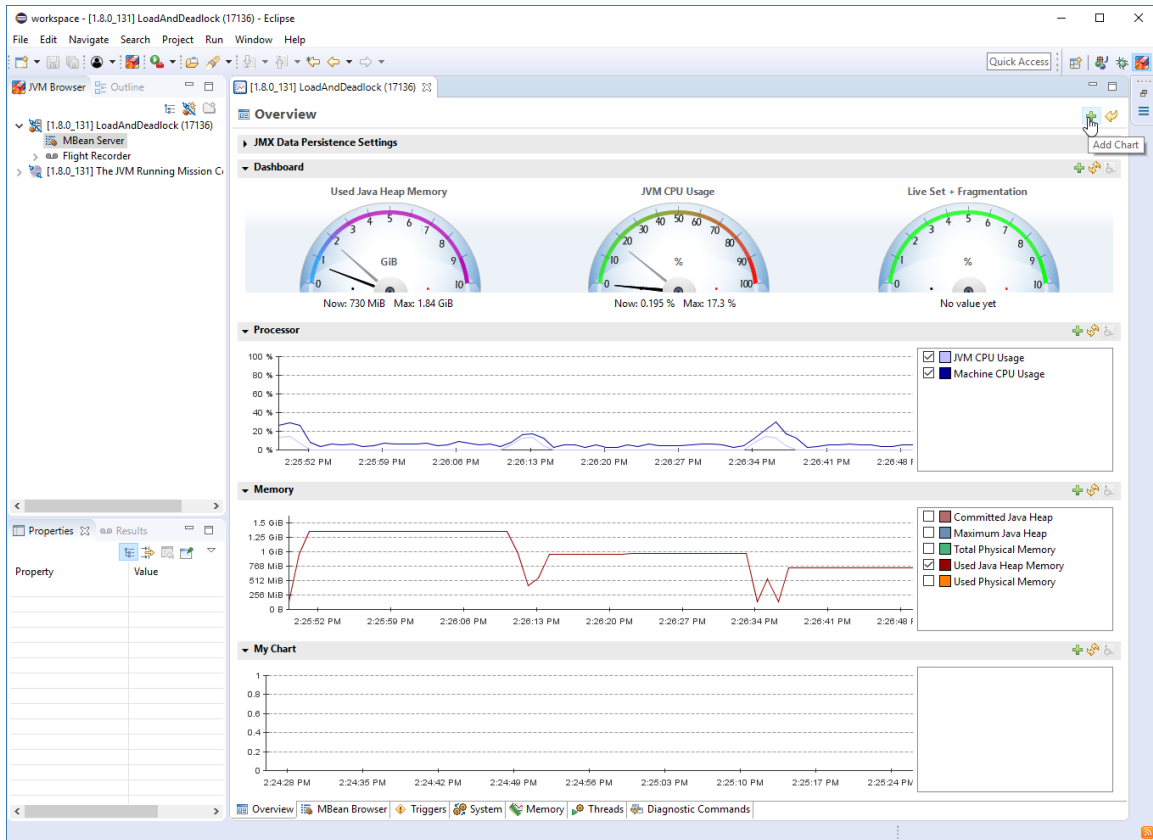
Start the LoadAndDeadlock program, like you did in Exercise 2.a. Then switch to the **Mission Control** perspective. After a little while you should see the JVM running the LoadAndDeadlock class appearing in the **JVM Browser**. Open a console by selecting **Start JMX Console** from the context menu of the JVM running the LoadAndDeadlock class, or by expanding the **LoadAndDeadlock** JVM in the **JVM Browser** and double clicking on the MBean Server.

You should now be at the Overview tab of the Management Console. You should see something similar to the picture below:

