JDK Mission Control 7.0 Tutorial

Index

This document describes a series of hands on exercises designed to familiarize you with some of the key concepts in JDK 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.

Introduction

JDK Mission Control is a suite of tools for monitoring, profiling and diagnosing applications running in production on the HotSpot JVM.

JDK 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 JDK Flight Recorder – a very low overhead profiling and diagnostics tool.

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

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

JDK 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:\jmc\bin
```

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

To do the exercises in this tutorial, you must first install Eclipse and Mission Control. Please follow the instructions in the README.md file in the root of the following GitHub repository:

https://github.com/thegreystone/jmc-tutorial

In this repository you will also find the files and code required to complete the tutorial.

JMC is still in development. We are currently working on JOverflow to work in JMC, but this is not yet the case. To run the JOverflow labs, you will therefore need to download an Oracle JDK 10 and install the JOverflow plug.

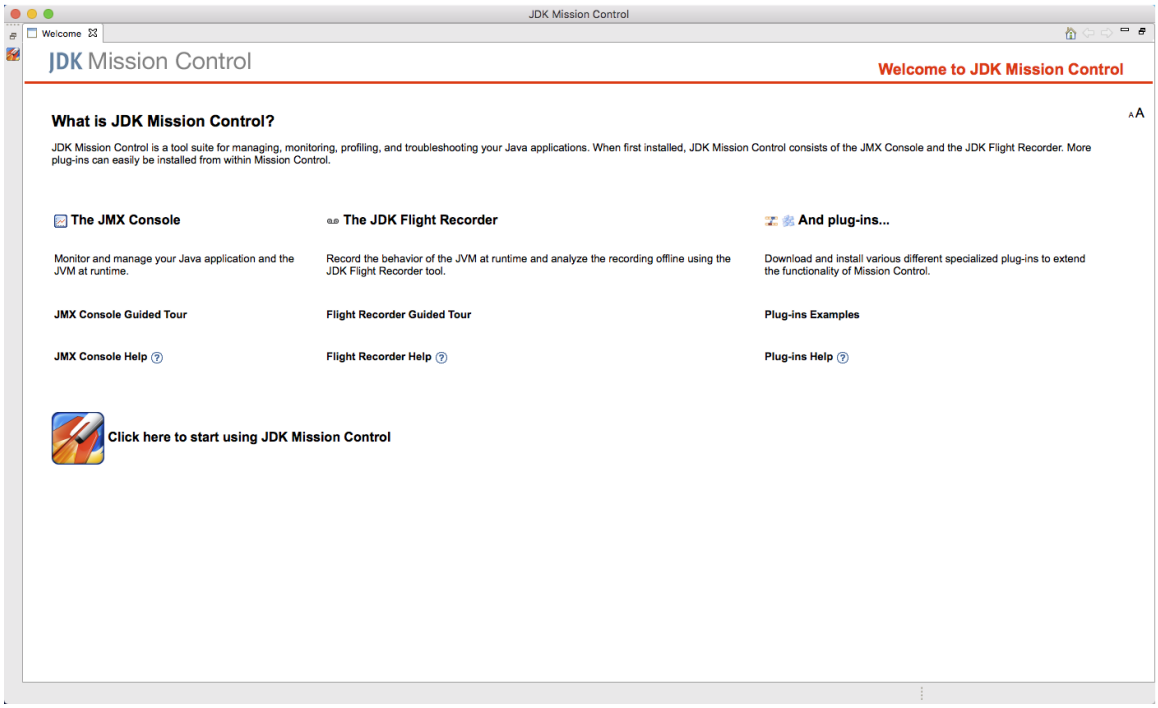
Starting JDK Mission Control

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

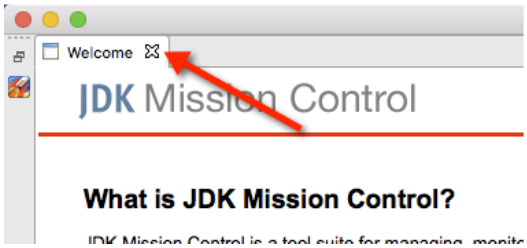
This exercise shows you how to start both the stand-alone and the Eclipse plug-in versions of JDK Mission Control.

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

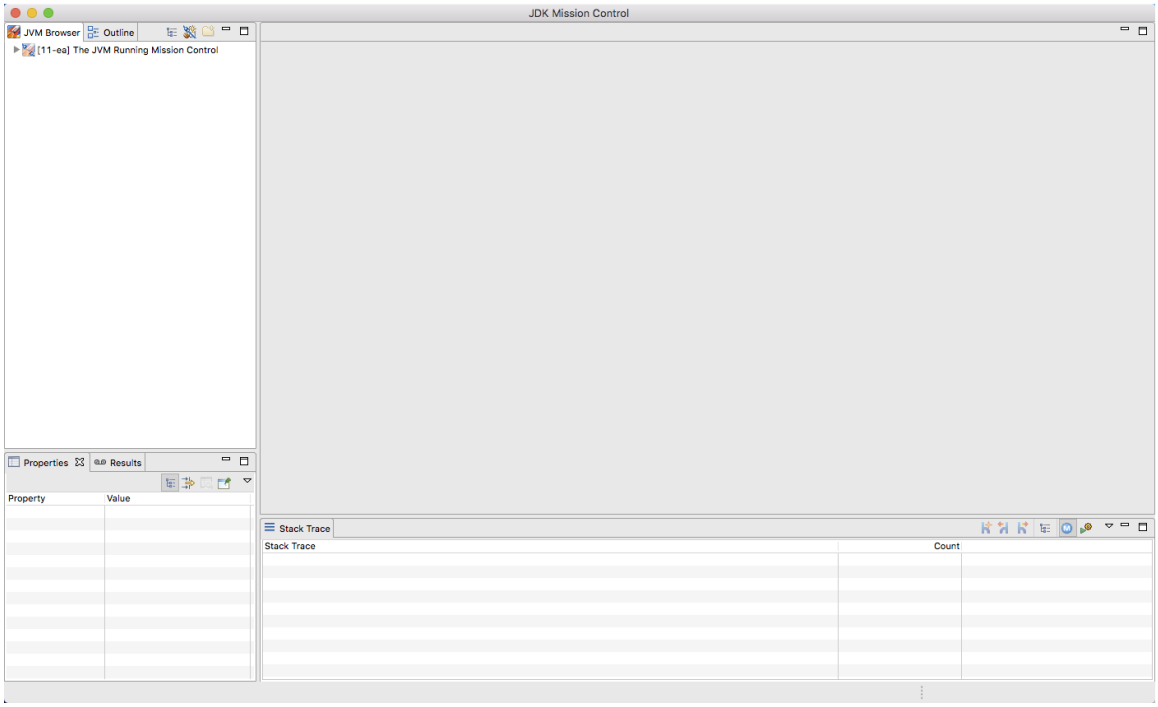
Go to the directory where you installed the stand-alone version of JMC. In the bin folder, you will find a jmc launcher. Either double-click it or run it from the command line. A splash screen should show, and after a little while you should be looking at something like this:



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



Closing the welcome screen will show the basic JDK 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 aggregated stack traces for anything selected in the editor.

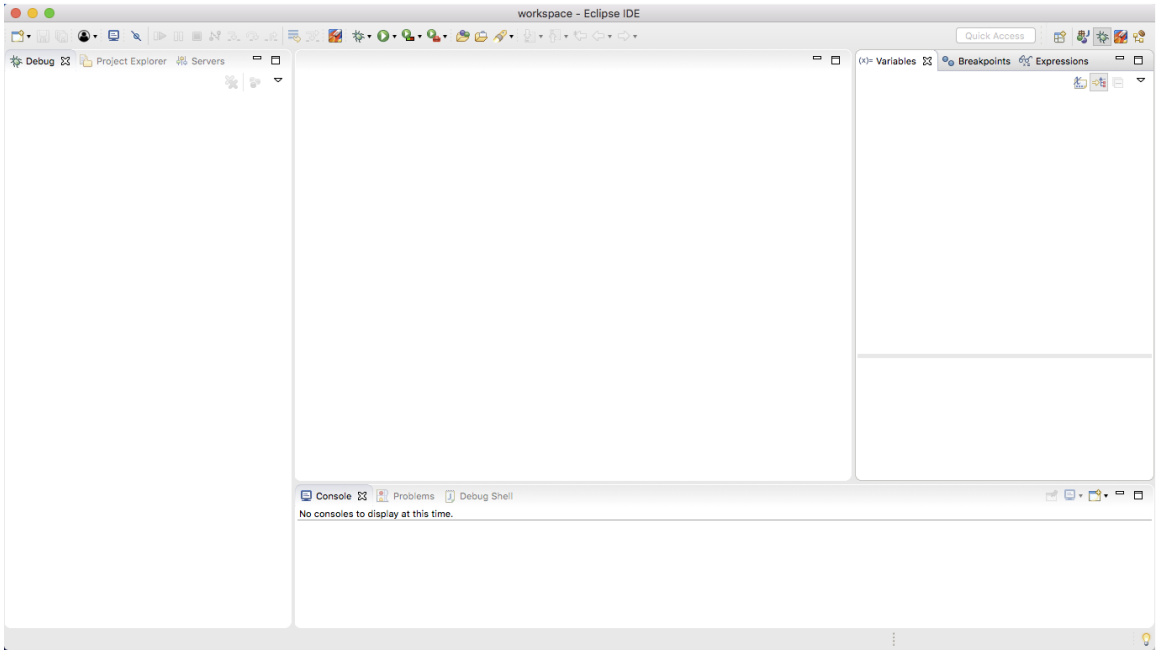
To launch the different tools, simply select a JVM in the JVM Browser and then select the appropriate tool from the context menu. For example, the Management Console can be started on a JVM by selecting the JVM in question in the JVM Browser and selecting Start JMX Console from the context menu.

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

Since we will be running the tutorial from within Eclipse, please ***shut down*** (⌘+Q on Mac, alt+F4 on Windows) the stand-alone version of JDK Mission Control.

Exercise 1b – Starting JMC in Eclipse

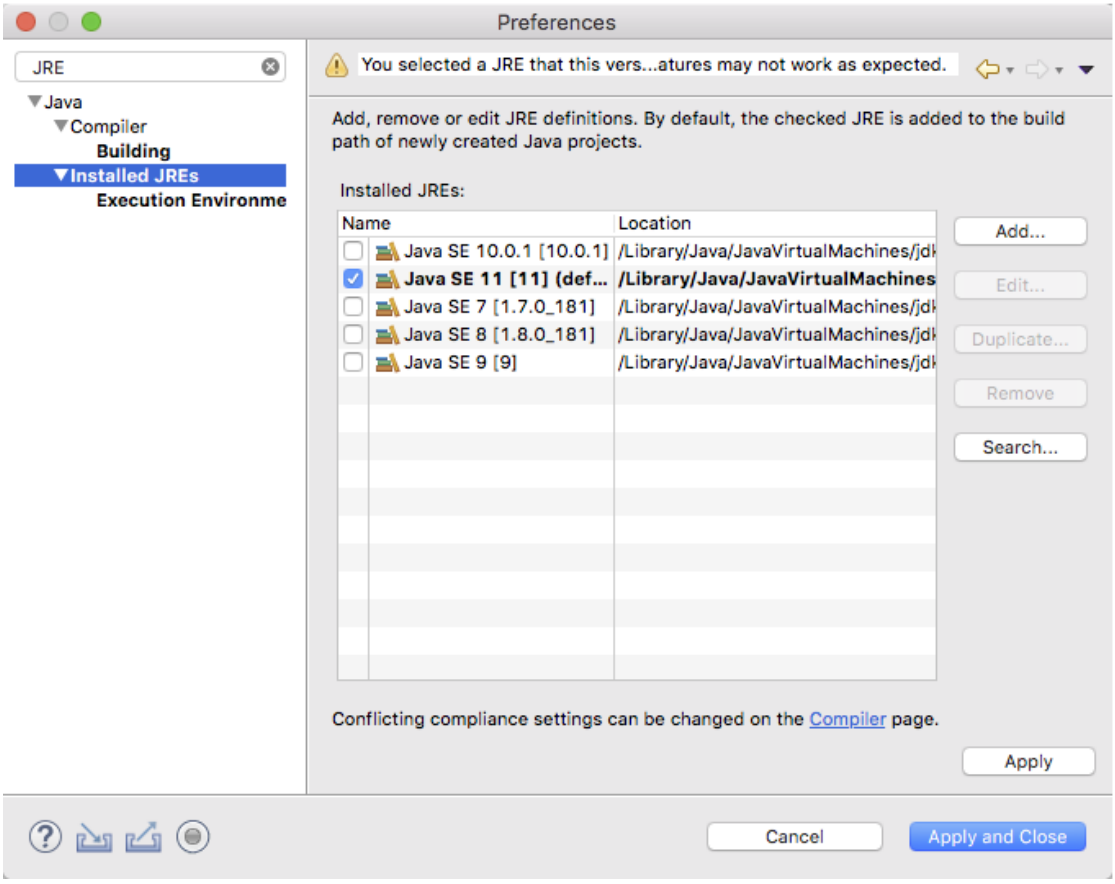
First make sure that you followed the instructions in the README.md file in the root of the GitHub repository to the letter, and that the JMC plug-ins are installed Next start Eclipse and switch to the Java perspective. You should now see something looking somewhat like below:



First we will need to verify that the JDK 11 runtime is properly configured and available for Eclipse to use.

1. Open the Preferences (Eclipse | Preferences… on Mac OS X, Window | Preferences… on Windows).

2. In the filter box in the upper left of the Preferences dialog, type JRE, then select Installed JREs.

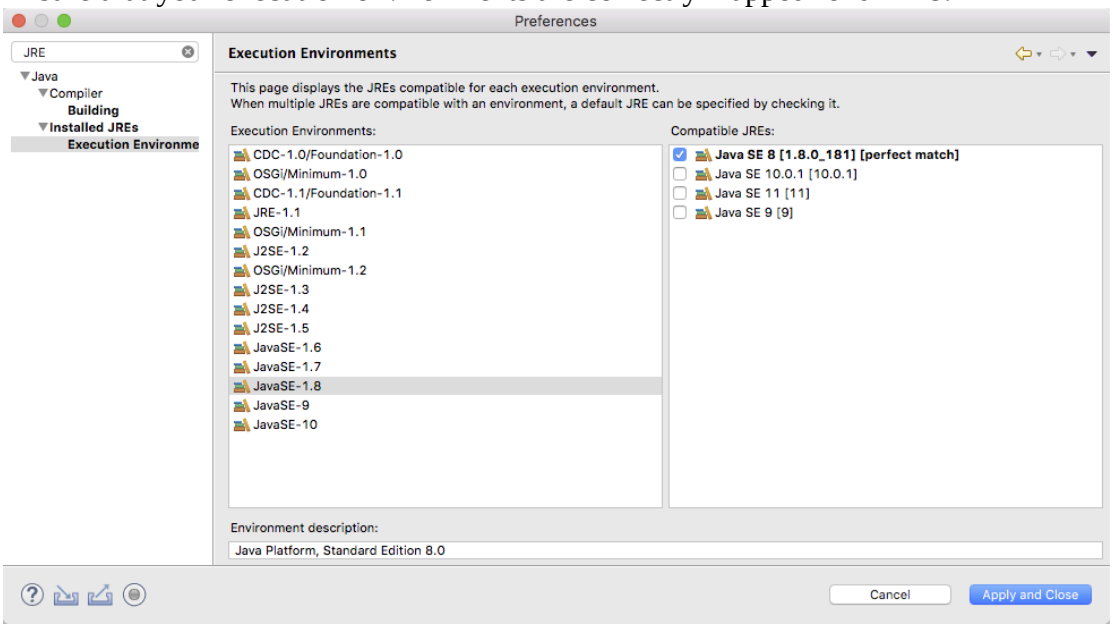


3. If you already have various JDKs installed, they may already show up here. If they don't, click add to add them. There is a JDK 11 provided in the HoL zip.

   Note: *when adding a JRE on Mac, select Mac OS X VM. Also note that you should browse to the <JDK>/Contents/Home folder.*

4. Ensure that your JDK 11 is the default by checking the checkbox next to it.

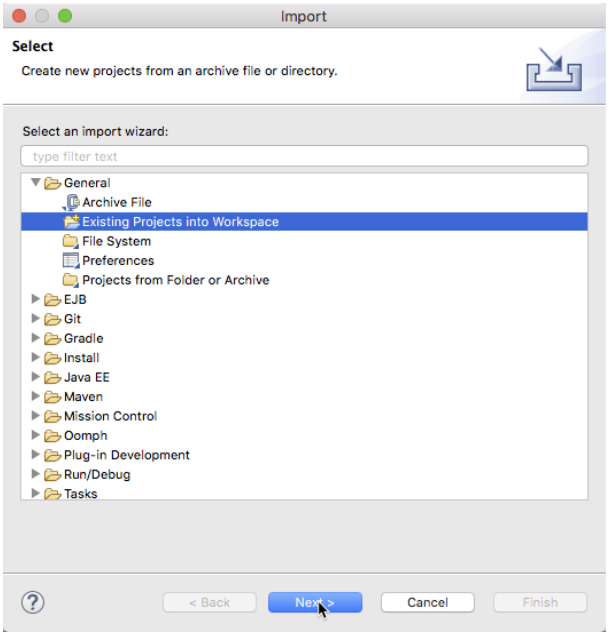
5. Ensure that your execution environments are correctly mapped for JDK 8.



Don't forget to click Apply and Close when done.

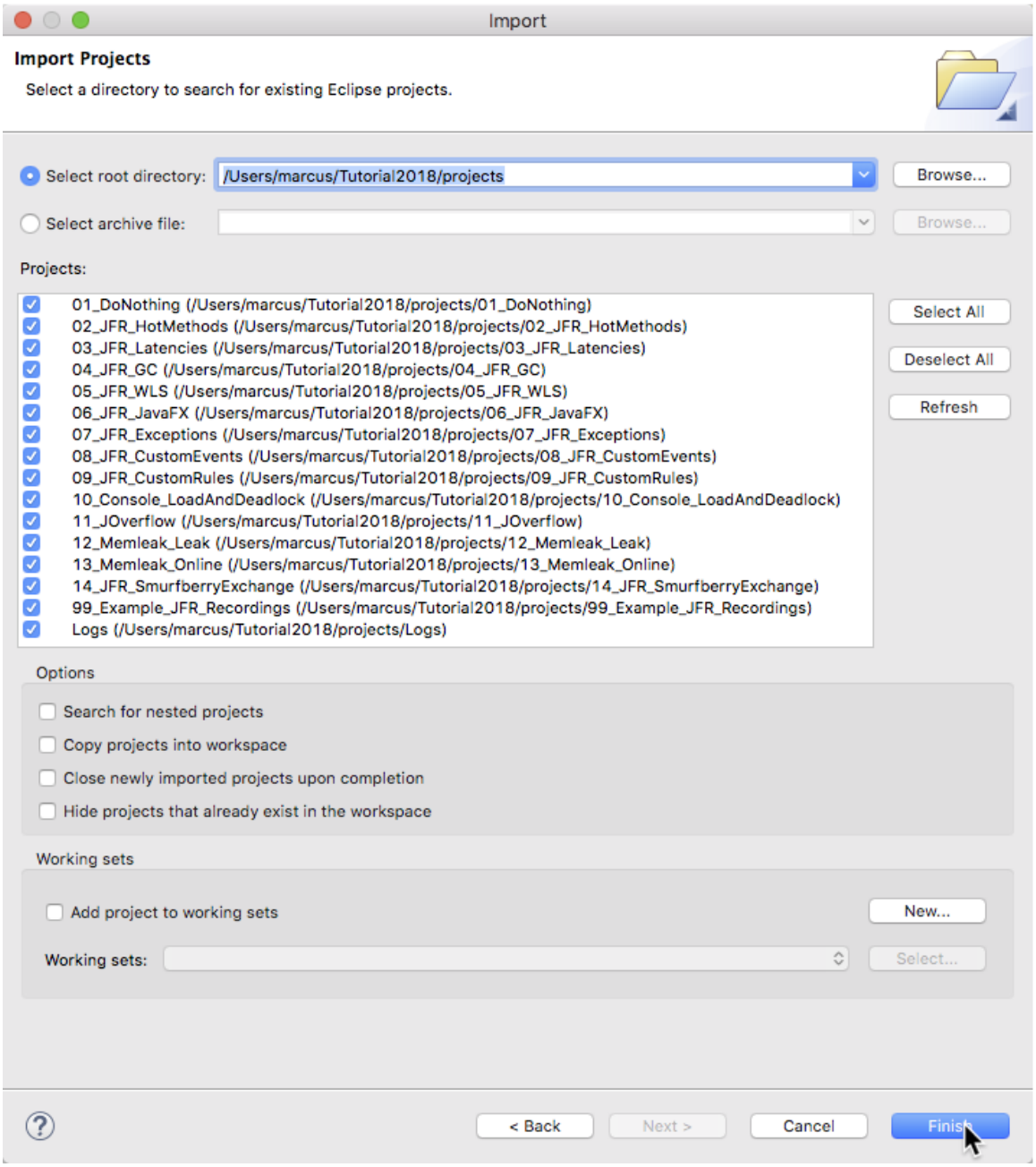
Next we will have to import the projects with all the examples into the workspace.

1. Select File | Import… from the menu.
2. In the Import dialog, select Existing Projects into Workspace and hit Next.

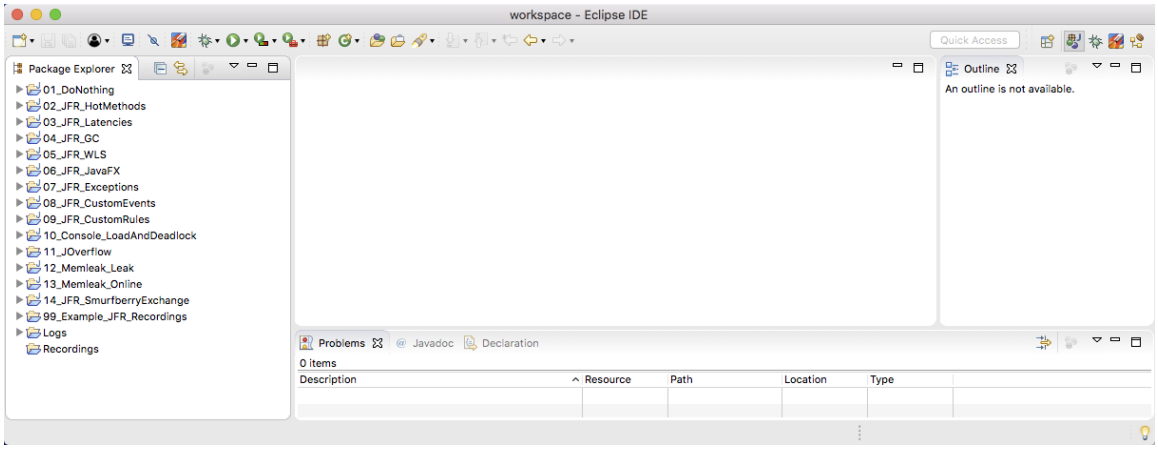


3. Browse to the projects folder in the root tutorial folder and click Open.

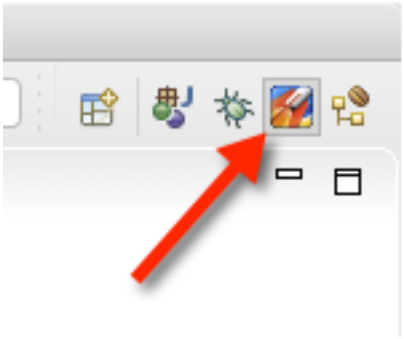
4. Select all projects and click Finish.



You should now see something similar to the following:

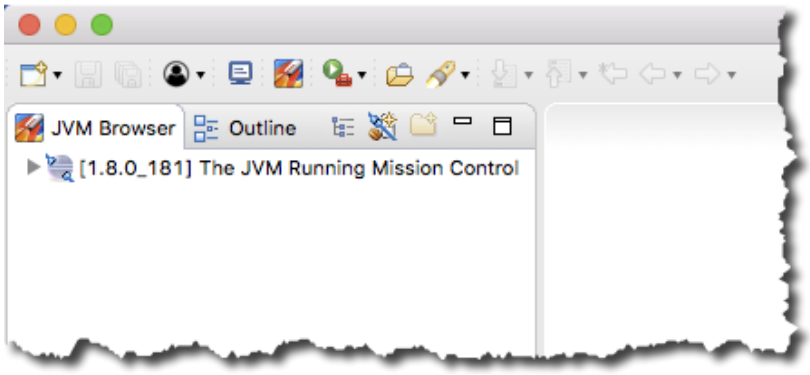


To the left the projects for the different exercises can be seen. In Eclipse a configuration of views is called a perspective. There is a special perspective optimized for working with JMC, called the JDK Mission Control perspective. To open the JDK Mission Control Perspective, click the Mission Control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

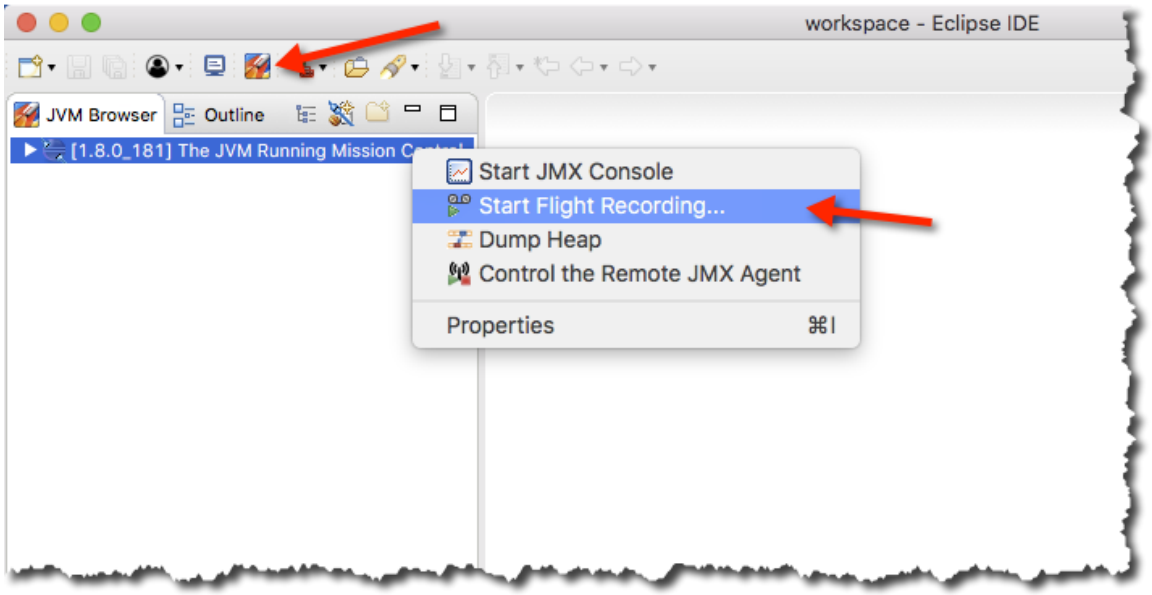


In this tutorial, you will be constantly switching between the Java perspective, to look at code and to open flight recordings, and the Mission Control perspective, to access the JVM browser and optimize the window layout for looking at flight recordings.

