Memory Profiling Using NetBeans Profiler

Sang Shin
JPassion.com
“Learn with Passion!”

Topics

- Memory profiling features of NetBeans
- Live memory profiling
    - Generations count
- Heap walker
- Dynamic attach

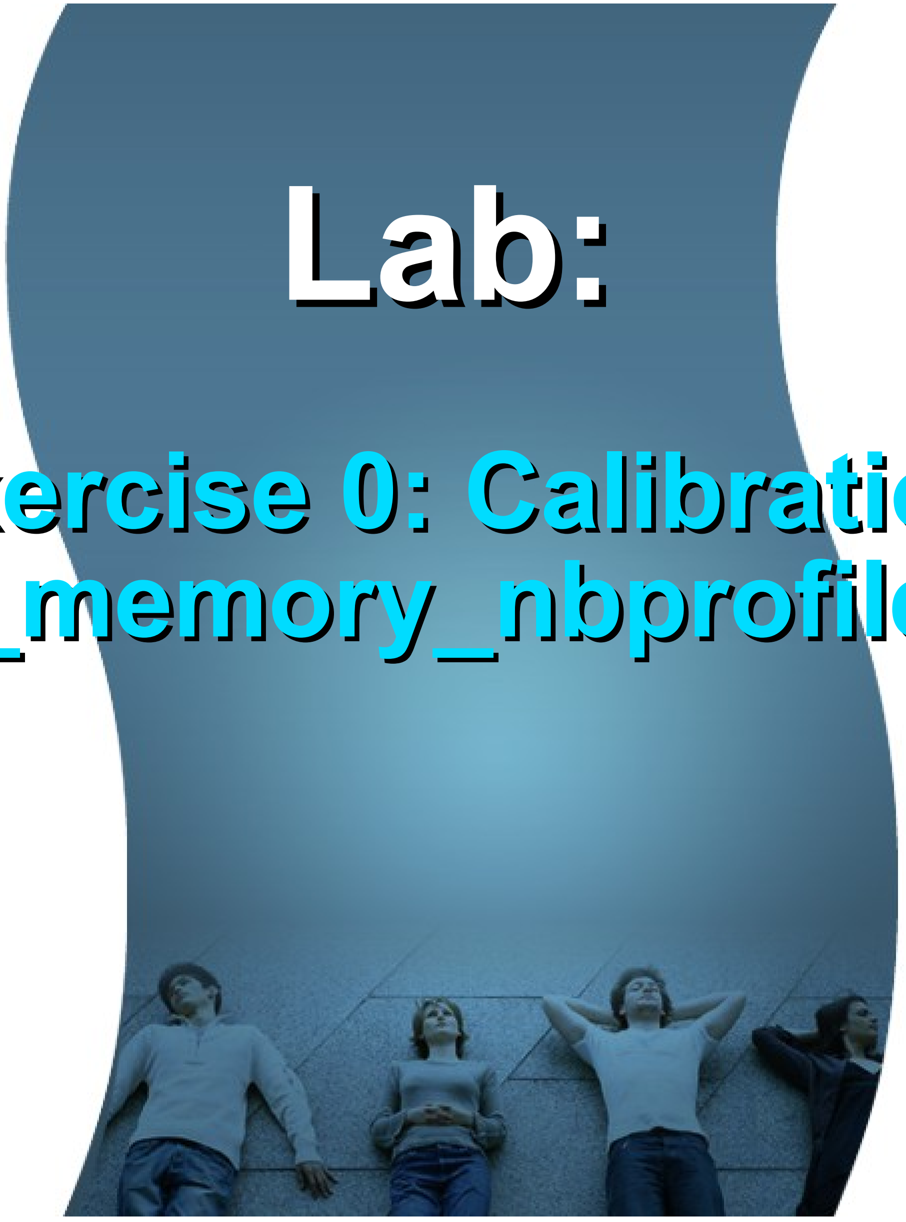
Memory Profiling Features of NetBeans

Memory Profiling Features of NetBeans

- Tracking down memory leaks on running application
  - You see allocations on the heap happen on running application
- Provides memory usage pattern
  - Detects memory leak even where each individual leak is very small
- Provides heap walker
  - Used to analyze a heap dump
  - Help you quickly identify memory leak – let you figure out why a particular object is not being garbage collected by the JVM
  - Eclipse Memory Analyzer is a more powerful heap walker with more features, however

How Does NetBeans Profiler Work?

