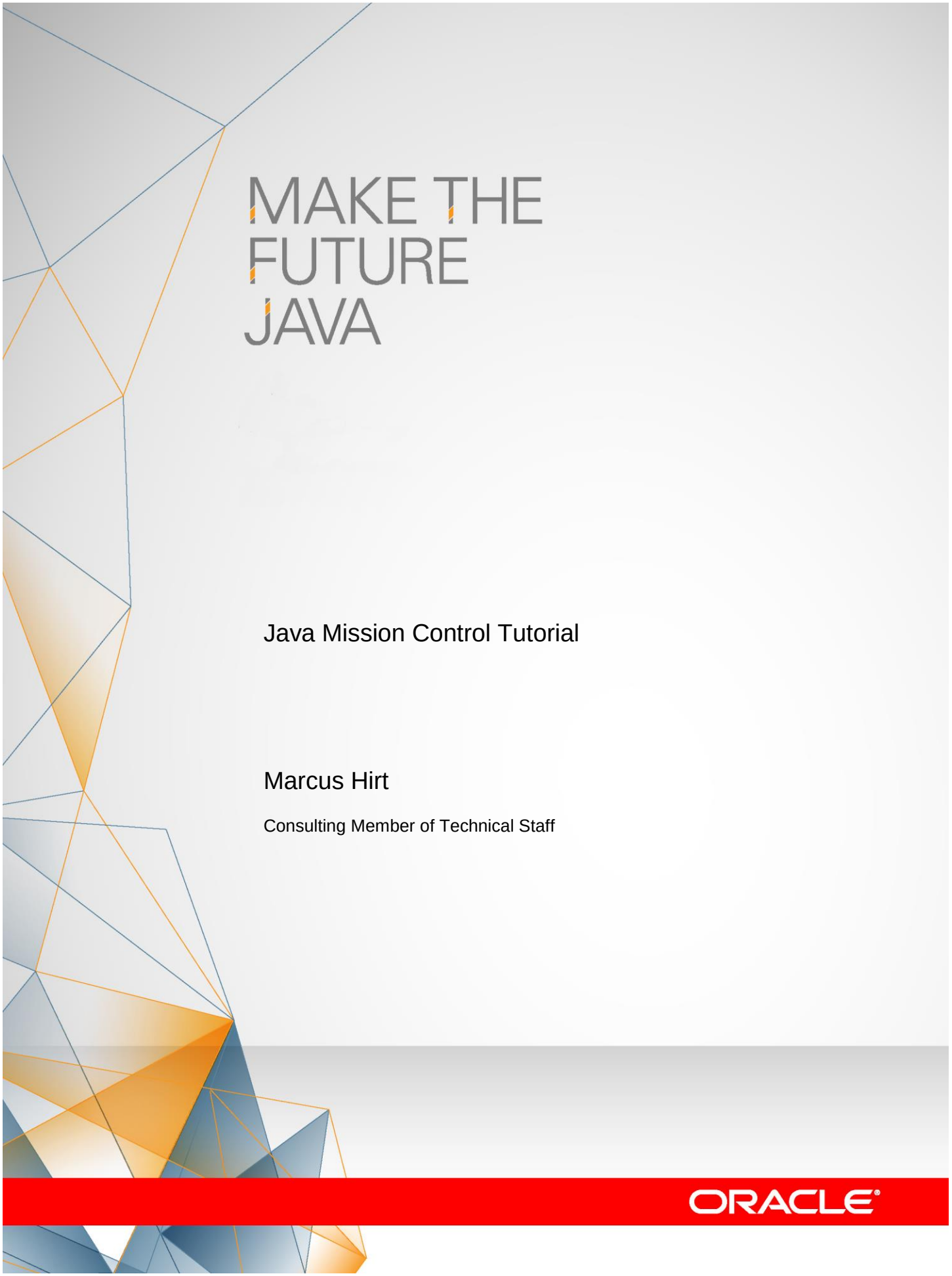


Java Mission Control Tutorial

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 may 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 standards exercises.

Introduction

Oracle Java Mission Control is a suite of tools for monitoring, profiling and diagnosing applications running in production. The tools suite is currently only for the Oracle 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, heap dump analysis and DTrace based profiling.

This tutorial will focus on the Java Flight Recorder, with bonus exercises for the heap dump analysis tool (JOverflow), the DTrace profiling tool and the JMX Console at 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:\jrmc\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

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 8u25.

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.

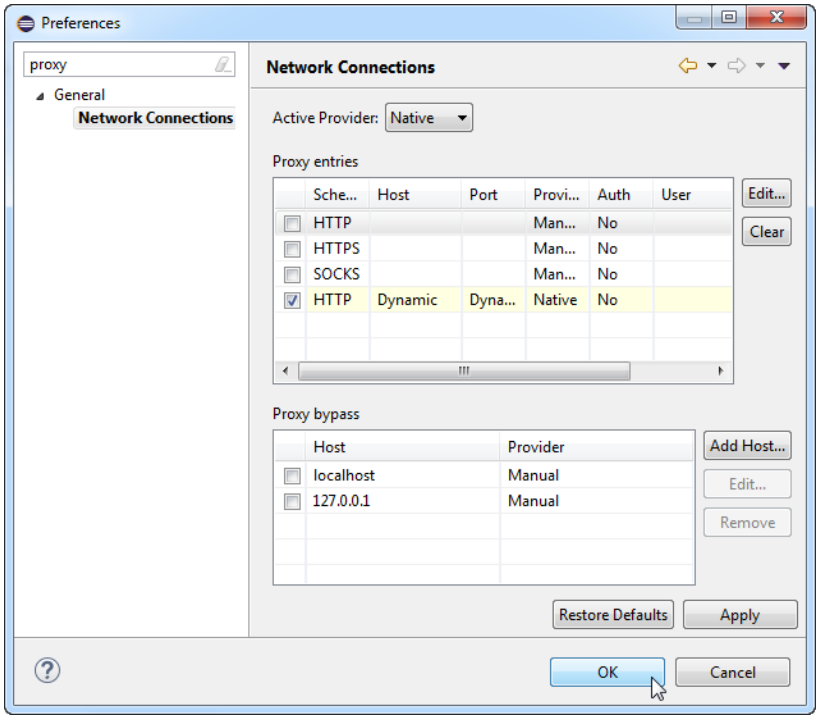
Here is how to get the latest Eclipse:

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.

Note: *if you are using an Eclipse 4 < 4.5 you may notice bad performance when switching between editors and tab groups. This problem will be fixed in Eclipse 4.5. Also, JMC 5.5.0 will contain a workaround that mostly alleviates this problem. If you are reading this before the release of either Eclipse 4.5 or JMC 5.5, you may be better off running JMC in Eclipse 3.8.2, or running the Stand Alone version delivered with the JDK.*

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 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 Experimental Update Site (the last bullet under the Overview section).
3. Click on Use Update Site under Download and Install.
4. Follow the instructions. Remember to install the WebLogic Tab Pack, JOverflow, Java FX and the D-Trace plug-in if you want to do the corresponding parts of the tutorial.

Update your eclipse.ini file (in your Eclipse root folder) to explicitly point to your new shiny JDK, and add appropriate VM arguments. These are the ones I typically use on a Windows box for this tutorial:

```
-vm
C:\Java\JDKs\jdk1.8.0_25\bin\javaw
-vmargs
-Dosgi.requiredJavaVersion=1.7
-Xms512m
-Xmx2048m
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200
-Dcom.jrockit.mc.allowdtrace=true
-XX:+UnlockCommercialFeatures
-XX:+FlightRecorder
```

If you wish to use these in your `eclipse.ini` file, first remove any `-vm` and `-vmargs` entries. Also remember to restart Eclipse after updating your `eclipse.ini` file.

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.

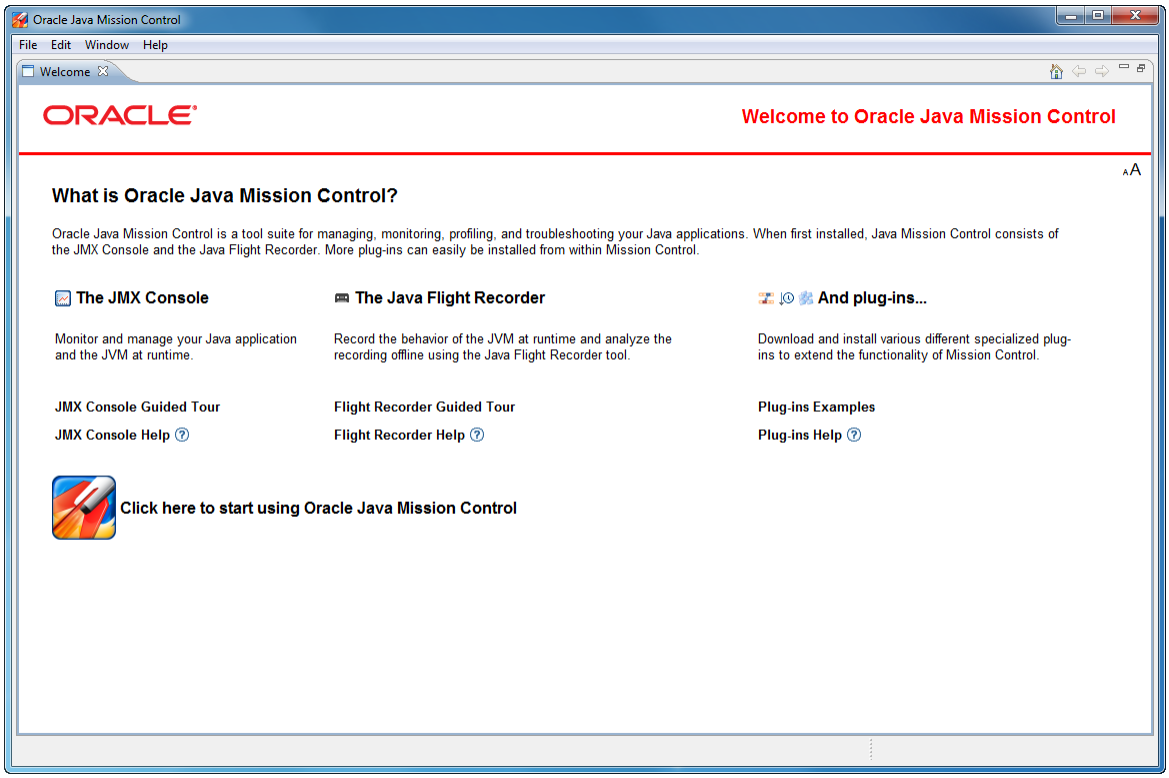
Starting Java Mission Control

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

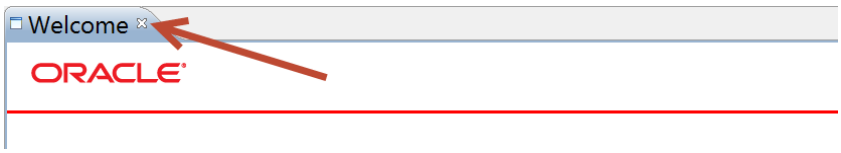
This exercise familiarizes you with the layout of the exercise collateral on disk, and shows you how to start both the stand alone and the Eclipse plug-in versions of Java Mission Control.

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

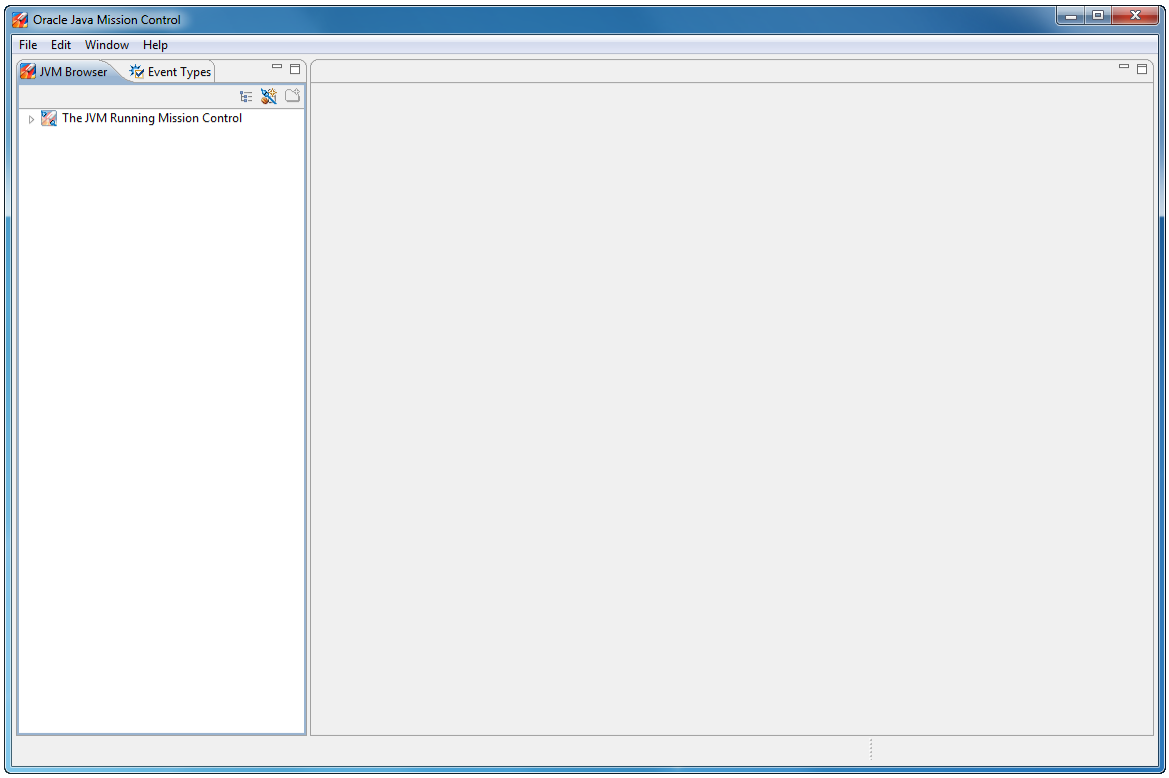
Go to the **`%JDK_HOME%\bin`** directory and double click the **`jmc`** executable. A splash screen should show, and after a little while you should be looking at something like this:



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.