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

):

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

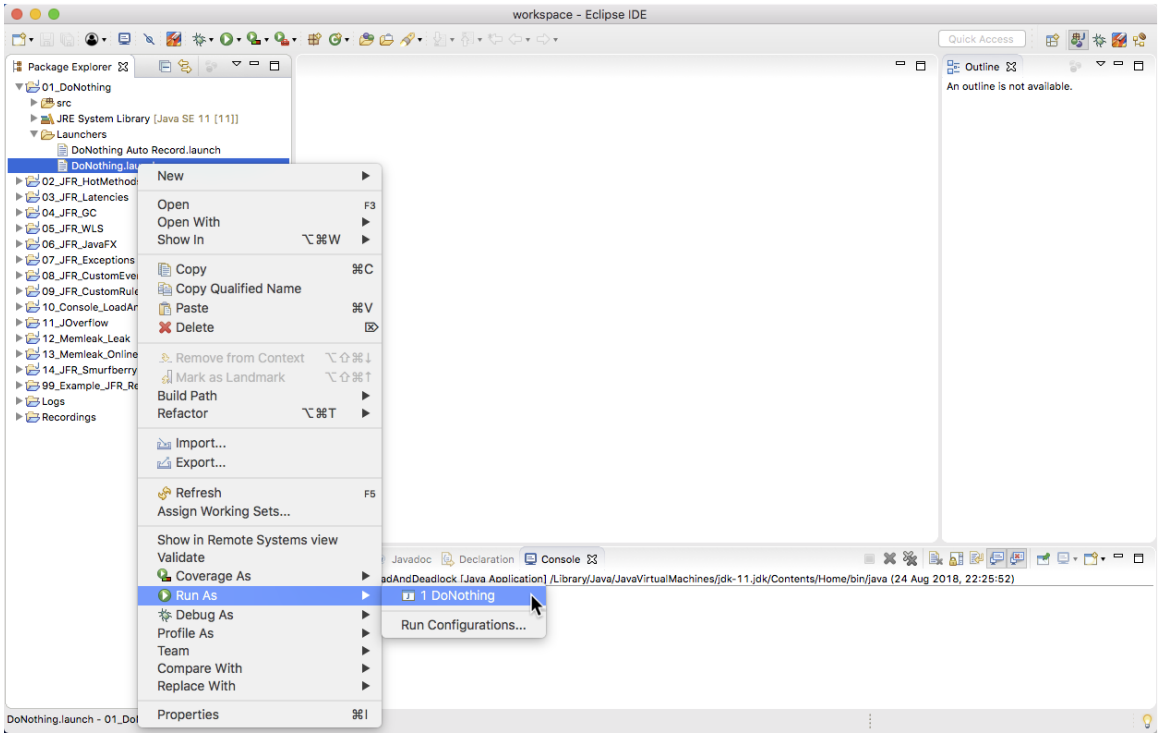
The JDK Flight Recorder

The JDK Flight Recorder (JFR) is the main profiling and diagnostics tool in JDK 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 2a – 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 JDK Mission Control.

First switch to the Java perspective. Note that there is an empty project named Recordings. Next start the DoNothing program by right clicking on the **`DoNothing.launch`** file in the **`Launchers`** folder of the 01_DoNothing project, as show below.

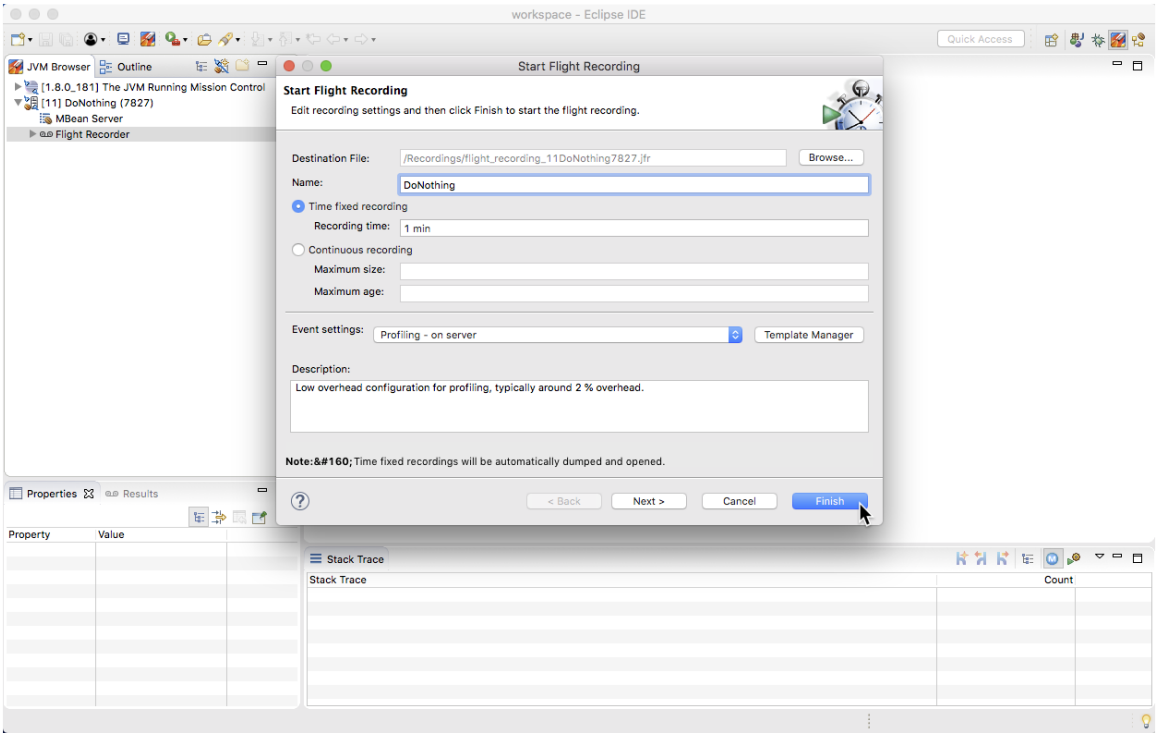


Note: *There are "Auto Record" versions of most launchers, which will launch the application and automatically create a recording in the* **`autorecordings`** *folder in the tutorial root. This is since, in certain environments (slow machines running a virtualized Windows for example), the highly loaded 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,*

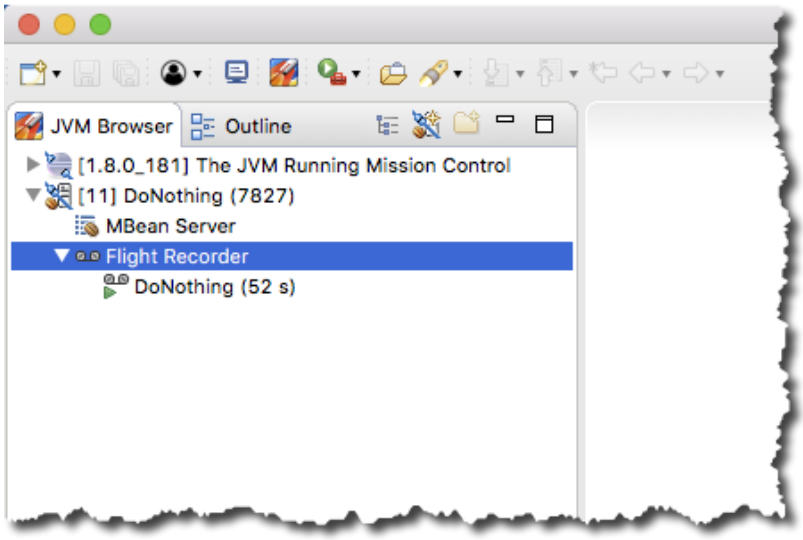
you know, if you're lazy. I will not judge. Just try doing this first exercise without Auto, as it is about doing recordings from JMC.

Switch to the Mission Control perspective and select the newly discovered JVM running the DoNothing 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.



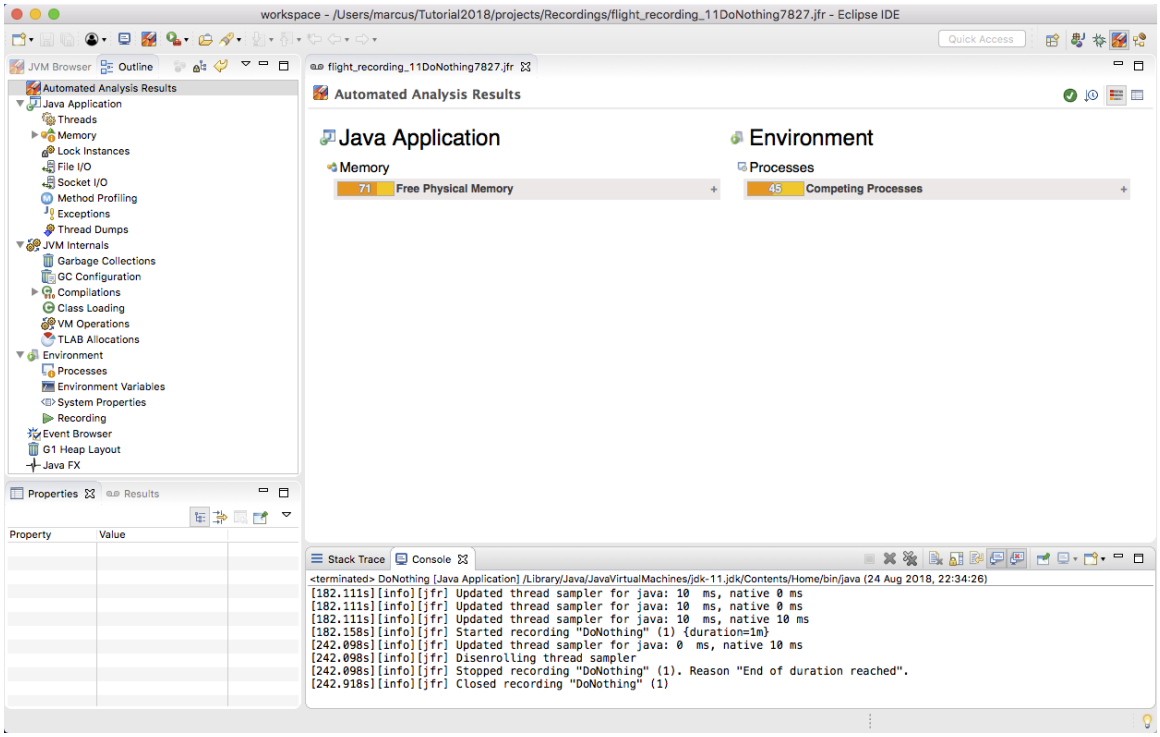
The progress of the recording can be seen in the JVM Browser, when the Flight Recorder node is expanded. It can also be seen in the status bar.



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 JDK Mission Control perspective.

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



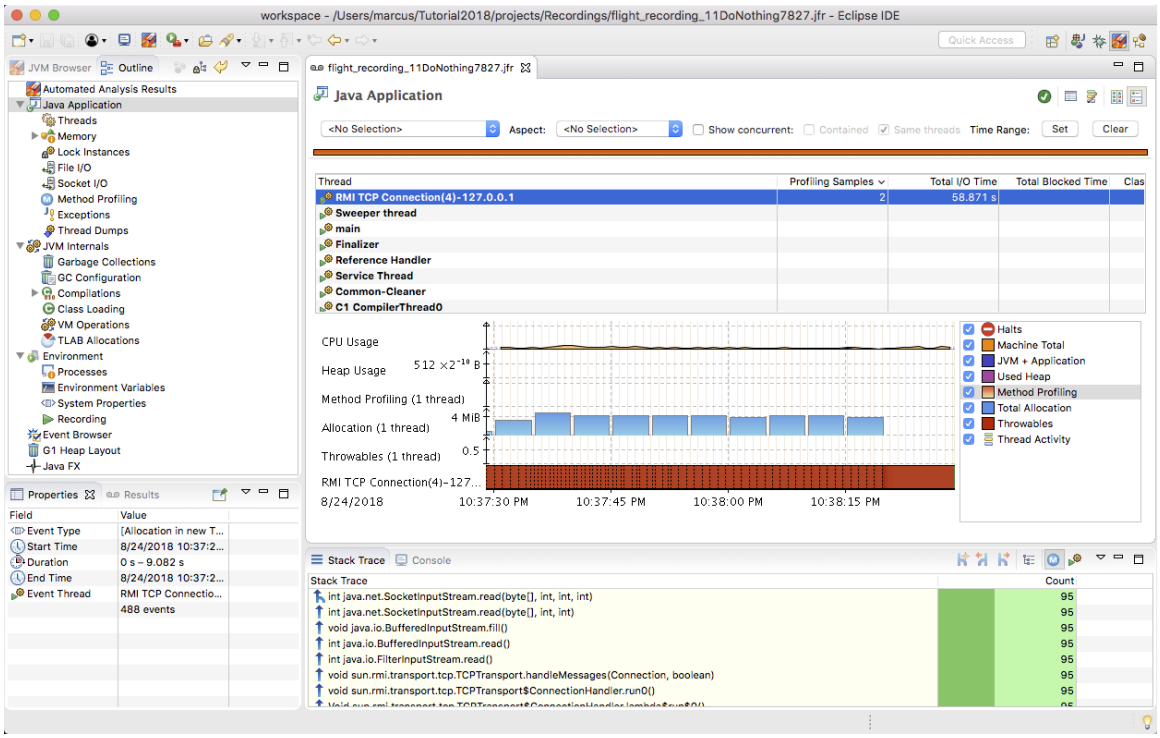
This exercise is just to familiarize you with one of the ways to create a flight recording. This will be a rather boring recording, in terms of results from the automated analysis, so don't mind the results of this analysis.

Since the recording was done on a process not doing any actual work, the auto-analysis didn't find anything other than that I should probably run less stuff on my laptop:

1. I am starting to run out of physical memory. I am guessing MS Word ate a good portion of it. ;)

2. I have a lot of other processes running aside from the JVM. Since this is my laptop, and not a dedicated server, I am not surprised.

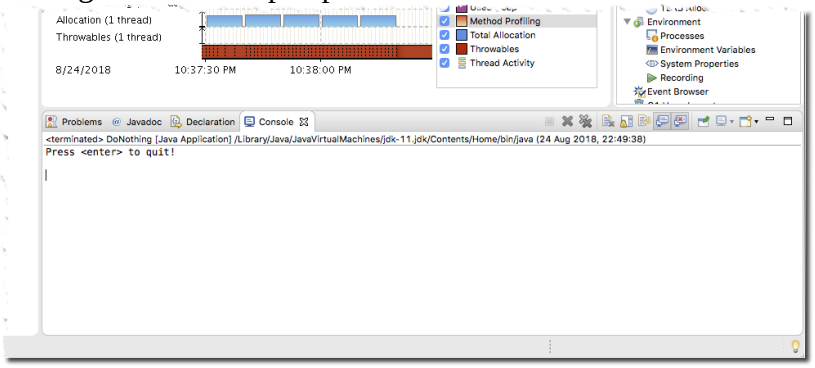
The Outline view shows the various ***pages*** in the JDK Flight Recorder user interface. Select Java Application.



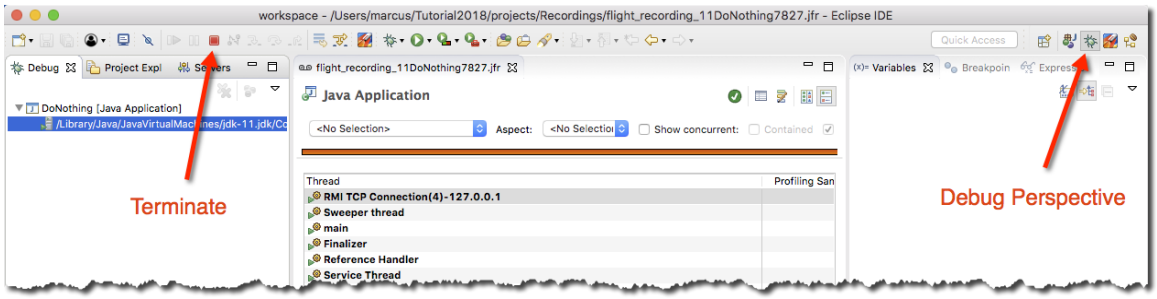
Here we get an overview of commonly interesting properties of the recording. This process didn't do much, but we can see that what little it did was spent transferring data.

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 DoNothing application.

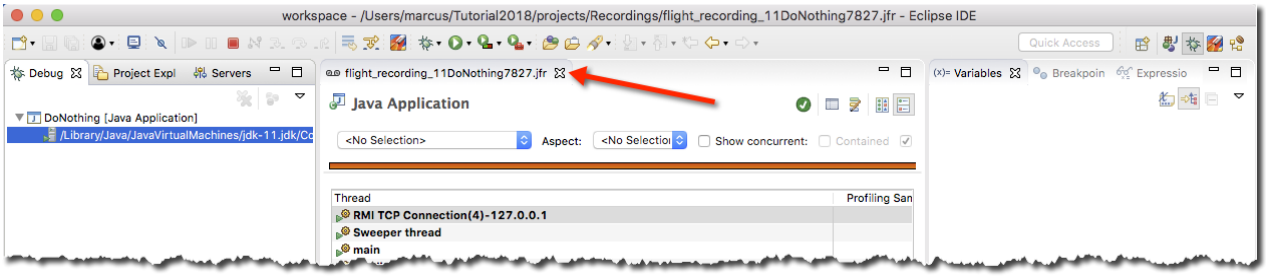
- Either go to the Java perspective and hit enter in the Console view,



- … 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.*



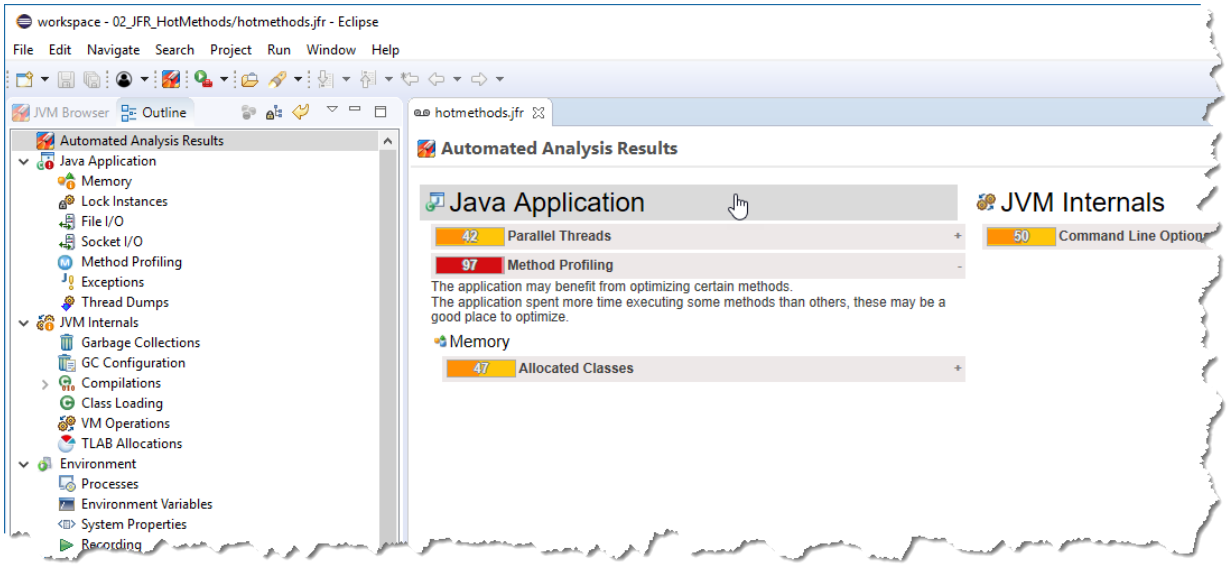
Exercise 2b – Hot Methods

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.

Like any good cooking show, we've provided you with pre-recorded recordings to save you from having to wait another few minutes for the recording to finish.

Open the *02_JFR_HotMethods*/hotmethods_before.jfr recording.

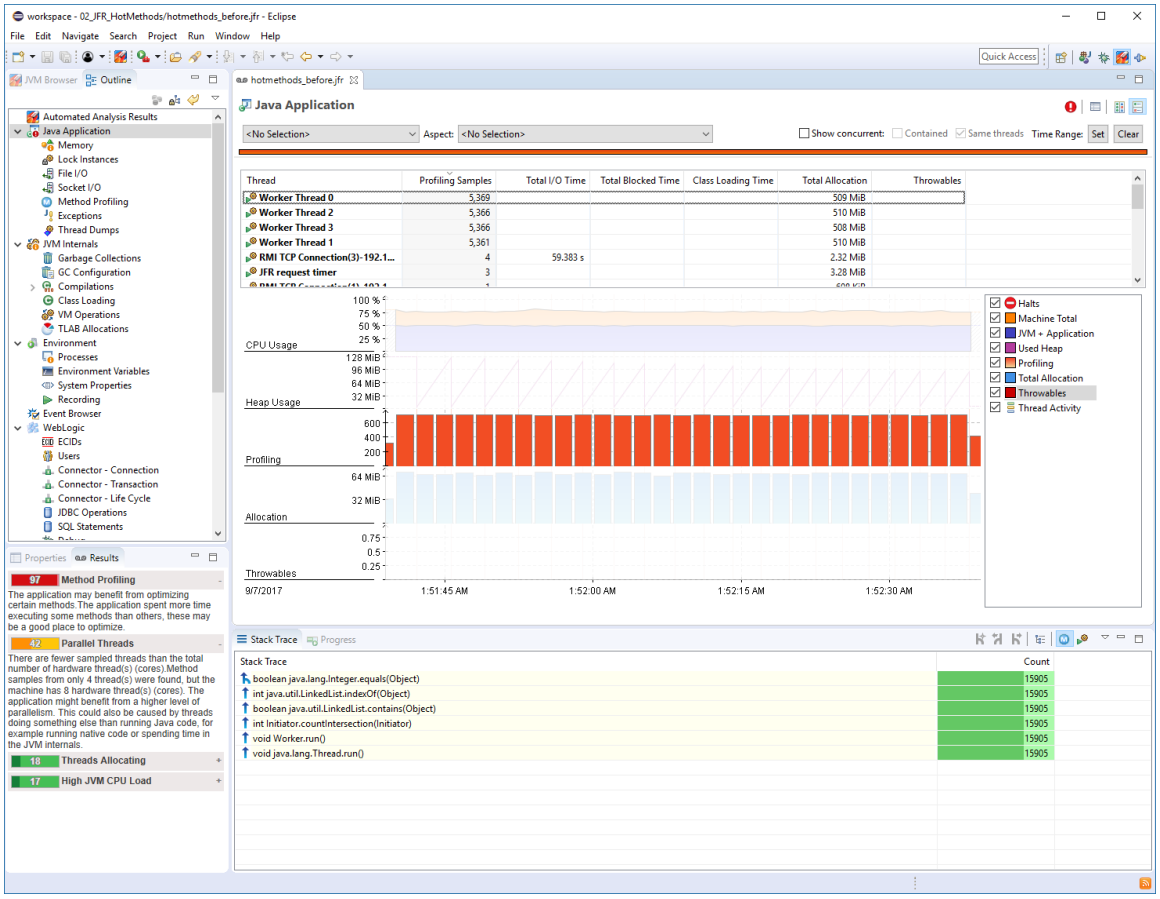
Note: *Switch to the Java perspective and open (double click) the recording in the 02_JFR_HotMethods project named hotmethods_before.jfr.*



Switch back to the JDK Mission Control perspective once the recording is open. Even though you will be able to see the automated analysis page directly in the Java Perspective, the JDK Mission Control places the support views needed to effectively navigate and analyze flight recordings in the right places. Therefore, always remember to switch to the JDK Mission Control perspective when analyzing recordings.

The automated analysis indicates that there is potential value in optimizing certain methods.

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



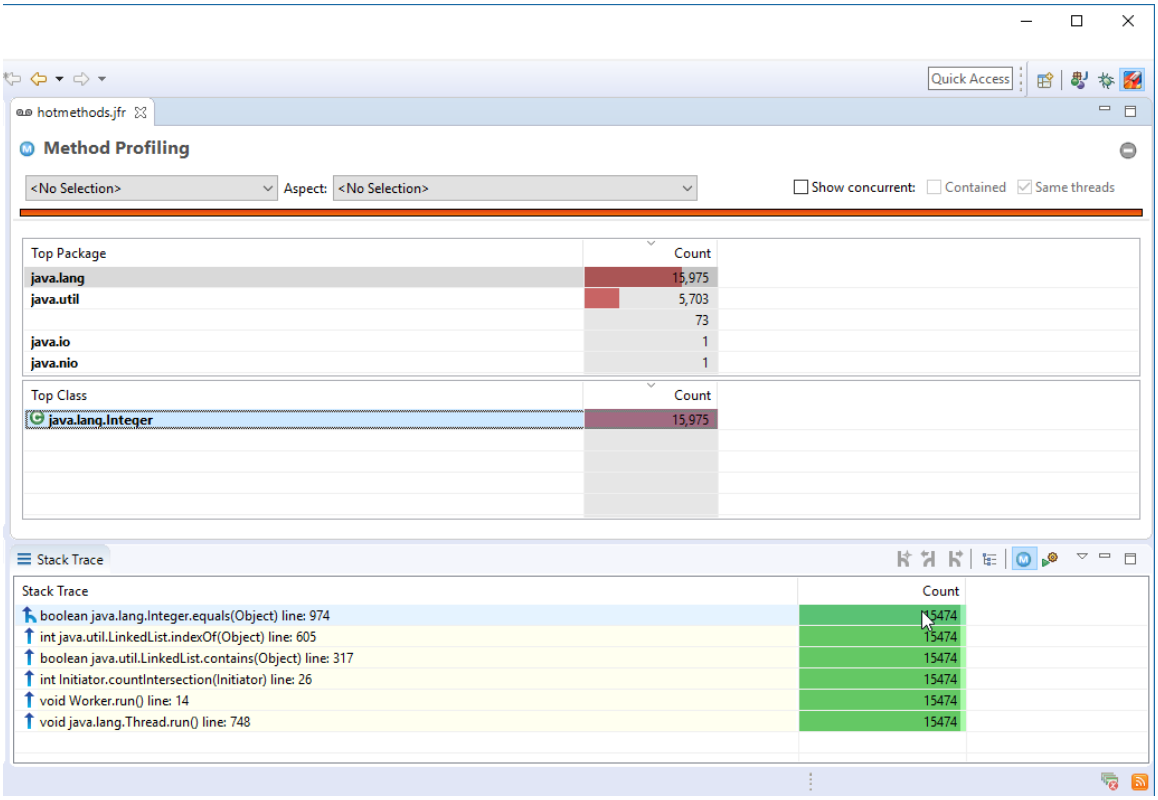
Next click on the Profiling lane. The stack trace view will show the aggregated stack traces of any selection in the editor. It will now show you the stack traces for the profiling samples.

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?

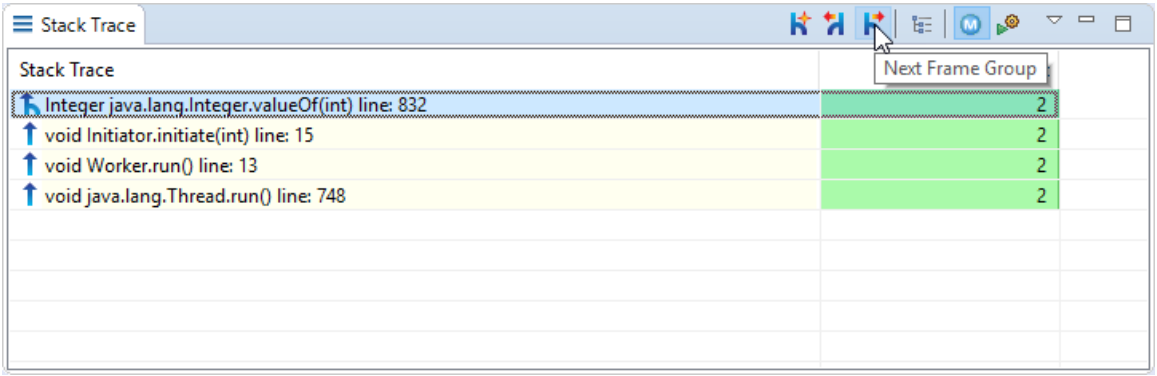
Which method do you think would be the best one to optimize to improve the performance of this application?

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

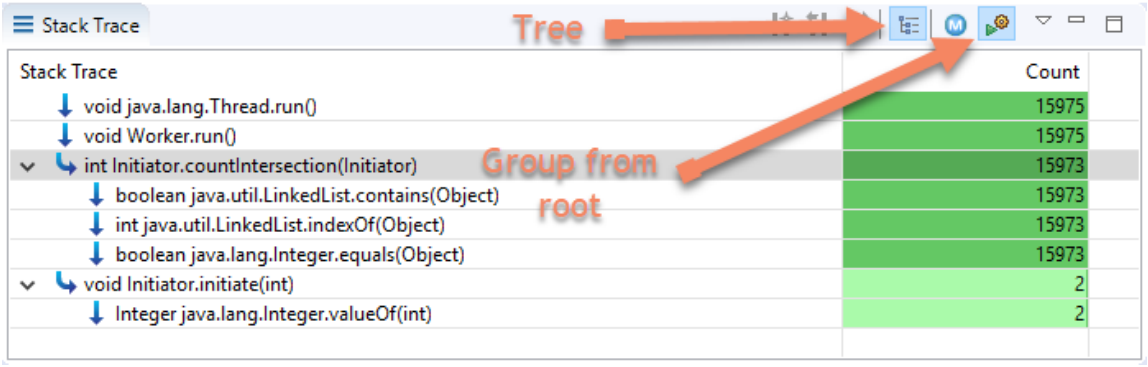
There is another page, Method Profiling, that makes it easy to break down the method profiling samples per package and class of where the sample was captured.