- Through byte code instrumentation
  - Byte code instrumentation is a technique for adding bytecode to a Java class during “run time”.
- Calibrating JDK needs to be done first
  - Instrumenting the bytecode of a running application imposes some overhead
  - To guarantee the high accuracy of profiling results, NetBeans Profiler needs to collect calibration data on a JDK in order to "factor out" the time spent in code instrumentation
  - You need to run the calibration process for each JDK you will use for profiling
  - The calibration data for each JDK is saved in the .nbprofile directory in your home directory

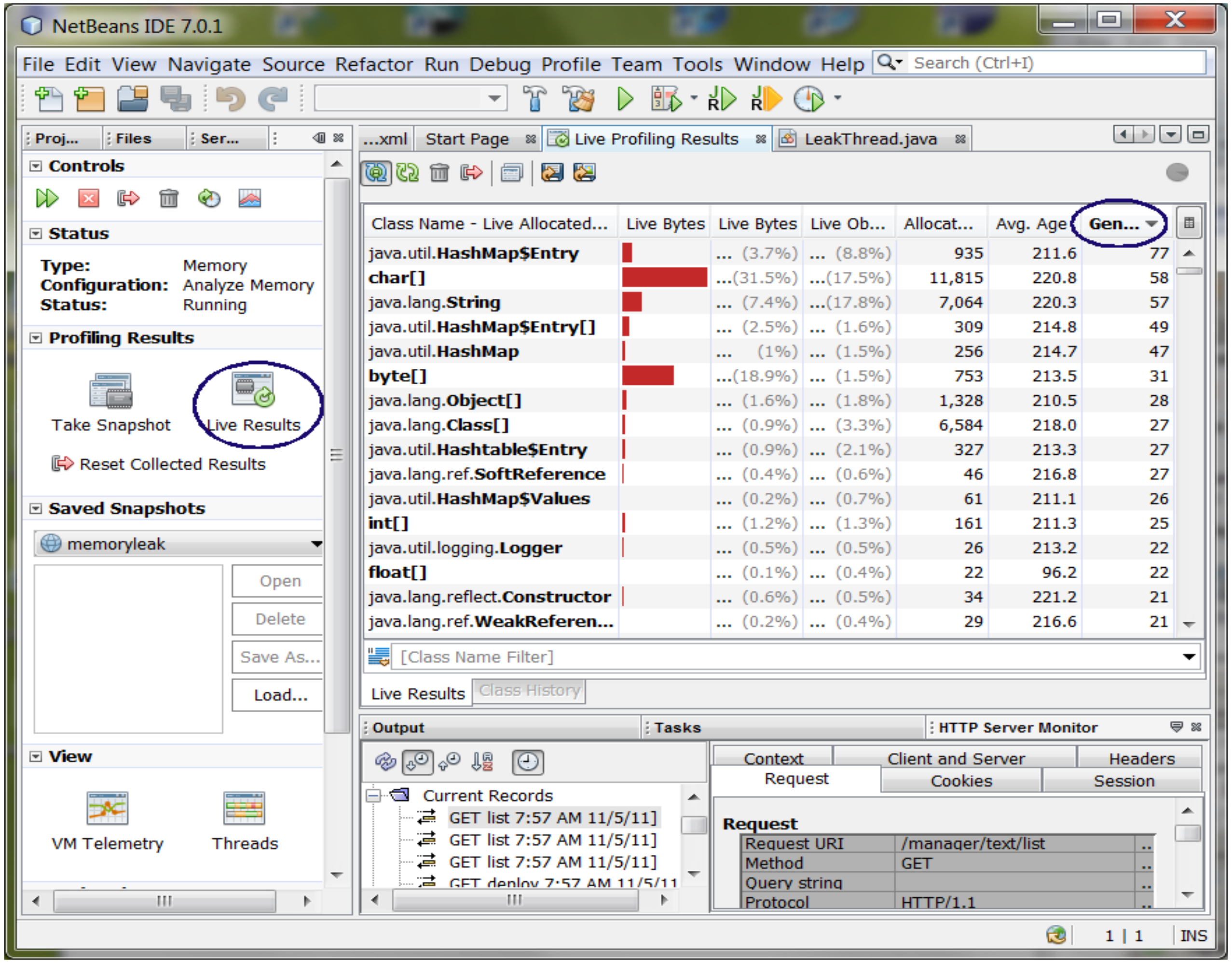


Lab:

Exercise 0: Calibration
5116_memory_nbprofiler.zip

Live Memory Profiling

Live Results (of Memory Profiling)



Status

Type:	Memory
Configuration:	Analyze Memory
Status:	Running

Class Name - Live Allocated...	Live Bytes	Live Bytes	Live Ob...	Allocat...	Avg. Age	Gen...
java.util.**HashMap$Entry**	🟥	... (3.7%)	... (8.8%)	935	211.6	77
char[]	🟥	...(31.5%)	...(17.5%)	11,815	220.8	58
java.lang.**String**	🟥	... (7.4%)	...(17.8%)	7,064	220.3	57
java.util.**HashMap$Entry[]**	🟥	... (2.5%)	... (1.6%)	309	214.8	49
java.util.**HashMap**	🟥	... (1%)	... (1.5%)	256	214.7	47
byte[]	🟥	...(18.9%)	... (1.5%)	753	213.5	31
java.lang.**Object[]**	🟥	... (1.6%)	... (1.8%)	1,328	210.5	28
java.lang.**Class[]**	🟥	... (0.9%)	... (3.3%)	6,584	218.0	27
java.util.**Hashtable$Entry**	🟥	... (0.9%)	... (2.1%)	327	213.3	27
java.lang.ref.**SoftReference**	🟥	... (0.4%)	... (0.6%)	46	216.8	27
java.util.**HashMap$Values**		... (0.2%)	... (0.7%)	61	211.1	26
int[]	🟥	... (1.2%)	... (1.3%)	161	211.3	25
java.util.logging.**Logger**	🟥	... (0.5%)	... (0.5%)	26	213.2	22
float[]		... (0.1%)	... (0.4%)	22	96.2	22
java.lang.reflect.**Constructor**	🟥	... (0.6%)	... (0.5%)	34	221.2	21
java.lang.ref.**WeakReferen...**		... (0.2%)	... (0.4%)	29	216.6	21

```
Current Records
    GET list 7:57 AM 11/5/11]
    GET list 7:57 AM 11/5/11]
    GET list 7:57 AM 11/5/11]
    GET deploy 7:57 AM 11/5/11
```

Request

Request URI	/manager/text/list
Method	GET
Query string	
Protocol	HTTP/1.1

Key Columns (of Live Results)

- Allocated objects - the number of objects that the profiler is monitoring
- Live objects - the number of the Allocated Objects that are still on the JVM's heap and are therefore taking up memory
- Average age - The age of each live object is the number of garbage collections that it has survived. The sum of the ages divided by the number of Live Objects is the Average age.
- Generations - The Generations count is the number of different ages for the Live Objects (we will talk more on this on the following slides)