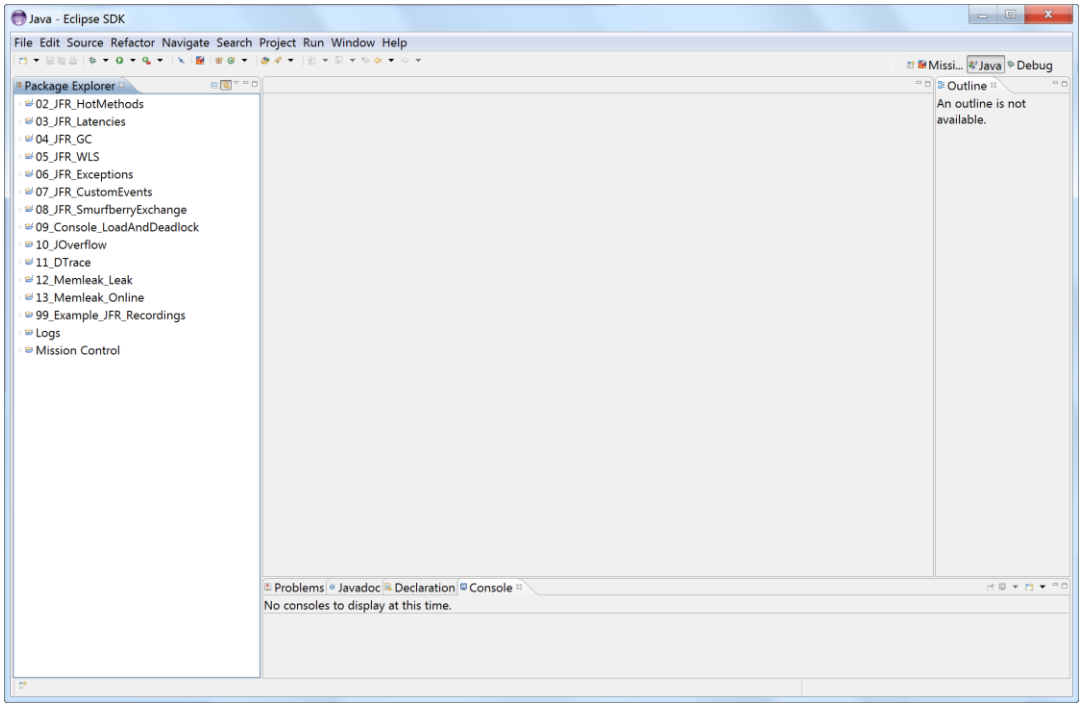
To launch the different tools, simply select a JVM in the JVM Browser and then select the appropriate tool from the context menu. For example, the Management Console can be started on a JVM by selecting the JVM in question in the JVM Browser and selecting Start JMX Console from the context menu.

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

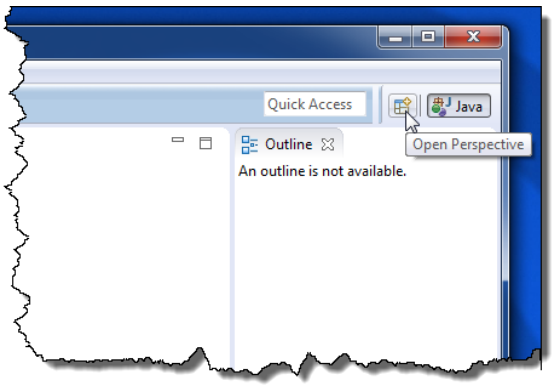
If you will be running the exercises from within Eclipse, now **shut down** the stand-alone version of Java Mission Control.

Exercise 1.b – Starting JMC in Eclipse (Optional)

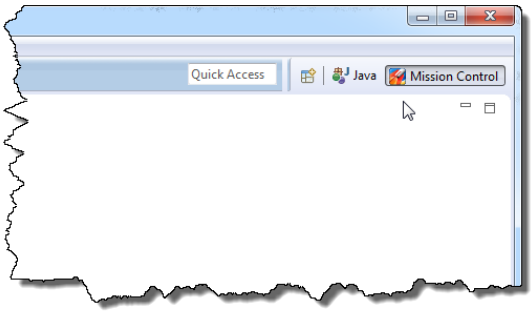
First start Eclipse by running your eclipse executable. You should now be presented with the Java perspective, looking somewhat like below:



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, either click the Open Perspective button, in the upper right corner of Eclipse, or select Window | Open Perspective | Other…

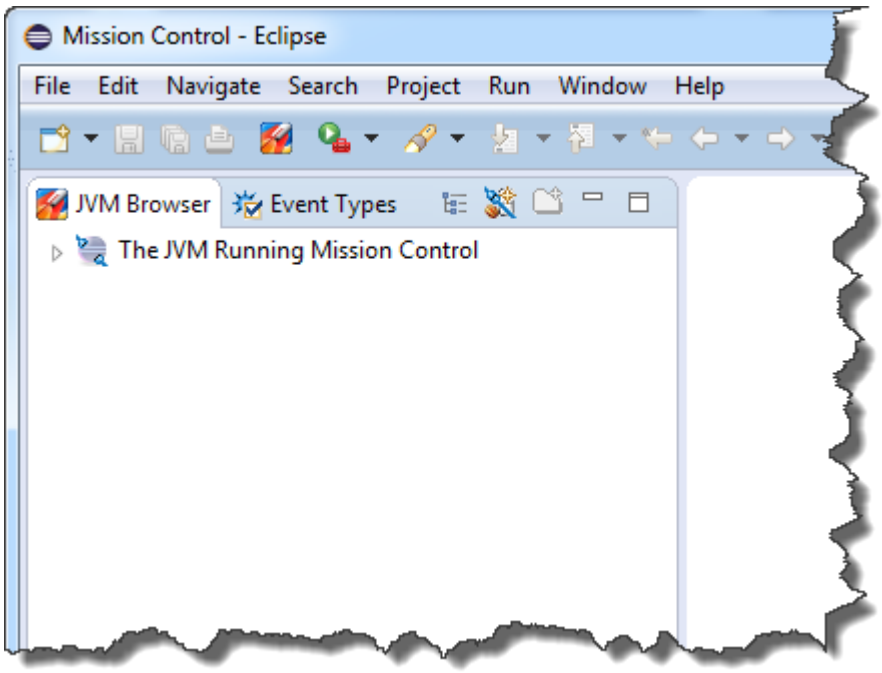


From the Open Perspective dialog, select the Mission Control perspective and hit ok. Now the Mission Control perspective should be easily available to you next to the Java perspective.



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.



Launching the tools work exactly the same as in the stand-alone version.

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

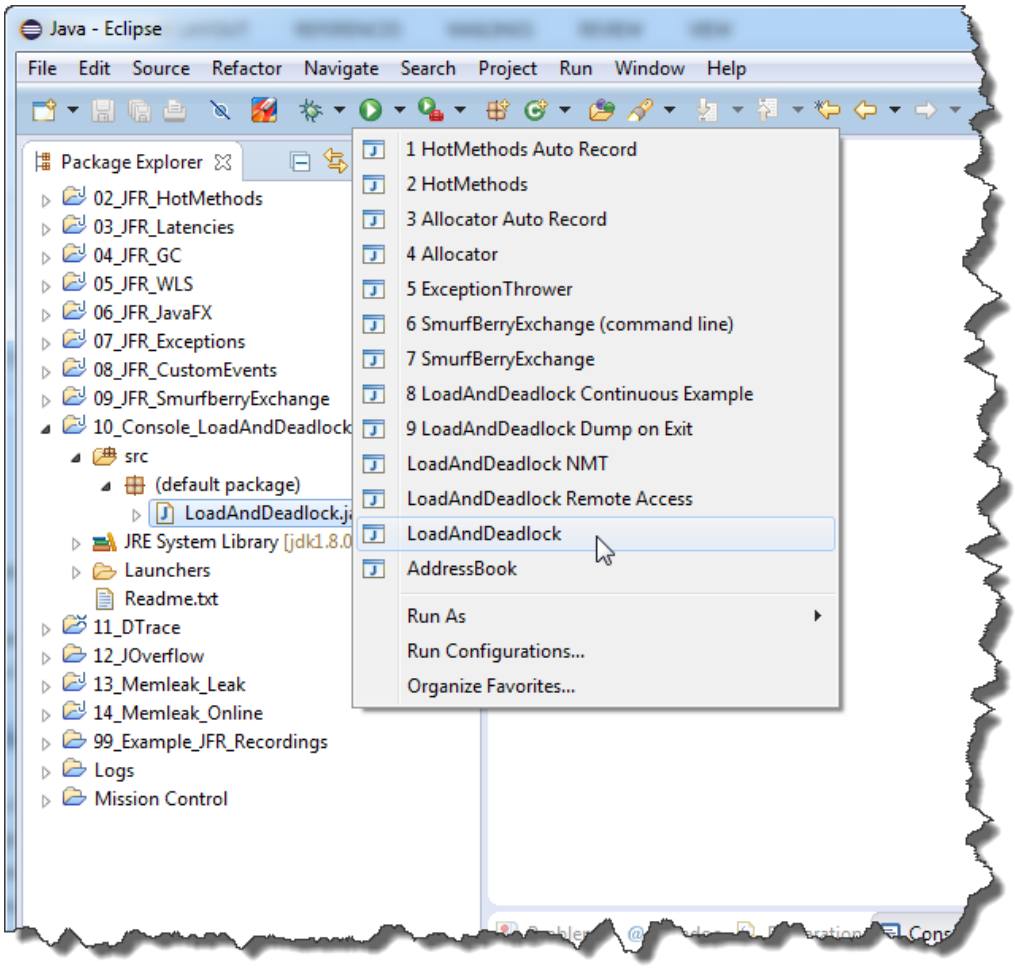
There are various different ways to start a flight recording. For this exercise we will use the Flight Recording Wizard built into Java Mission Control.

If running from within Eclipse:

First switch to the Java perspective (upper right corner 

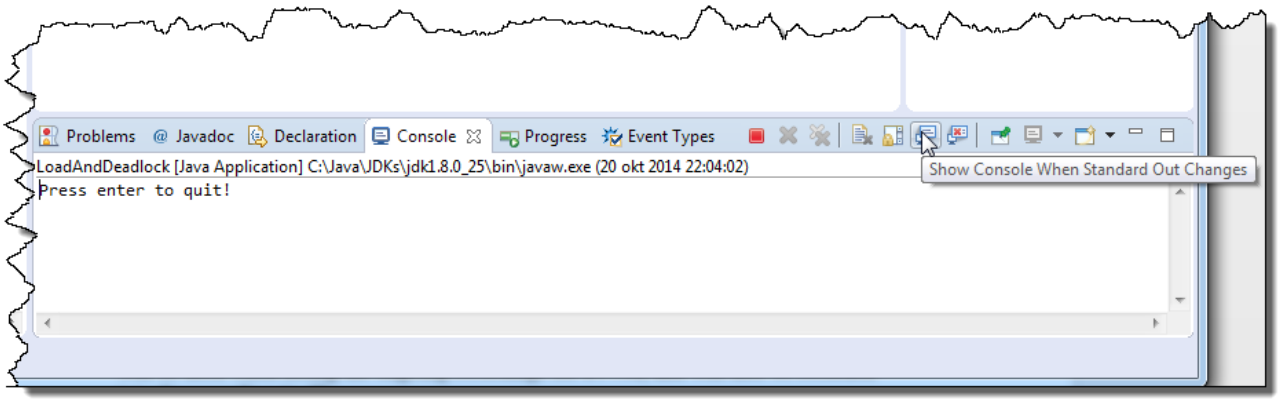
). Create a new project where you want to store your Flight Recordings (JMC 5.4.0 is not too happy about projects not residing in the same space as the workspace when running in Eclipse). File | New Project… -> Next -> Enter a project name and hit Finish.