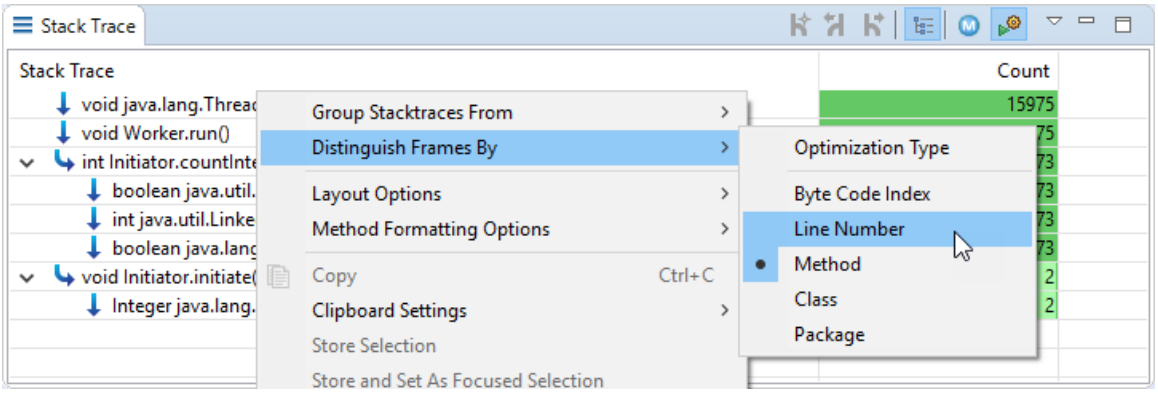
In the stack trace view, the most commonly traveled path is shown by default, effectively giving you the most common stack trace directly. Wherever the path branches, there is a branch icon (,). You can use the right and left arrow keys to select between the different branches (Frame Groups), or use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

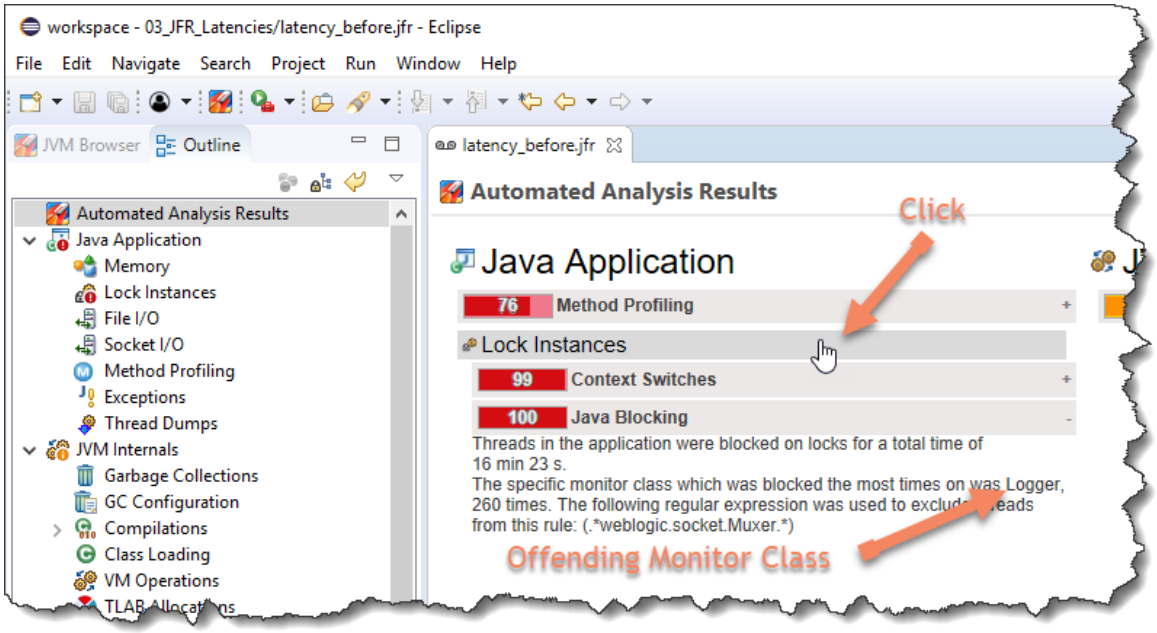
   Note**:** *The application generates custom events for each unit of "work" done. This makes it easy to compare the time it takes to complete a unit of work before and after the code change. Would it be possible to decide how faster the program became without these custom events?*

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.

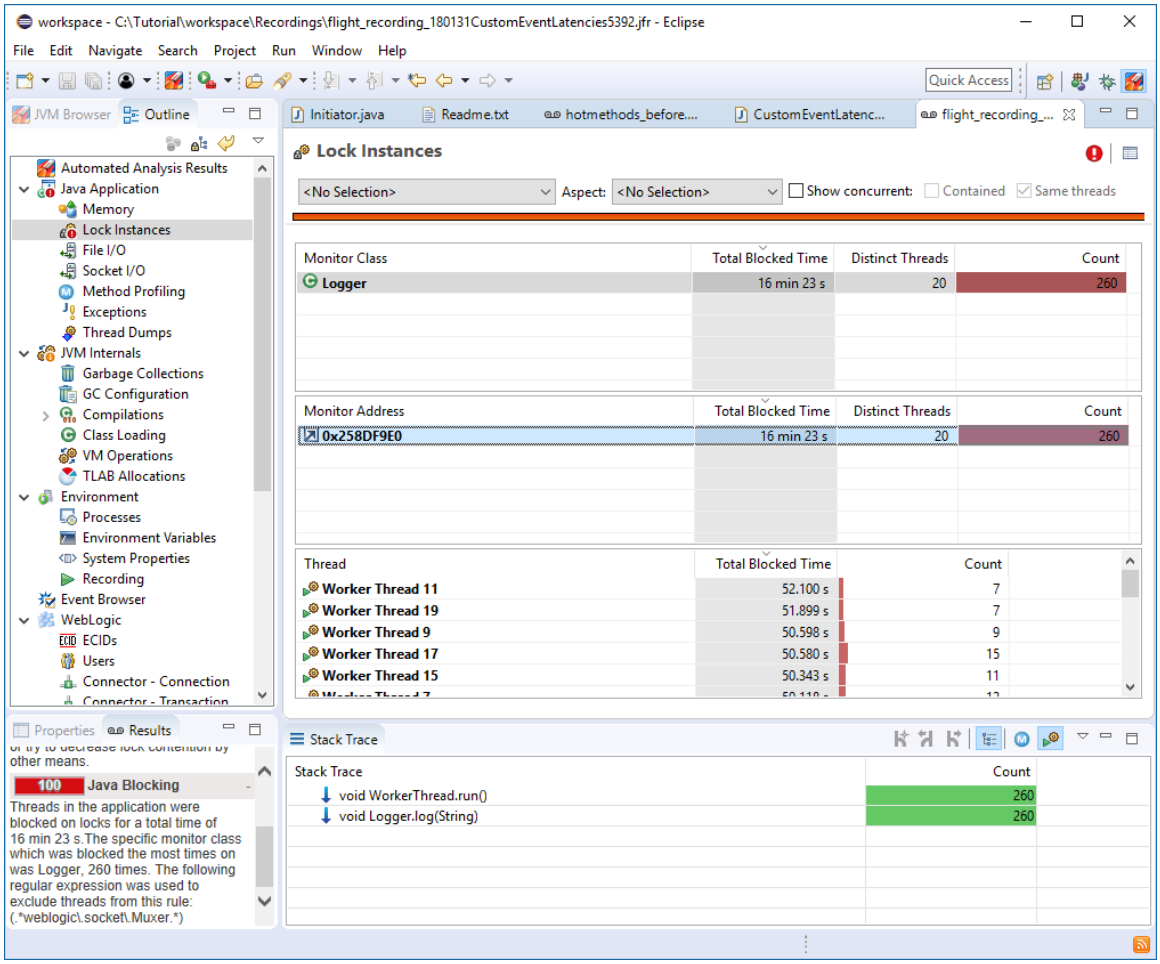
Exercise 3 – Latencies

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.

Open the **`03_JFR_Latencies/latency_before.jfr`** recording (same procedure as when opening the hotmethods_before.jfr 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.



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.*

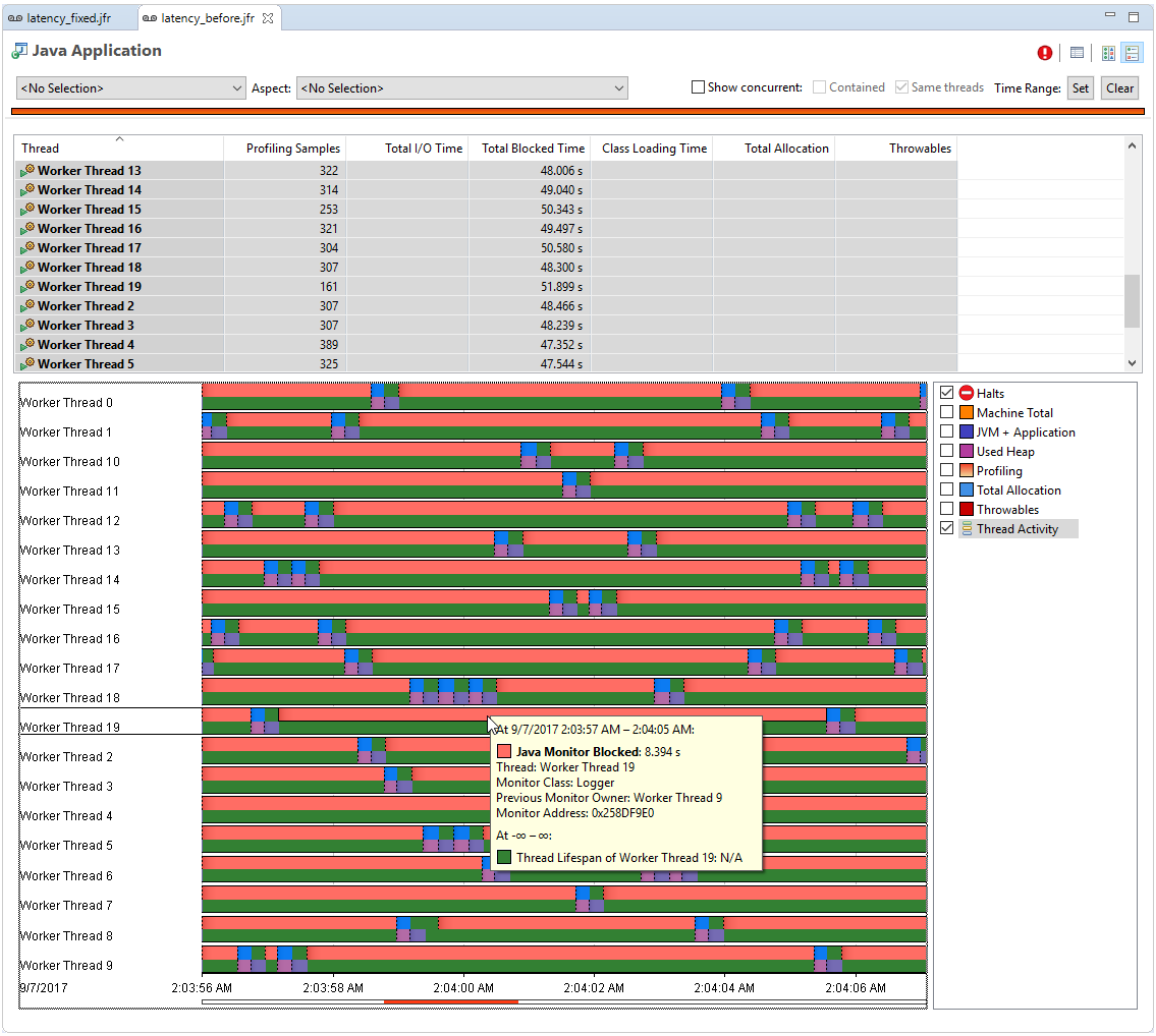
It seems most of these blocking events come from the same source.

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?

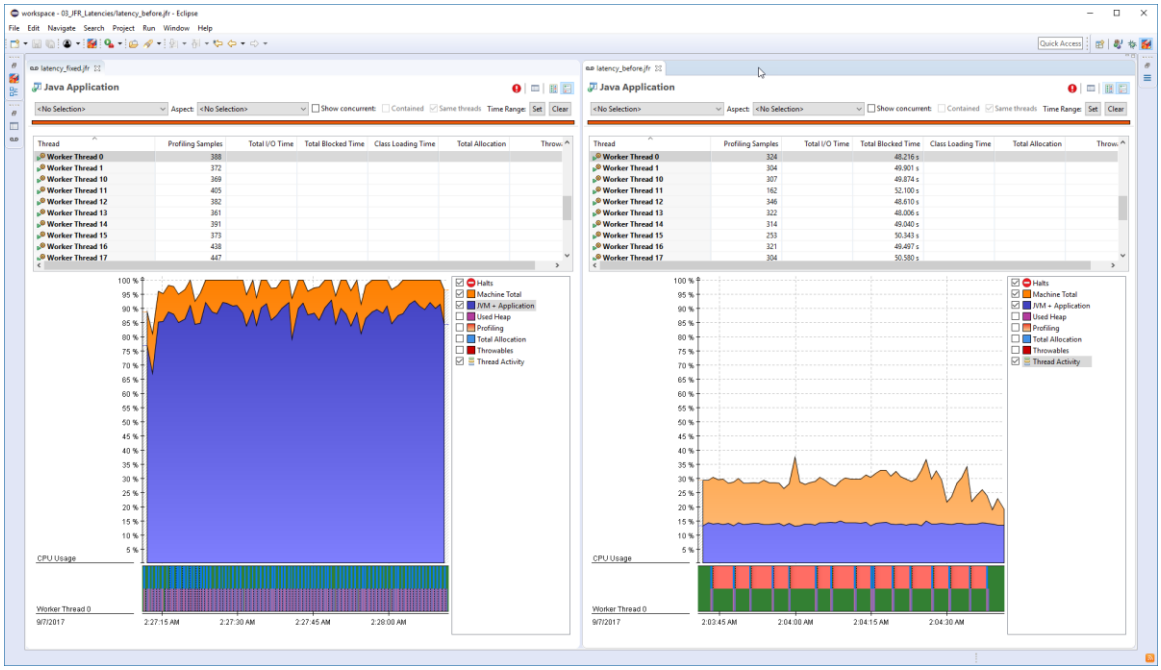
Note: *Right click on the* `Logger.log(String)` *method and select Open Method if running JMC in Eclipse. If not running in Eclipse open the source file and take a look at it. We get several matches; select the one in 03_JFR_Latencies. The method is synchronized.*

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.

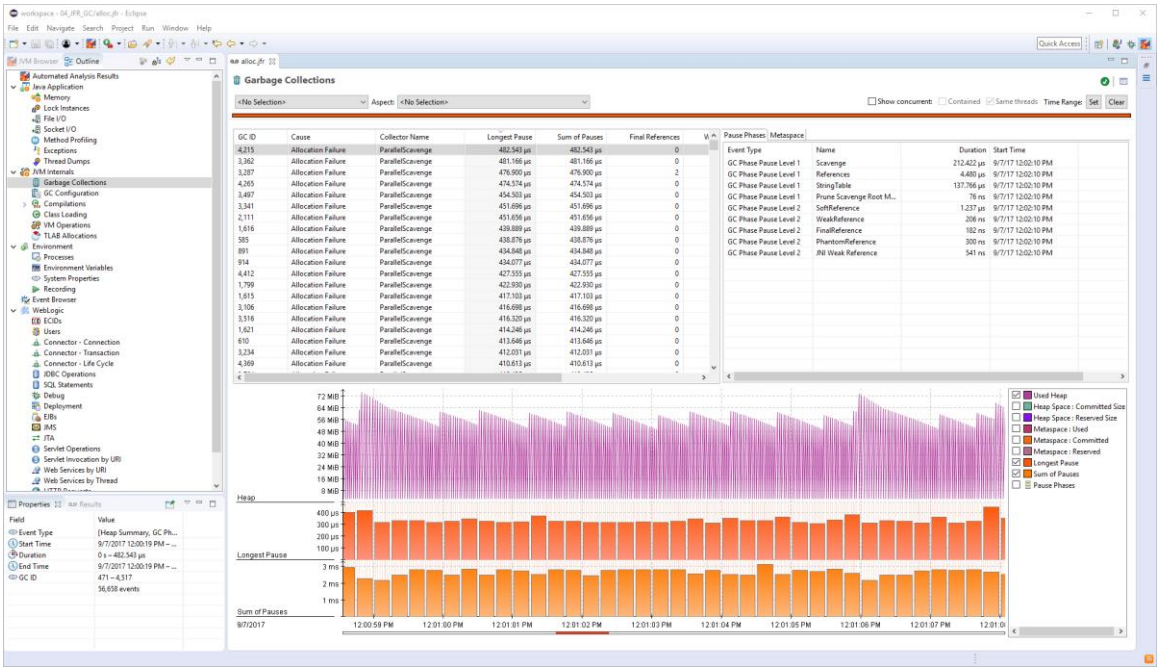
Exercise 4 (Bonus) – Garbage Collection Behavior

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, go to the Garbage Collections page.

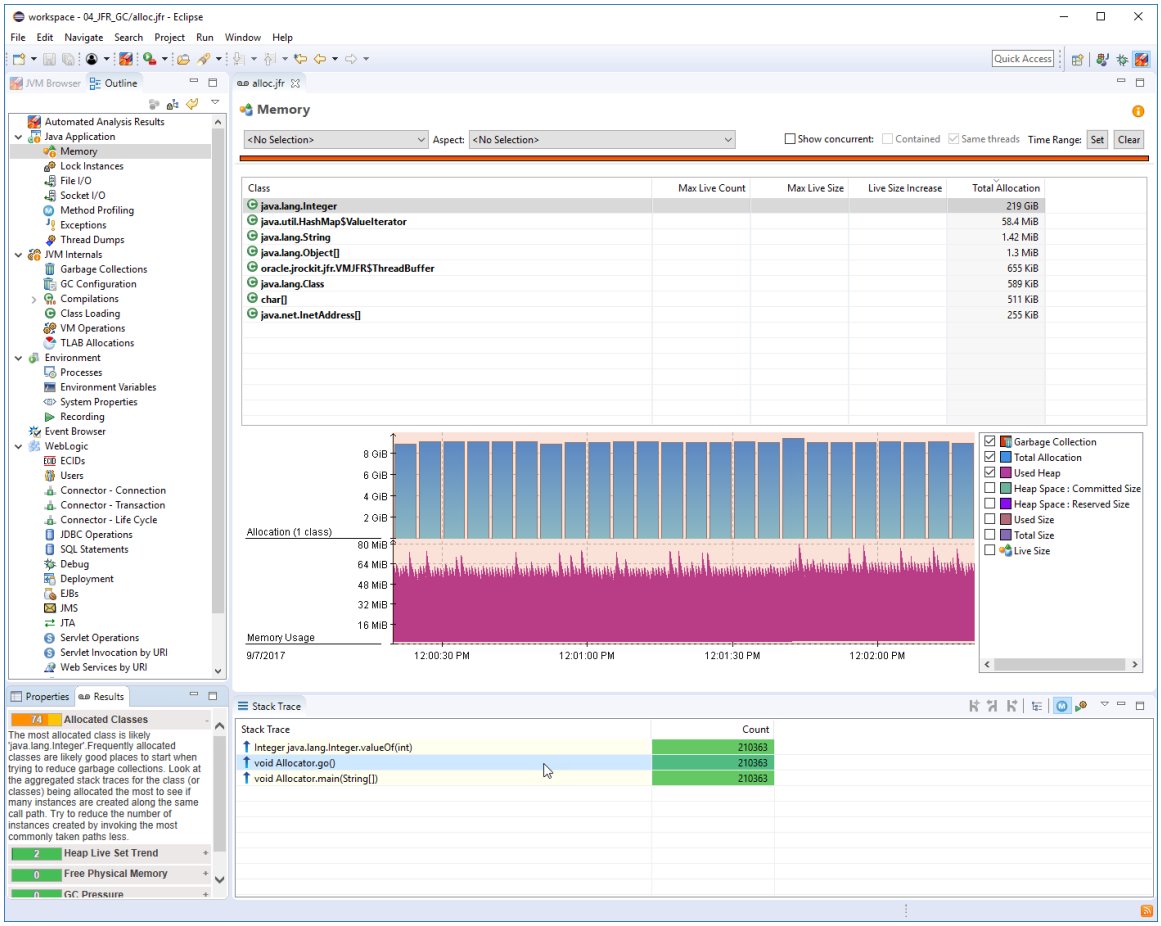
In this page you can see many important aspects about each and every garbage collection that happened during the recording. As can be seen from the graph, garbage collections occur quite frequently.

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:

3. 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*

4. 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) – Memory Leaks

One commonly recurring problem is the memory leak. In Java, memory leaks are created by unintentionally retaining objects that are no longer of any use to the program.

Some examples of common Java memory leaks include:

- Putting objects in a collection (for example a HashMap being used as a cache), but never removing them.
- Adding a listener to an object in a framework, but never removing it.
- Putting an object in a java.lang.ThreadLocal, but never removing it.
- Keeping references to classes loaded in class loaders that are no longer supposed to be in use, or just simply keeping references to classes no longer required.

Unless the heap behavior of an application is actively monitored, a memory leak is commonly discovered post-hoc, when the JVM exits due to there no longer being any heap memory available (`OutOfMemoryError`).

One way to resolve memory leaks is to analyze the heap contents by capturing a heap dump (see JOverflow). There are, however, many practical problems with heap dumps and heap dump analysis:

1. Heap dumps represent all of the heap content – they tend to get very large.
2. Heap dumps of large heaps take some time to produce, during which the JVM is stalled.
3. Since heap dumps, when originating from a production system, normally come from beefier computers than your laptop, they can be hard to process on a developer system at all.
4. Heap dumps contain all the (potentially confidential) data on the heap. All your customer information, your company secrets… Be very careful when you handle heap dumps. Maybe they should not reside on your laptop at all.
5. Multiple dumps represent a vast amount of data, making differential analysis very expensive in terms of both memory and processing.

Because of all of the above, you typically only do a heap dump when you know you will need one. This may require you to try to reproduce a problem that led to an `OutOfMemoryError`.

It would be quite useful to have some information from the production system allowing you to resolve the problem without having to go off and trying to reproduce it.

Since JDK 10 the JVM has a new event available – the Old Object Sample event. It is very useful little event that can help solve memory leaks without having to do heap dumps.

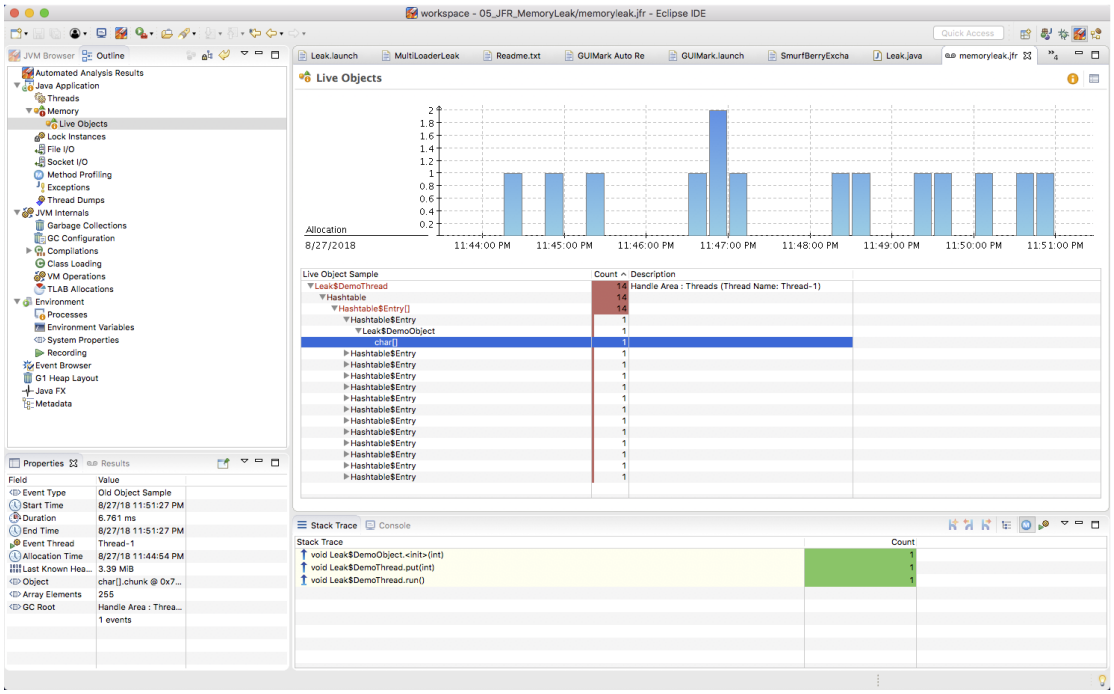
The Old Object Sample events will contain a sampling of the objects that have survived. The event contains the time of allocation, the allocation stack trace, the path back to the GC root. Since you may want to keep sampling, but perhaps not calculate the paths back to the GC roots on every dump, you need to specify the `path-to-gc-roots` parameter when dumping the flight recorder.

For more information on Old Object Sample events, please see the Old Object Sample article on Marcus' blog (hirt.se/blog).

Project 05_JFR_MemoryLeaks contains memory leaking applications. Open the memoryleak_before.jfr recording. It was taken before the memory leak was fixed.

Select an old object sample somewhere in the middle of the interval.

JMC-6127 is currently being fixed – instead look at the full interval.



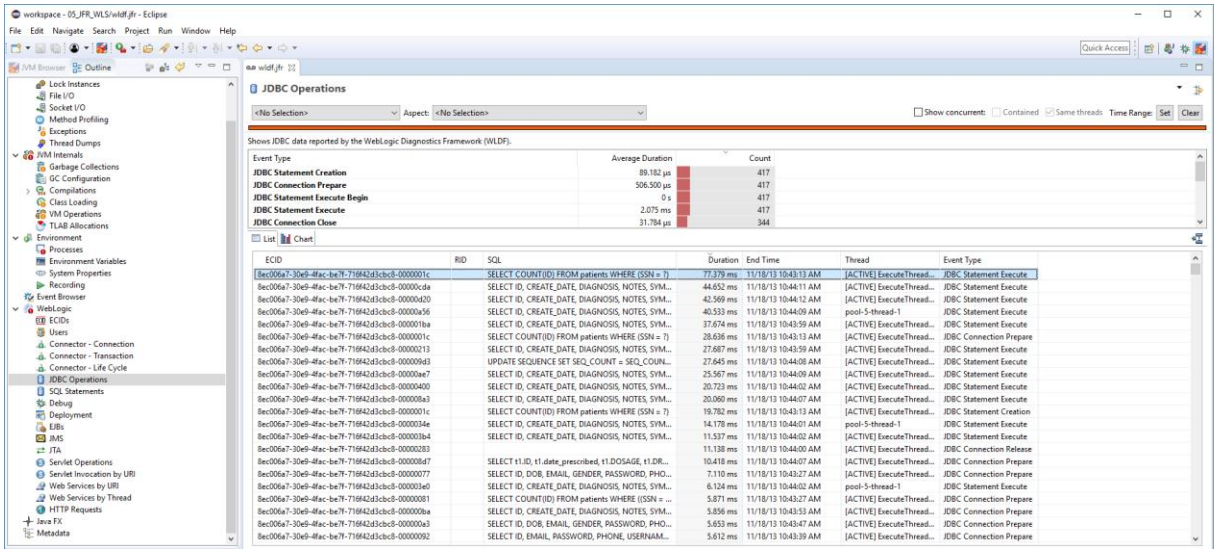
- Can you tell when it was allocated?
- Can you tell where it is being held on to?
- Looking at the source, can you figure out how to fix the problem?
- How many allocation events does the recording have?
- How many Old Object Samples are kept?
- Open the memoryleak_fixed.jfr. Do you see any differences?