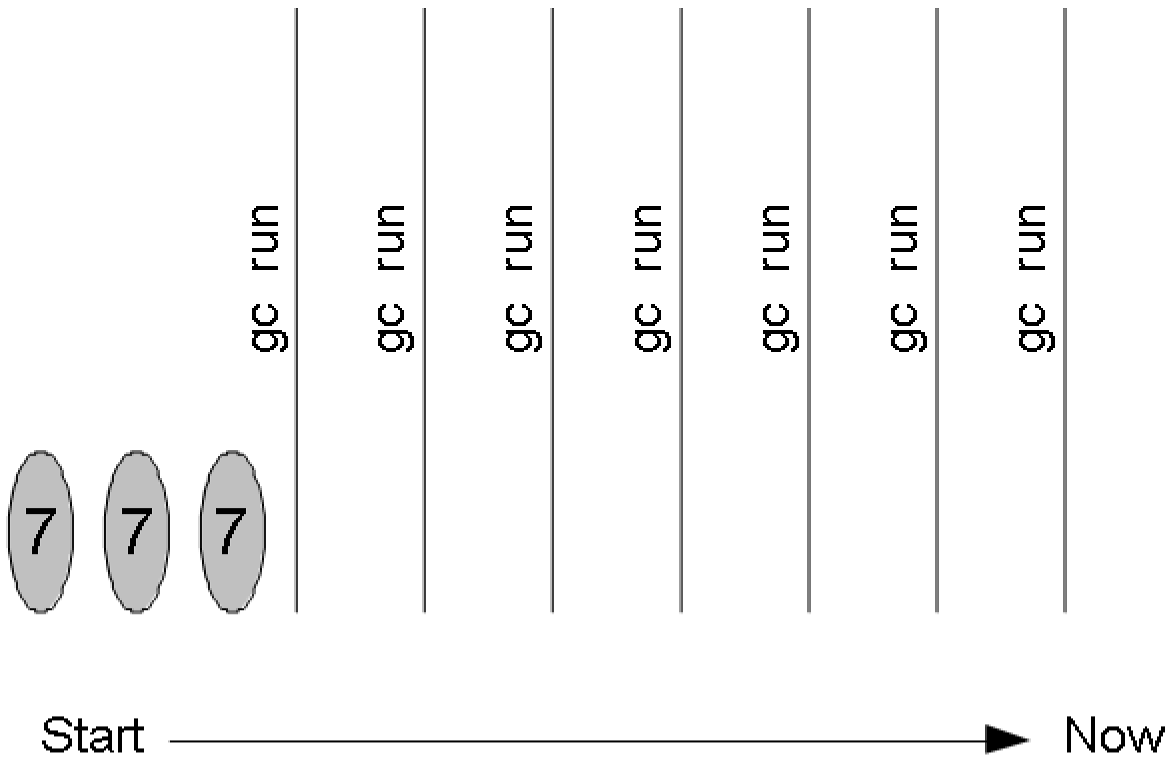
Generations Count

Generations Count & Memory Leak

- The Generations count is the number of different ages for the Live Objects
- If the Generations count keeps increasing, it indicates that new objects are being created while the old ones are still in memory
  - Indicates memory leak is occurring
- Types of objects in typical application
  - Long-lived objects: created at the beginning and stay alive for a long time
  - Short-lived objects: created and destroyed in short duration

Generation Count: Long Lived Objects

- One Example of Healthy Behavior:



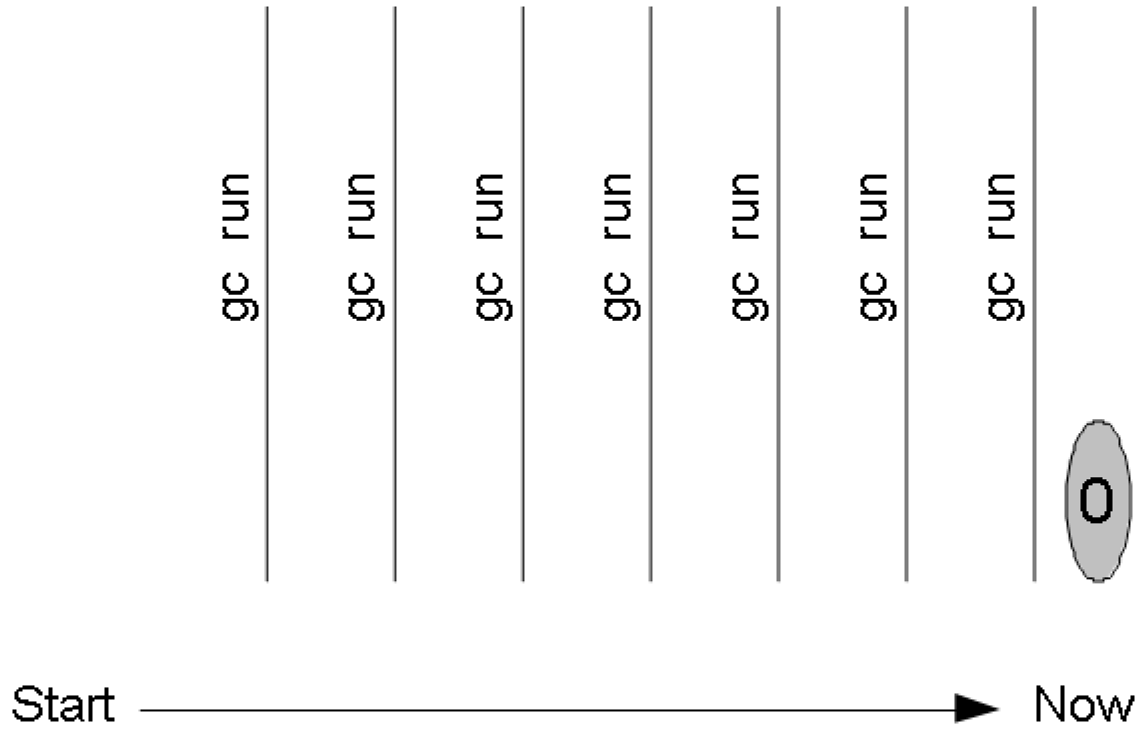
Long-lived objects.

