




ORACLE®

Introduction of Java GC Tuning and Java Mission Control

Leon Chen (陳逸嘉)
Principal Sales Consultant
Oracle



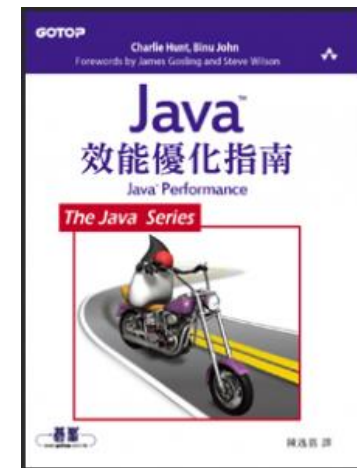
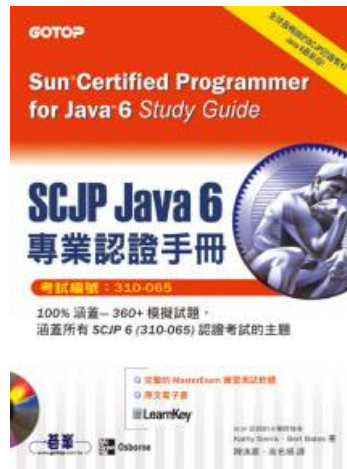
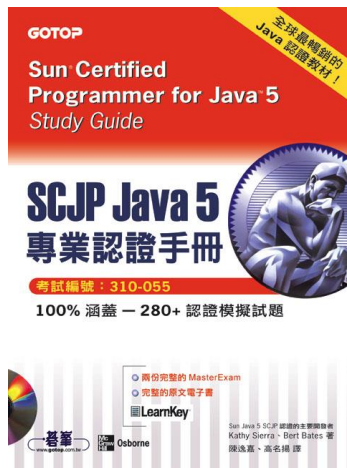


The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Leon Chen (陳逸嘉)

- More than 14 years in Java/JEE, as programmer, architect, leader and consultant, in finance/telecom domain.
- 工研院
- 得捷
- 新光人壽
- Ericsson
- Oracle

- **TW Patent 182927**, 一種文章切割方法, **2003**, 陳逸嘉, 林一中
- **TW Patent 206819**, 自動服務組合方法及系統, **2004**, 陳逸嘉, 許維德, 洪鵬翔
- **US Patent 7,617,174**, Method and system for automatic service composition, **2009**, Leon Chen, Wei-Tek Hsu, Peng-Hsiang Hung