Next start the LoadAndDeadlock program by selecting it from the drop down menu next to the run icon () as show below.



Note: *If the launchers do not show up, go to the* Run Configurations… *and look for it there.*

TIP: *Whilst in the Java perspective you may also want to go to the Console view and turn off the automatic showing of the console view when the standard out and standard error changes, as shown below.*



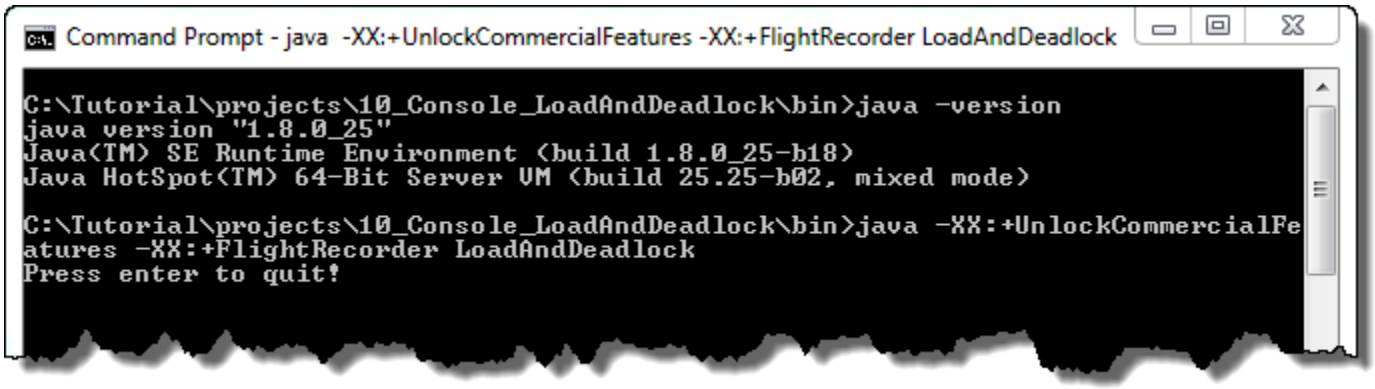
Switch to the Mission Control perspective and select the newly discovered JVM running the LoadAndDeadlock class.

If running the stand-alone version of JMC:

Open a terminal. Change to the directory where you unpacked the tutorial. Go to `10_Console_LoadAndDeadlock/bin` folder. Check that you are using the latest version of Java, by invoking `java -version`.