Example: Three object instances created at startup.

Their ages continue to increase, but generation count remains stable (at 1)

Generation Count: Short-lived objects

- Another Example of Healthy Behavior:



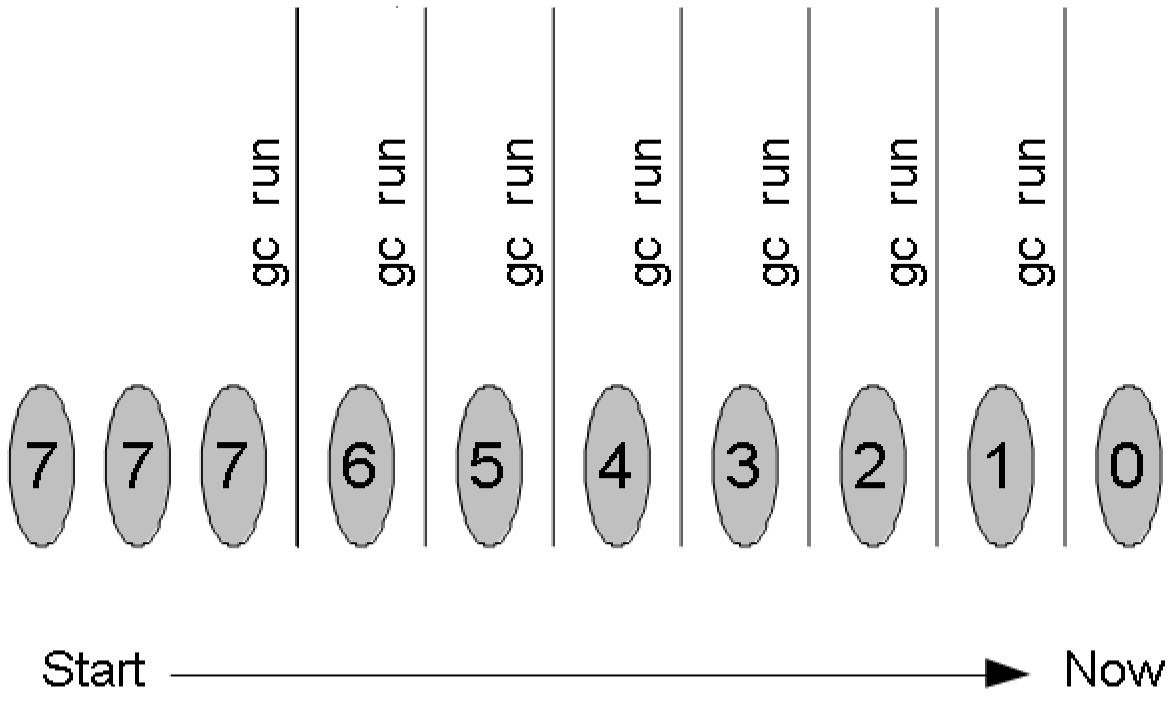
Short-lived objects

Example: Create an object, use it and then immediately let go of all references to it.