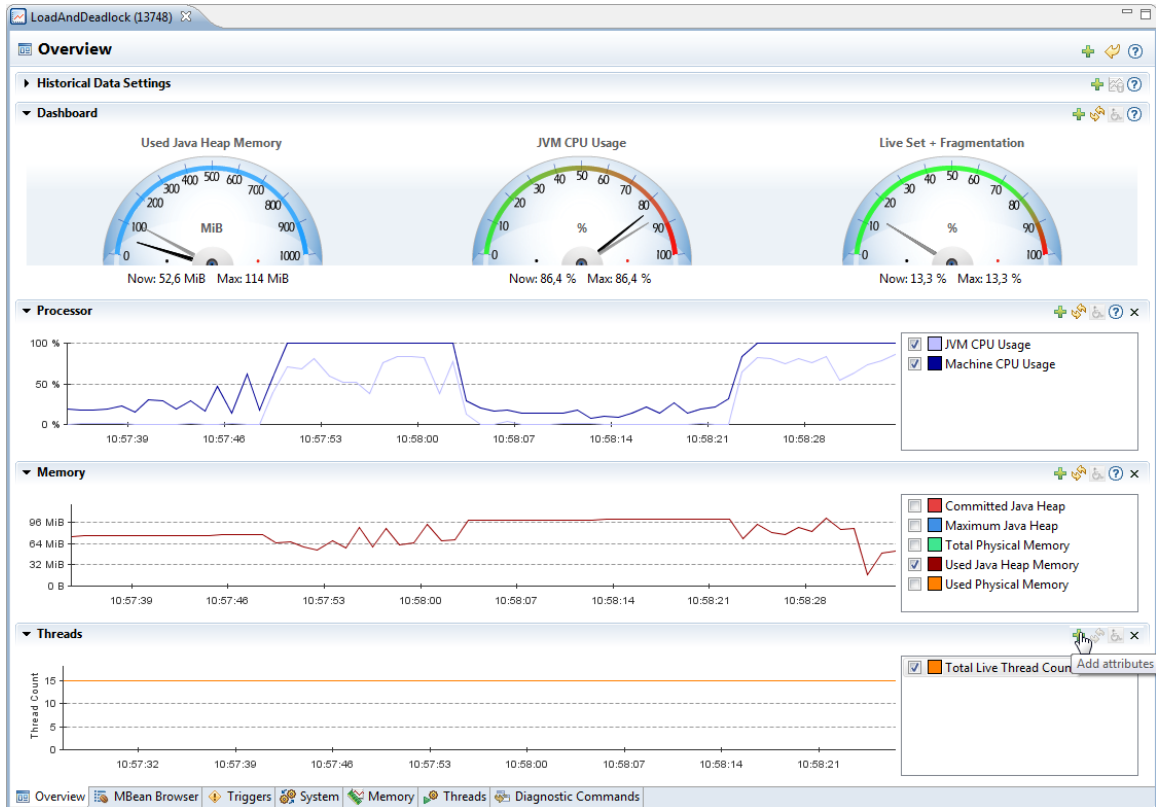


In the overview tab you can remove charts, add new charts, add attributes to the charts, plot other attributes in the velocimeters, log the information in the charts to disk, freeze the charts to look at specific values, zoom and more.

Click on the Add chart button in the upper right corner of the console. This will add a new blank chart to JMC.



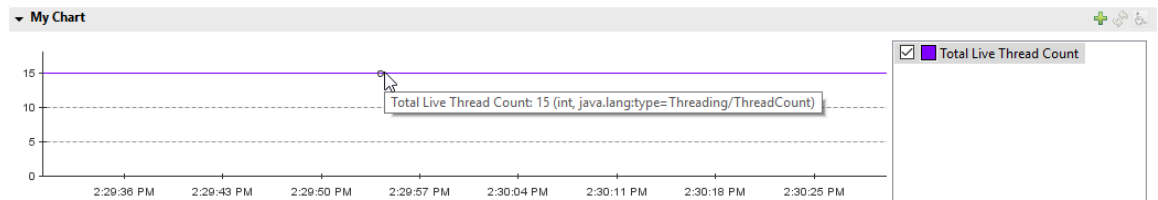
Click the **Add...** button of the new chart. In the attribute selector dialog, go to the **Filter** text field and enter “Th” (without quotation marks). Select the **ThreadCount** attribute, and press ok. You should now see the thread count.



Note: You can use the context menu in the attribute list to change the color of the thread count graph. To change the titles in the chart, use the context menu of the chart.

Deep Dive exercises:

15. Try changing the color of the chart.
16. Sometimes it can be hard to read the precise value in a chart. Freeze the graph and hover with the pointer over the thread count graph for a little while. What is your exact thread count?