Next start LoadAndDeadlock by running:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder
LoadAndDeadlock
```

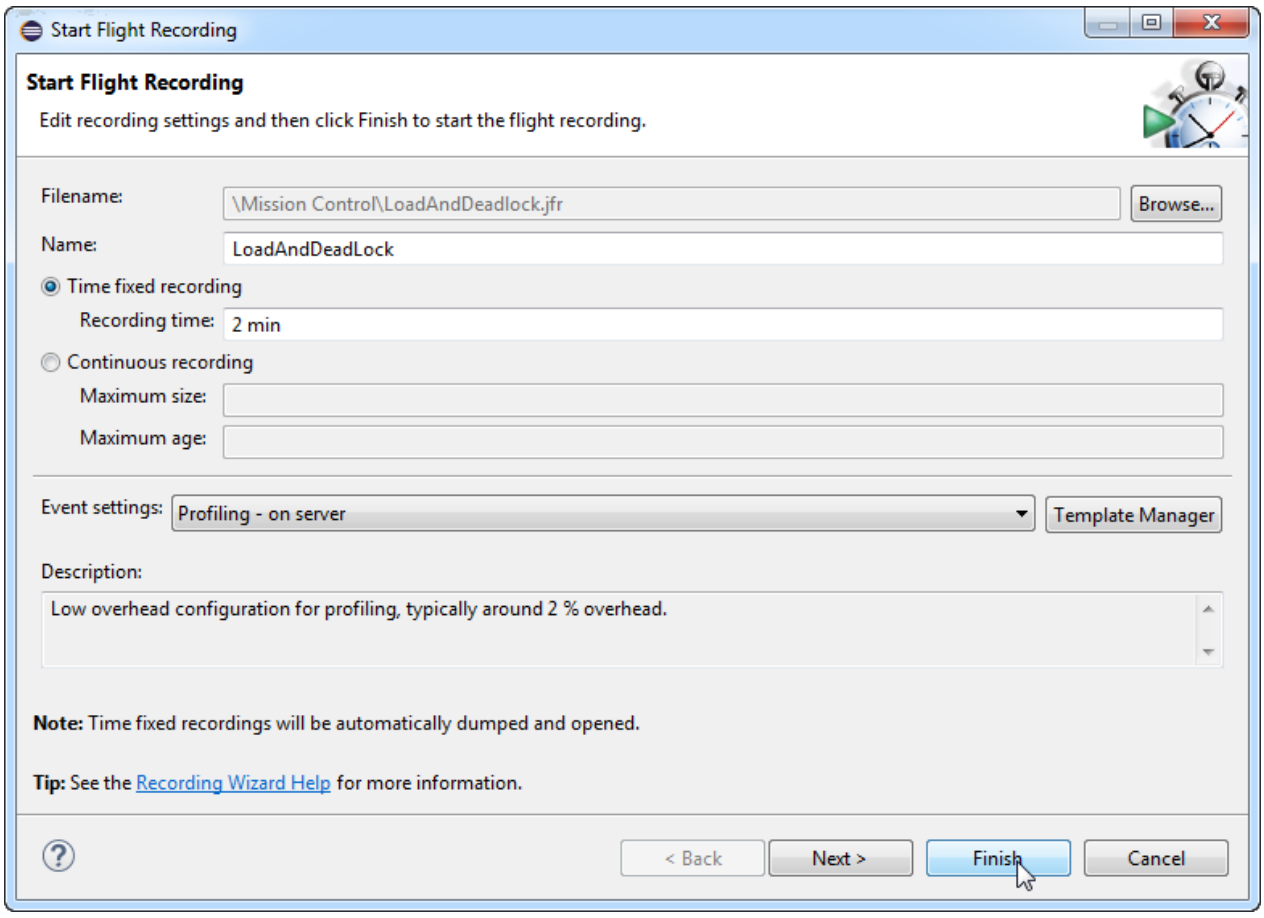


Next (all versions):

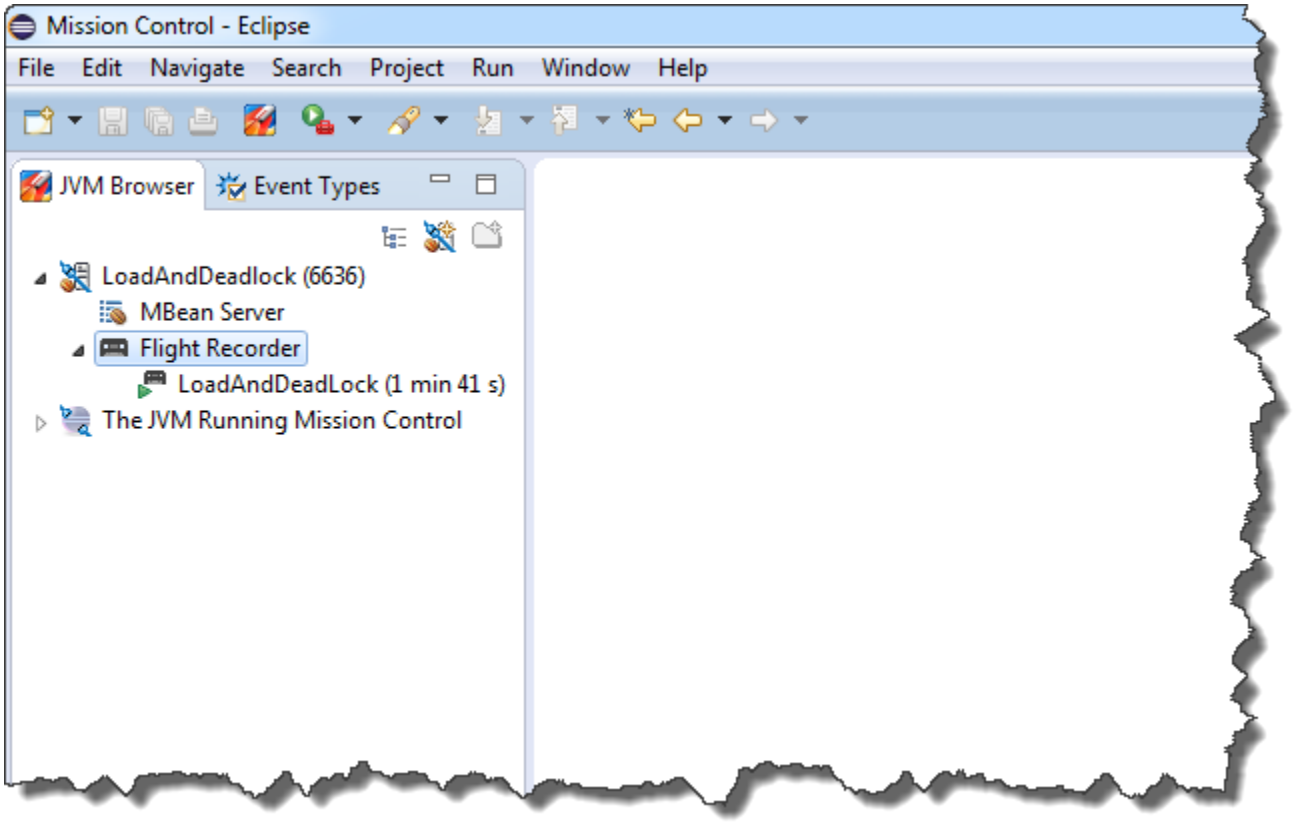
Wait for the JVM running the LoadAndDeadlock class to appear in the JVM Browser, and select Start Flight Recording…. The Flight Recording Wizard will open.

Click Browse to find a suitable location (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.

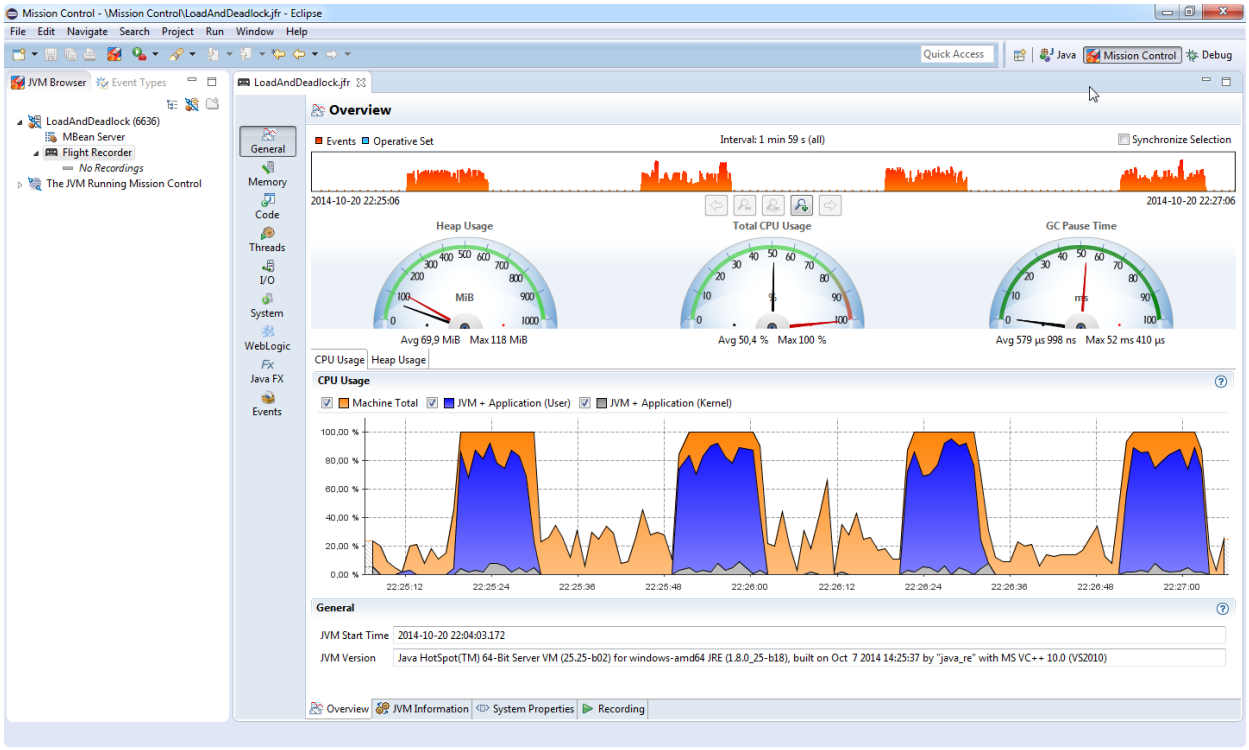


The progress of the recording can be seen in the JVM Browser, when the Flight Recorder node is expanded. It can also be seen in the status bar.



Use the two minutes 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

After the two minutes, the recording will be downloaded to your Mission Control client, and opened. You should be looking at a graphical overview of the recording.



Once you are done with the recording, remember to shut down the LoadAndDeadlock application.

<u>If running from within Eclipse:</u>

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

<u>If running the stand-alone version of JMC:</u>

Press enter in the terminal

Note: *Also, close the recording window and* ***try not to have more than 2 or 3 recordings opened*** *at any given time. The user interface is rather large and resource hungry on some platforms you can quickly run out of graphics resources.*

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.jfr recording.

If running from within Eclipse:

Switch to the Java perspective and open (double click) the recording in the 02_JFR_HotMehods project named hotmethods.jfr.

If running the stand-alone version of JMC:

Go to File | Open File… and open the `projects/02_JFR_HotMethods/hotmethods.jfr` file.

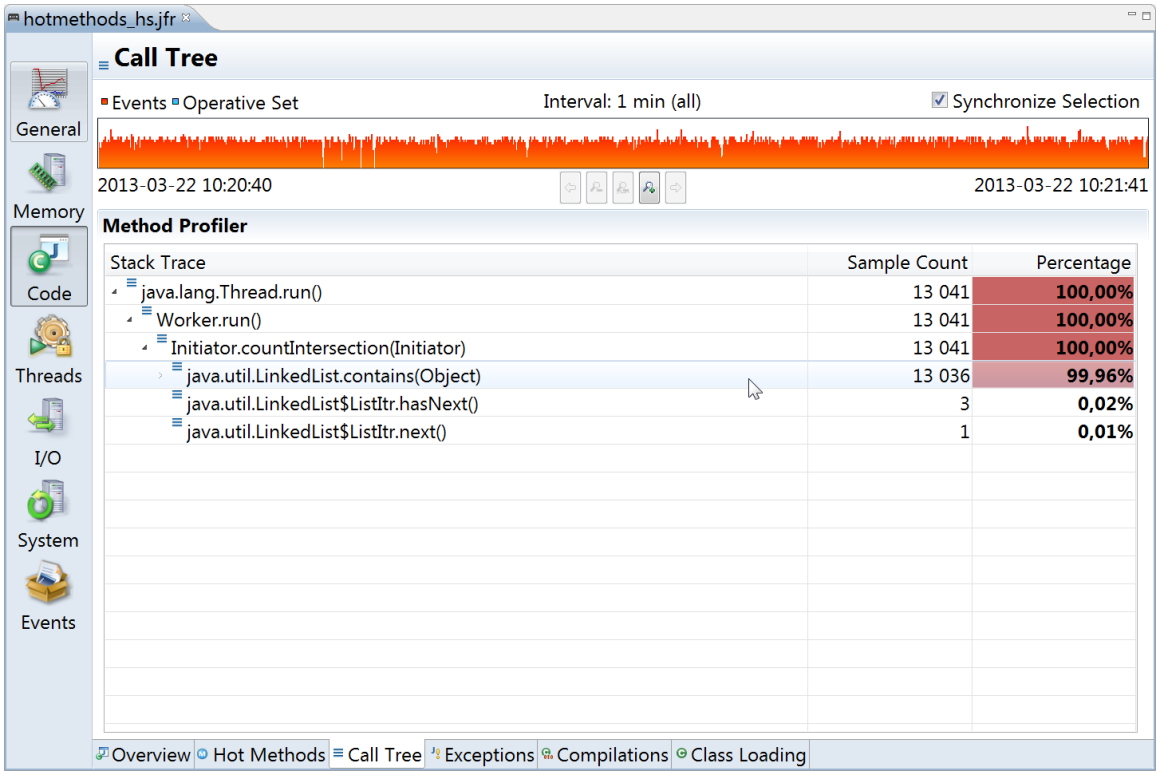
Next (all versions):

Go to the Hot Methods tab in the recording you just opened (it is under the Code tab group).

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.*



Bonus 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?

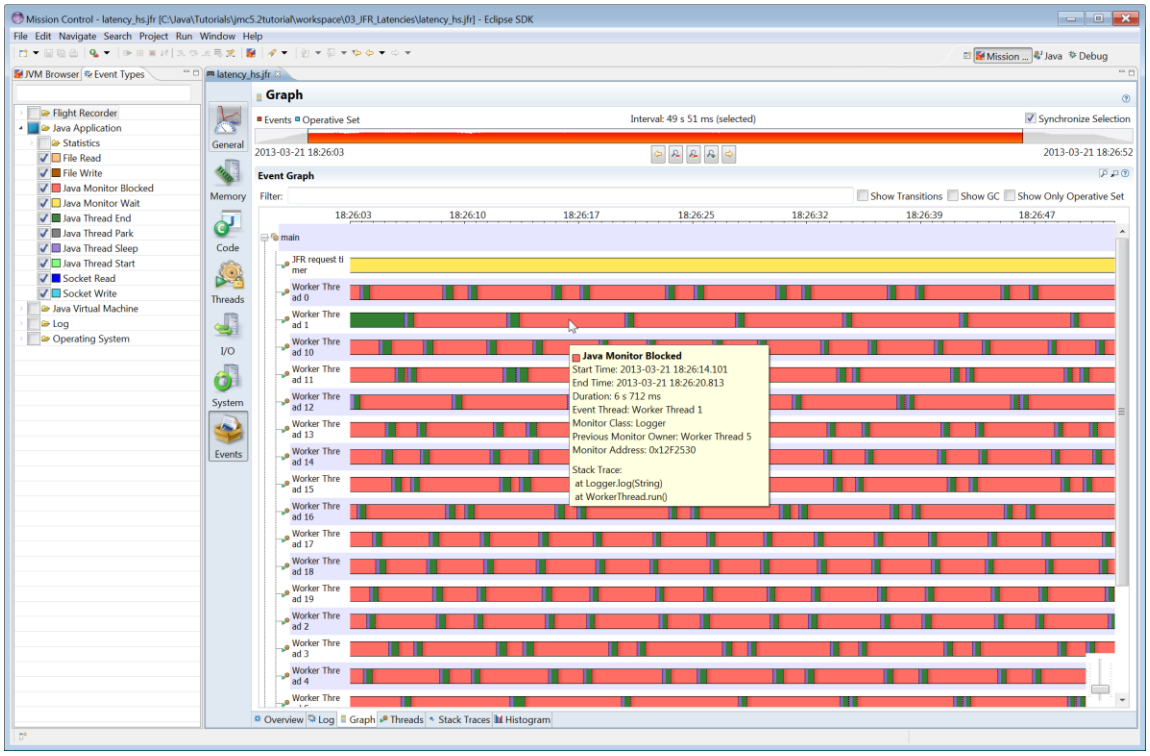
   Note**:** *The application generates custom events for each unit of "work" done. This makes it easy to compare the time it takes to complete a unit of work before and after the code change. Would it be possible to decide how faster the program became without these custom events?*
   Note**:** *Events in the general Events tab group is affected by the Event Types filter. By default, the filter will only show Java Application 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 behaviour 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.jfr`** recording (same procedure as when opening the hotmethods.jfr recording in the previous exercise). Then switch back to the Mission Control perspective. Click on the Events tab group, and then select the Graph tab at the bottom.



Ensure that the Event Types view is visible in the top left corner, by clicking on its tab. What type of event seems to be the dominating one? Of what class is the lock we're blocking on? From where in the code is that event originating?

Note: Look in the traces tab and look only at the blocking events, or go to the CPU/Threads tab group and select the Contention and/or Lock Instances tab.