Generation count remains stable (at 1)

Generation Count: Leaking Objects

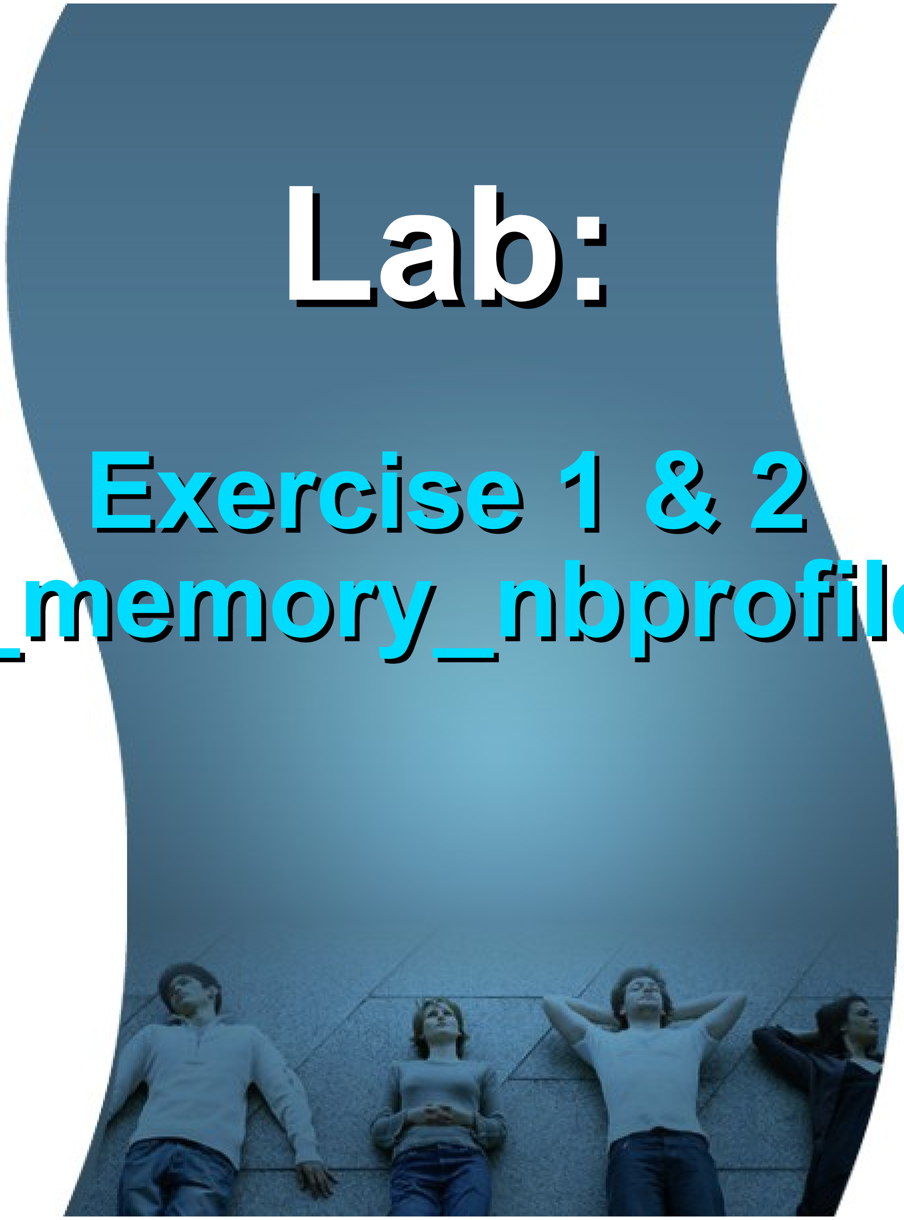
- Unhealthy Behavior (Memory Leak):



Example: Continue to allocate objects without letting go of all references.

Ten objects with eight ***different*** ages.

Generation count is ***always increasing (1 to 8)***.