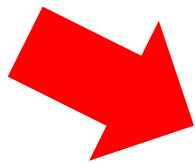




Java Roadmap



JRockit



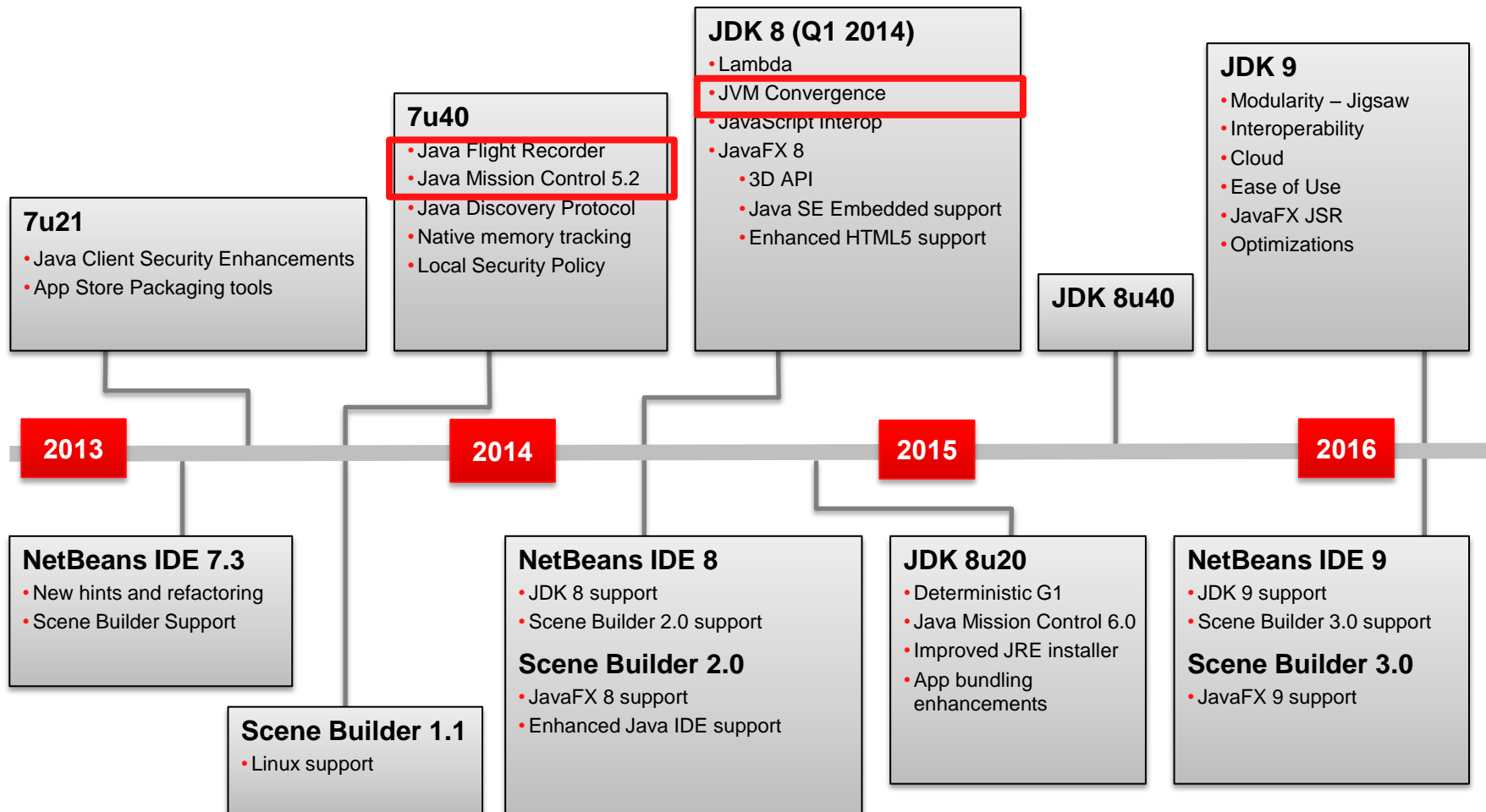
Hotspot



HotRockit

(JDK7_update 40 almost; JDK 8, 2014 暫定)