Move over to the Stack Traces tab. Click the Stack Trace Tree column header Total to sort the traces on the total latency contributed. Expand the trace that contributes most to the latency.

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: *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: *Green means the thread is happily running along. In the latency.jfr recording only one thread is running at any given time, the rest are waiting.*

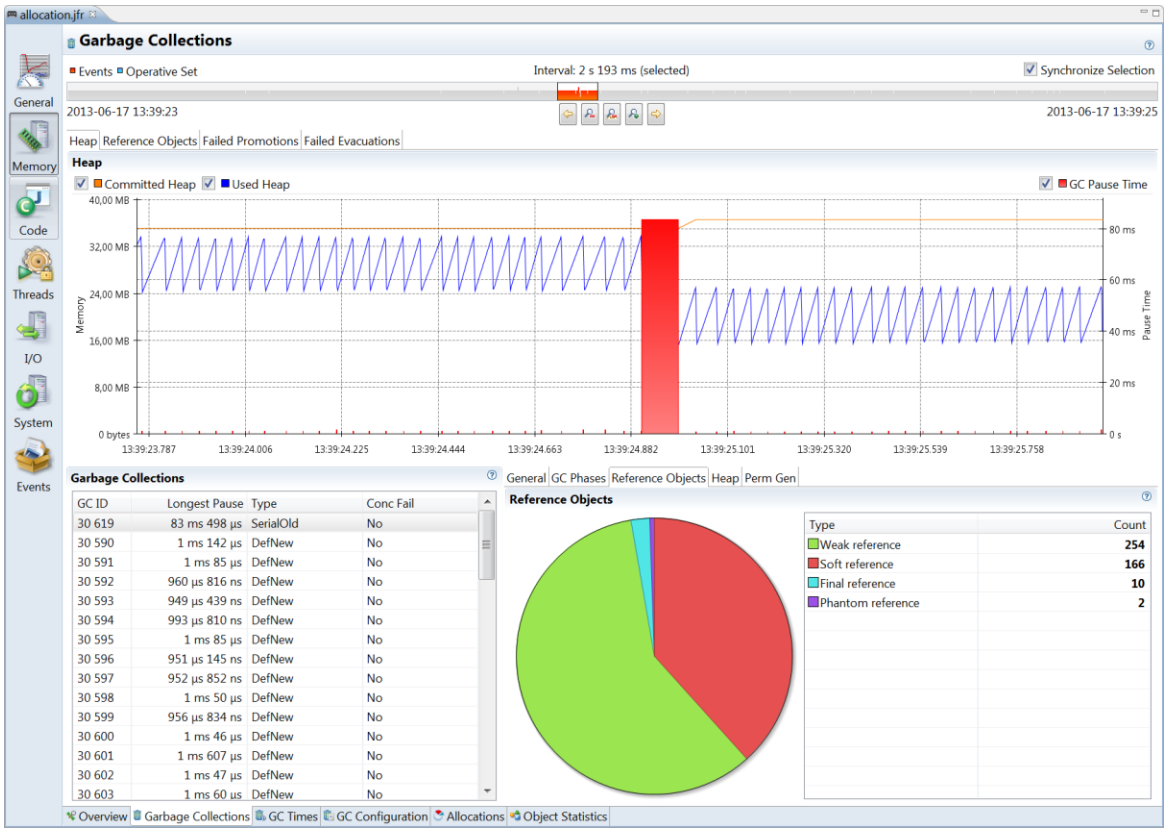
Note: *The CPU load can be seen in the Overview tab.*

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

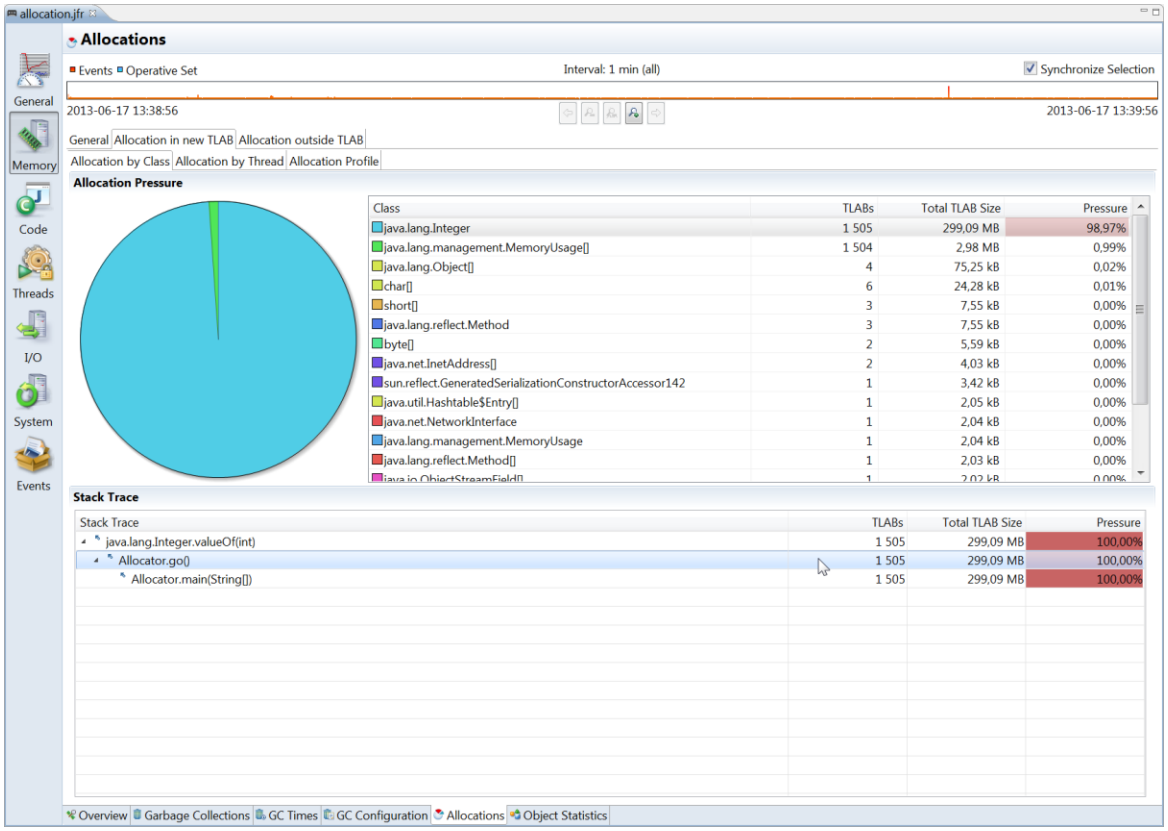
Exercise 4 (Bonus) – Garbage Collection Behaviour

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.jfr recording in the 04_JFR_GC project. Switch to the Mission Control perspective (if in Eclipse) and go to the Memory | Garbage Collections tab. In this tab 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.



Go to the Memory | Allocations tab. 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.*

Bonus Exercise:

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?

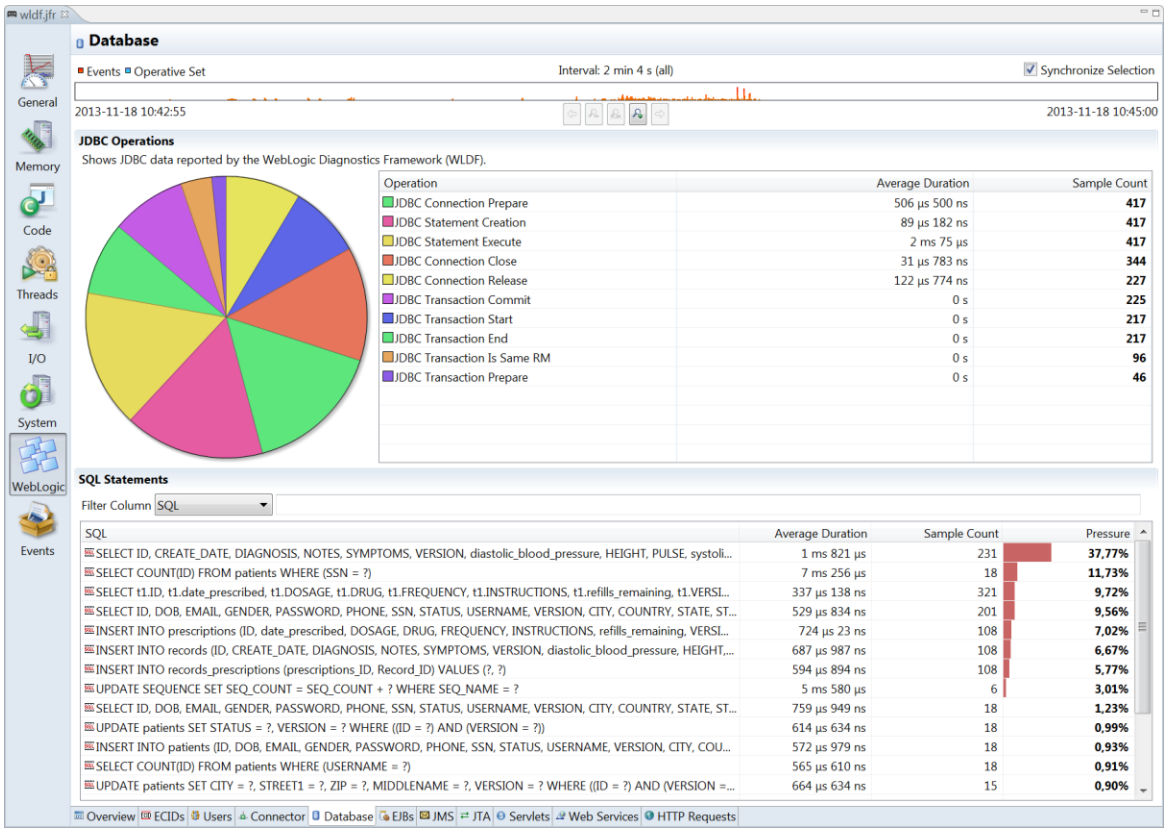
   Note: *Hints in the Readme.txt*

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.

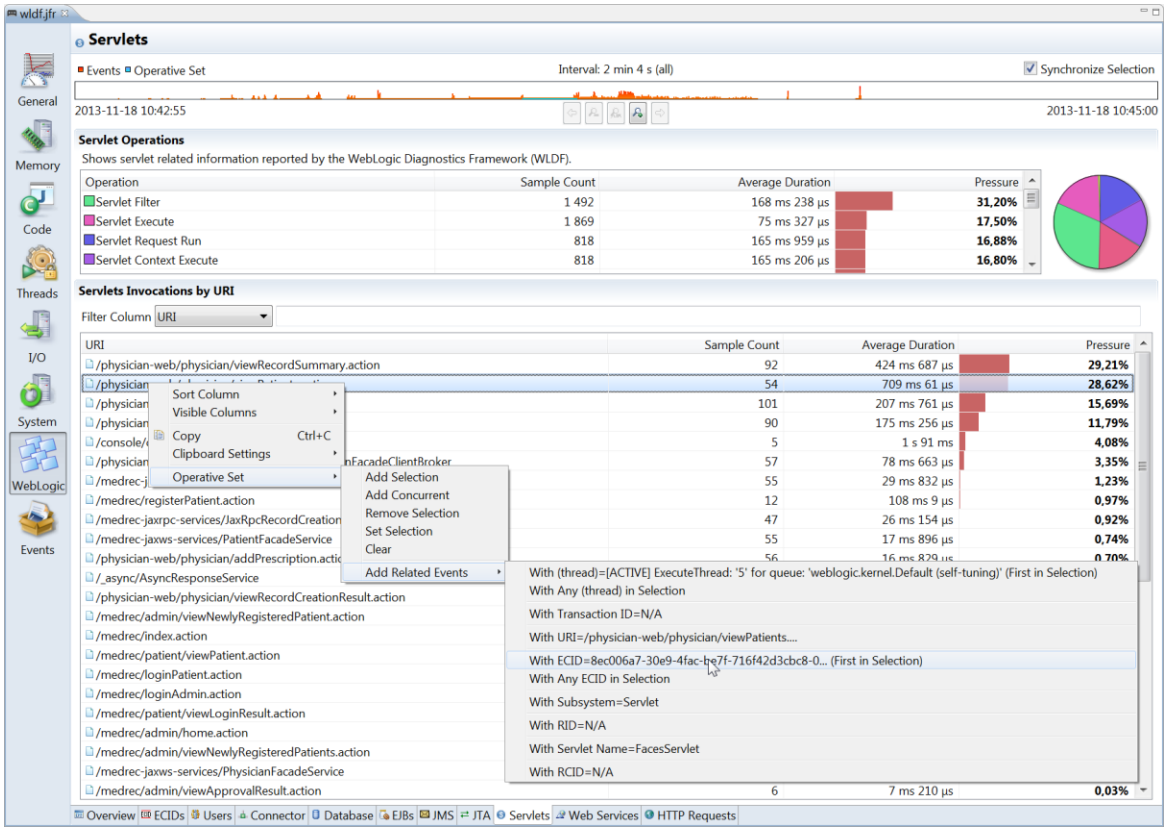
Exercise 5 (Bonus) – WebLogic Server Integration

This exercise will familiarize you with the capabilities of the integration with the WebLogic Diagnostics Framework (WLDF). First open the file named `05_JFR_WLS/wldf_medrec.jfr`. This recording contains, aside from the standard flight recorder events, events contributed by WLDF.

Open the WebLogic tab group, and switch to the database tab. Can you tell what SQL statement usually took the longest to execute, on average?



Open the Servlets tab. What are the events related to the first invocation of the /admin/viewPatients.action URI?



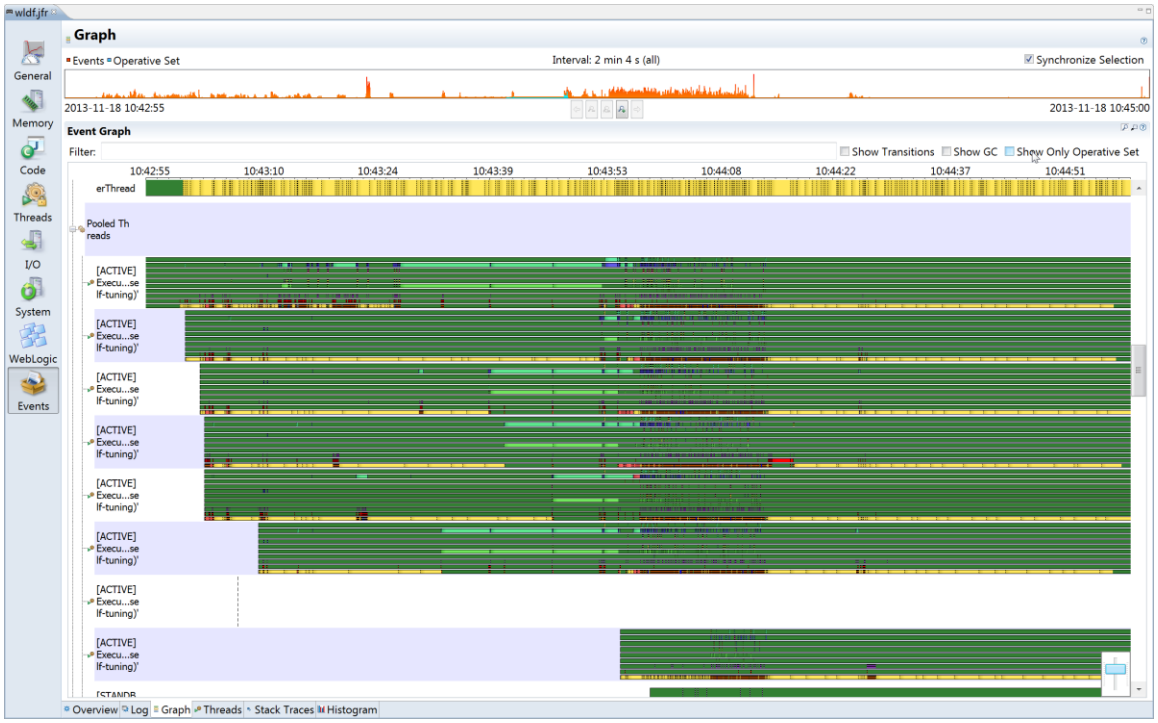
Hint: *Right click on the line representing the URI in the Servlet Invocations by URI histogram, and select Operative Set | Add Related Events | With ECID = 8ec006a7…(a very long id)*

Go to the Events tab group, select Log and make sure the Show Only Operative Set checkbox is checked. Also, make sure that you have selected to enable the WebLogic Server events in the Events Types view.

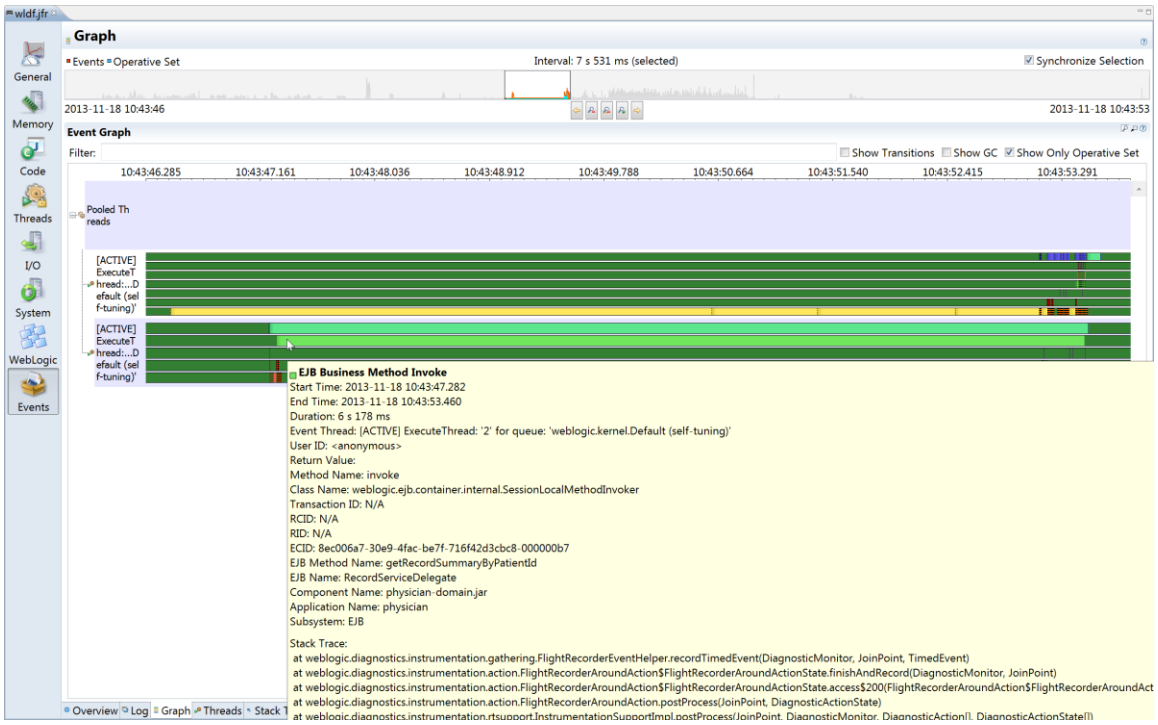
What was going on in the other layers during that first invocation of the viewPatients.action?

Hint: *In the log view, select all events (ctrl-A, make sure that Show Only Operative Set is checked), then select Operative Set | Add Concurrent. You should now be looking at the previous events, plus all the events happening concurrently as the WLS events, i.e. the events taking place in the same thread during the same time as the WLS events.*

Once all the related events are in the operative set, we can look at them in the various events tab. Go to the Events | Graph view. When viewing all the events in a recording, this view can be rather complex and daunting:



Click the box in the upper right corner for selecting to see only the operative set. Next zoom in on the group of events for a view of the events captured that were involved with serving the servlet:



Bonus Exercises:

4. Can you find the aggregated stack traces for where from code the SQL statement that took longest to execute on average originated.