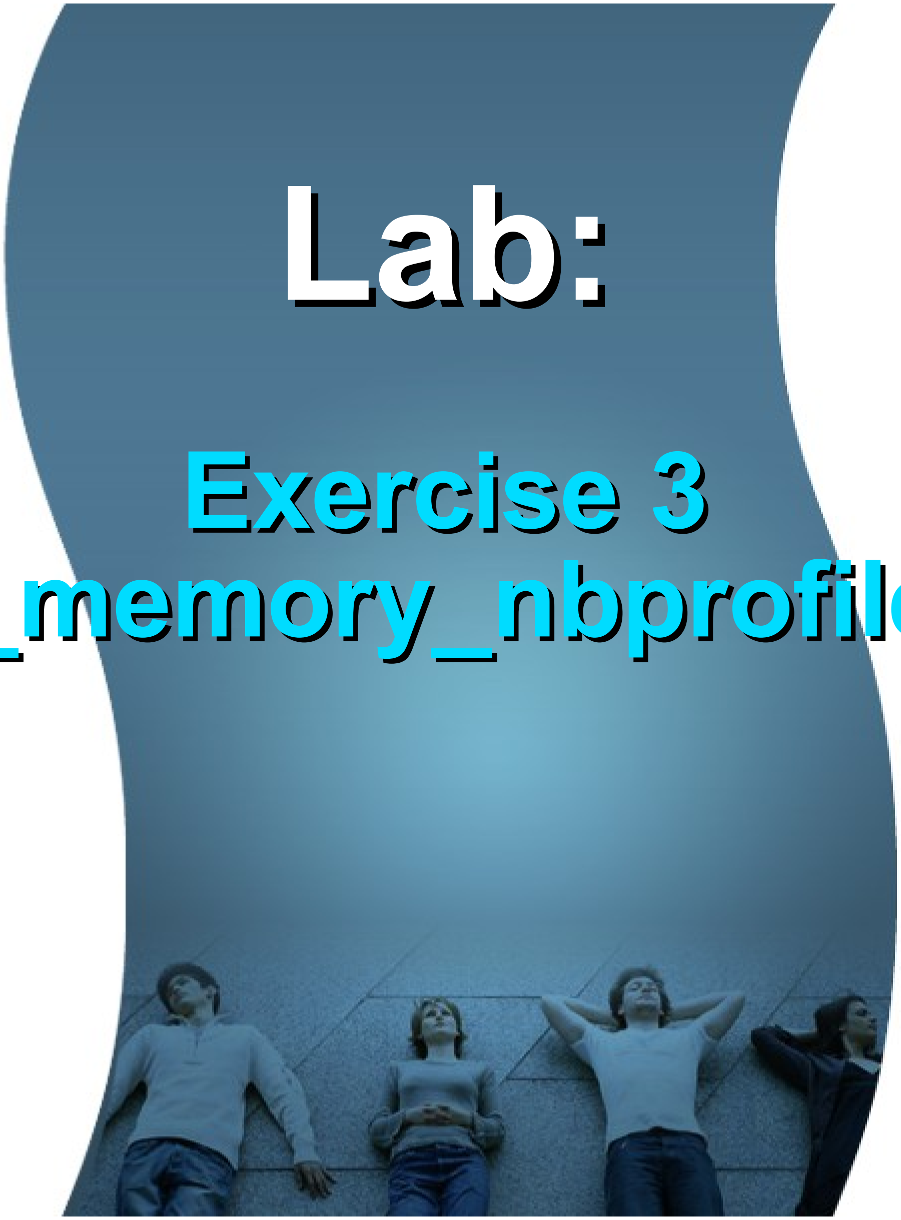
Lab:

Exercise 1 & 2
5116_memory_nbprofiler.zip

Heap Walker

What does Heap Walker do?