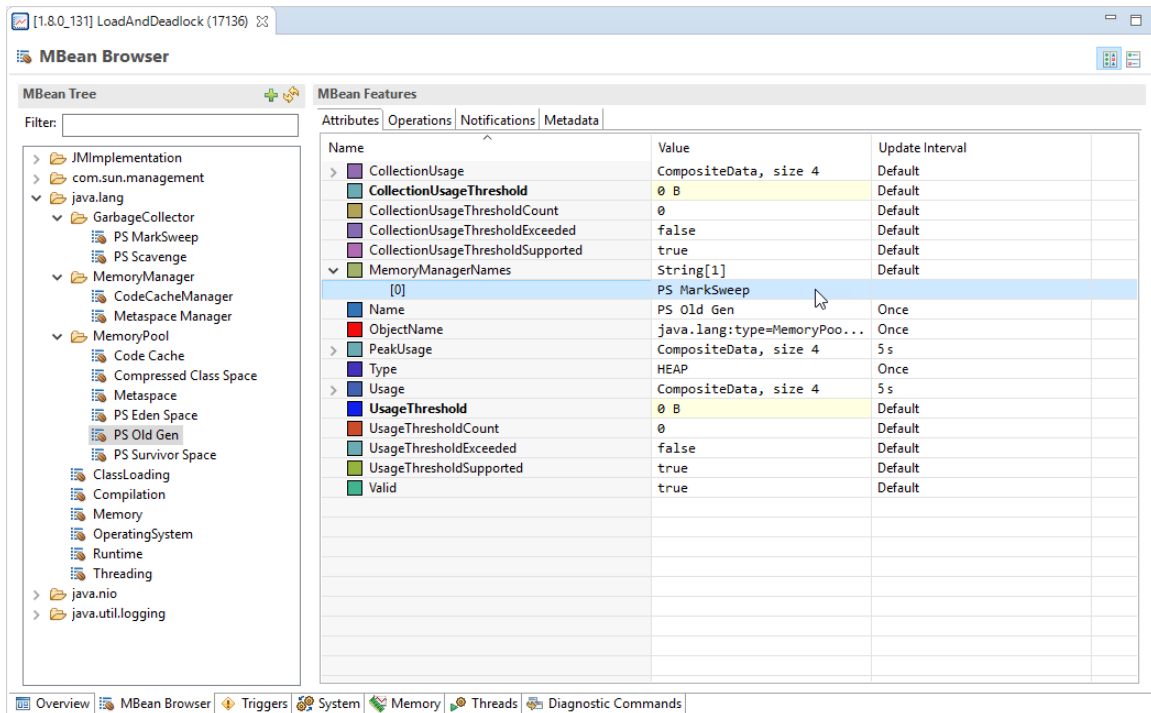


17. You decide that you dislike the live set attribute and warm up a bit to the Thread Count attribute. Remove the Live Set velocimeter in the upper right corner of the **Overview** tab, and instead add one for the Thread Count attribute.

Exercise 10.b – The MBean Browser

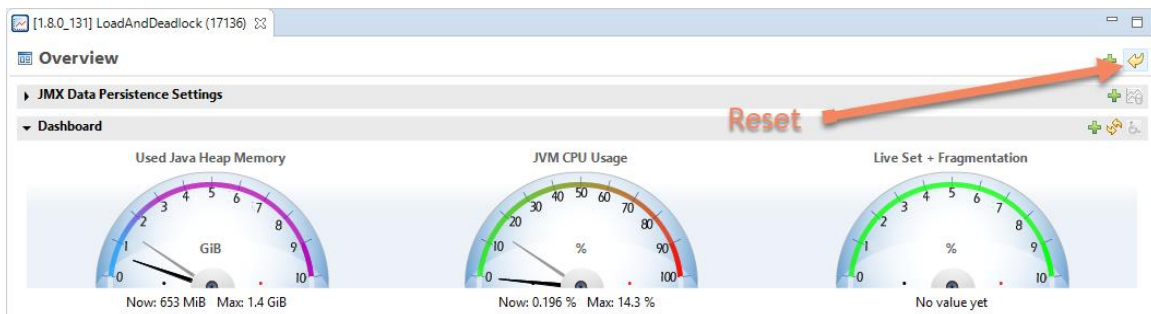
The MBean browser is where you browse the MBeans available in the platform MBean server. If you expose your own application for monitoring through JMX and register them in the platform MBean server, your custom MBeans will show up here. You can use the MBean browser to look at specific values of attributes, change the update times for attributes, add attributes to charts, execute operations and more. Go to the MBean browser by clicking the **MBean Browser** tab.

What is the current garbage collection strategy for the old generation?



Note: Go to the `java.lang` domain, select the proper memory pool MBean and look for the `MemoryManagerNames` attribute in the **Attribute** table.