   Hint: *Go to the Database tab, find the line in the lower table representing the events with the longest average duration. Right click to add those events to the operative set. Go to the Events tab group, and select the Traces tab. Check the Show Only Operative Set checkbox. Also, make sure that you have selected to enable the WebLogic Server events in the Events Types view.*

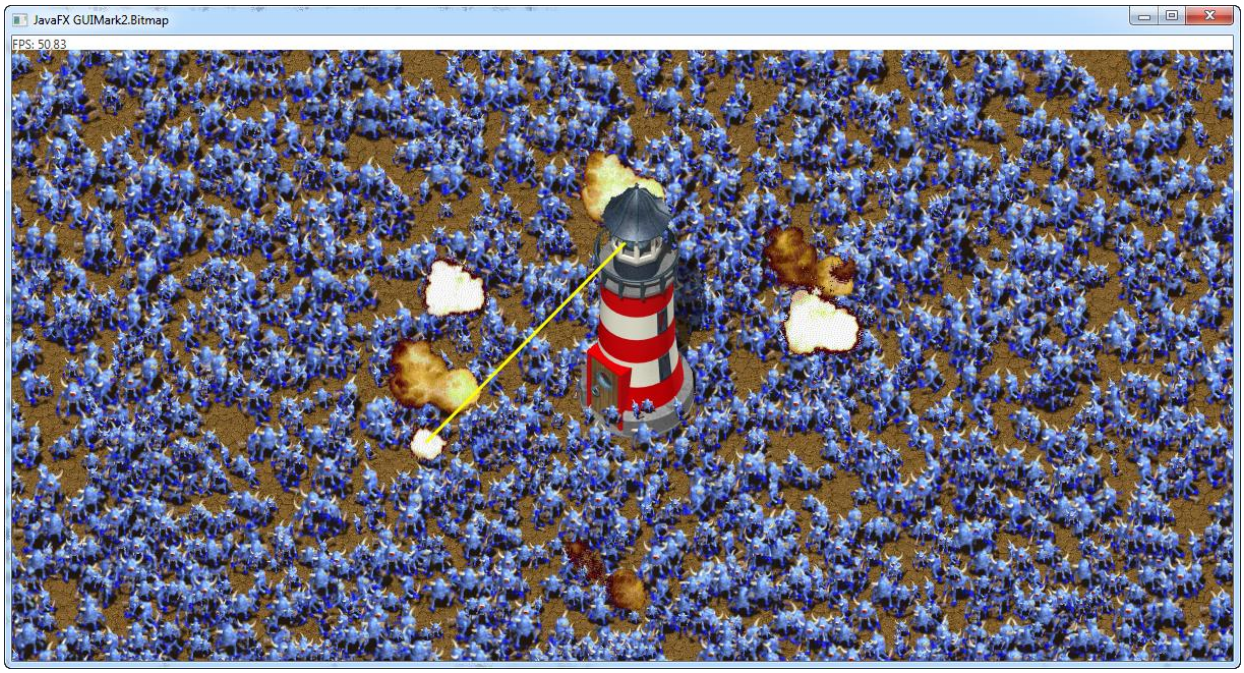
5. Can you find out which EJB the application seems to be spending the most time in?

6. Which user seems to be starting the most transactions?

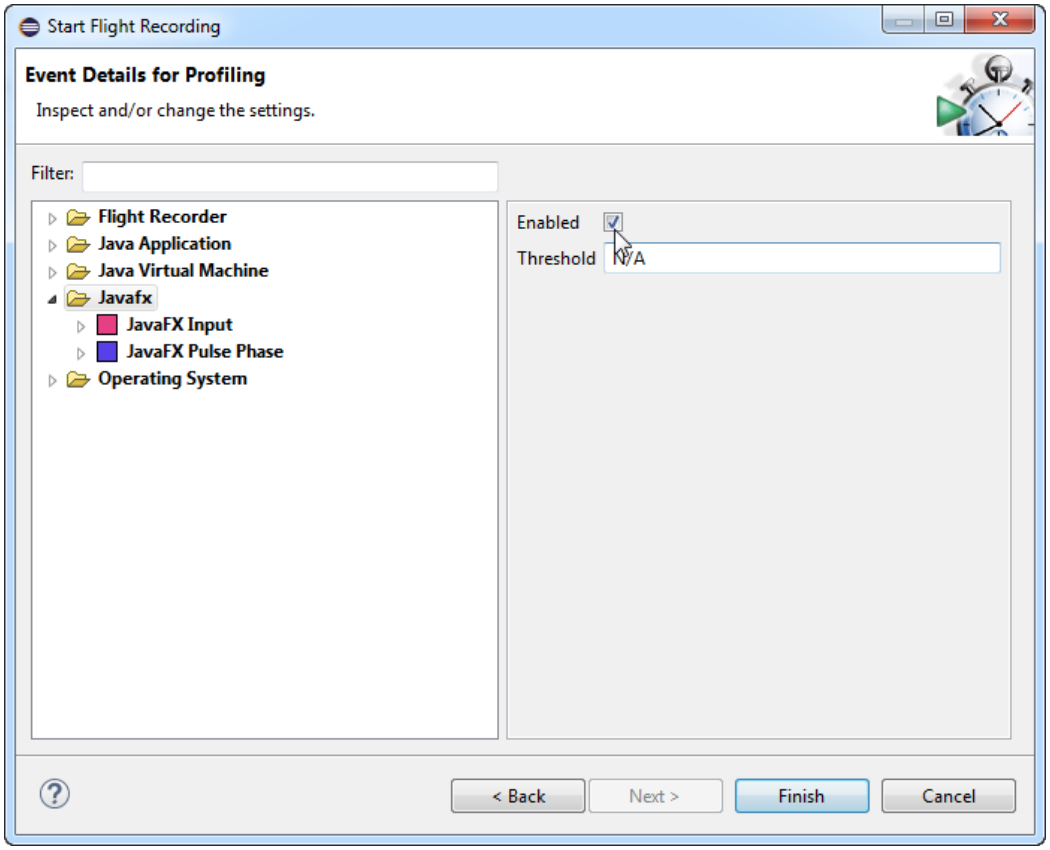
The moral of this exercise is that there are tools available that extend flight recorder and that can be quite useful/powerful. Also, using the operative set in conjunction with such tabs can be a nice way to visualize and drill down on a set of events with very specific properties.

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 lasers at monsters.



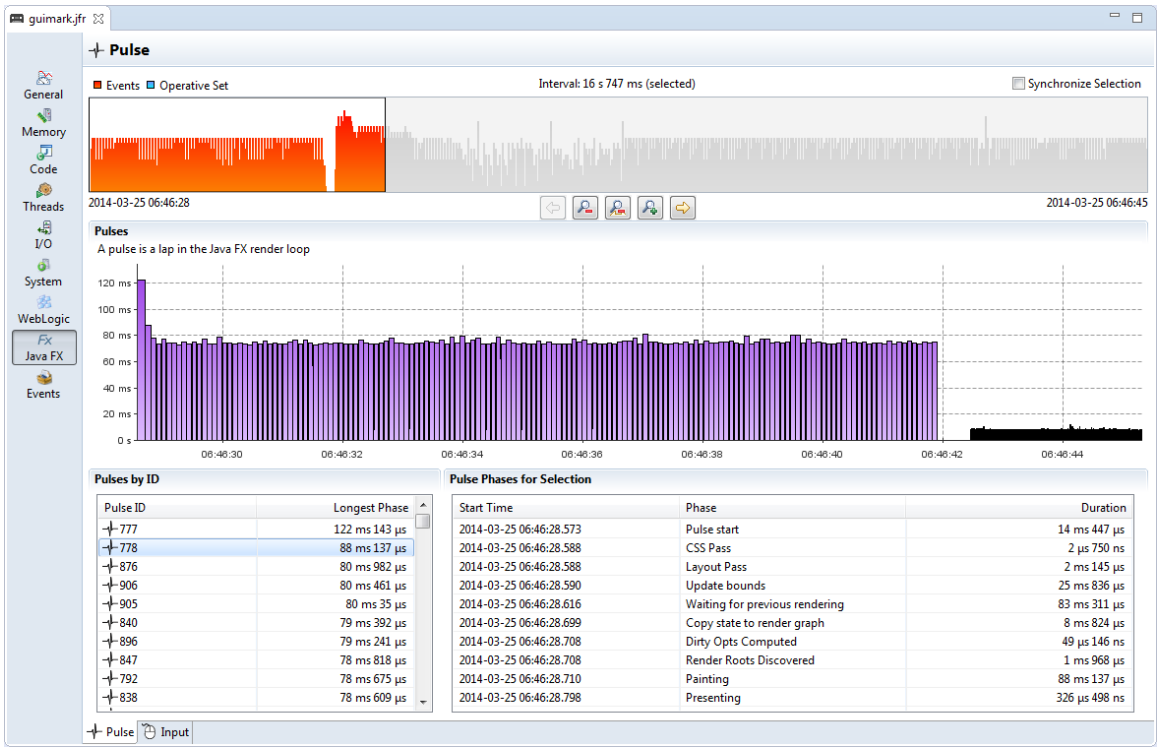
Next start a flight recording on the guimark2.BitmapTest application. This time in the recording wizard, select the server side profiling template again, however next click twice to get to the advanced, per event type settings.



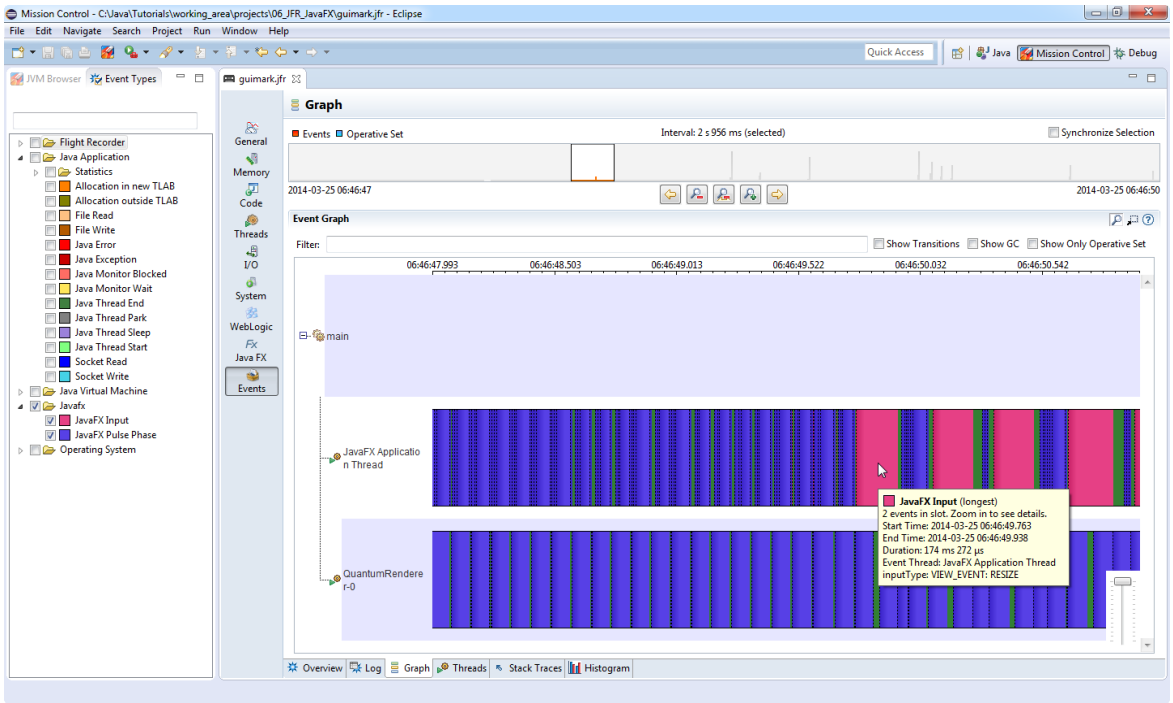
Enable the Java FX events.

During the recording, try to resize the test application window a few times.

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



How can you easily configure the thread graph view to show you only the Java FX events, and thus let you view the interaction between the Java FX application thread and the render thread?

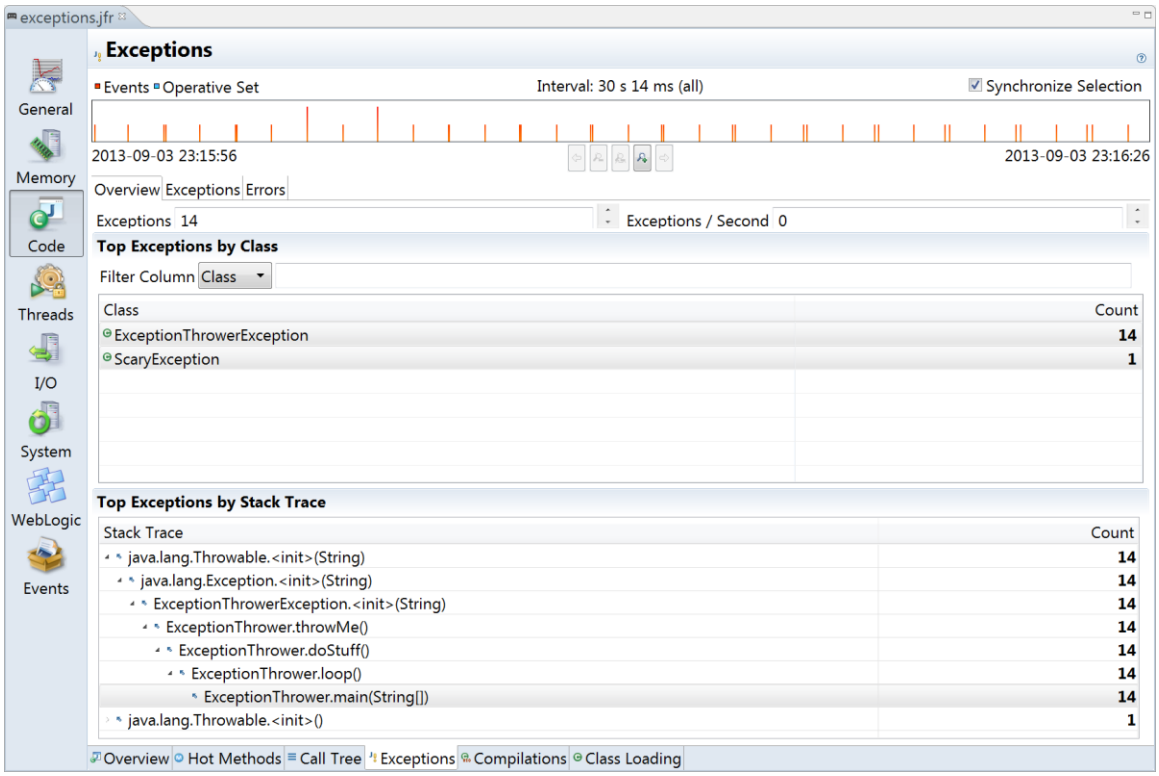


Exercise 7 (Bonus) – Exceptions

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 06_JFR_Exceptions project. Go to the Code | Exceptions tab.

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 to determine where it was thrown. You can also select multiple classes to see the aggregate traces for all classes in the selection.*

You should have at least one instance of the ScaryException. Can you tell exactly what time they are thrown? From where in the code are the instances of ScaryException thrown?

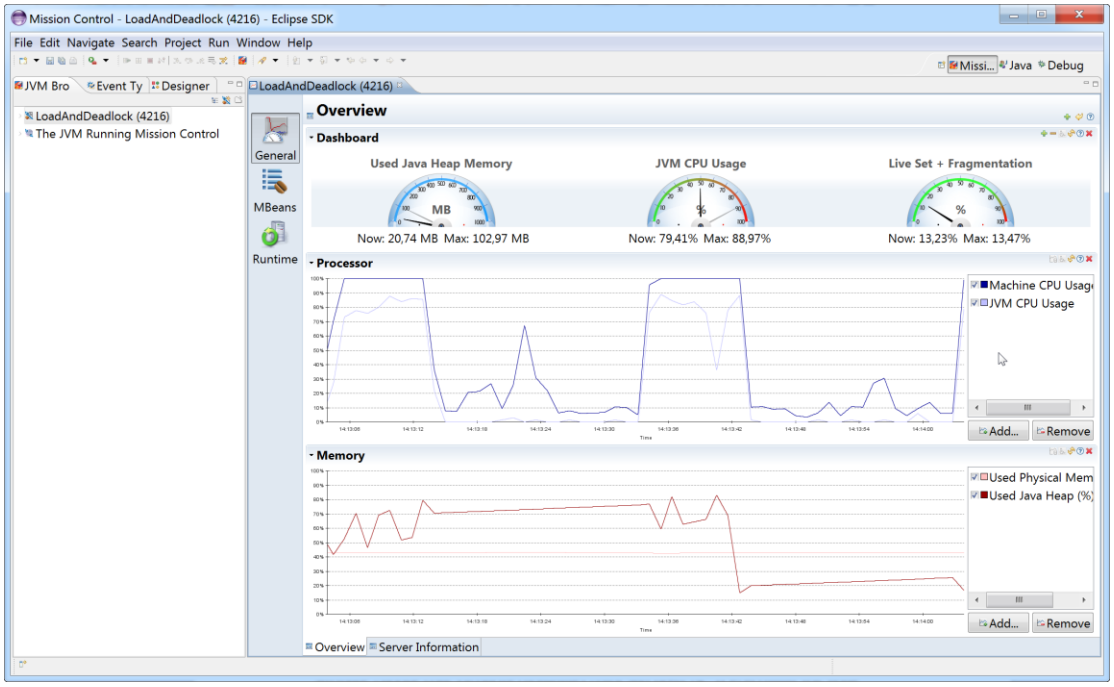
Note: *Add the events for the scary exceptions to the operative set by right-clicking the class* `ScaryException` *and selecting* Operative Set | Add selection *from the context menu. That will highlight the scary exceptions in the range selector at the top of the flight recorder editor. To get exact information, go to the* Events | Log *tab, and check* Show only operative set.

The Management Console (Bonus)

Exercise 10.a – The Overview

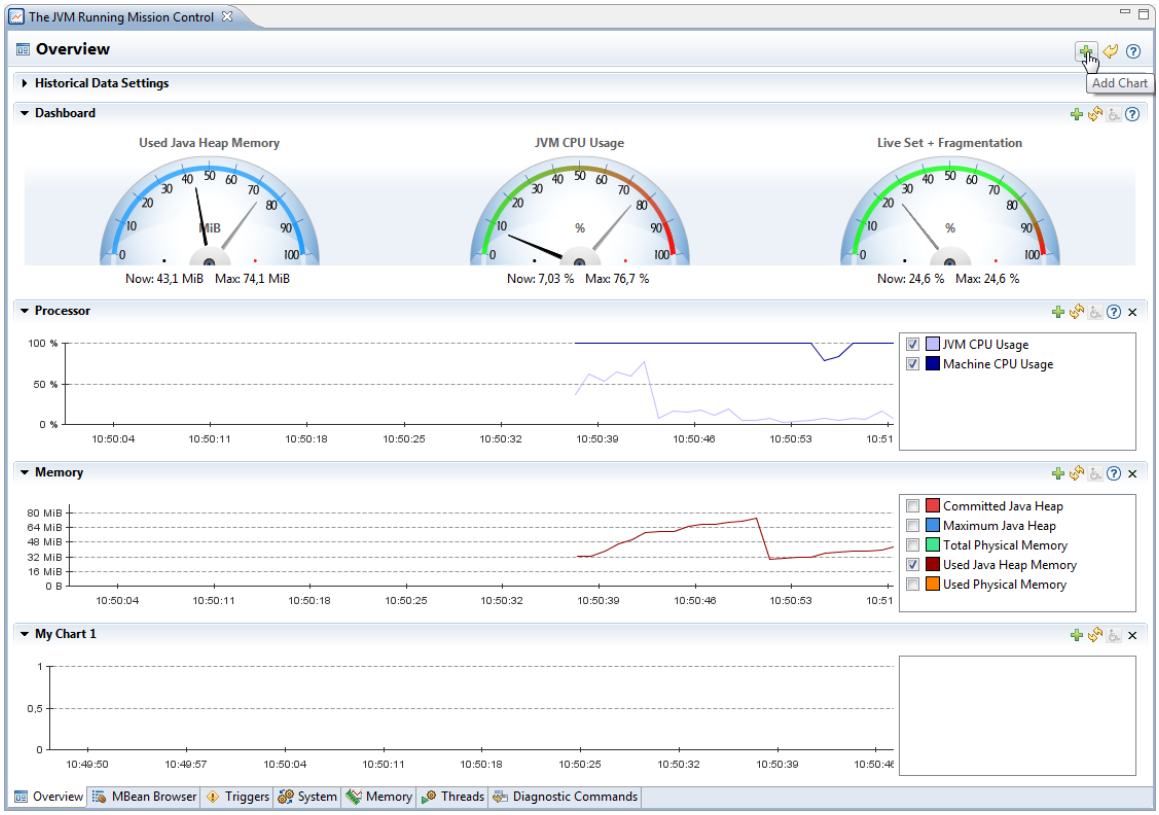
Start the `LoadAndDeadlock` program, like you did in Exercise 2.a. Then switch to the Mission Control perspective. After a little while you should see the JVM running the `LoadAndDeadlock` class appearing in the JVM Browser under the **`Discovered/Local`** node. Open a console by selecting Start Console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

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.

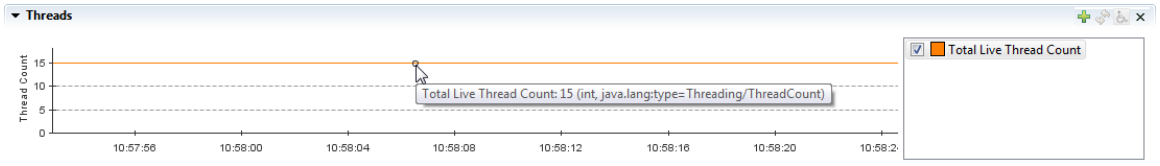