Exercise 6 (Bonus) – WebLogic Server Integration

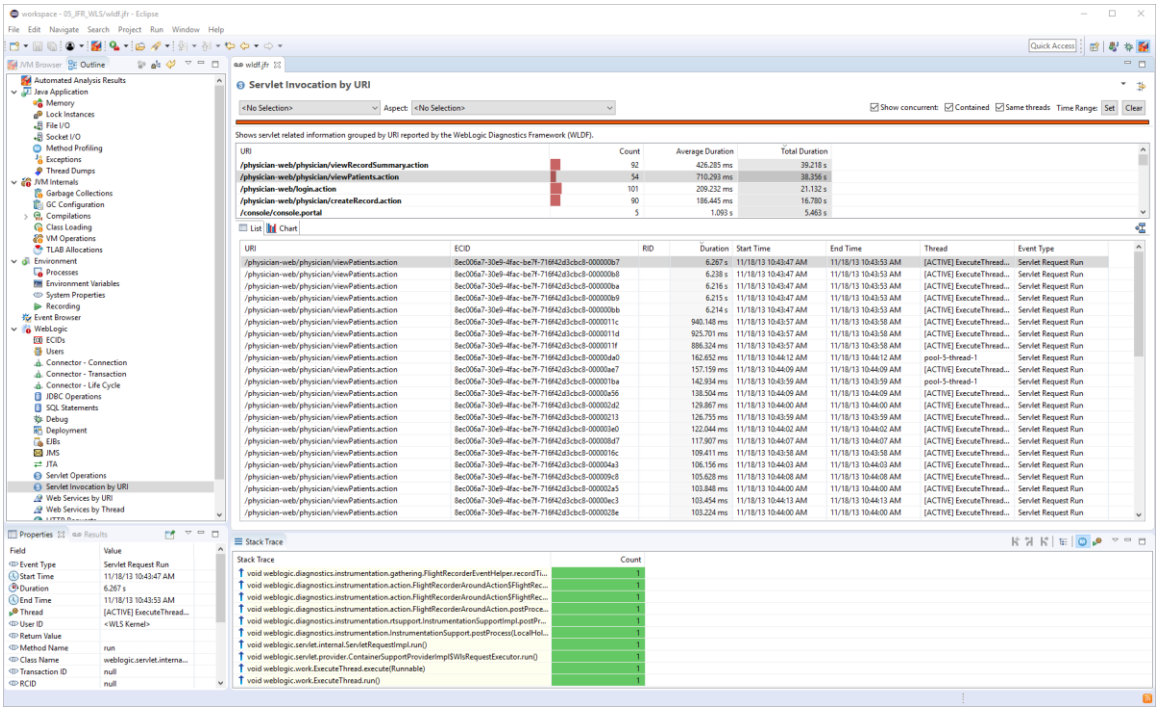
This exercise will familiarize you with various elements of the new user interface. It also shows that it is possible to create integration with JFR and JMC from a party outside of the JDK, in this case the WebLogic Diagnostics Framework (WLDF). Even if you are not using WLS, this is a good exercise,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

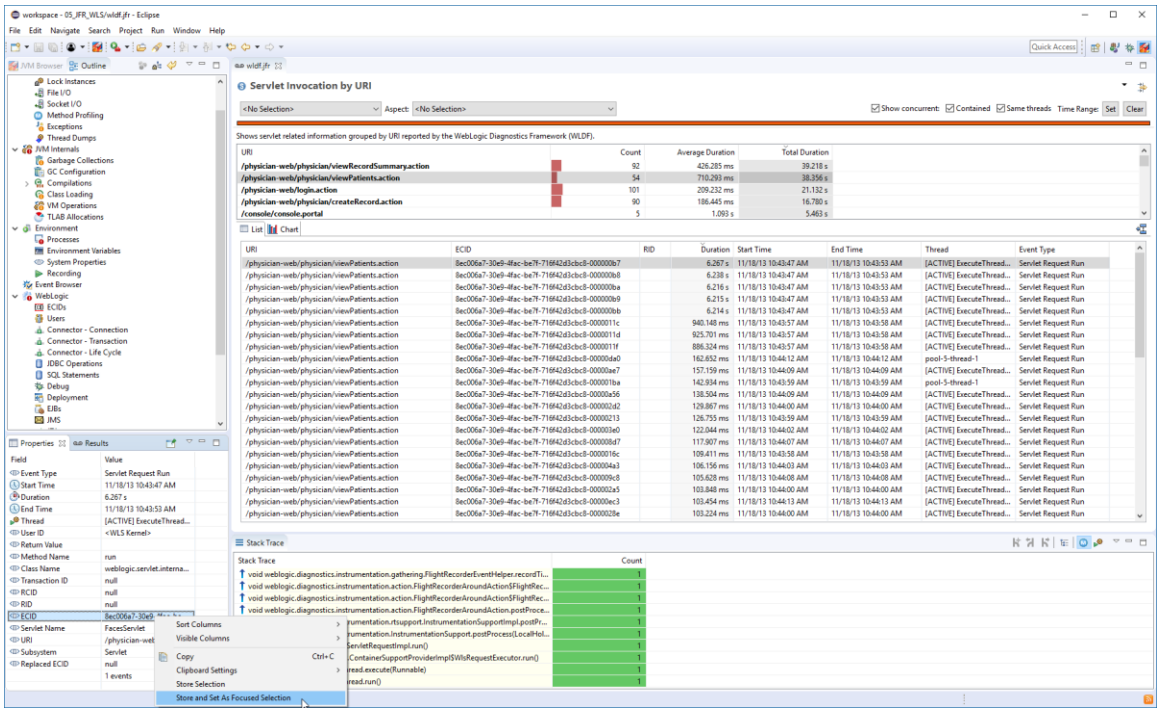


Open the WebLogic / Servlet Invocation by URI page. Can you tell which invocation of the viewPatients servlet took the longest time? How long did it take?



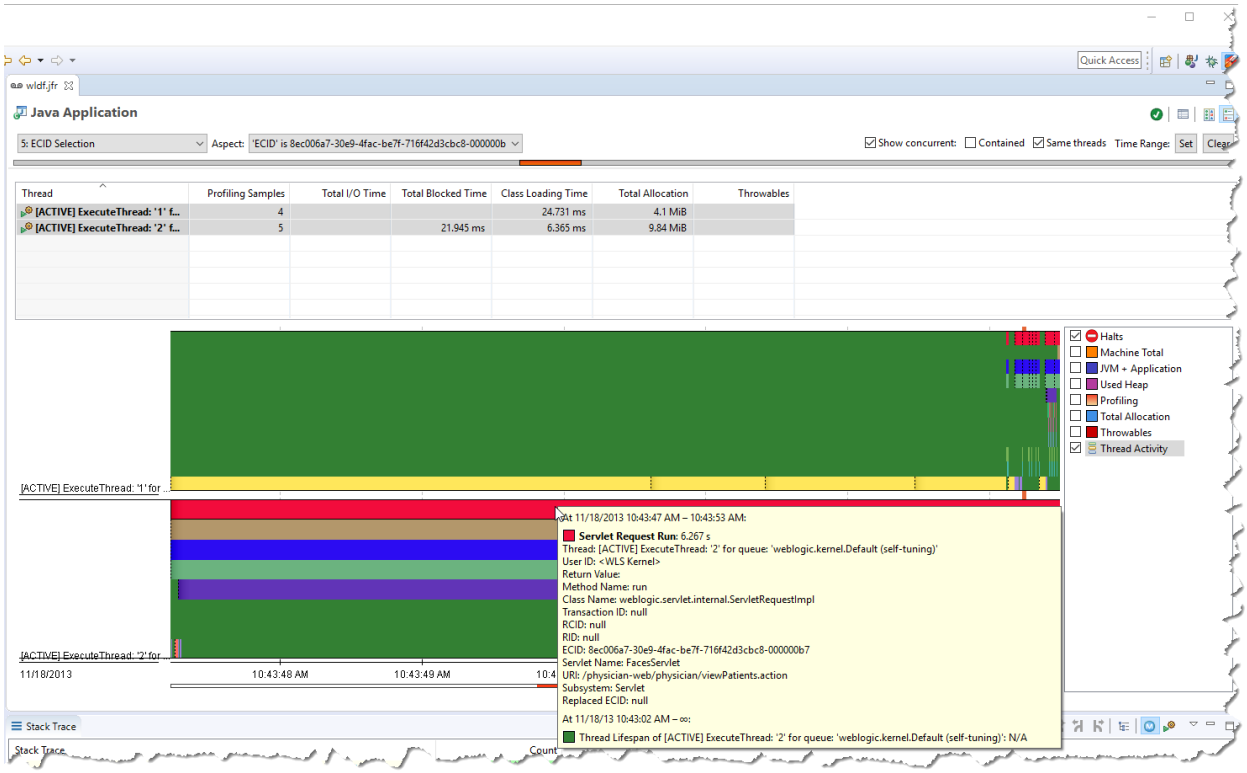
Let’s take a look at everything that was going on during that request. Select the longest lasting viewPatients servlet, and select the ECID (Execution Context ID) in the Properties view. An ECID is an identifier which follows a request through the system across process and thread boundaries. A little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

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



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.

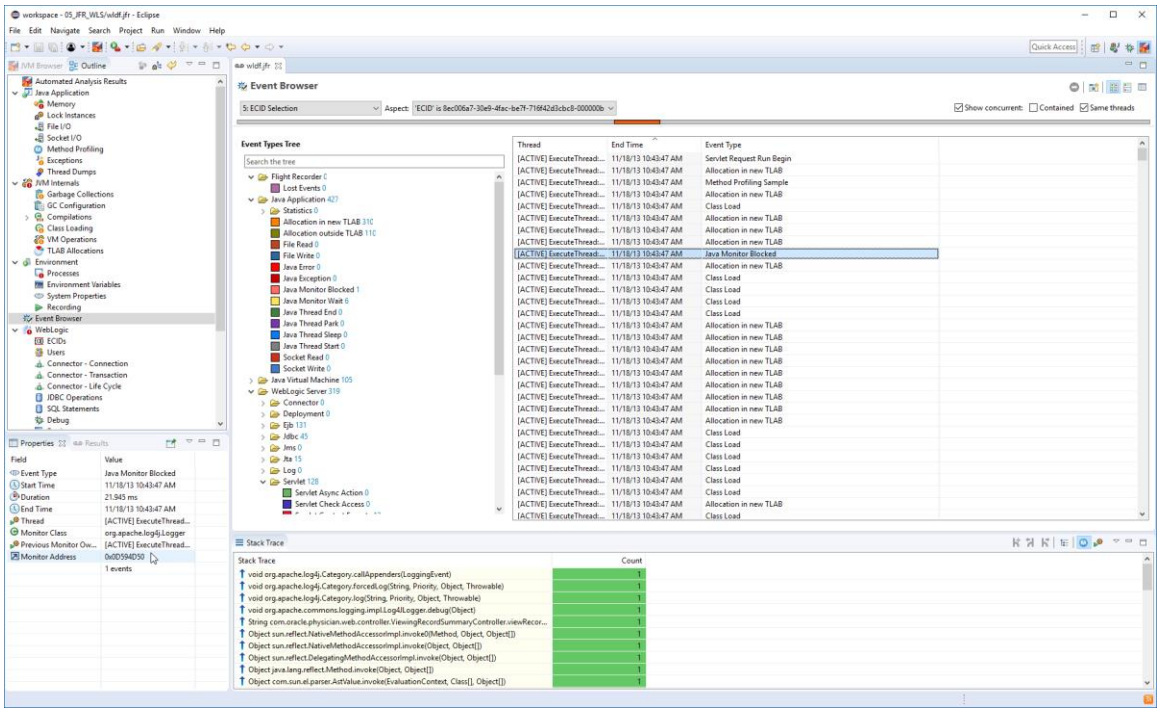


Also note that we can select other aspects of the selection to determine what the UI shows. Some pages may only be able to act on certain aspects of a selection. If the Show concurrent box is checked, events concurrent (happening during the same time interval) to the selection will also be showed. If 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:

5. 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.*

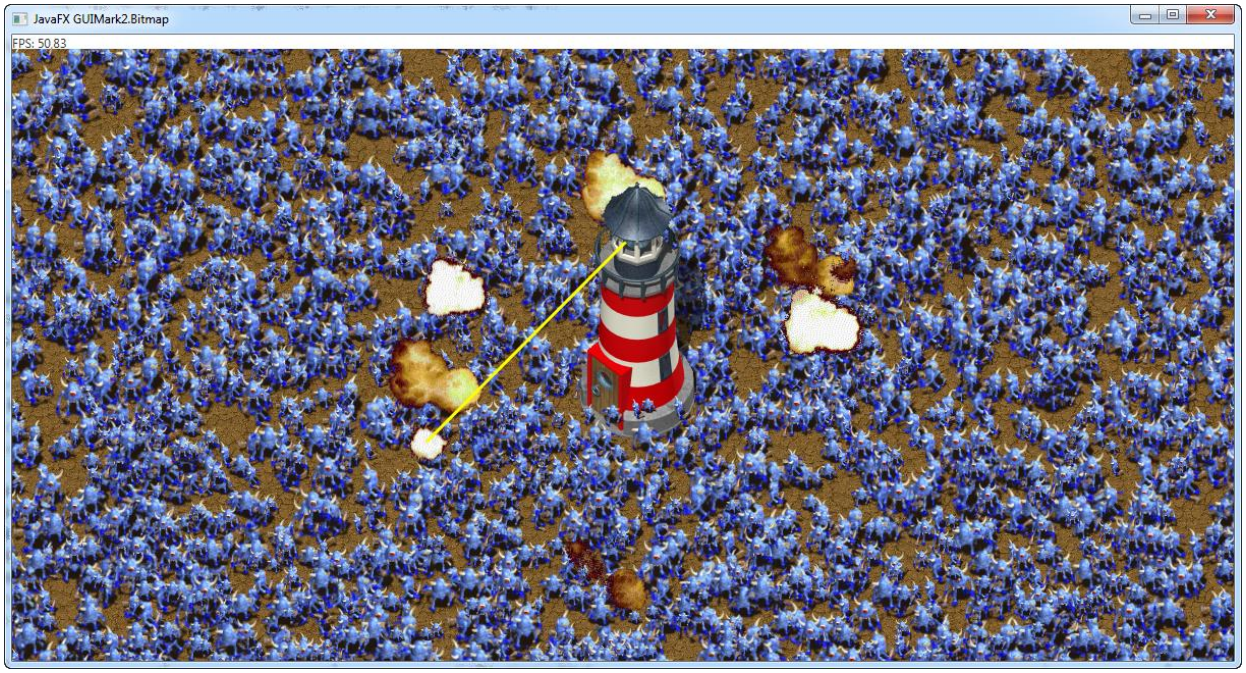
6. Can you find out which EJB the application seems to be spending the most time in on average?

7. 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.

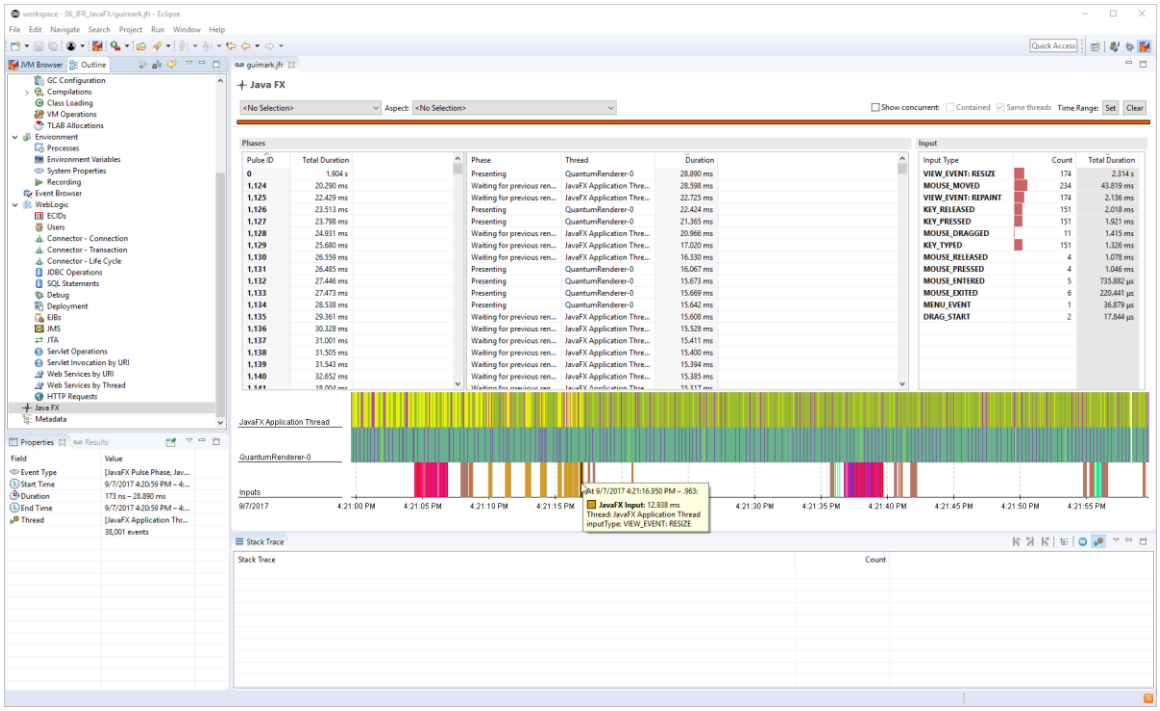
Exercise 7 (Bonus) – JavaFX

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

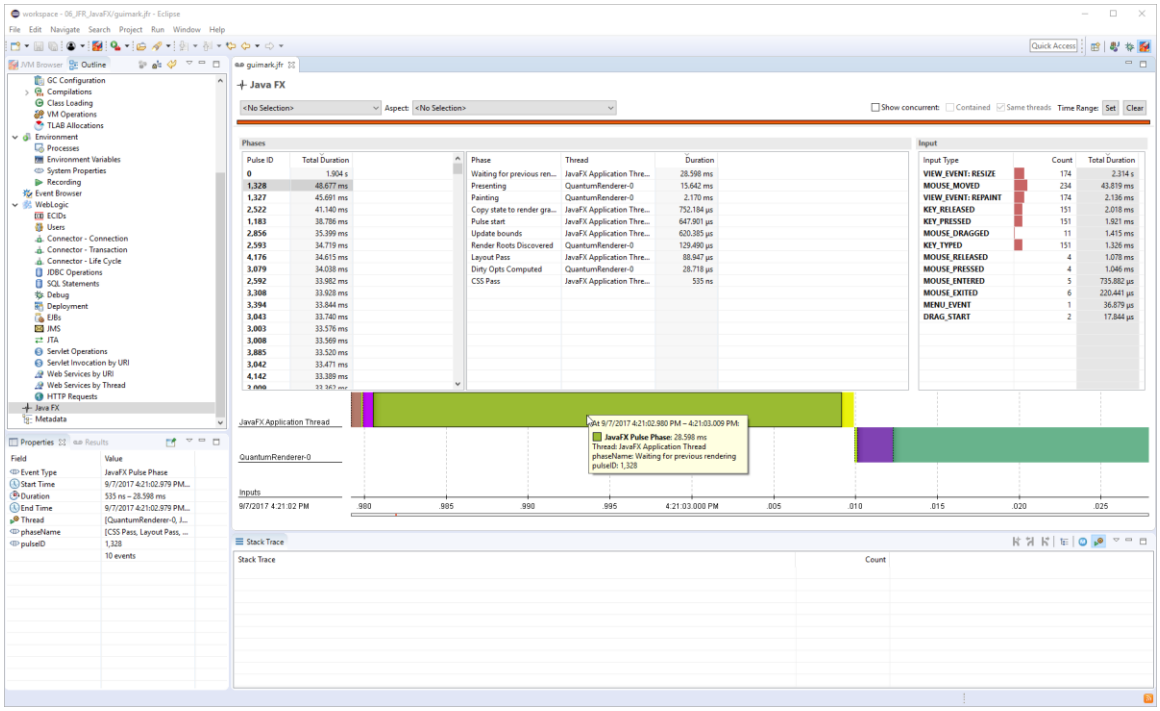


We will use a pre-recorded recording for this exercise, **`06_JFR_JavaFX/guimark.jfr`**, so you starting GUIMark was mostly pointless. But, come on – lasers? Monsters? It had to be seen.

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?



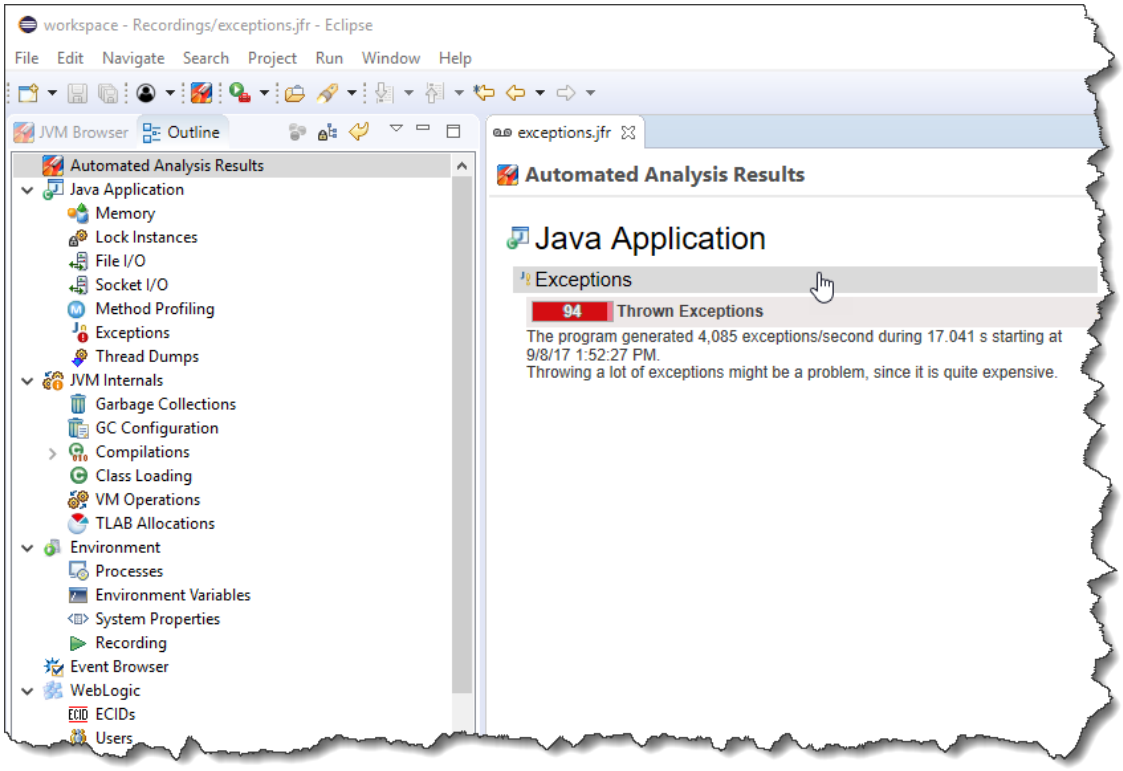
Exercise 8 (Bonus) – Exceptions

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.

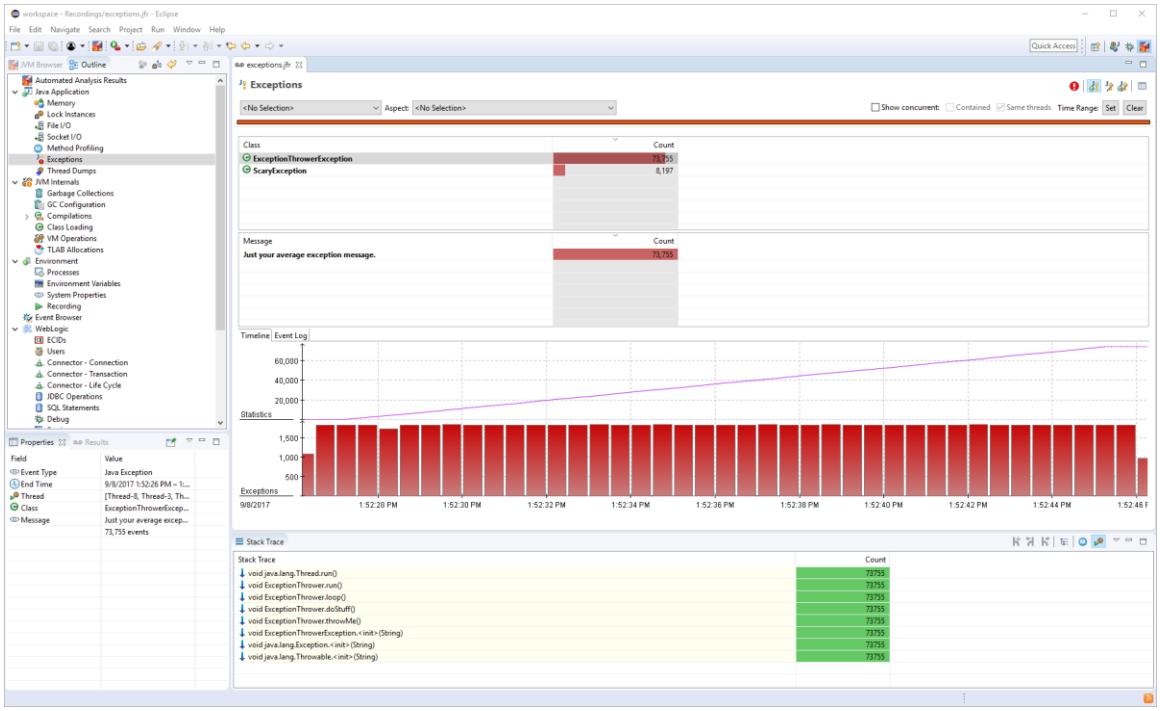
Open up the flight recording named `exceptions.jfr` in the 07_JFR_Exceptions project.

The automated analysis should indicate that things could be better:

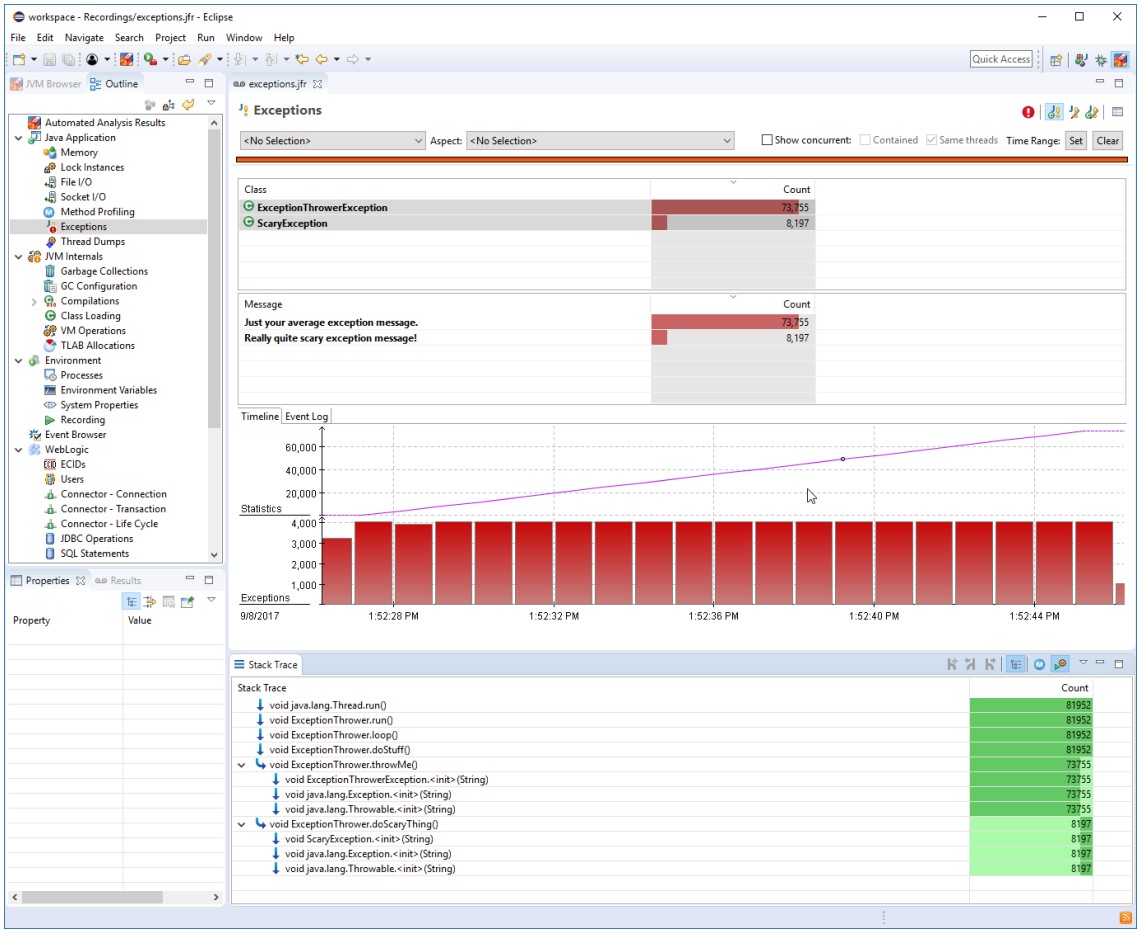


Click on the Exceptions header above the rule result (or the Exceptions page in the Outline) to go to the Exceptions page.