Whilst browsing the `java.lang.Threading` MBean, you encounter your old friend the `ThreadCount` attribute. You decide that you enjoy it so much that you wish to add it to yet another chart on the **Overview** tab. Right click on the attribute, select **Visualize...** Select **Add new chart** and click **OK**. Go back to the **Overview** tab and enjoy the Dual ThreadCount Plotting Experience™ for a brief moment. Then reset the user interface by clicking the **Reset to Default Controls** button in the upper right corner.



Note: In JMC charts must contain values of the same content type. That is the reason why you cannot plot the `ThreadCount` attribute in the same chart as, say, the `Memory` attributes.

Deep Dive Exercises:

18. Get a thread stack dump by executing the DiagnosticCommand `print_threads`.

Note: Browse to `com.sun.management.DiagnosticCommand`, select the `operations` tab, select the `threadPrint` operation. Press the `Execute` button. You will get a new time-stamped result view for each invocation of an operation.

19. Can you find a much simpler way of executing the Diagnostic Commands?

Note: Use the `Diagnostic Command` tab.

Exercise 10.c – The Threads View

Short on time as we are, we skip to the Threads view. Rejoice at the discovery of our old friend the Thread Count attribute in the upper chart (needs to be unfolded)! In the threads view we can check if there are any deadlocked threads in our application. Turn on **deadlock detection** by checking the appropriate checkbox.

Next click on the **Deadlocked** column header to bring the deadlocked threads to the top.

***Note:** You can also turn off the automatic retrieval of new stack traces by clicking the **Refresh Stack Traces** icon next to the deadlock detection icon on the toolbar. This is usually a good idea while investigating something specific, as you may otherwise be interrupted by constant table refreshes.*

What are the names of the deadlocked threads? In which method and on what line are they deadlocked?

The screenshot shows the Mission Control Threads view for a Java application. The top section displays a table of threads with the following columns: Thread Name, Thread State, Blocked Count, Total CPU Usage, Deadlocked, Allocated Memory, and Lock Owner Name. The threads are sorted by the Deadlocked column, with Thread-4 and Thread-3 at the top, both in a BLOCKED state. Below the table, the 'Stack Traces for Selected Threads' section shows the stack trace for Thread-4 [16] (BLOCKED), which is LoadAndDeadlock\$LockerThread.run line: 41. The bottom toolbar includes icons for Overview, MBean Browser, Triggers, System, Memory, Threads, and Diagnostic Commands.

Thread Name	Thread State	Blocked Count	Total CPU Usage	Deadlocked	Allocated Memory	Lock Owner Name
Thread-4	BLOCKED	1	Not Enabled	true	Not Enabled	Thread-3
Thread-3	BLOCKED	1	Not Enabled	true	Not Enabled	Thread-4
RMI TCP Connection(4)-192.168.56.1	RUNNABLE	461	Not Enabled	false	Not Enabled	
RMI TCP Connection(2)-192.168.56.1	TIMED_WAITING	2,134	Not Enabled	false	Not Enabled	
JMX server connection timeout 21	TIMED_WAITING	6,778	Not Enabled	false	Not Enabled	
RMI Scheduler(0)	TIMED_WAITING	0	Not Enabled	false	Not Enabled	
RMI TCP Accept-0	RUNNABLE	0	Not Enabled	false	Not Enabled	
Thread-2	TIMED_WAITING	0	Not Enabled	false	Not Enabled	
VM JFR Buffer Thread	RUNNABLE	0	Not Enabled	false	Not Enabled	
JFR request timer	WAITING	0	Not Enabled	false	Not Enabled	
Attach Listener	RUNNABLE	0	Not Enabled	false	Not Enabled	
Signal Dispatcher	RUNNABLE	0	Not Enabled	false	Not Enabled	
Finalizer	WAITING	2	Not Enabled	false	Not Enabled	
Reference Handler	WAITING	2	Not Enabled	false	Not Enabled	
main	RUNNABLE	0	Not Enabled	false	Not Enabled	

Stack Traces for Selected Threads

Stack traces for selected threads 2:43:40 PM

Thread-4 [16] (BLOCKED)