Click the Add… button of the new chart. In the attribute selector dialog, go to the Filter text field and enter “Th” (without quotation marks). Select the ThreadCount attribute, and press ok. If everything has gone according to plan, you should see the thread count weighing in at around 20.

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.*

Bonus exercises:

7. 20-ish is not very precise though. Freeze the graph and hover with the pointer over the thread count graph for a little while. What is your exact thread count?

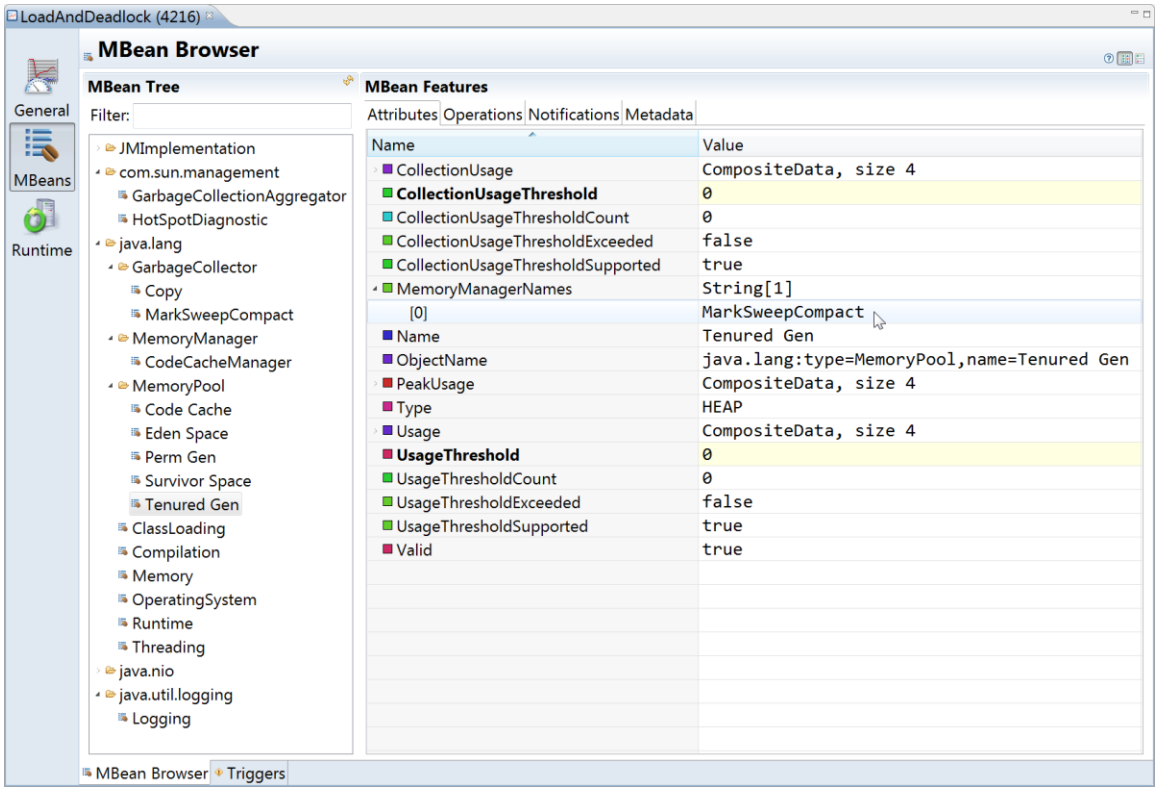


8. You decide that you dislike the live set attribute and warm up a bit to the Thread Count attribute. Remove the Live Set velocimeter in the upper right corner of the Overview tab, and instead add one for the Thread Count attribute.

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 MBeans category. The first tab should be the MBean Browser.

What is the current garbage collection strategy for the old space (the tenured 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 remove both of the new charts.

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.*

Bonus exercises (these are for JDK 8/JMC 5.3.0 and later):

9. 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* Invoke *button. You will get a new time-stamped result view for each invocation of an operation.*

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

    Note: *Use the* Diagnostic Command *tab.*

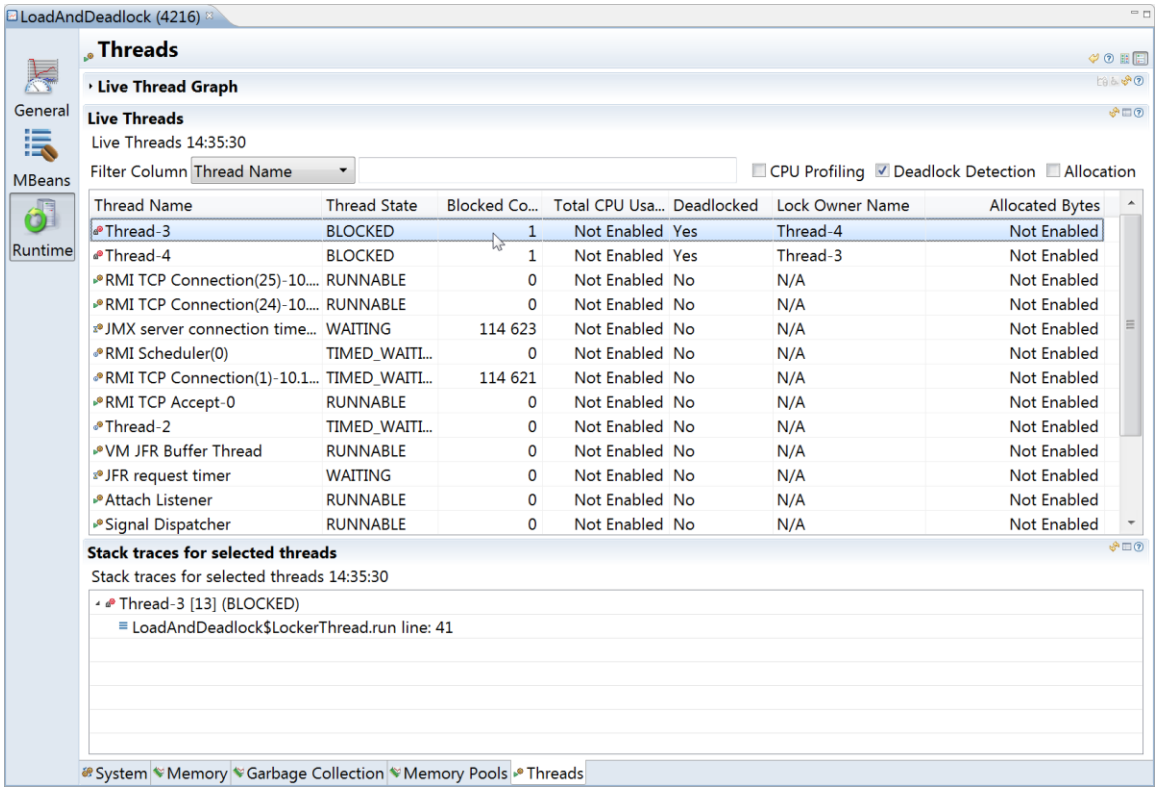
Exercise 10.c – The Threads View

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?



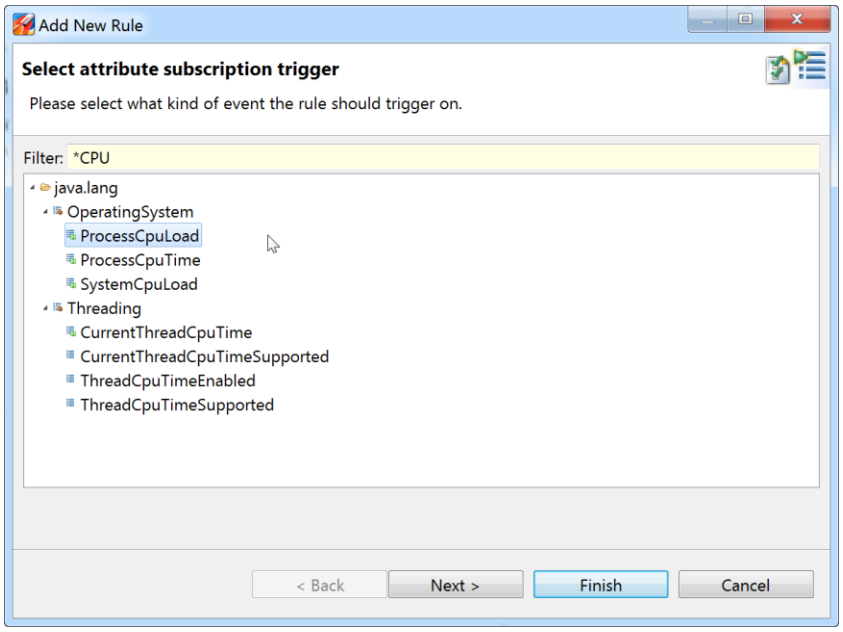
Note: *In most tables in Mission Control, there are columns that are not visualized by default. Click the little table settings icon in the upper right corner of the table to alter what is shown, or change the visibility from the context menu in the table.*

Bonus exercises:

11. 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 in the MBeans tab group. Click the Add… button. Select the **`ProcessCPULoad`** attribute and hit Next.



Select the Max trigger value to be 0.3 (30%). Click **Next**. You can see a few different actions that can be taken. Let's stick with the default (Application alert). Click Next. You can add constraints for 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.

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

DTrace (Bonus)

Since JRockit Mission Control 4.1, there is an experimental plug-in available which provides DTrace integration. DTrace is a profiling framework available on Solaris and Mac OS X. Experimental plug-ins can be downloaded into JMC using either the update site (the Eclipse plug-in version), or directly from the stand alone tool (using the Help | Install Plug-ins… menu).

The DTrace plug-in is fairly advanced. It provides DScript language extensions that make it easy to define new probes that record events in a manner that JRMC can consume. The plug-in also provides a user interface that is very similar to the Flight Recorder user interface, and just as with flight recorder, the user interface can be re-designed from within itself to accommodate for custom events.

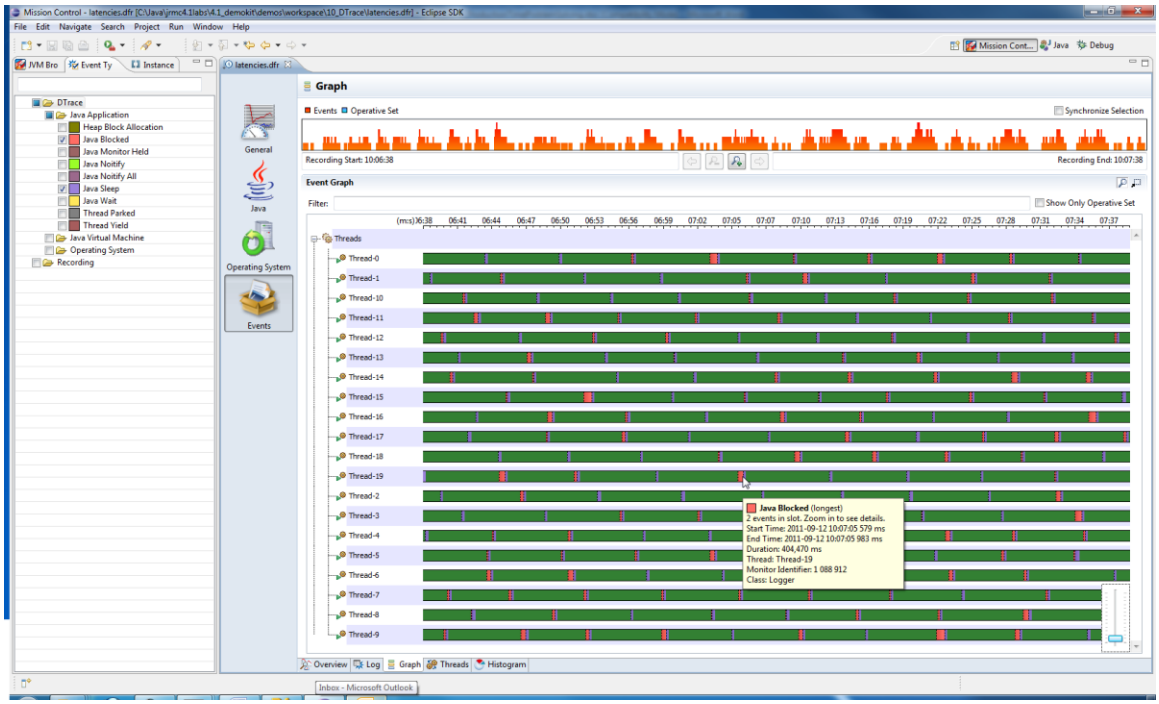
> Note:
>
> The use case for DTrace differs quite radically from flight recorder. The flight recorder is designed to be running constantly with minimal overhead, and the data dumped and analyzed when something interesting occurs. DTrace is primarily for shorter profiling sessions and where a higher overhead can be tolerated.