Can you tell how many exceptions were thrown? Where did the exceptions originate in code?

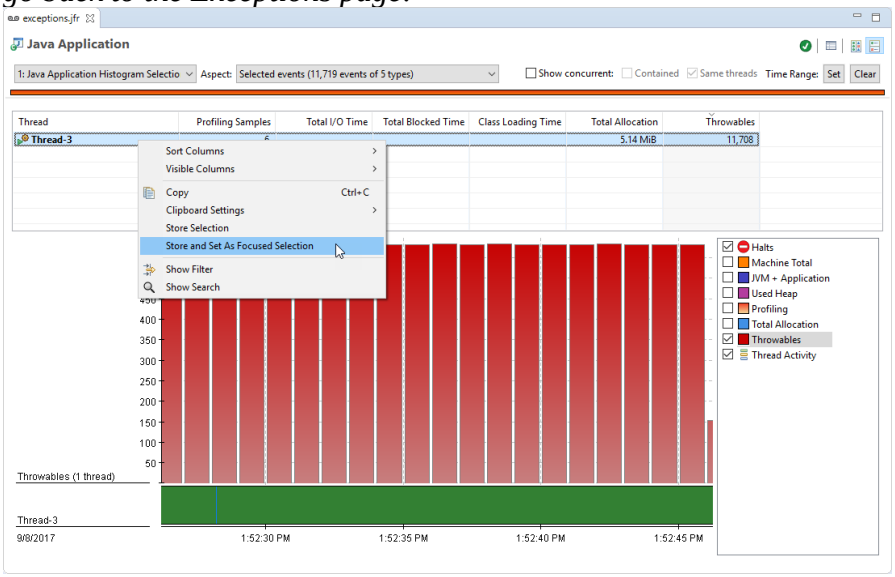


Note: *Click the exception class you want stack traces for. You can also select multiple classes to see the aggregate traces for all classes in the selection.*

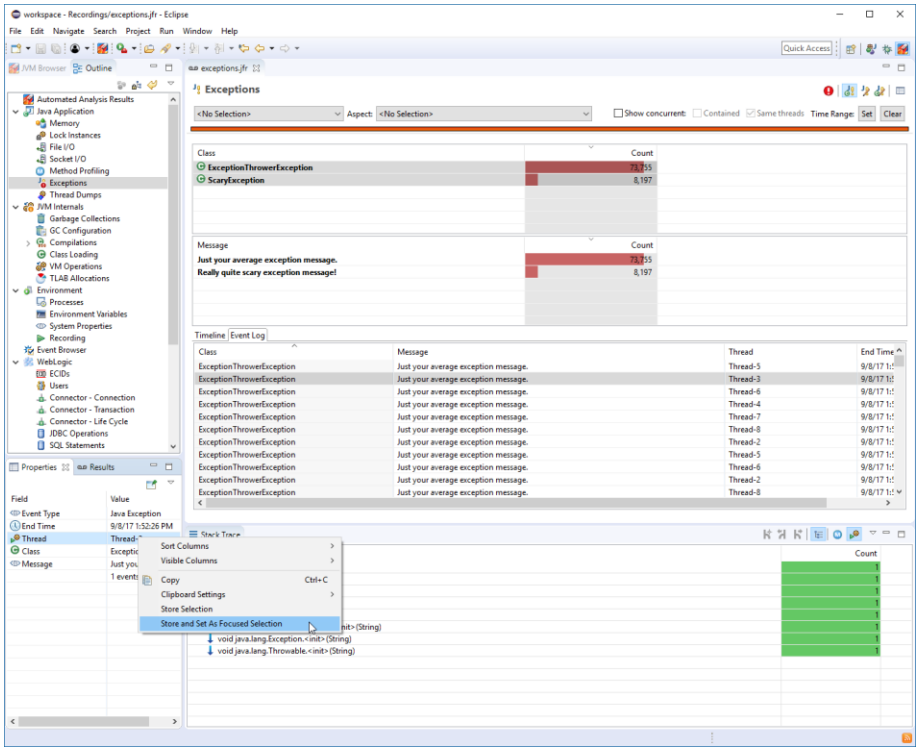


Can you tell exactly where the Scary exceptions are originating? In what threads are they originating? Can you study the time line for the exceptions in just one of the threads?

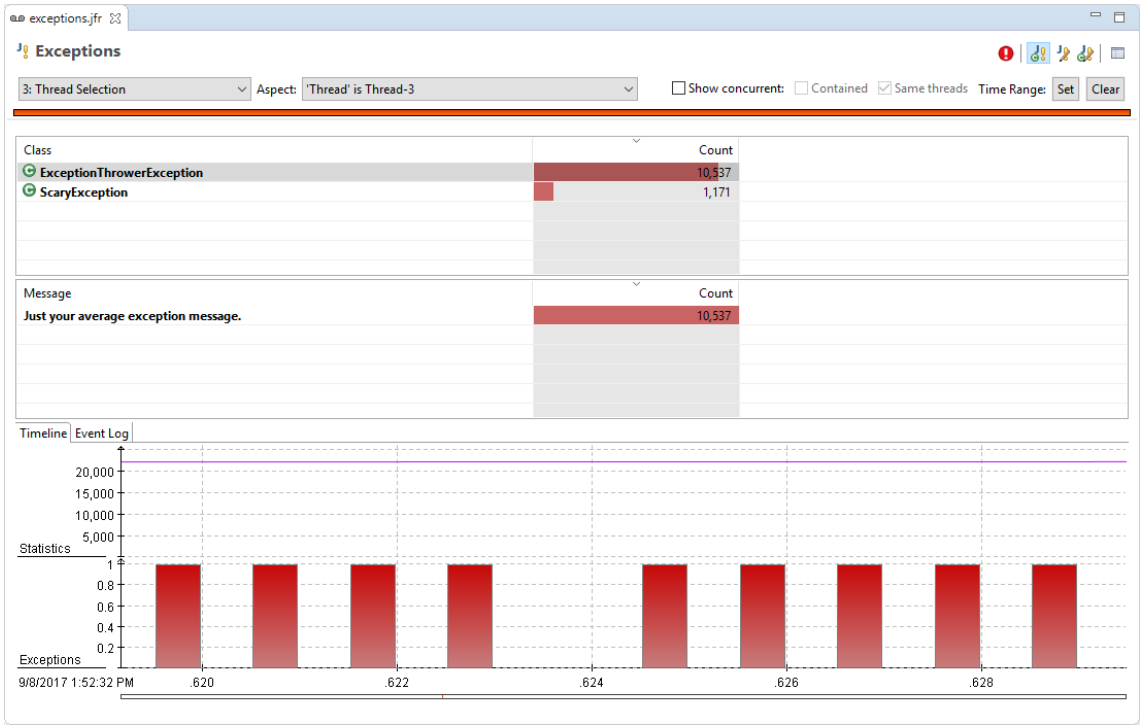
Note: *Either go back to the Java Application page class, sort on Throwables in the thread table, and pick a Thread and select Store and Set As Focused Selection then go back to the Exceptions page:*

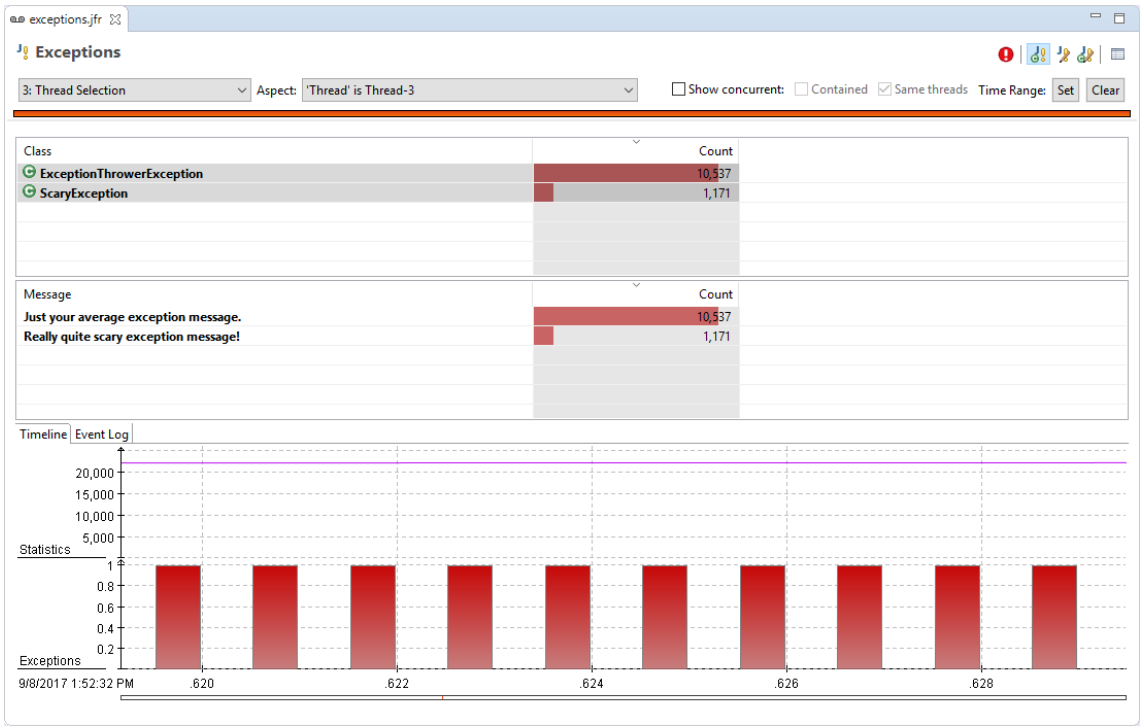


…or pick an event in the Event Log directly on the Exceptions page, then select the Thread in the Properties view and use Store and Set As Focused Selection from the context menu:



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





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

Exercise 9 – Custom Events in JDK 9 (Bonus)

In this exercise we will use an example from a different repository – the java-svc repo. Clone the following repository:

https://github.com/thegreystone/java-svc

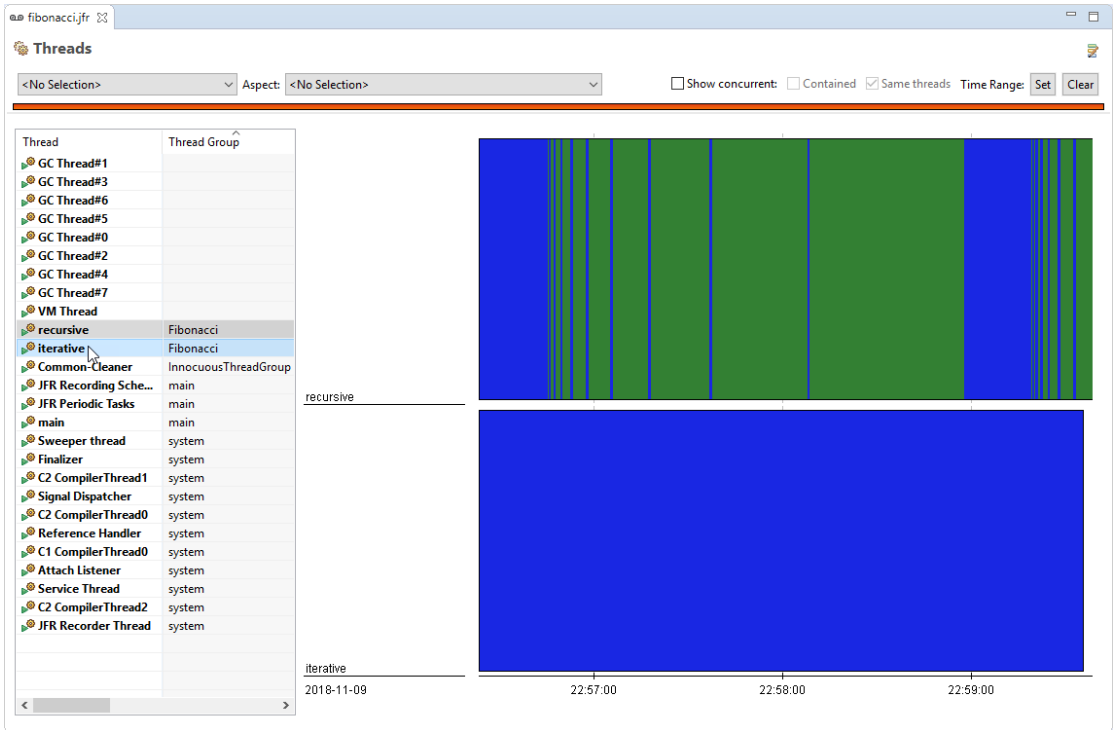
Follow the instructions to build. Make sure you use a JDK 11 or later.

Open a command line and go to the **`java-svc/jfr`** folder.

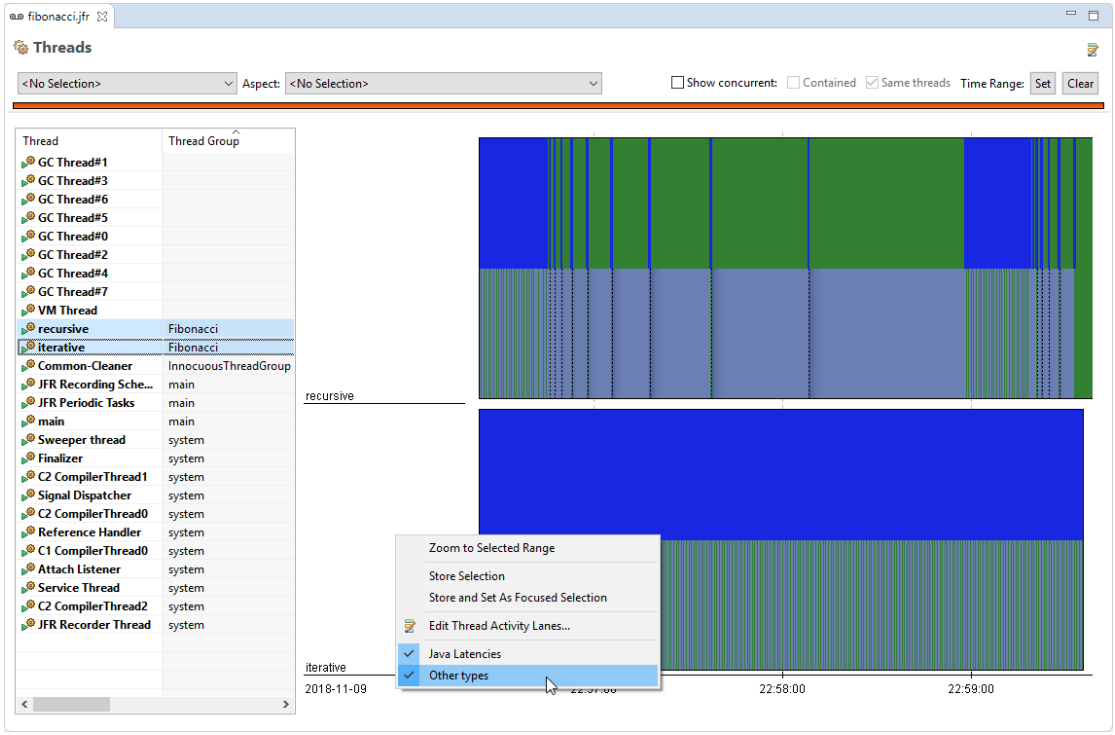
Next run `target/bin/runFibonacci.`

Connect to the process with JMC. The process starts with a recording already running, so you can simply dump the whole thing when you feel ready. Either let it run for a few minutes and then dump it, or go with the pre-recorded recording present in the 09_JFR_CustomEvents project.

Open the Threads view. Sort on Thread Groups and shift-click to select the two threads in the Fibonacci thread group:



Right click on the graph and enable the viewing of other event types in a separate lane per thread:



You can now hover over the events to see what they contain.

Deep Dive Exercises:

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

Hint: *Go to the Event Browser, click on the Fibonacci Event. Sort on Duration.*

9. Verify that the iterative version is not cheating by making sure that the calculation of both the iterative and recursive version of the algorithm come to the same value.

Hint: *Go to the Event Browser, click on the Fibonacci Event. Sort on Fibonnaci Number.*

10. Did you notice any difference in performance between the two algorithms?

Exercise 10 – 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, 9, 10 and 11 recordings transparently. It is the parser used in JDK 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 JDK Mission Control rules headless using the JMC core libraries. A custom rule will also be created, using the JDK Mission Control Eclipse PDE support.

JMC in JShell

First we will clone a different repository:

> **`https://github.com/thegreystone/jmc-jshell`**

Next follow the instructions in the README.md to try some of the various things that can be done using the JMC core API.

In JDK Mission Control 7 there are two built in classes for producing HTML reports.

The `JfrRulesReport` class can output the results of the automated analysis in xml and html, and which can be configured with a custom xslt.

The `JfrHtmlRulesReport` class will output html very similar to the Automated Analysis Results page in JMC.

Deep Dive Exercises:

11. Try analysing some of the recordings you have been analysing so far.

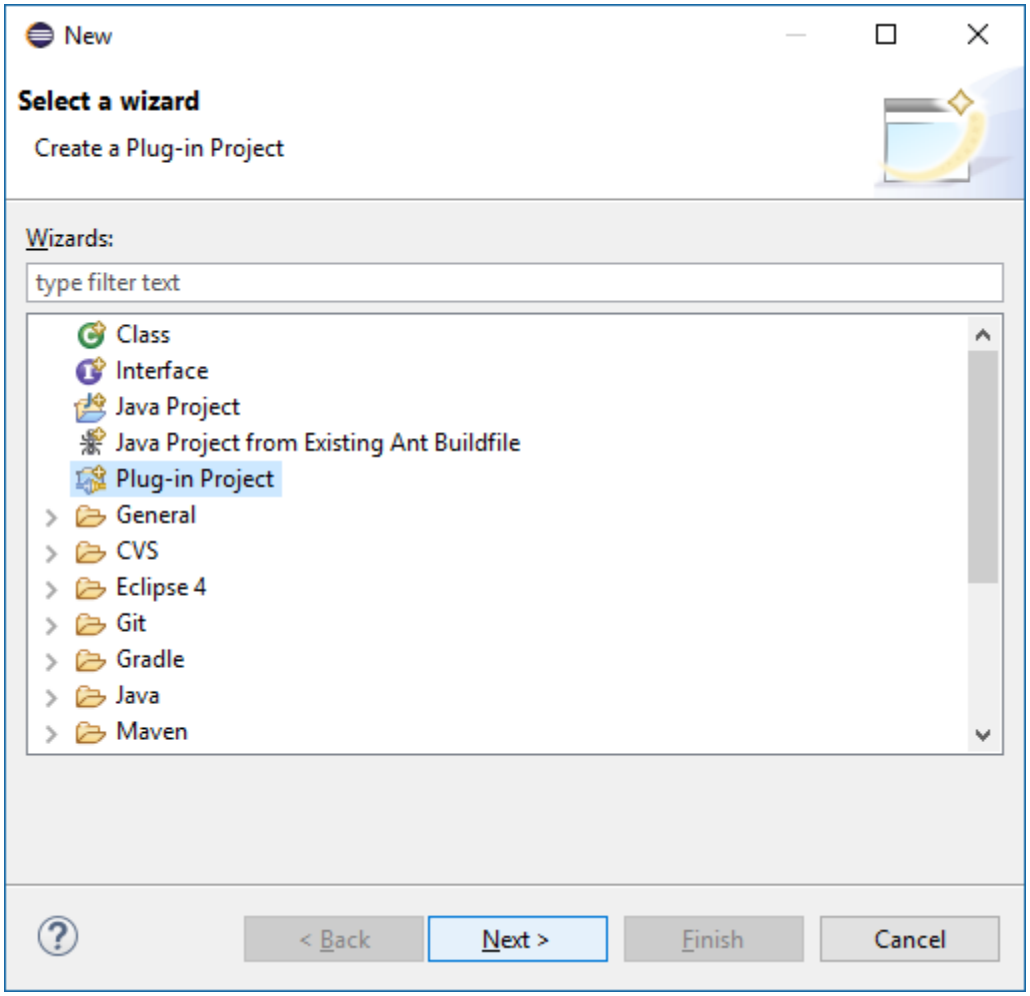
12. Try using both JfrRulesReport and JfrHtmlRulesReport, with various arguments, to perform the analysis.

Creating Rules

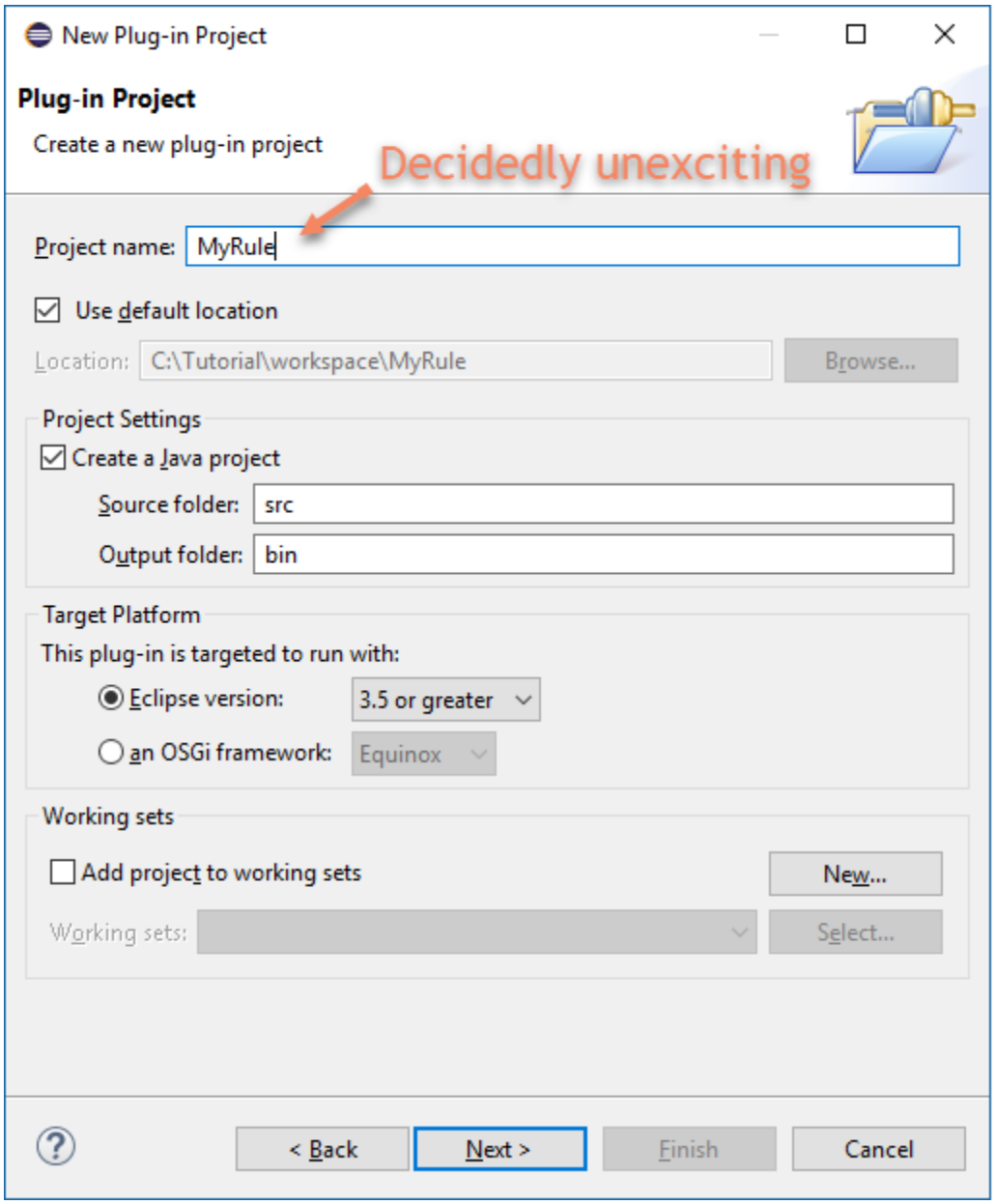
The easiest way to get started creating rules is to use the PDE plug-in for Eclipse.

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).

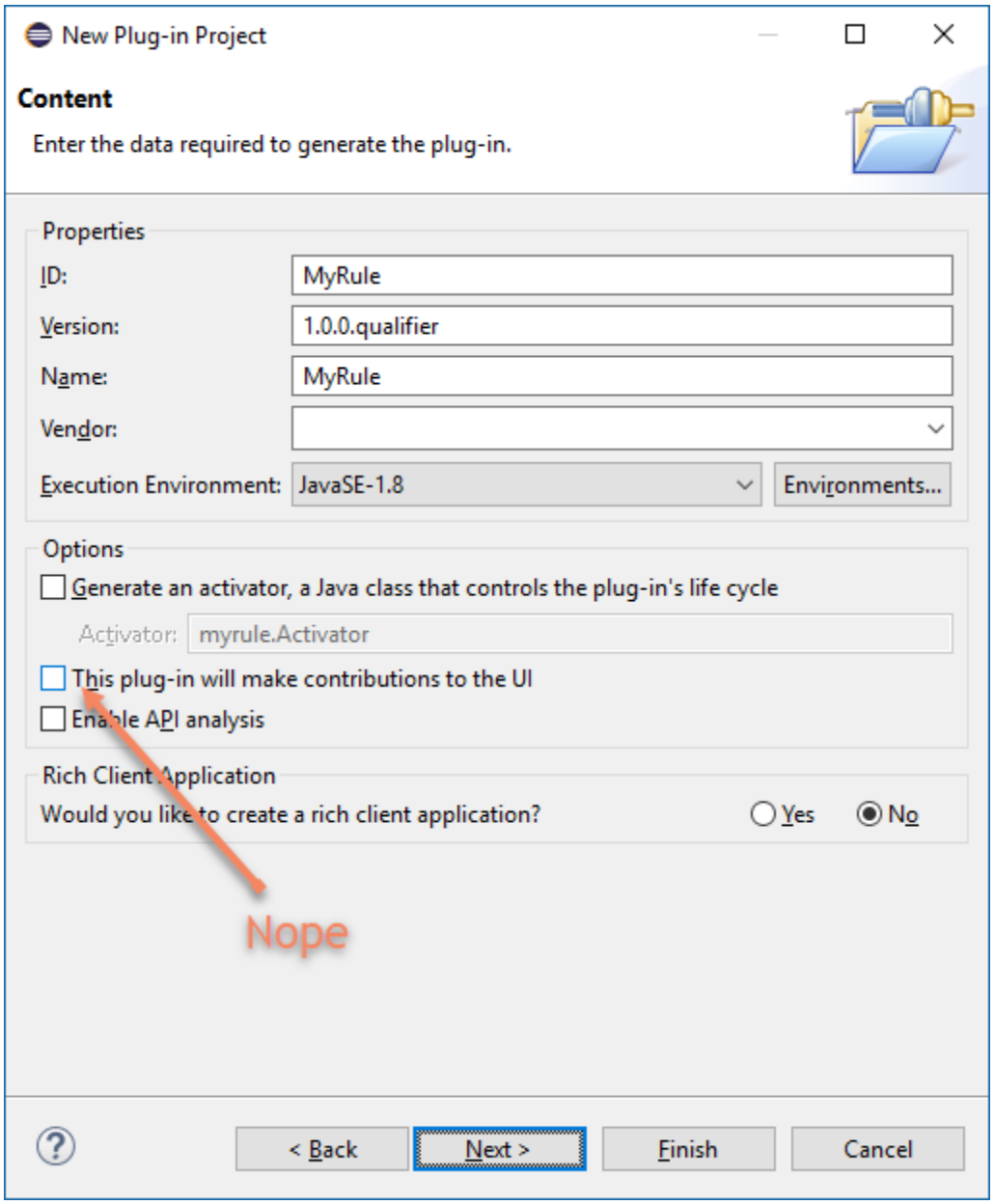
Press ctrl-n (or click the File | New | Other… menu) to bring up the New wizard.