Java SE Roadmap



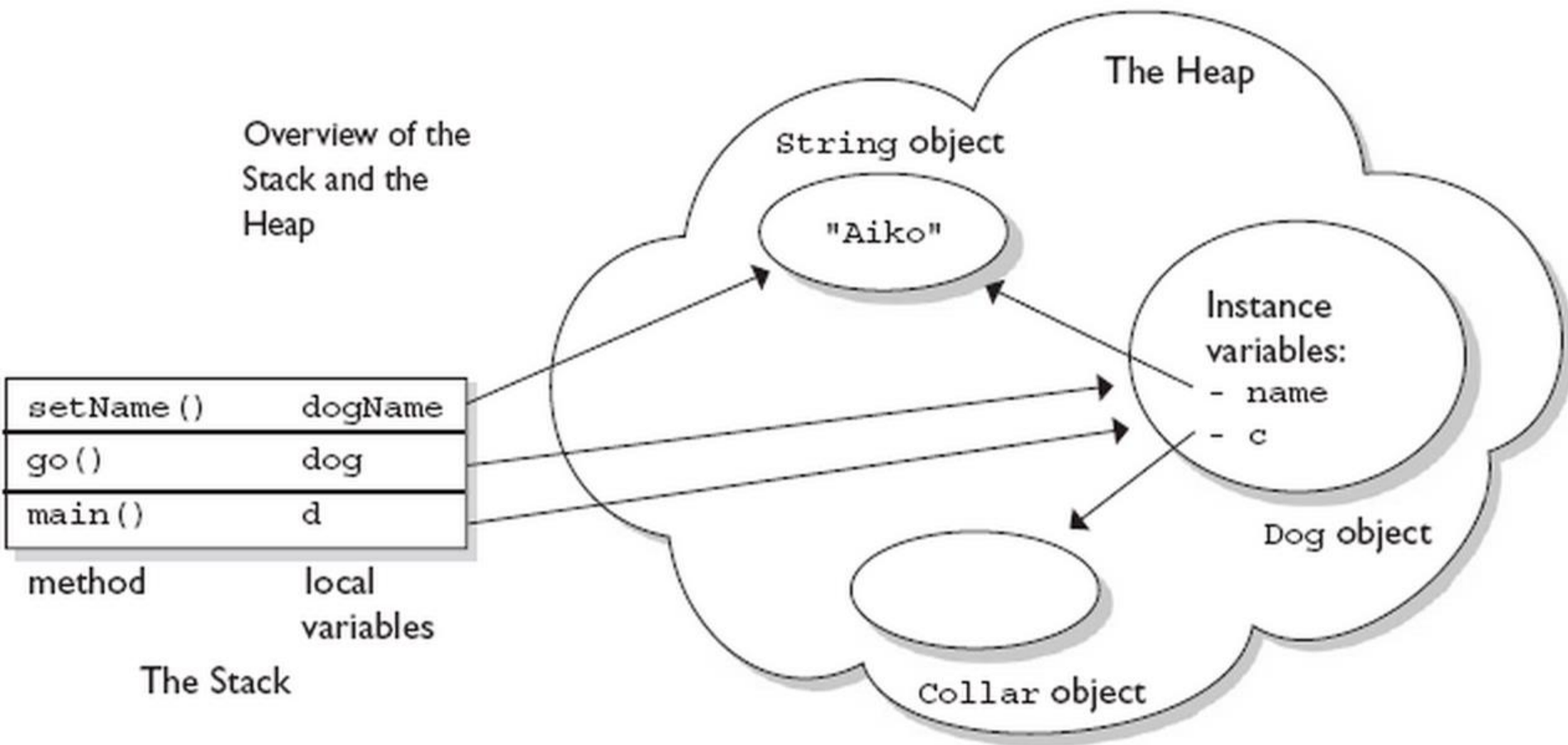


JVM GC Tuning

32-bit VS. 64-bit

- 32-bit
 - For heap sizes up to 2.5G/3G or so
- 64-bit
 - Windows/Linux: different JVM, Solaris: add `-d64`
 - `-XX:+UseCompressedOops` (default JDK6_23+)
 - Compressed references: 32GB Max (26GB best)
 - `-Xmx`: 26G (compressed) / unlimited (regular)
- 32-bit → 64-bit migration
 - Higher heap size requirements (around 20%)
 - Slight throughput impact (**without** compressed refs)
- 64-bit preferred for today's servers
 - Only option starting with Fusion Middleware 12c