- It reads the heap dump and provides a complete picture of the objects on the heap and the references between the objects
  - Useful to analyze a heap dump produced when an *OutOfMemoryError* occurs
- The “Find Nearest GC Root” feature can help you track down the cause of memory leaks by showing the owner of the reference that prevents an object from being garbage collected
  - Garbage Collection (GC) roots are the objects that never get removed from the heap - they are the starting point for the JVM's garbage collector.
  - Any object that is reachable from a GC root cannot be removed from the heap



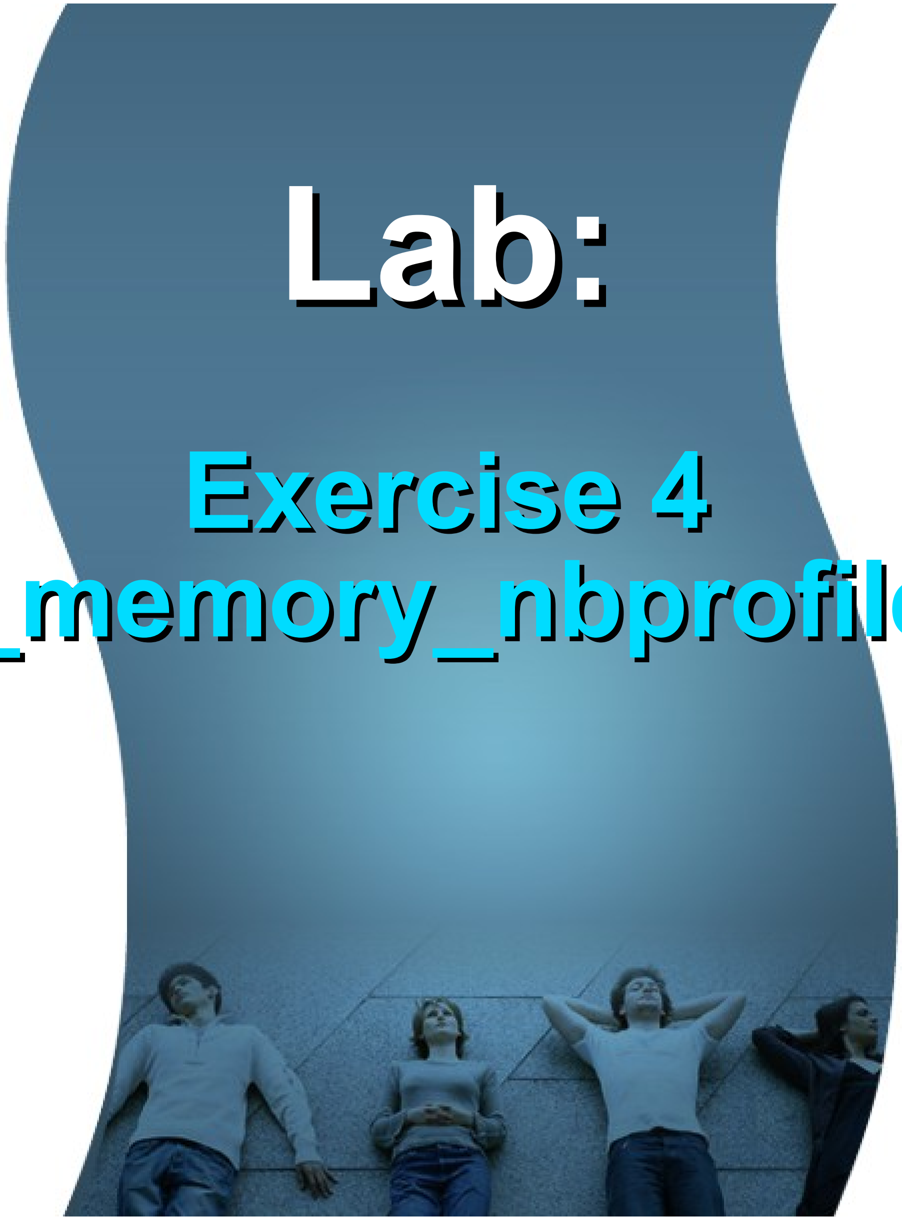
Lab:

Exercise 3
5116_memory_nbprofiler.zip

Dynamic Attach

Attaching Profiler

- With JDK 6+, you can attach the profiler to an application that is already running
  - No special JVM command line flags are necessary when you start that target application
  - Useful for profiling production application
  - It is called Dynamic attach



Lab:

Exercise 4
5116_memory_nbprofiler.zip

Learn with Passion!
JPassion.com