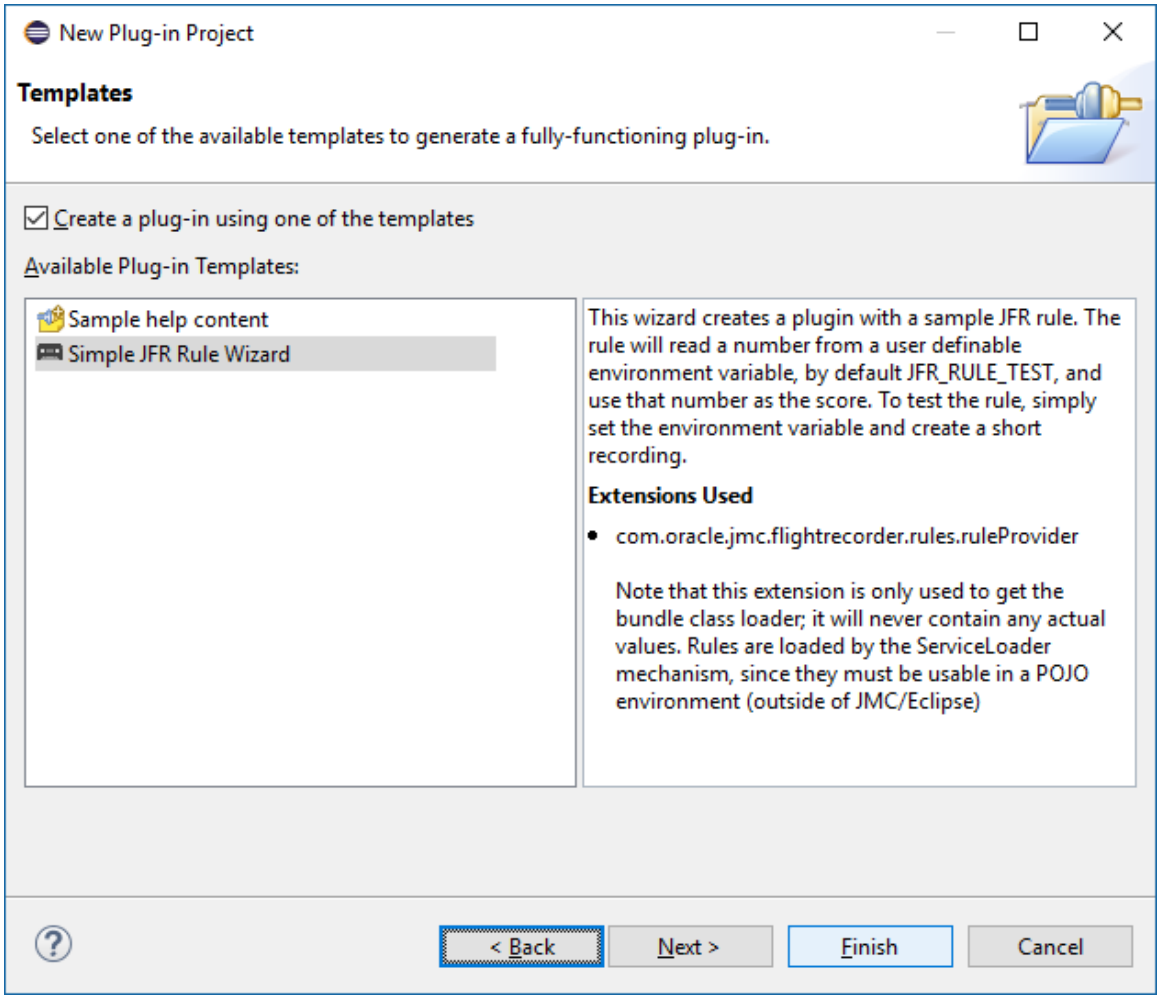
Select Plug-in Project and hit Next. Name your rule project something exciting.



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

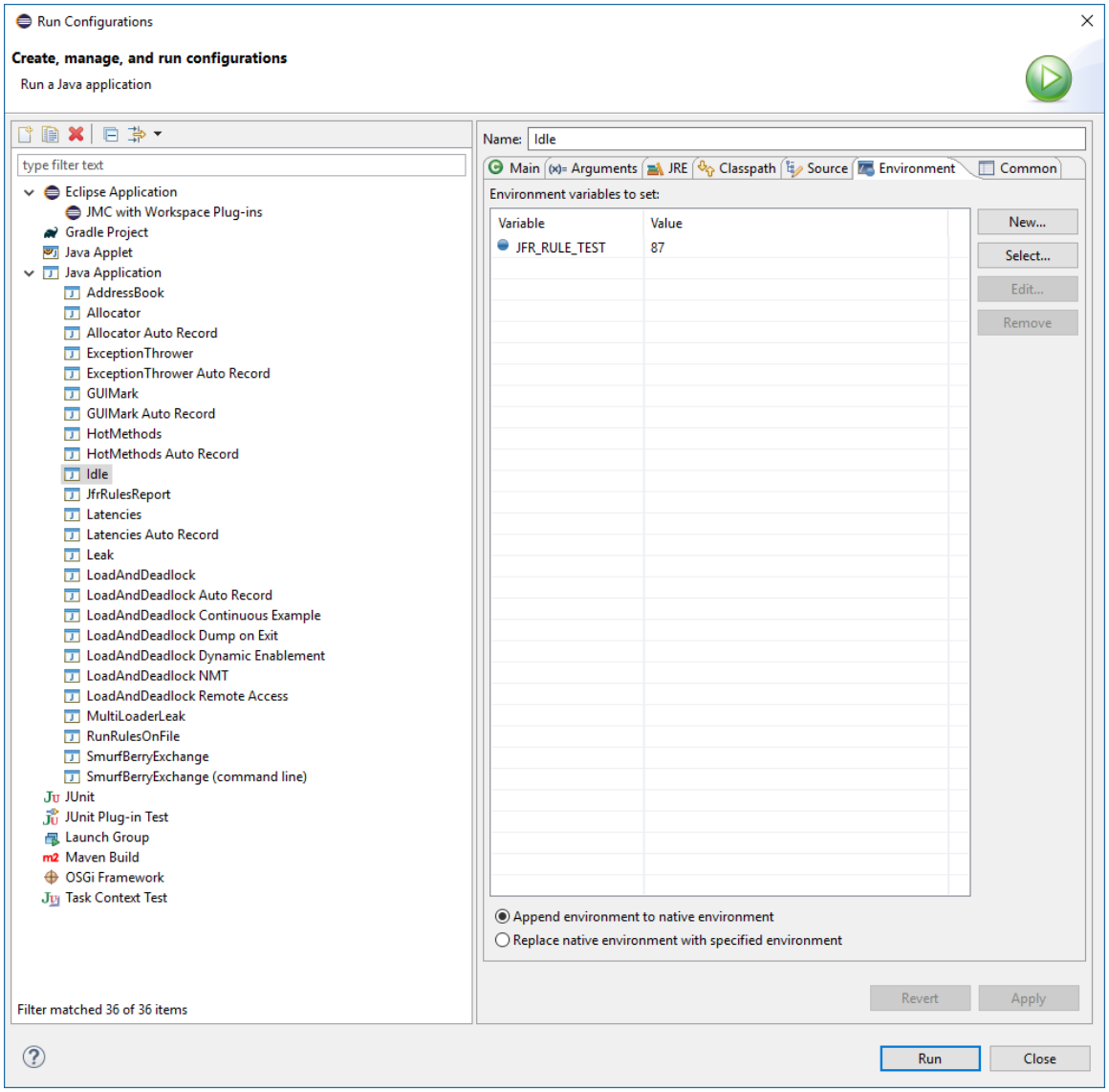


Next select the Simple JFR Rule Wizard and click Finish (or Next, if you really wish to do some further customizations).

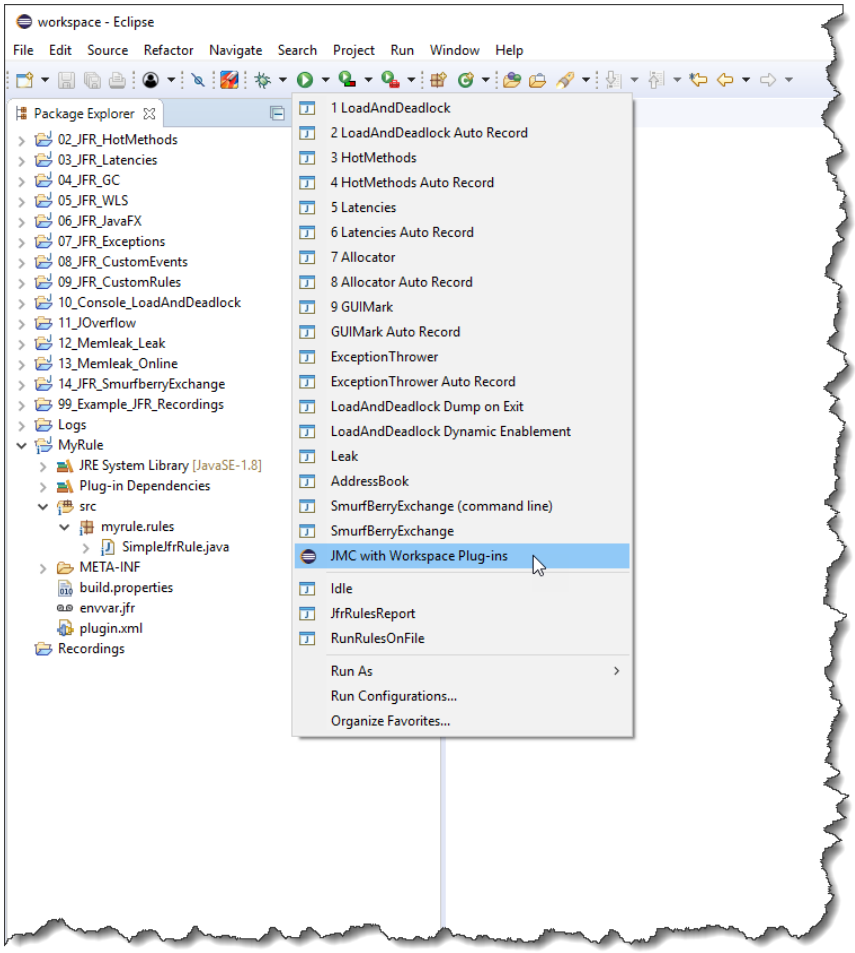


You will now have a new project in your workspace. Open up the project and study the source of the rule. Can you see what it does?

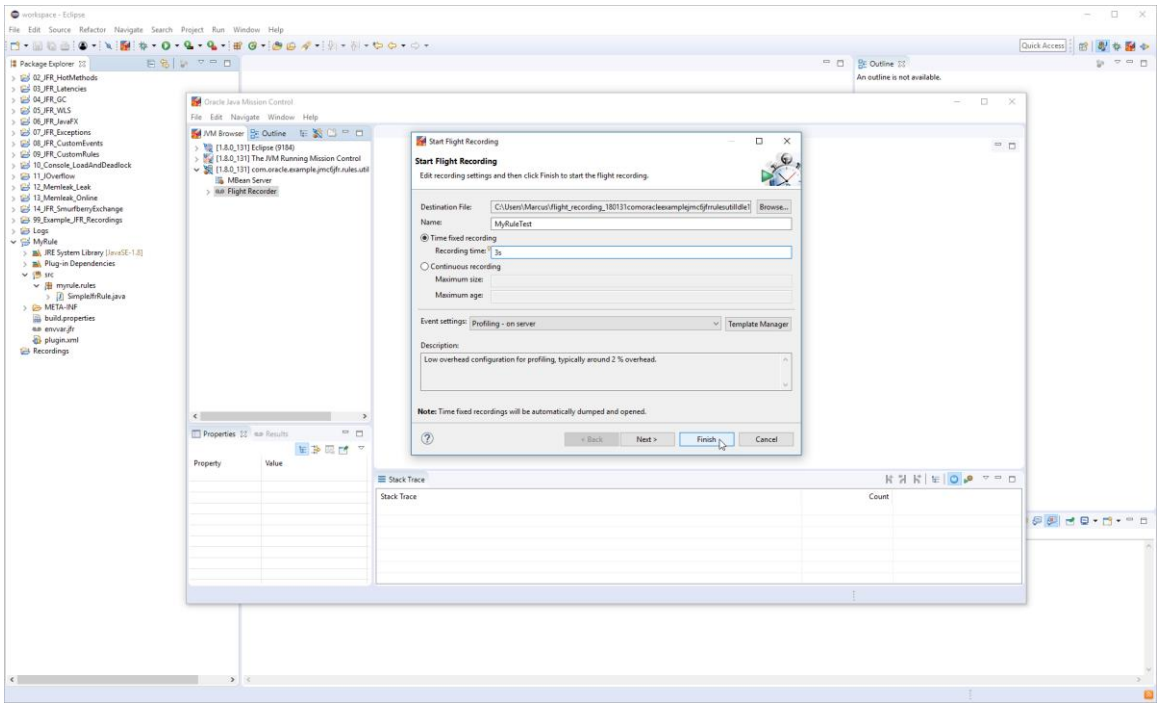
Go to the Run Configurations and check out the Idle launcher. In the Environment tab you can set up an environment variable. Launch the Idle launcher by hitting Run.



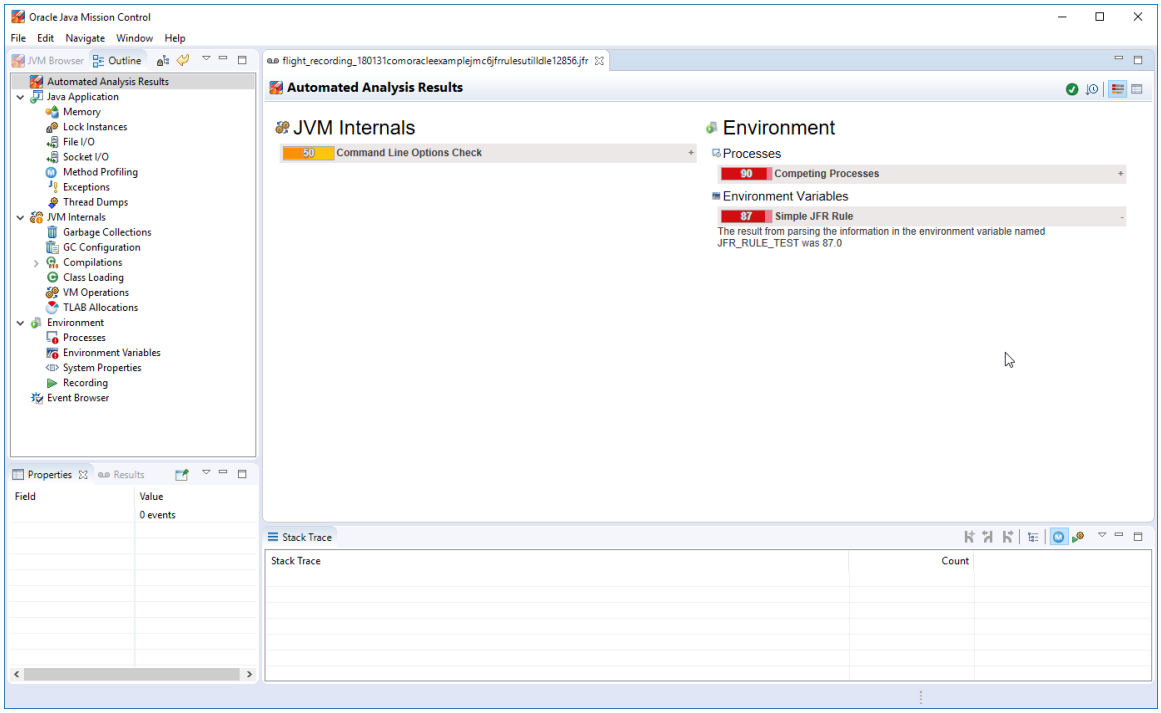
Next launch a JMC with your rule running by launching JMC with Workspace Plug-ins.



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

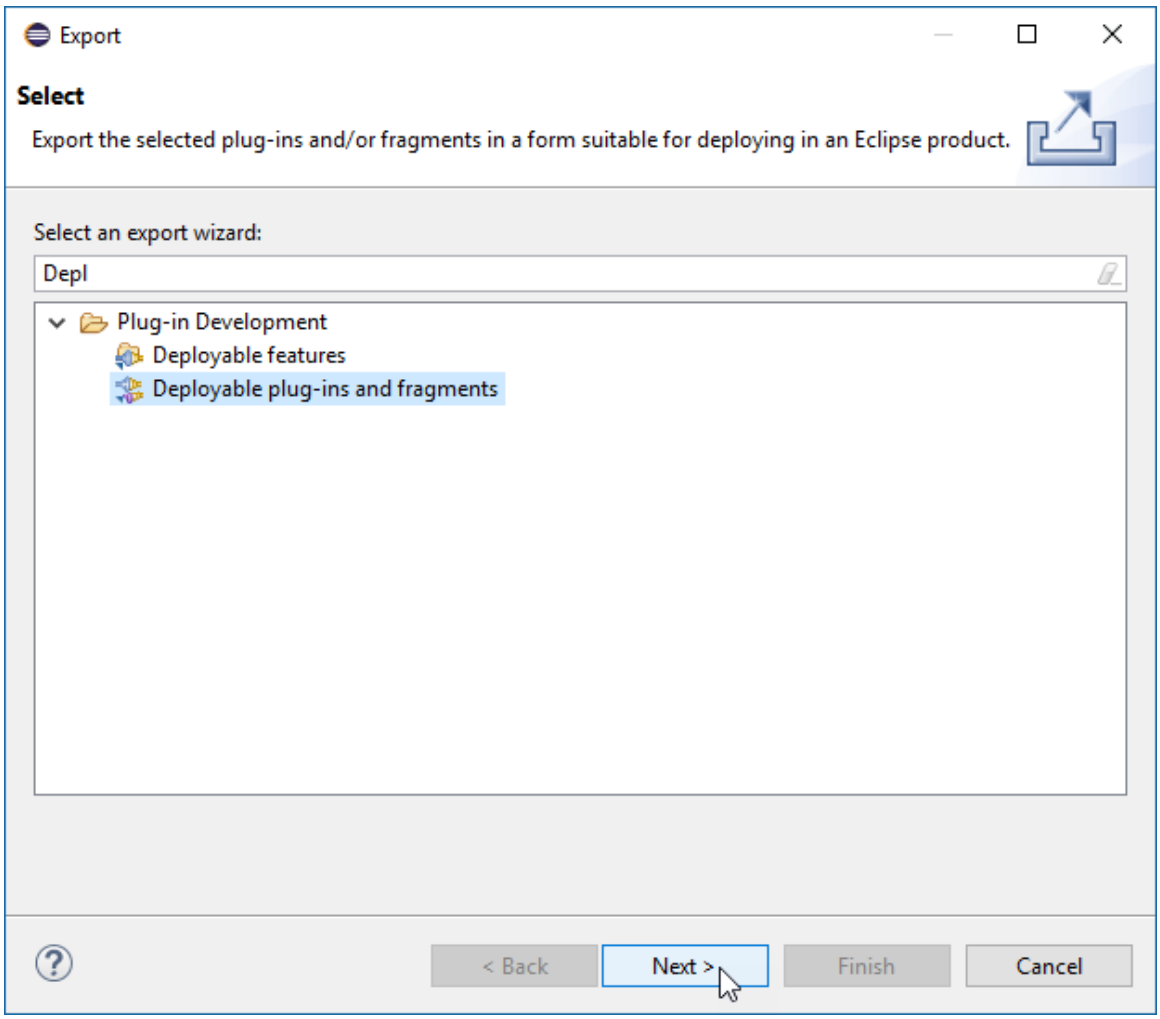


Depending on the value you set the environment variable to, you should now see your rule triggering:



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.

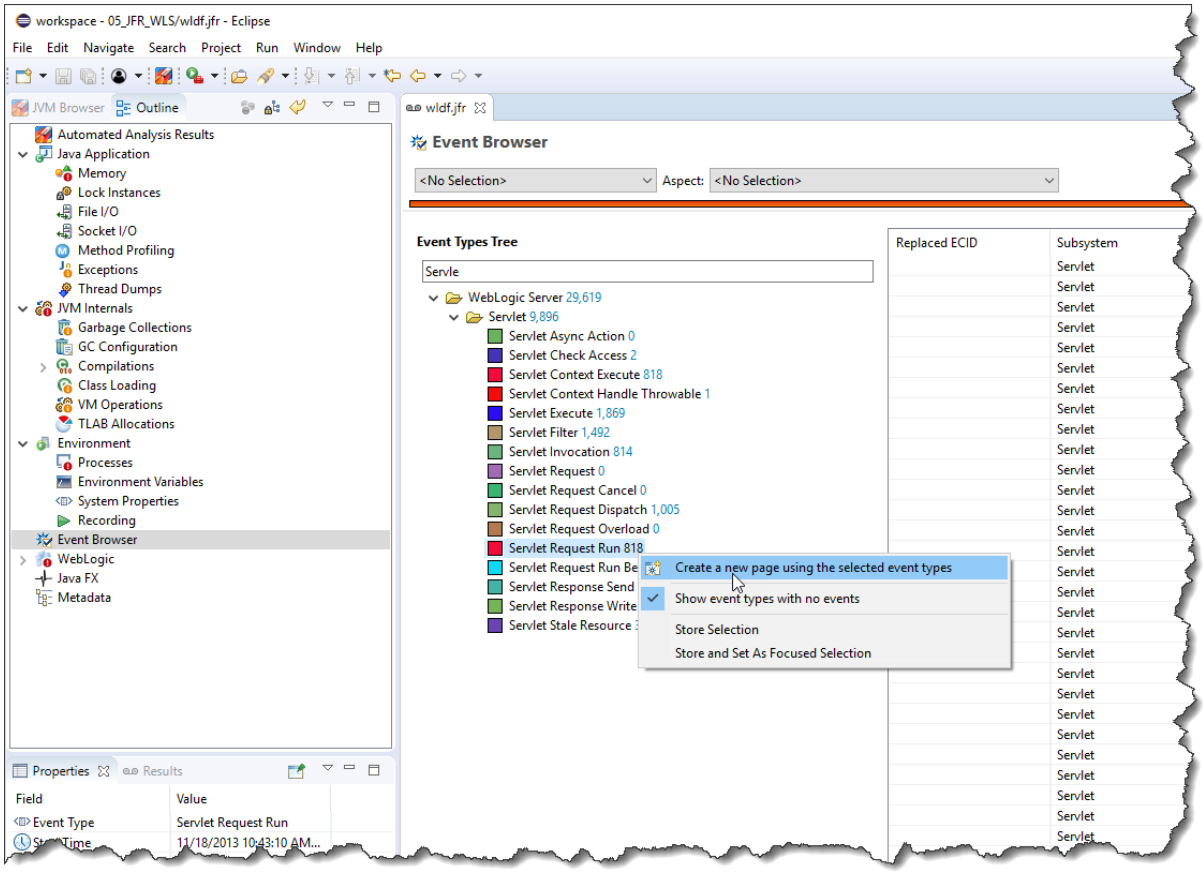
Exercise 11 – Custom Pages

After some time with JDK 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.

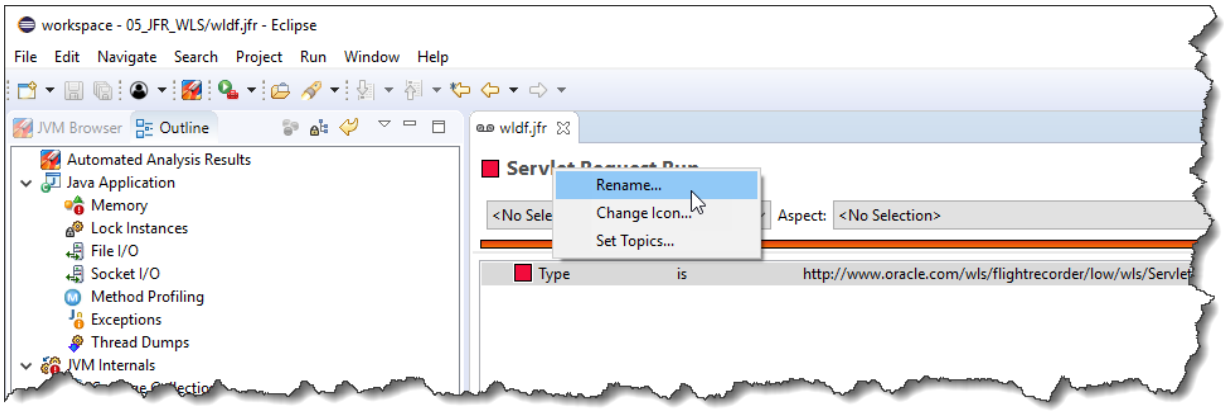
We will use the recording from the 05_JFR_WLS project for this exercise. Open the **`wldf.jfr`** recording, and switch to the JDK Mission Control perspective.

Filters

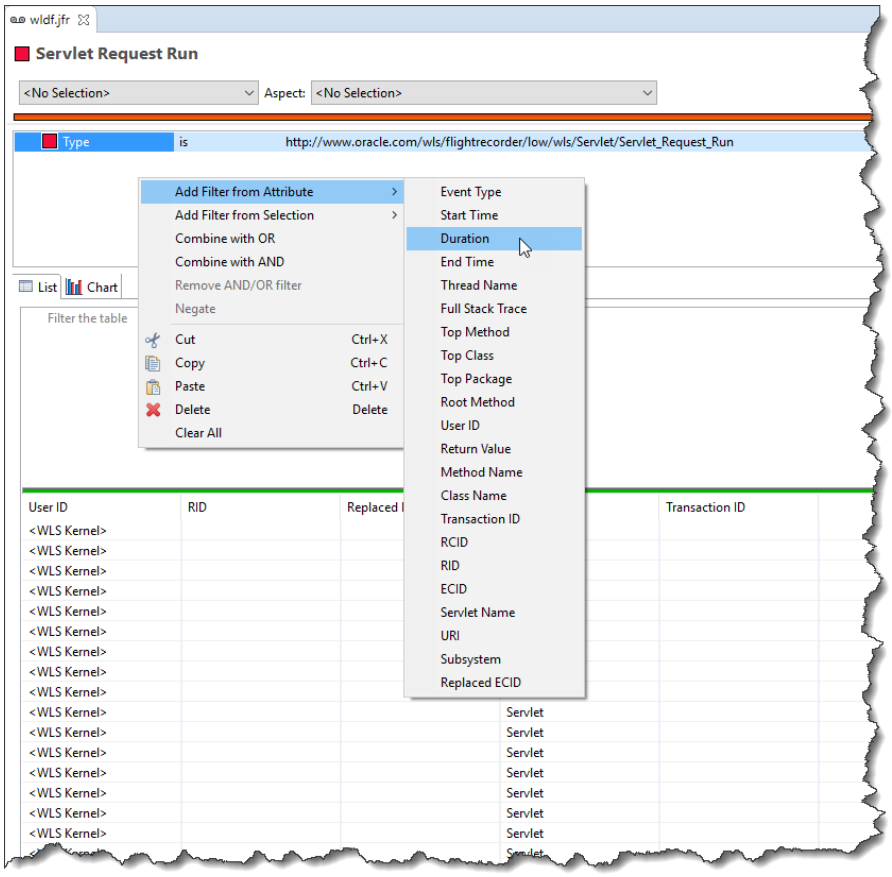
We would like a nice view of the servlet requests taking longer than 2 seconds. Start by going to the Event Browser page. Use the filter box to quickly find the Servlet Request Run event type, and then use the context menu to create a new page using the selected event type.