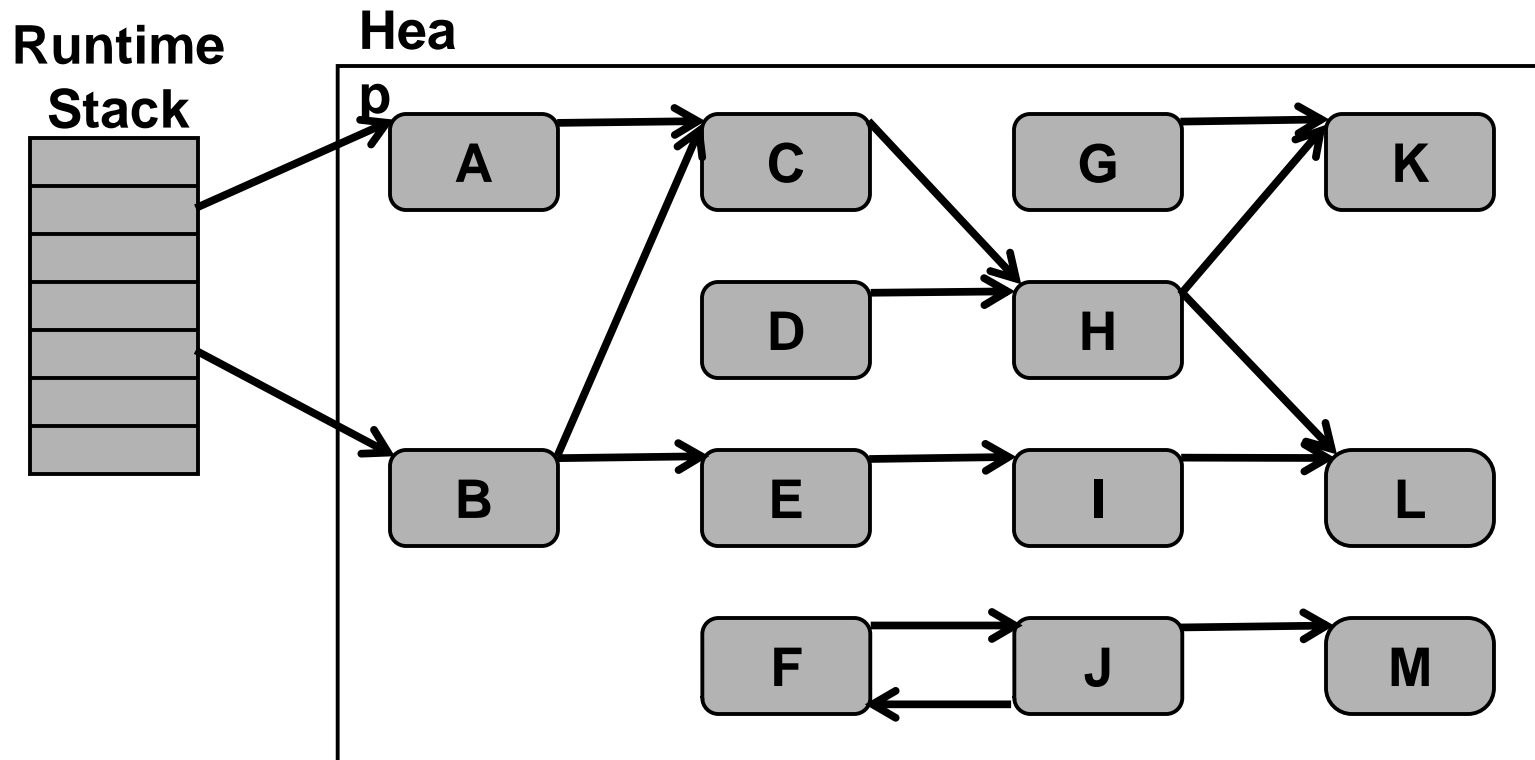
Java Memory Management



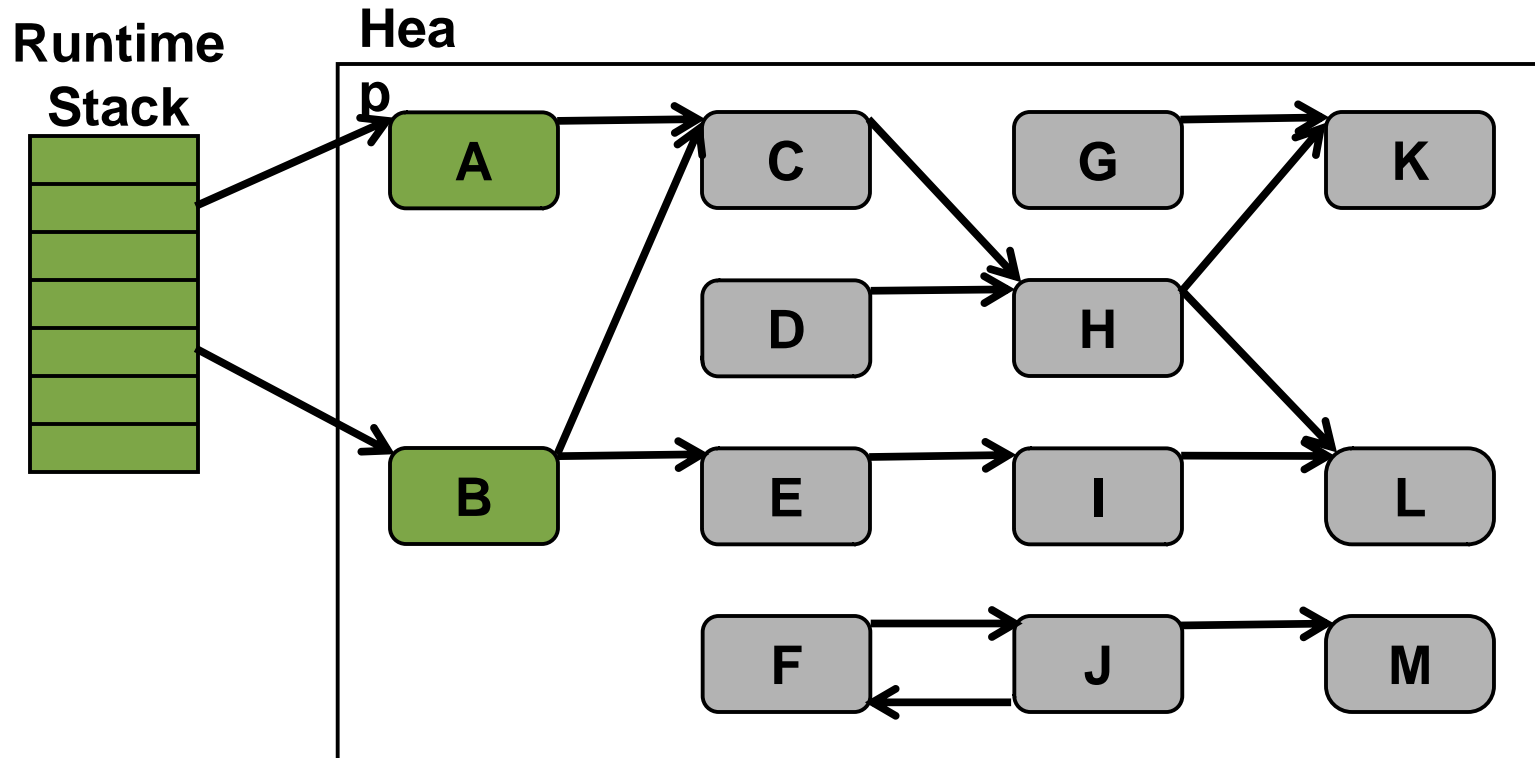
Reference video: <http://www.youtube.com/watch?v=VQ4eZw6eVtQ>

```
class Test {  
    private List<User> userList =  
        new ArrayList<User>();  
  
    ...  
    public void execute() {  
        User user = new User();  
        userList.add(user);  
        ...  
        user = null;    //No use  
    }  
    ...  
}
```