- LoadAndDeadlock\$LockerThread.run line: 41

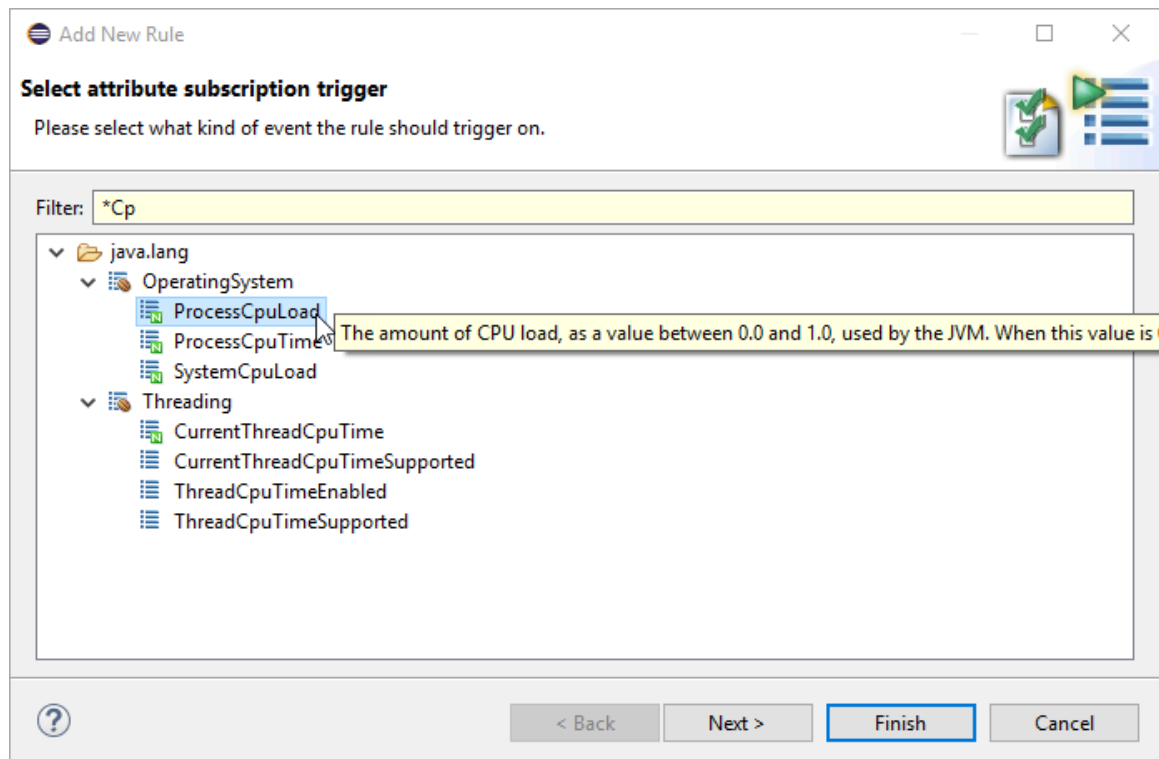
***Note:** In most tables in Mission Control, there are columns that are not visualized by default. The visibility can be changed from the context menu in the table.*

Deep Dive Exercises:

- 20.** If you run this from within Eclipse, you can jump to that line in the source and fix the problem. Right click on the offending stack frame and jump to the method in question.

Exercise 10.d (Bonus) – Triggers

Let's set up a trigger that alerts us when the CPU load is above a certain value. Go to the **Triggers** tab. Click the **Add...** button. Select the **ProcessCPULoad** attribute and hit **Next**.




Select the **Max trigger value** to be 0.3 (30%). And set the limit to once per second. Click **Next**.

There are a few different actions that can be taken when the rule triggers. There are custom actions downloadable from the update site, and it is also possible to add your own.


Let's stick with the default (**Application alert**). Click **Next**. Constraints can be added to constrain when the action is allowed to be taken. We do not want any constraints for this trigger rule. Click **Next** once more. Enter a name that you will remember for the trigger rule, then hit **Finish**.








Trigger rules are by default inactive. Let's enable the trigger by clicking the checkbox next to its name. The rule is now active. Move over to the Overview and wait for one of the computationally intense cycles to happen. The Alert dialog should appear and show you details about the particular event. If that isn't enough to generate the necessary CPU load, try resizing Eclipse like crazy for more than a second.

 Trigger Alerts

Trigger Alerts

This dialog shows the alerts that have triggered since the start of Java Mission Control. Triggers can be configured in the Trigger tab of the Java Mission Control JMX Console.