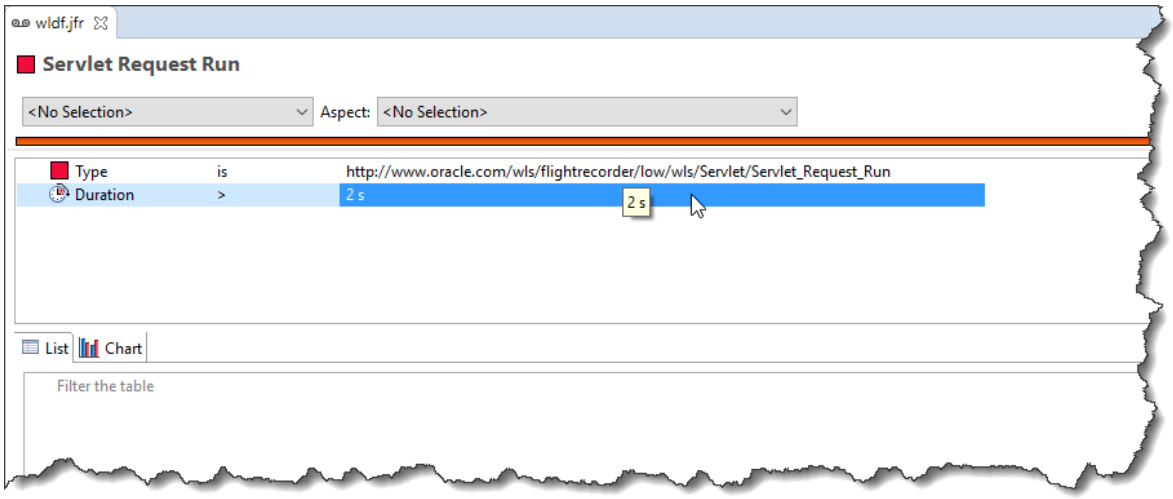
Right click on the name of the new page, and name it “Long Lasting Servlets”:



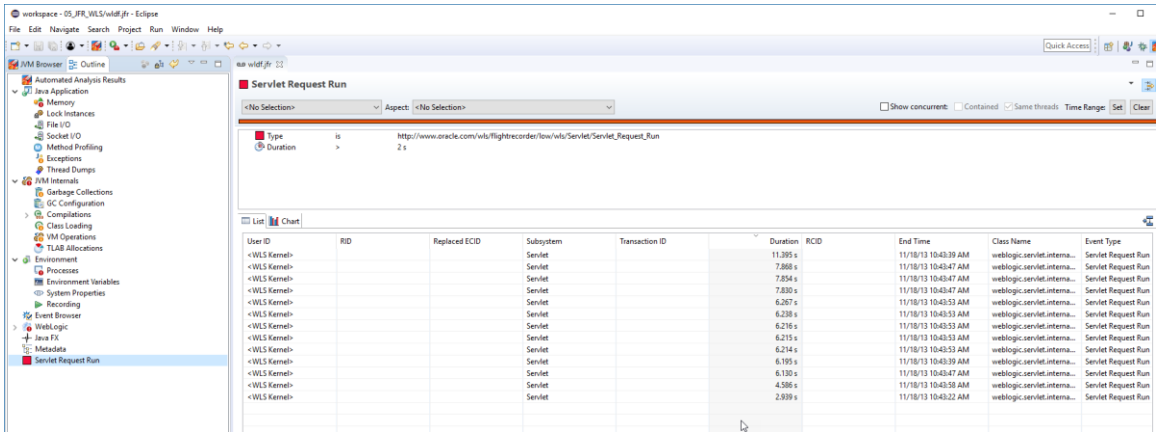
Add a new attribute filter on Duration:



And set it to > 2 seconds:

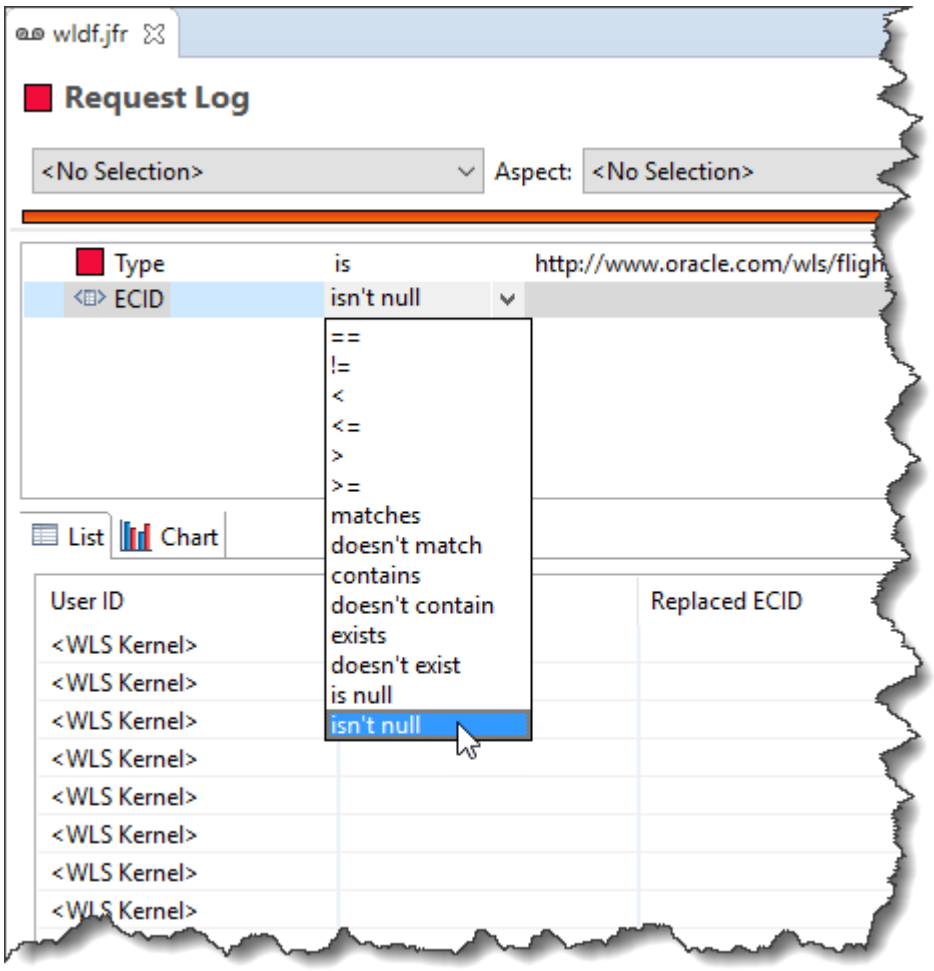


Now we have a custom page showing the longest lasting servlet requests:

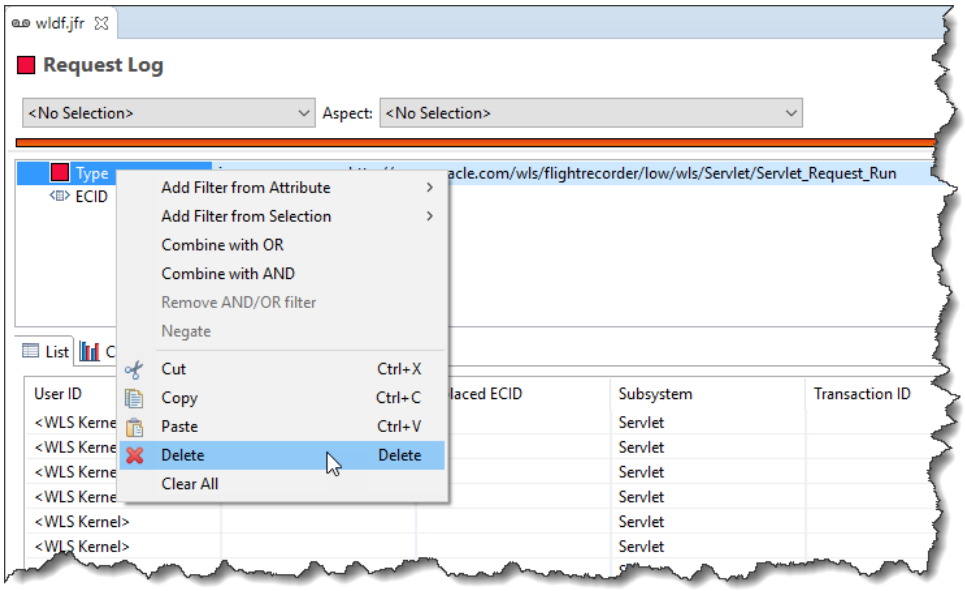


Grouping

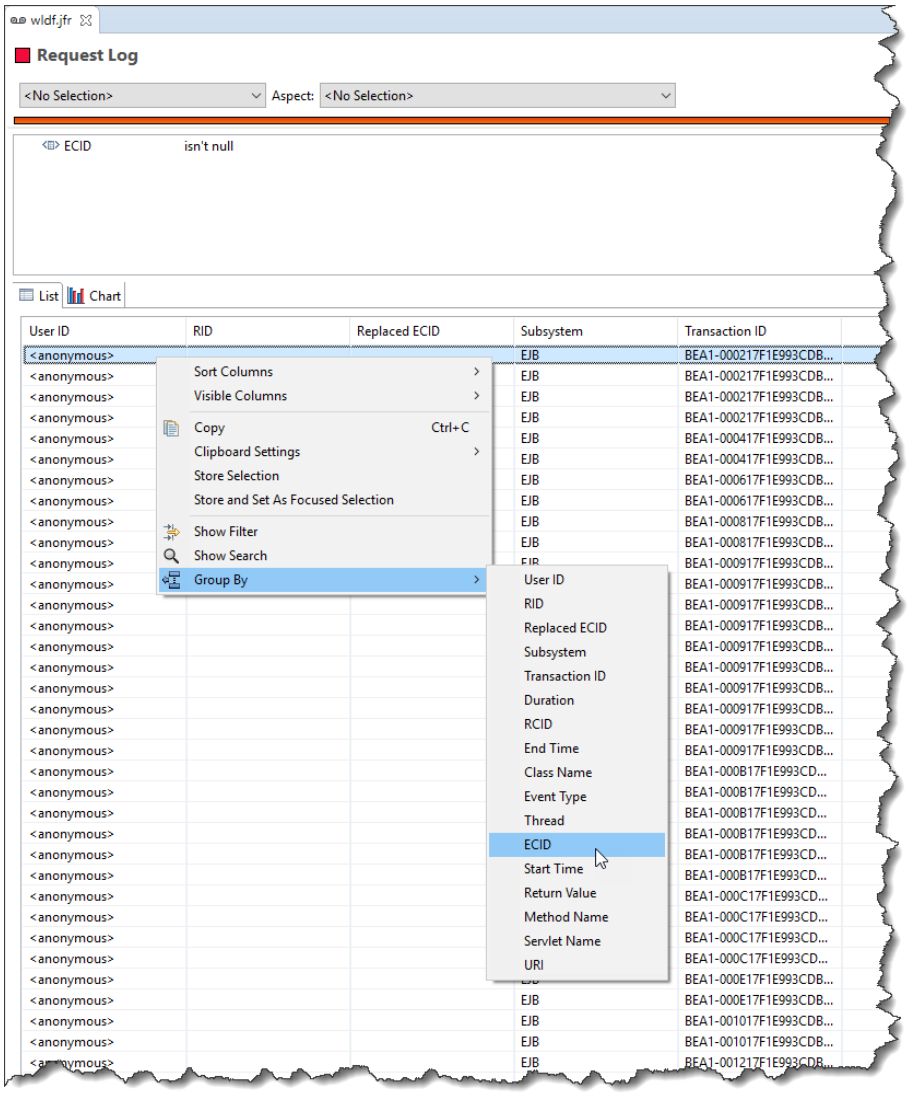
In the custom page, events can also be grouped. Create a new page, using the Servlet Request Run event again. Name the page Request Log. Add a new filter from attribute, this time ECID. Select isn't null as the predicate relation.



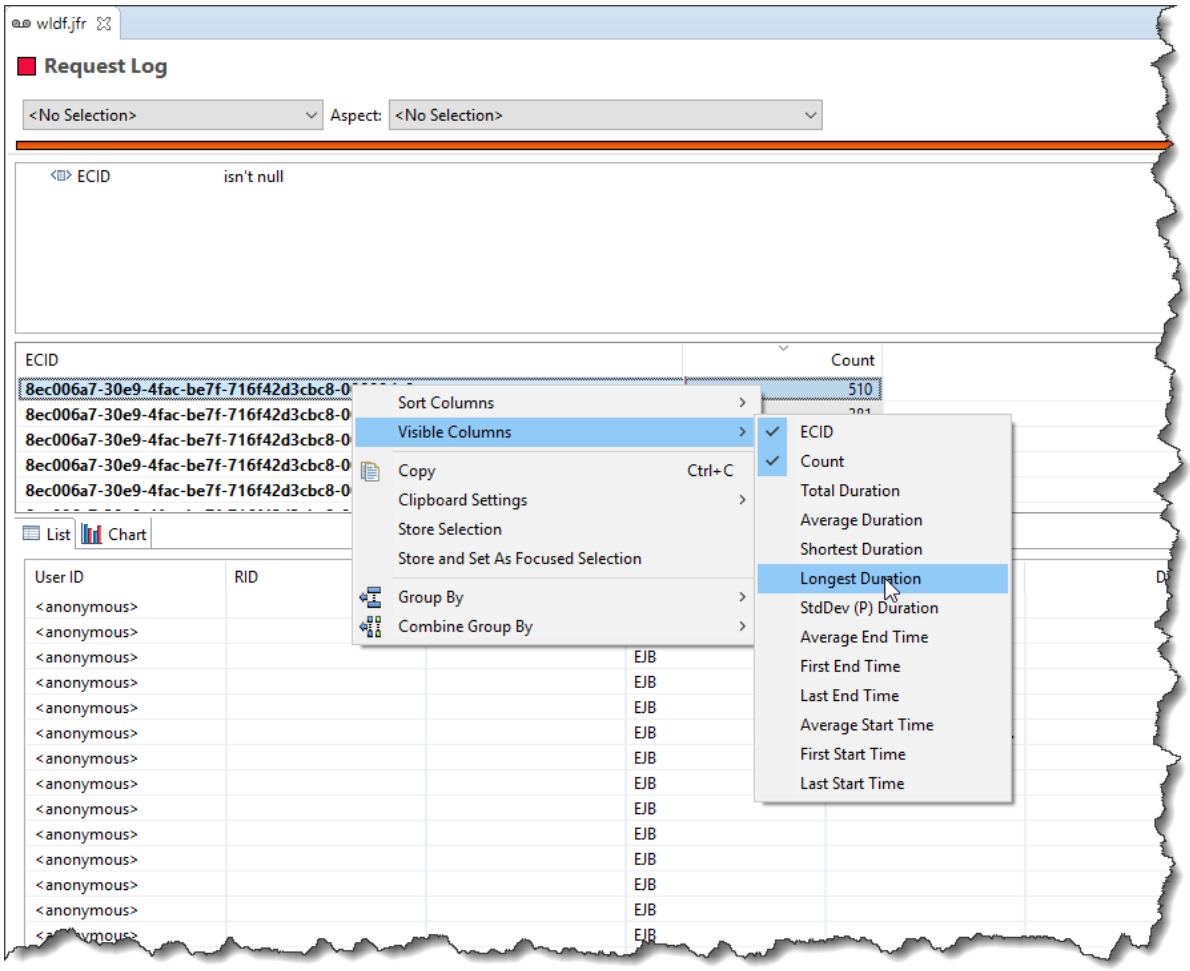
Now remove the Type filter:



Use the context menu in the table to group by ECID:

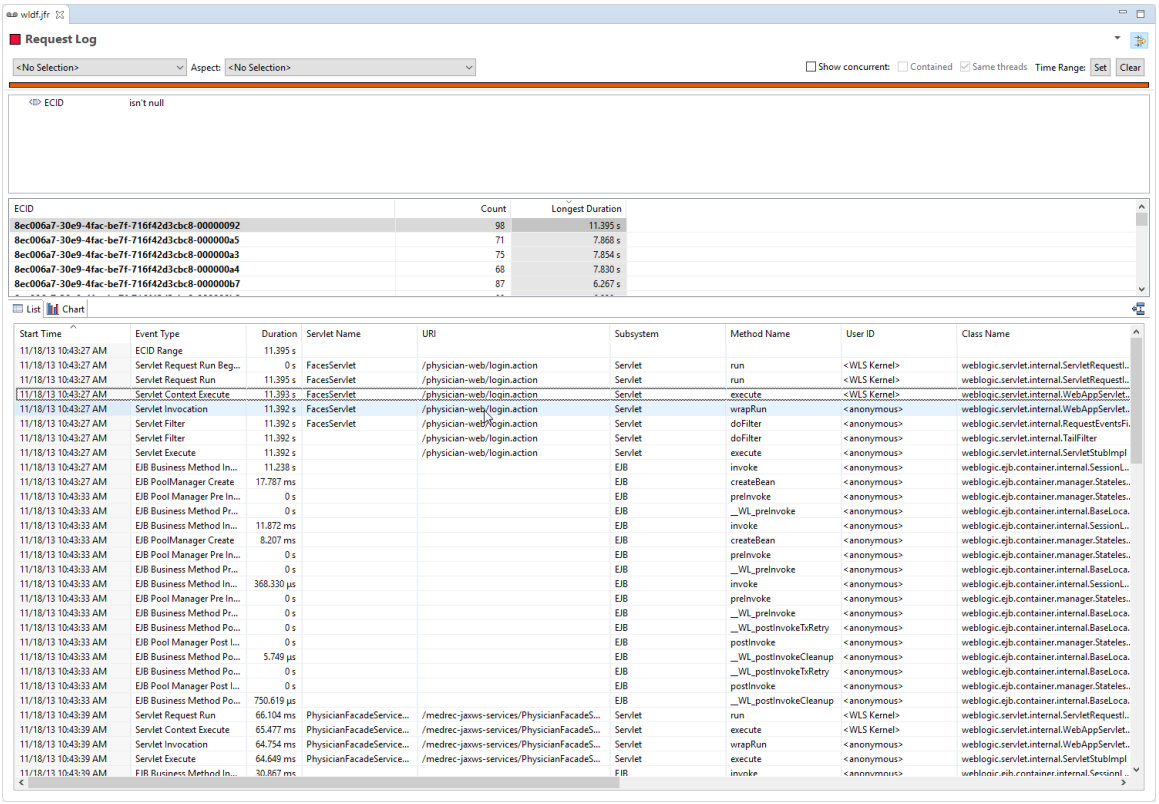


Add a column for the longest duration in the ECID table:



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.

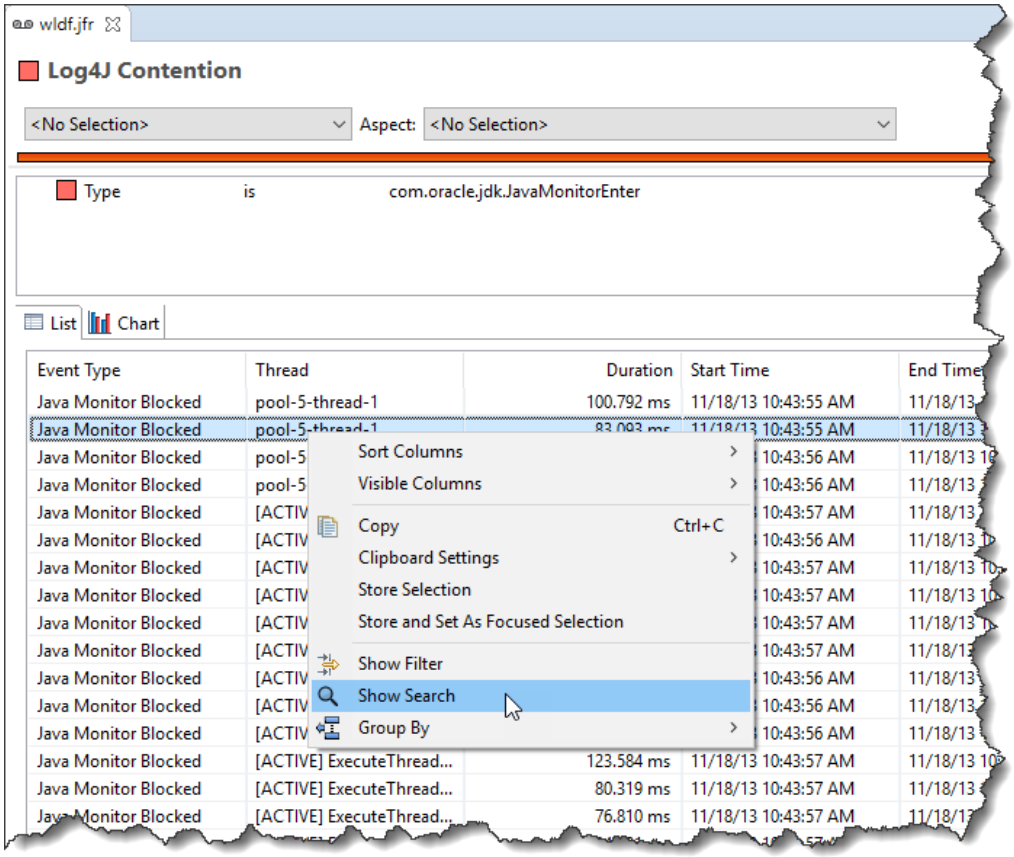


Boolean Filter Operations

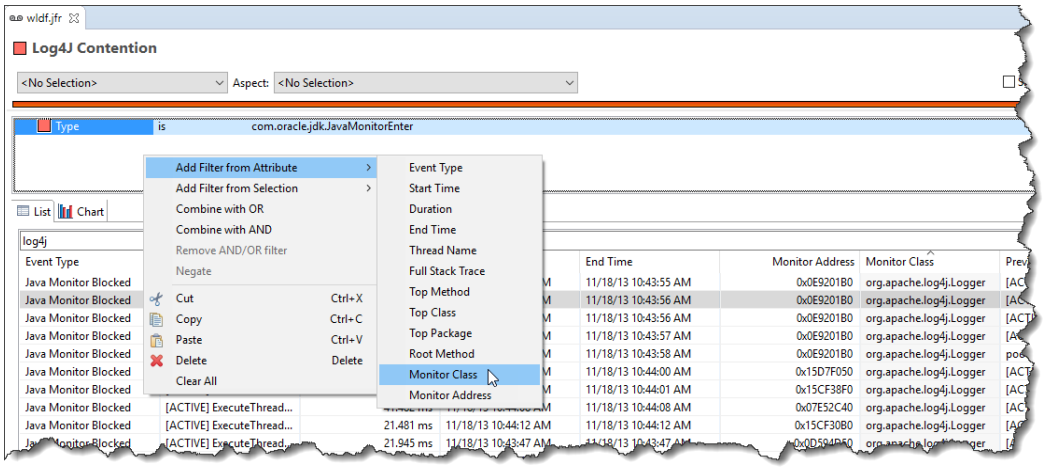
Lastly we will build a custom view to find any contention on Log4J lasting longer than 400ms, or 200ms if the contention is somewhere else, as this will neatly illustrate the use of Boolean filter operations.

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.)

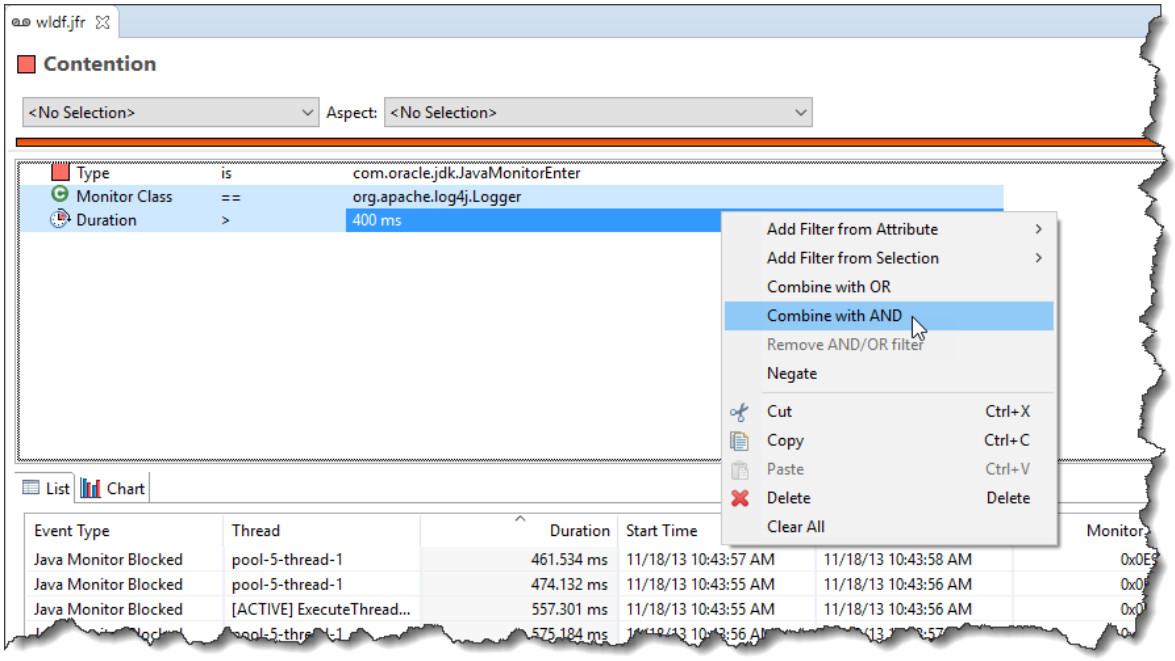


In the search box, type log4j, then select one of the events. Next add a filter from the attribute Monitor Class.



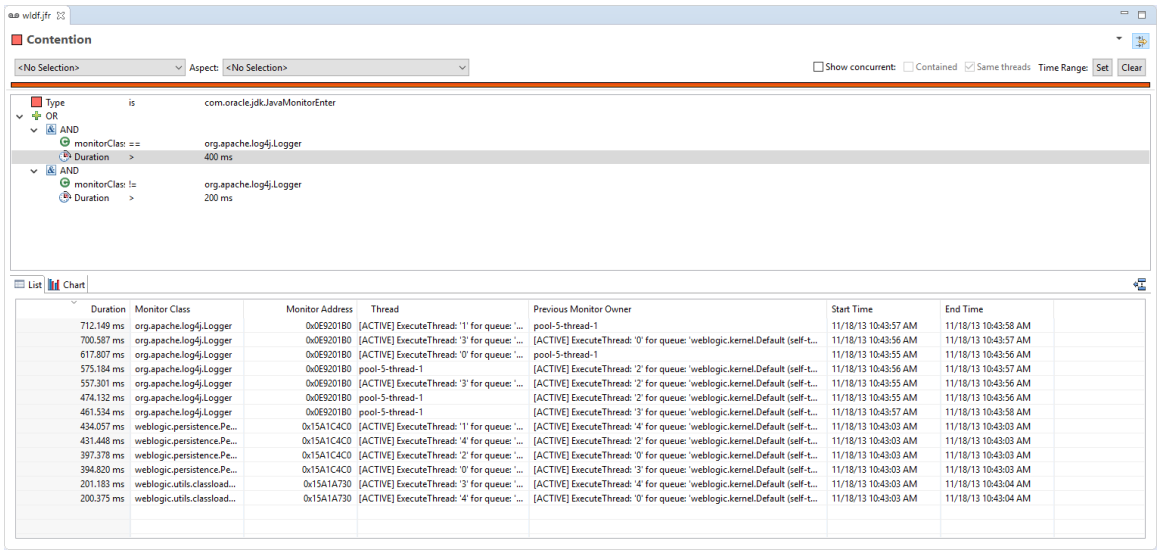
The log4j class should automatically be added for you. Disable the search from the context menu in the List table by using the context menu.

Add a new filter from the attribute Duration. Set the filter predicate to be more than 400 ms.

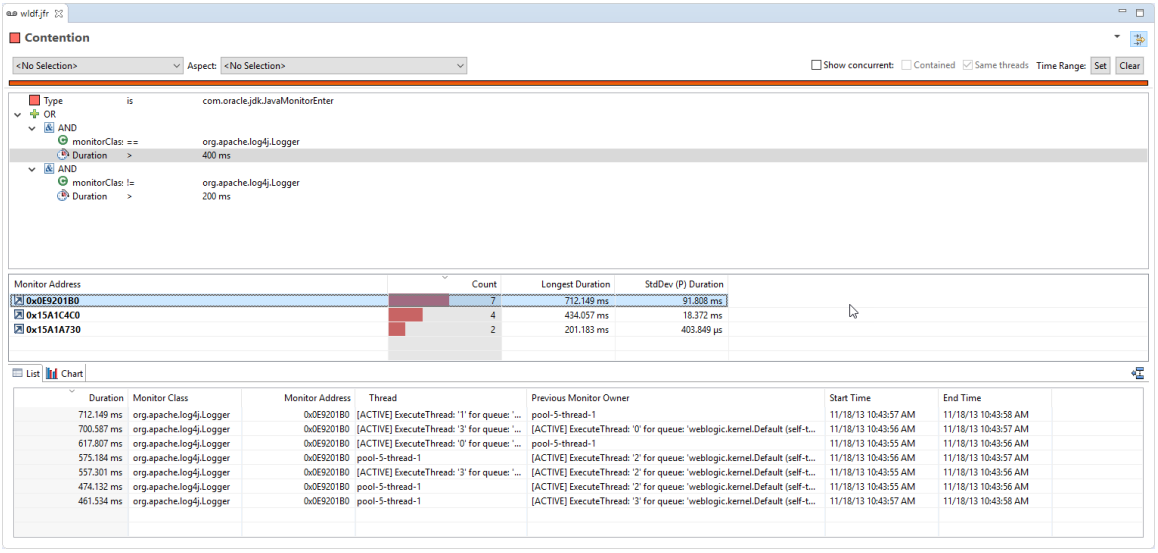


Select the Monitor Class filter and the Duration filter both, and select Combine with AND from the context menu. Next add a filter for Monitor Class != org.apache.log4j.Logger, and Duration > 200 ms. Combine them with an AND filter, and finally combine the both AND filters with an OR filter.