Exercise 11.a – Blocking Calls

In exercise 3 we solved a latency problem using the flight recorder. In this exercise we will look at the same problem using a DTrace recording of the same application.

Switch to the Java perspective and open up the **`11_DTrace/latences.dfr`** recording by double clicking it.

Can you see the blocking events? Is there a difference between the data provided by the flight recorder and DTrace.



Note: *Ensure that only the Java Blocked and Java Sleep events are enabled in the EventTypes view.*

Note: *The difference is due to several different things. Not only do the DTrace probes work differently than the corresponding Flight Recorder ones, but the HotSpot and JRockit JVMs differ as well. Here are some differences between DTrace and Flight Recorder:*

- *DTrace has access to a richer set of Operating System related data, as well as data from other processes.*
- *Flight Recorder was designed to be a black box recorder. Recording is done in self containing chunks, i.e. the recording can run continuously (all metadata and constants needed to resolve a chunk is included in the chunk).*
- *Overhead is usually less with Flight Recorder*
    - *The Flight Recorder records data about the JVM from within the JVM in thread local buffers.*
    - *Flight Recorder employs specialized assembly code for getting timestamps cheaply.*
    - *Data that the JVM needs to capture anyway can be used to emit events almost for free. For instance, piggybacking on the GC to know the heap layout, or piggybacking on the hotspot detector to do method profiling.*

- *Flight recorder events usually contain more data relevant from a Java point of view, such as the stack trace from where the event originated.*

Exercise 11.b – Building New Scripts (Bonus)

One great feature in DTrace is that there are literally thousands of probes already available from the operating system. If you know DTrace, the default template responsible for generating the events can easily be modified. Open **`11_DTrace/default.de`** file. It is currently a copy of the standard template distributed with the plug-in.

Note that the editor supports our language with syntax highlighting. Other common operations, as getting hyperlinks for jumping to declarations by holding down ctrl are also supported.

Find the template named Hotspot Profiling JDK6/JDK7. The template contains references to several different sets of probes. Can you find what probes are in the probe set named IO?

Hint: *Press and hold control, then click on IO. This trick can be used throughout the exercise.*

Look at the probe definitions. As you can see it is DScript, but with important additions. The emit keyword is new, and will emit an event in the log. Can you find out what the event that is emitted will contain?

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 in the same manner as the previous labs.

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.

Bonus exercises:

12. 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?

13. 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.*

14. 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	- Java Mission Control articles
http://twitter.com/hirt	- My twitter account