Date	Rule	Source
 Sep 8, 2017 2:56:15 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)
 Sep 8, 2017 2:56:18 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)
 Sep 8, 2017 2:56:37 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)
 Sep 8, 2017 2:56:41 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)
 Sep 8, 2017 2:57:00 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)
 Sep 8, 2017 2:57:04 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)
 Sep 8, 2017 2:57:23 PM	My Rules\My CpuLoad ...	[1.8.0_131] LoadAndDeadlock (17136)

The notification rule is: My CpuLoad Rule

Type description:
 attribute://java.lang:type=OperatingSystem/ProcessCpuLoad
 Rule trigger condition: value > 0.03

The actual trigger value: 0.07163235202865546


☒ Show dialog on alerts

Clear

Close

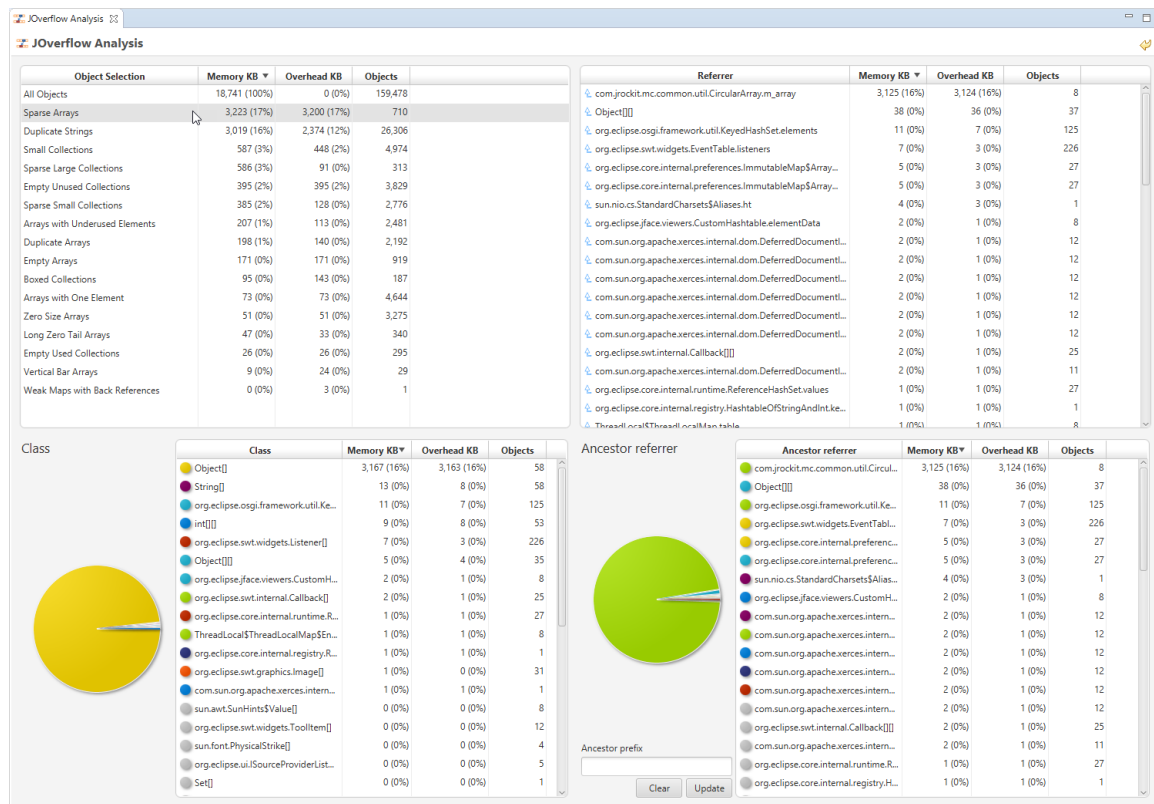
Disable or remove the rule when done to avoid getting more notifications.

Heap Waste Analysis (Bonus)

There is an experimental plug-in available for Java Mission Control which provides heap waste analysis. Heap waste analysis aims to find inefficient use of Java heap memory, and provides suggestions on how to improve the density of an application.

To use it, it must usually first be installed. For this JavaOne Hands-on-Lab, however, it has already been installed into the Eclipse lab environment.

Open the file `11_JOverflow/jmc41dump.hprof` by double clicking on it. This is a dump from an earlier version of Mission Control, which traded quite a lot of memory for a dubious performance gain in the JMX Console.