Note: *There is a bug (fixed in JMC 7) 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:



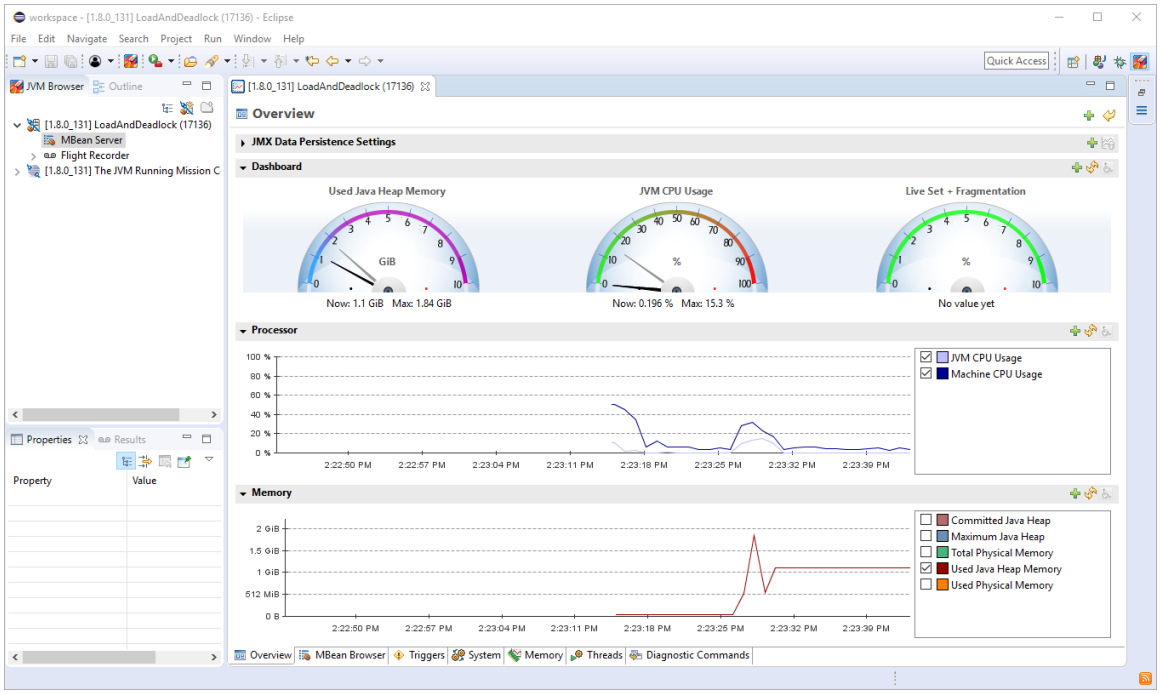
The Management Console (Bonus)

JDK 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 12a – The Overview

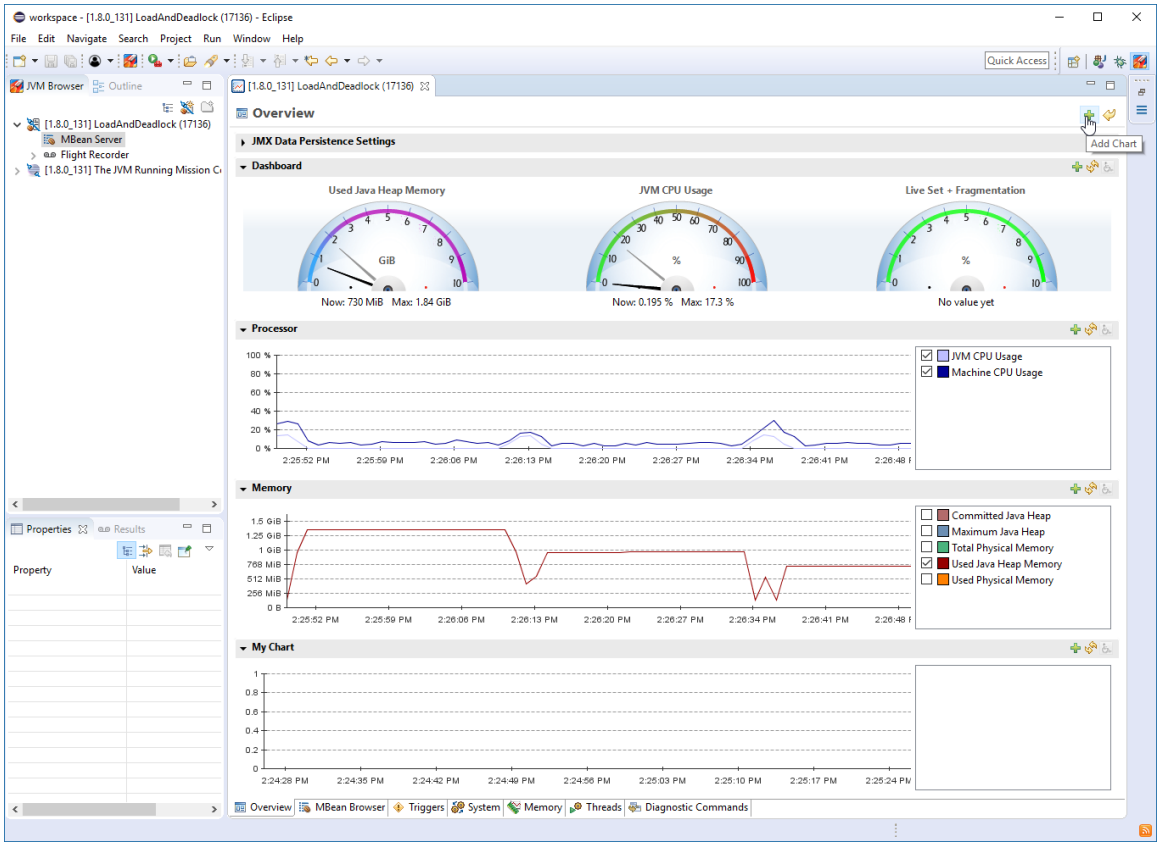
Start the `LoadAndDeadlock` program, like you did in Exercise 2.a. Then switch to the Mission Control perspective. After a little while you should see the JVM running the `LoadAndDeadlock` class appearing in the JVM Browser. Open a console by selecting Start JMX Console from the context menu of the JVM 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.



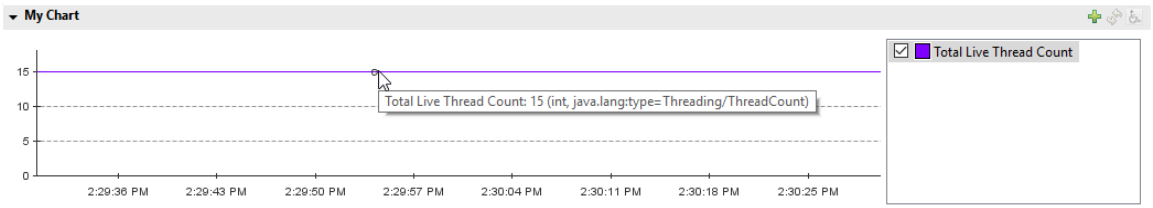
Click the Add… button of the new chart. In the attribute selector dialog, go to the Filter text field and enter “Th” (without quotation marks). Select the ThreadCount attribute, and press ok. You should now see the thread count.

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?

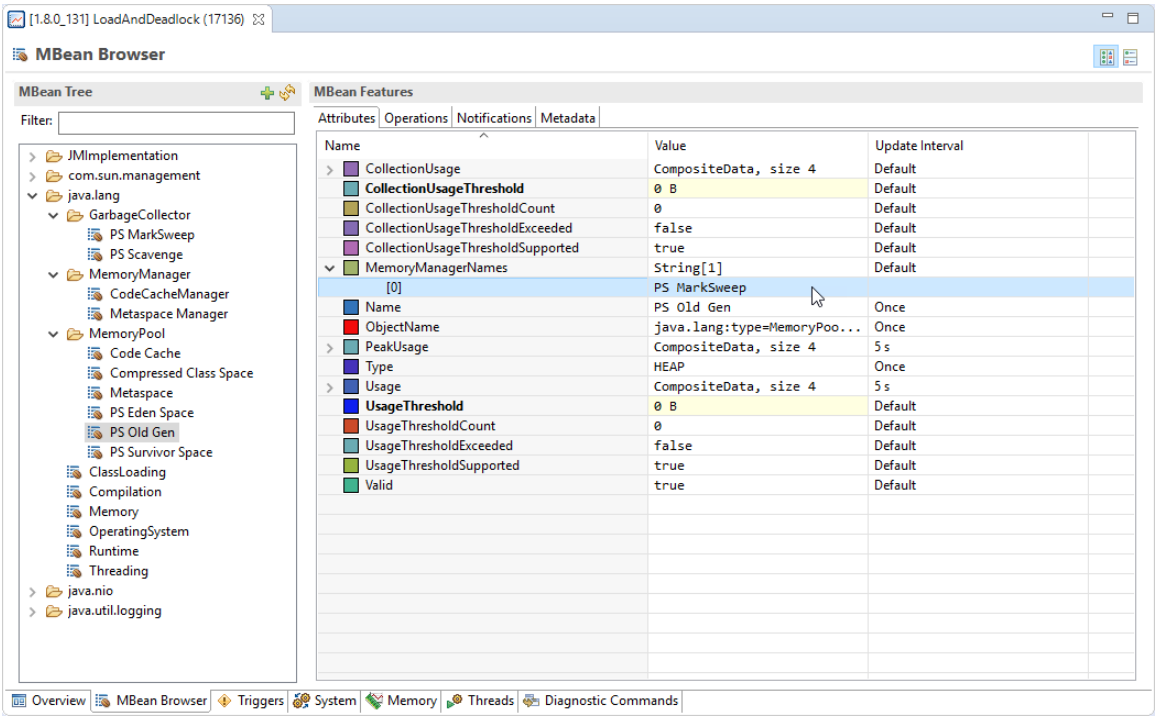


17. You decide that you dislike the live set attribute and warm up a bit to the Thread Count attribute. Remove the Live Set velocimeter in the upper right corner of the Overview tab, and instead add one for the Thread Count attribute.

Exercise 12b – The MBean Browser

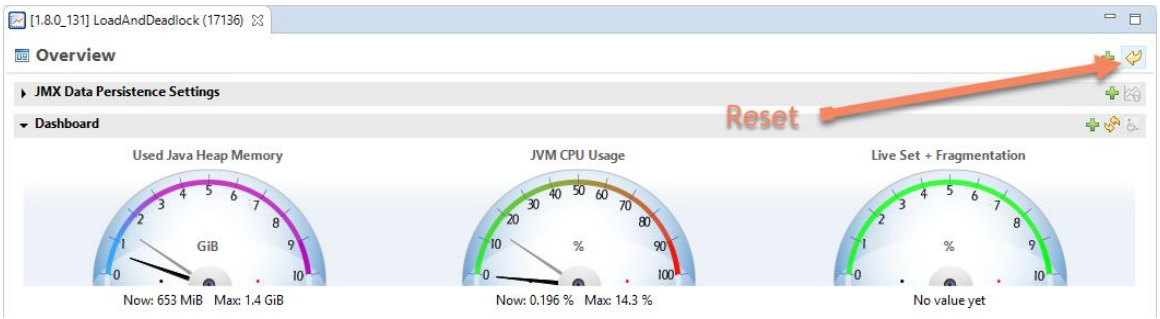
The MBean browser is where you browse the MBeans available in the platform MBean server. If you expose your own application for monitoring through JMX and register them in the platform MBean server, your custom MBeans will show up here. You can use the MBean browser to look at specific values of attributes, 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.*

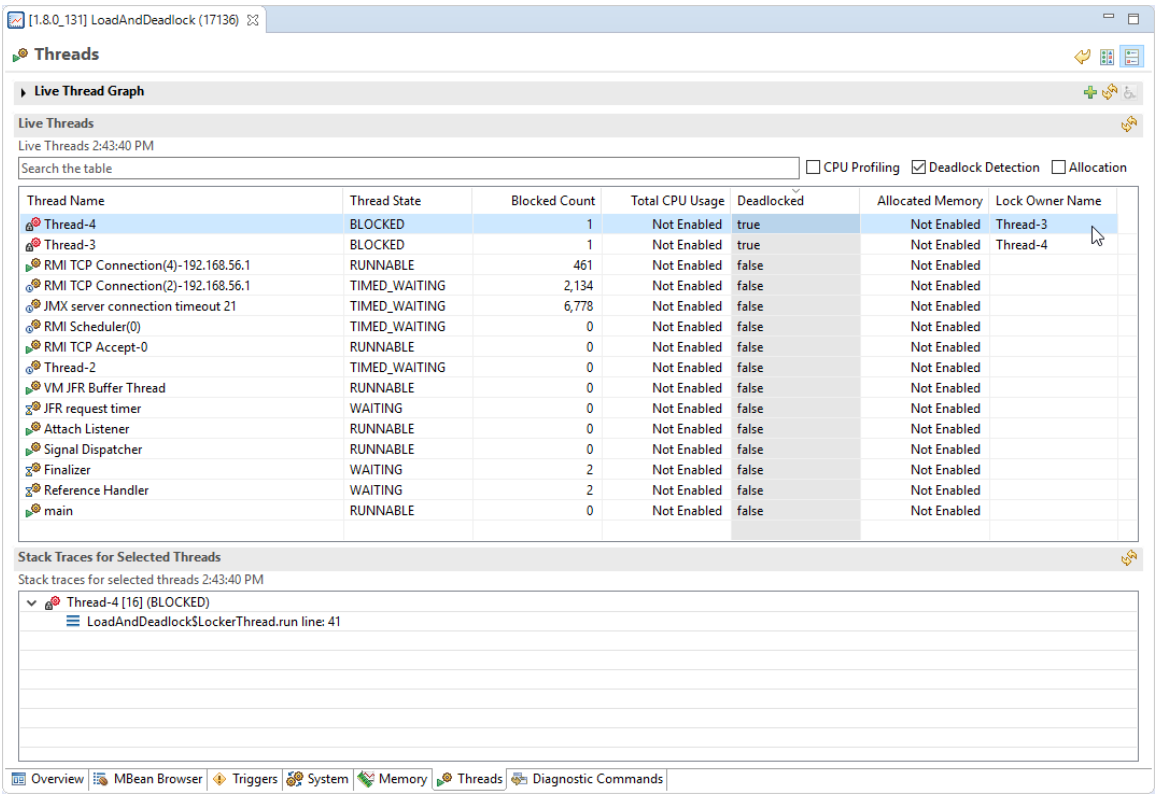
Exercise 12c – The Threads View

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?



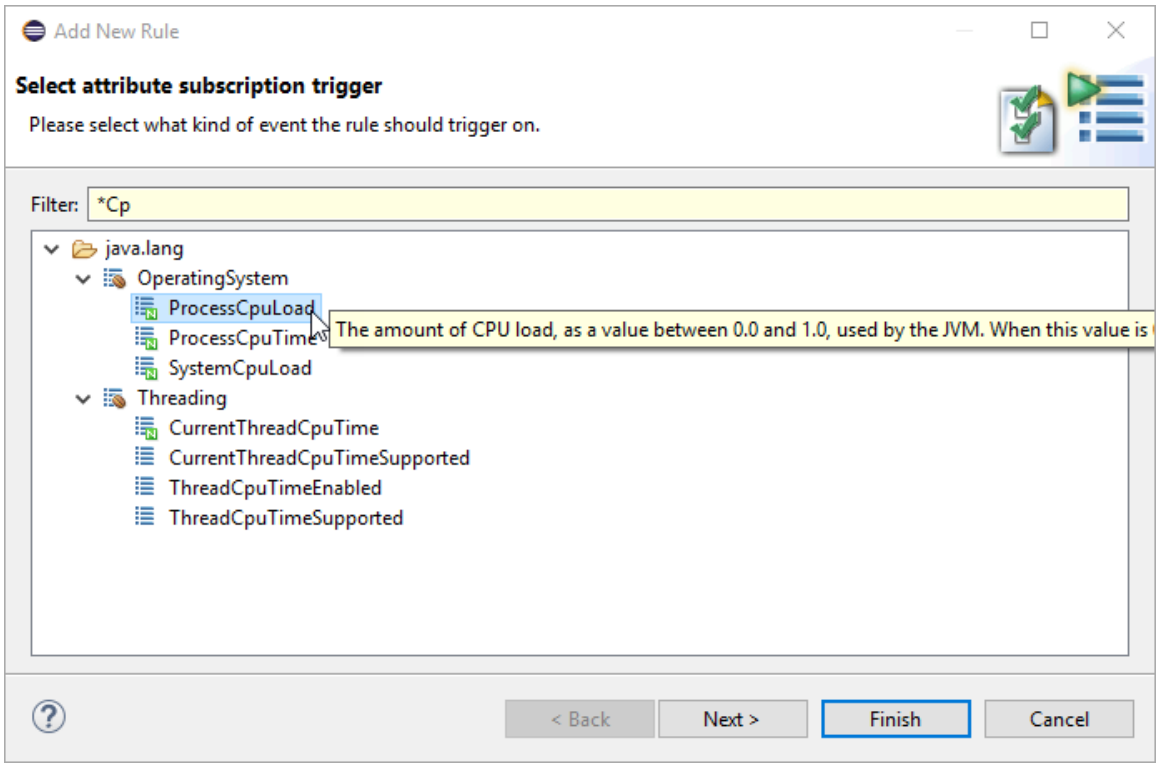
Note: *In most tables in Mission Control, there are columns that are not visualized by default. The visibility can be changed from the context menu in the table.*

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 12d (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.

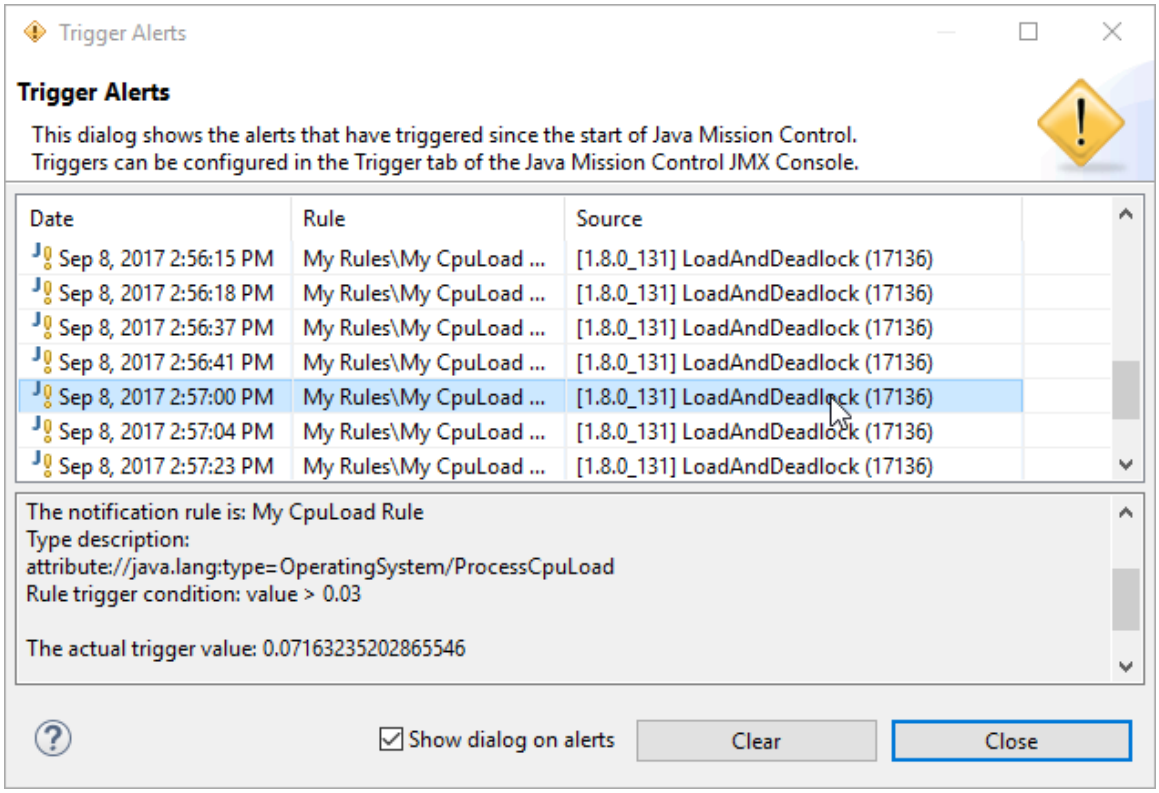


Select the Max trigger value to be 0.3 (30%). And set the limit to once per second. Click **Next**.

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.



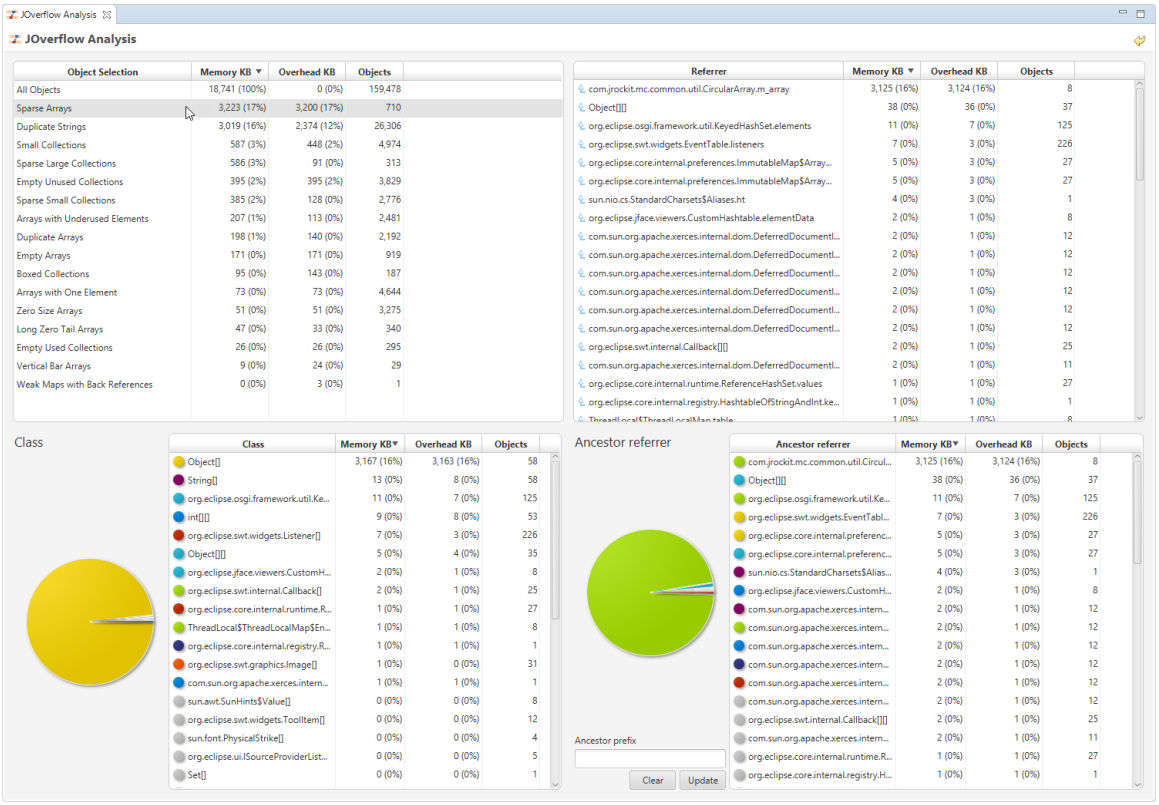
Disable or remove the rule when done to avoid getting more notifications.

Heap Waste Analysis (Bonus)

There is an experimental plug-in available for JDK 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.



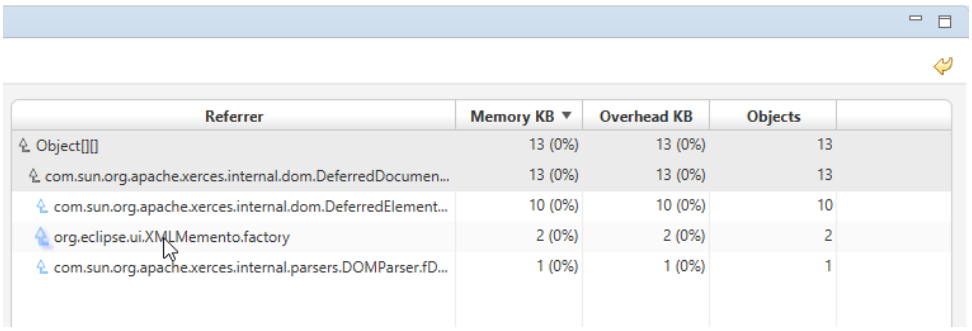
JOverflow will open, and show the contents of the heapdump. There are four quadrants in the JOverflow user interface. Also notice the little reset button in the upper right corner ( ). It will reset all the selections in the user interface.

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.



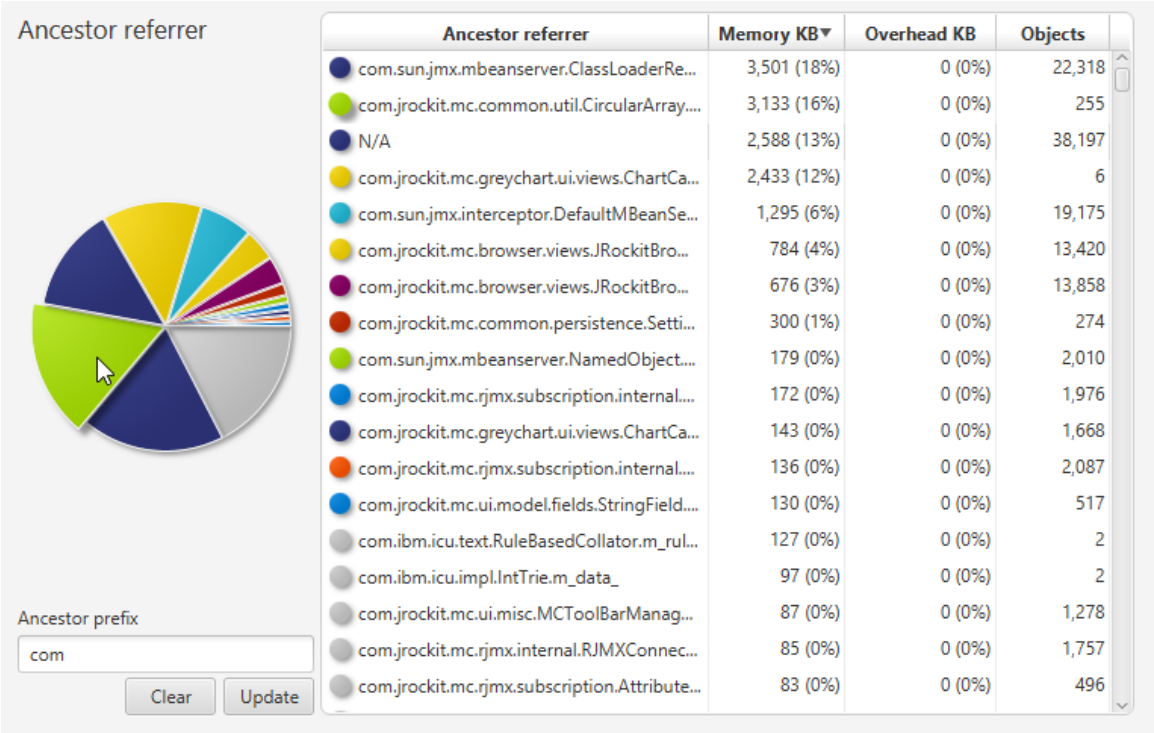
(Screenshot showing multiple available paths to select from)

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 13 – 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 JDK 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 holing 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

The JDK Mission Control EA builds:
http://jdk.java.net/jmc/

The JDK Mission Control wiki:
https://wiki.openjdk.java.net/display/jmc/Main

The JDK Mission Control source:
http://hg.openjdk.java.net/jmc/jmc/

The OpenJDK Bug System:
https://bugs.openjdk.java.net/

The Oracle JDK Mission Control homepage:
http://oracle.com/missioncontrol

The JDK Mission Control twitter account:
http://twitter.com/javamissionctrl

JDK Mission Control on Facebook:
https://www.facebook.com/javamissionctrl/

Marcus Hirt's JDK Mission Control articles:
http://hirt.se/blog

Marcus Hirt's twitter account:
http://twitter.com/hirt