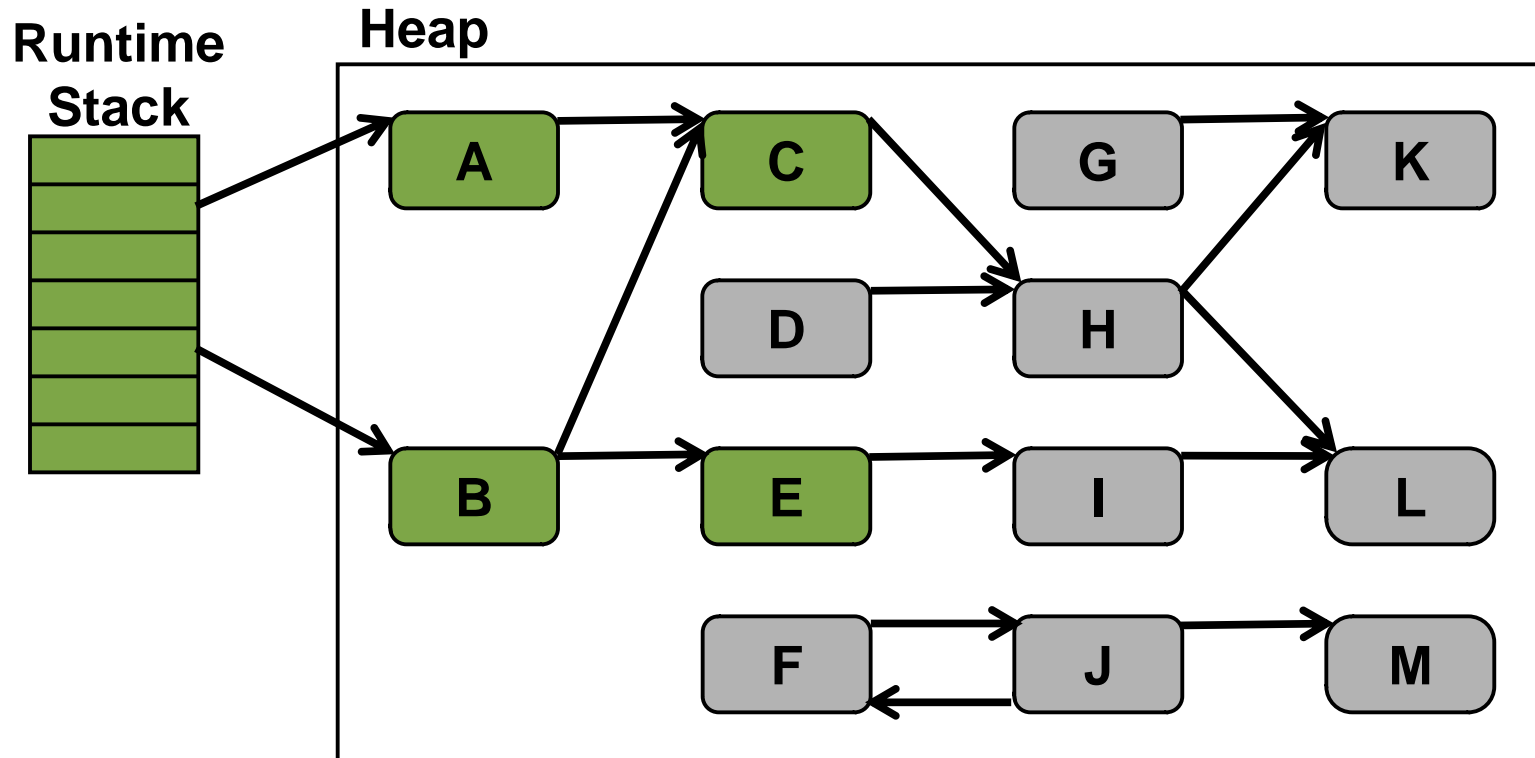
Tracing GC Example



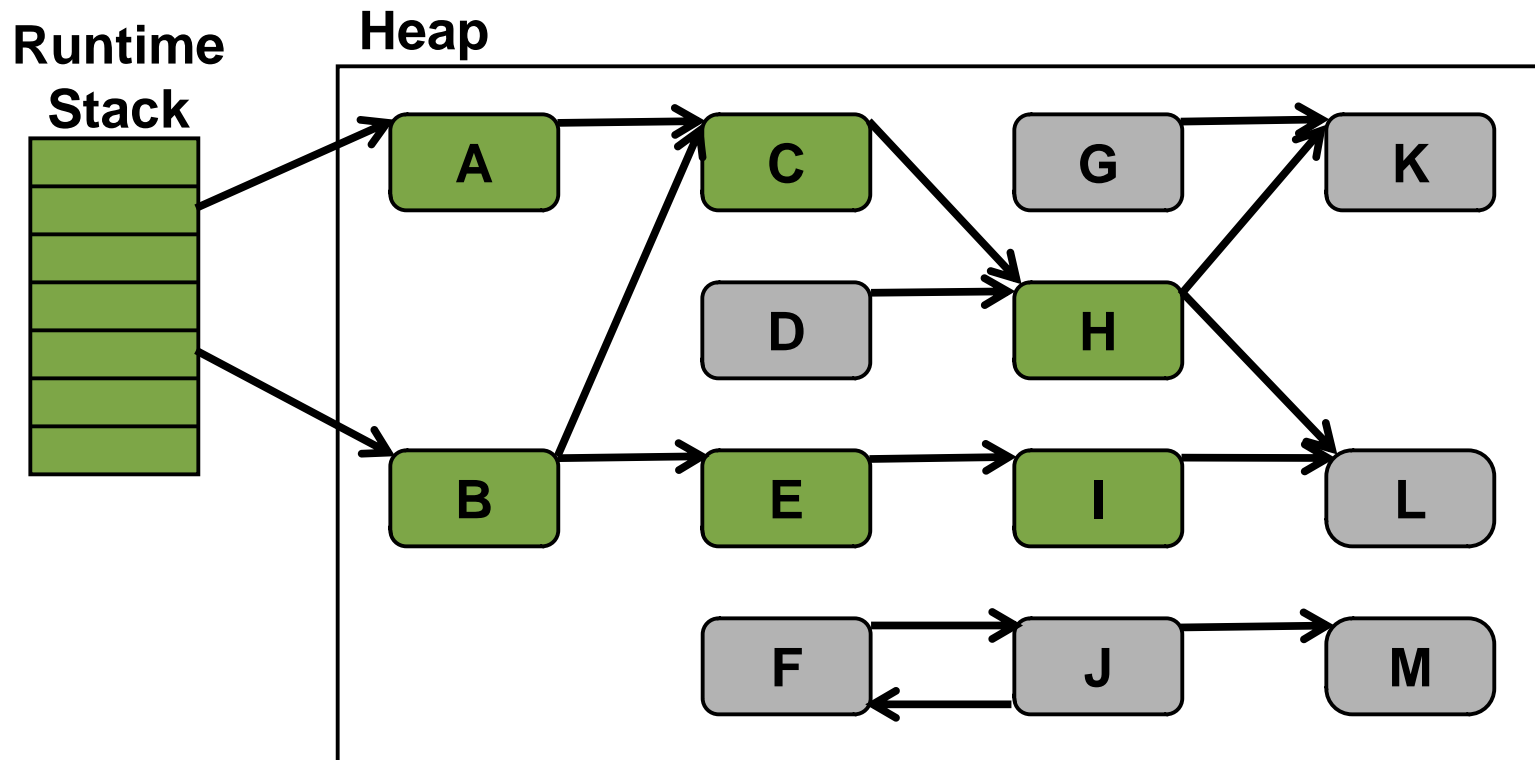