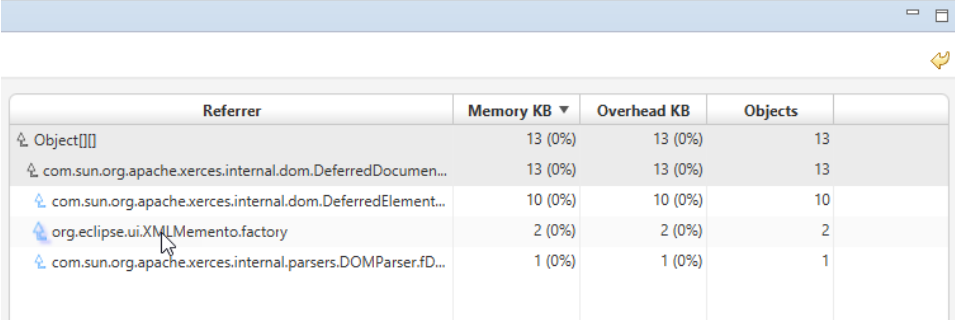
JOverflow will open, and show the contents of the heapdump. There are four quadrants in the JOverflow user interface. Also notice the little reset button in the upper right corner (👉). It will reset all the selections in the user interface.

Object Selection

The top left quadrant, **Object Selection**, will show you what heap usage anti-patterns the analysis has found. The first column in the **Object Selection** table show the kind of objects found. The second how much memory they use in total. The third column, **Overhead**, shows how much of the memory was wasted, in percent of the total heap used.

Referrer Tree-table

The top right quadrant contains the **Referrer** tree-table. This tree-table will show the aggregated reference chains for whatever is selected. Note that the way to reset the selections in the **Referrer** table-tree is to **right click in the table**. This is since you can make multiple consecutive selections to arrive at the reference chain you are interested in.

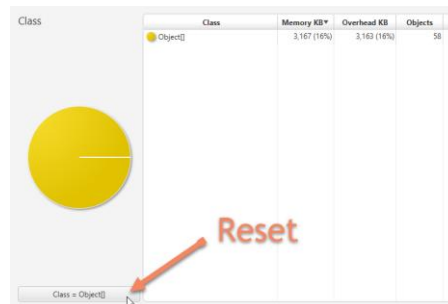


Referrer	Memory KB	Overhead KB	Objects
Object[]	13 (0%)	13 (0%)	13
com.sun.org.apache.xerces.internal.dom.DeferredDocumen...	13 (0%)	13 (0%)	13
com.sun.org.apache.xerces.internal.dom.DeferredElement...	10 (0%)	10 (0%)	10
org.eclipse.ui.XMLMemento.factory	2 (0%)	2 (0%)	2
com.sun.org.apache.xerces.internal.parsers.DOMParser.fD...	1 (0%)	1 (0%)	1

(Screenshot showing multiple available paths to select from)

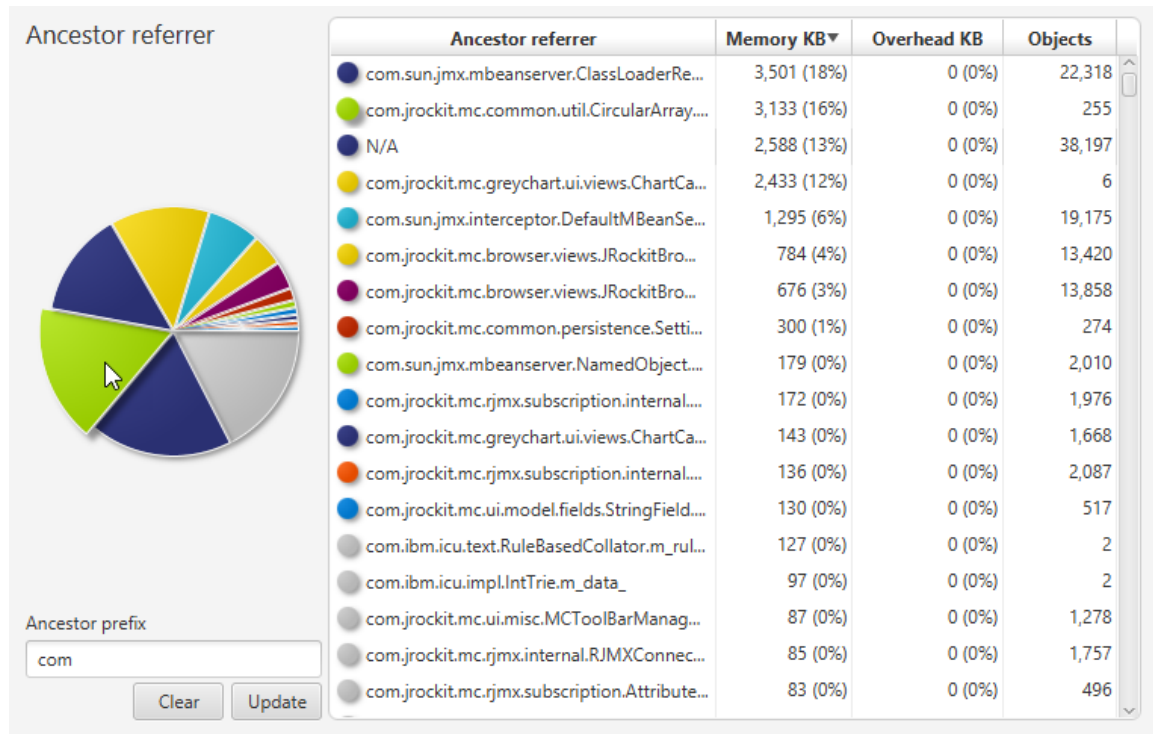
Class Histogram

The lower left shows a class histogram for whatever is selected, allowing you to filter on class. If you want to reset your selection, click the button representing the selection you have made.



Ancestor referrer

The final table, in the lower right, will show the objects grouped by the closest ancestor referrer. It provides a pie chart to show the memory distribution, and filter box, making it easy to home in on to instances of classes belonging to specific packages.



Note: it is possible to directly select a piece in the pie chart.

Exercise 11a – Reducing Memory Usage

It seems that quite a few objects used by this old version of JMC, are Sparse Arrays. This means that there are arrays with very few actual instances referenced to from them. In other words, they are mostly empty.

- How much memory (in percent of the total heap used) would be saved if the Java Mission Control 4.1 JMX Console switched to a more compact representation?
- How many instances are holding on to all that memory?
- What is the name of the field holding on to those instances?
- Can you, just by looking at the names in the reference chain, figure out how these sparse arrays were used?

JCMD (Java CoMmanD) (Bonus)

This exercise will explain the basic usage of the JDK command line tool `jcmd`. You can find it in the JDK distribution under `JDK_HOME/bin`. It will already be on the path if you open the command line interface by double clicking `C:\Tutorial\cmd.exe`.

Start any Java application. If you already have Eclipse or the stand-alone version of Mission Control running, you are already running one and can skip this step.

Next open a terminal. At the prompt type `jcmd` and hit enter. Assuming you have `jcmd` on your path, this will list the running java processes and their Process IDs (PID). If not, either add it to your path, or specify the full path to `JDK_HOME/bin/jcmd`. Since `jcmd` uses Java, and it is running, it will list itself as well.

The `jcmd` uses the PID to identify what JVM to talk to. (It can also use the main class for identification, but let's stick with PID for now.) Type `jcmd <PID> help`, for example `jcmd 4711 help`. That will list all available diagnostic commands in that particular Java process. Different versions of the JVM may have different sets of commands available to them. If `<PID>` is set to 0, the command will be sent to all running JVMs.

Attempt to list the versions of all running JVMs.

Deep Dive Exercises:

21. Start the Leak program. Use the `GC.class_histogram` command. Wait for a little while, and then run it again. Can you find any specific use for it?
22. You decide that you want your friend to access a running server that has been up for a few days from his computer to help you solve a problem. Oh dear, you didn't start the external agent when you started the server, did you? Can you find a solution that doesn't involve taking the server down?

***Note:** If you want to try the solution without specifying keystores and certificates, make sure you specify `jmxremote.ssl=false jmxremote.authenticate=false`. Also, specifying a free port is considered good form. Using `jmxremote.ssl=false jmxremote.authenticate=false jmxremote.port=4711` should be fine.*

23. Could you start flight recordings using `jcmd`? How?

***Note:** Have you noticed that there is a very similar feature set available from the Diagnostic Commands discussed in Exercise 10.b and `jcmd`. As a matter of fact, everything you can do from `jcmd` you can do using the `DiagnosticCommand` MBean and vice versa.*

More Resources

<http://oracle.com/missioncontrol>

- The Oracle Java Mission Control homepage

<http://twitter.com/javamissionctrl>

- The Java Mission Control twitter account

<http://hirt.se/blog>

- Marcus Hirt's Java Mission Control articles

<http://twitter.com/hirt>

- Marcus Hirt's twitter account