Tracing GC Example



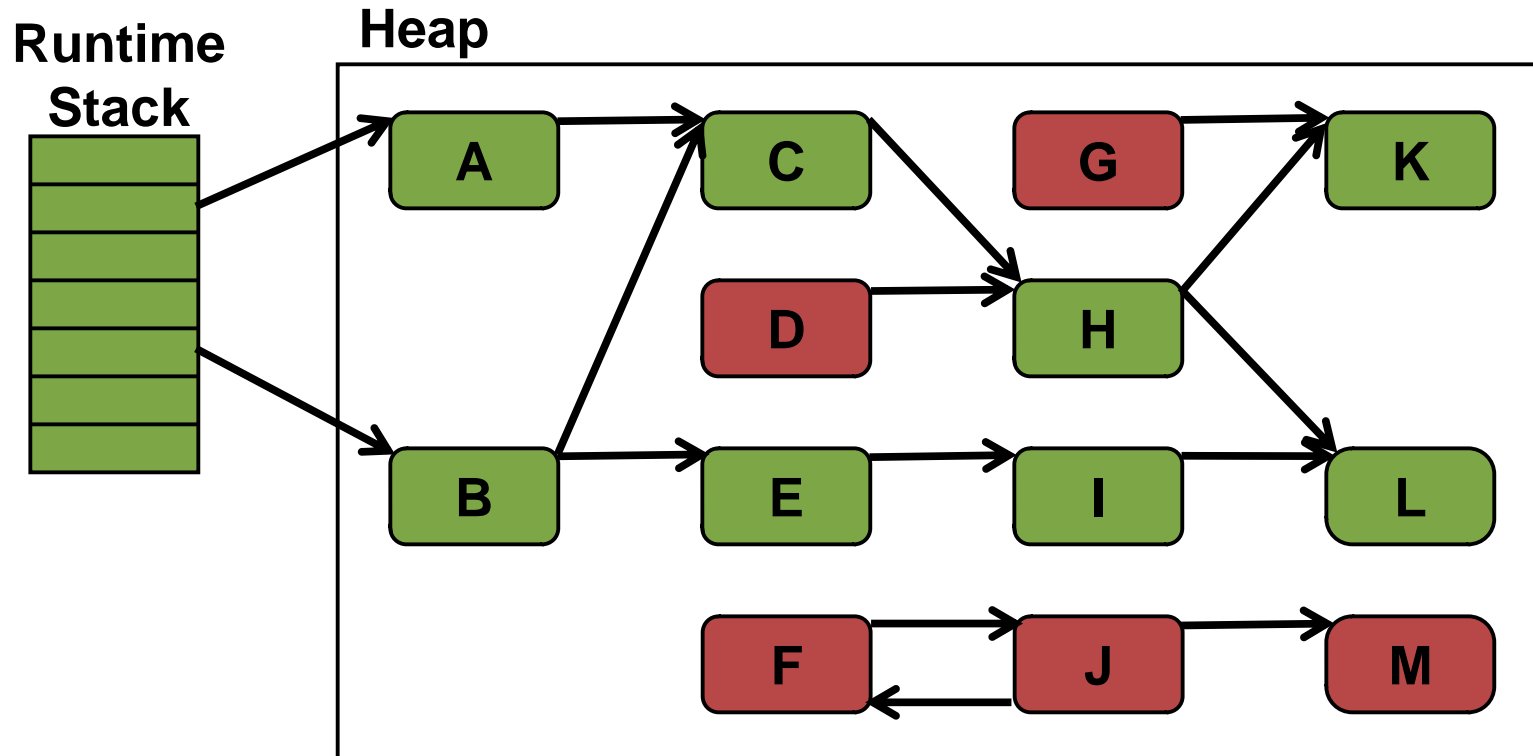
Tracing GC Example



Tracing GC Example

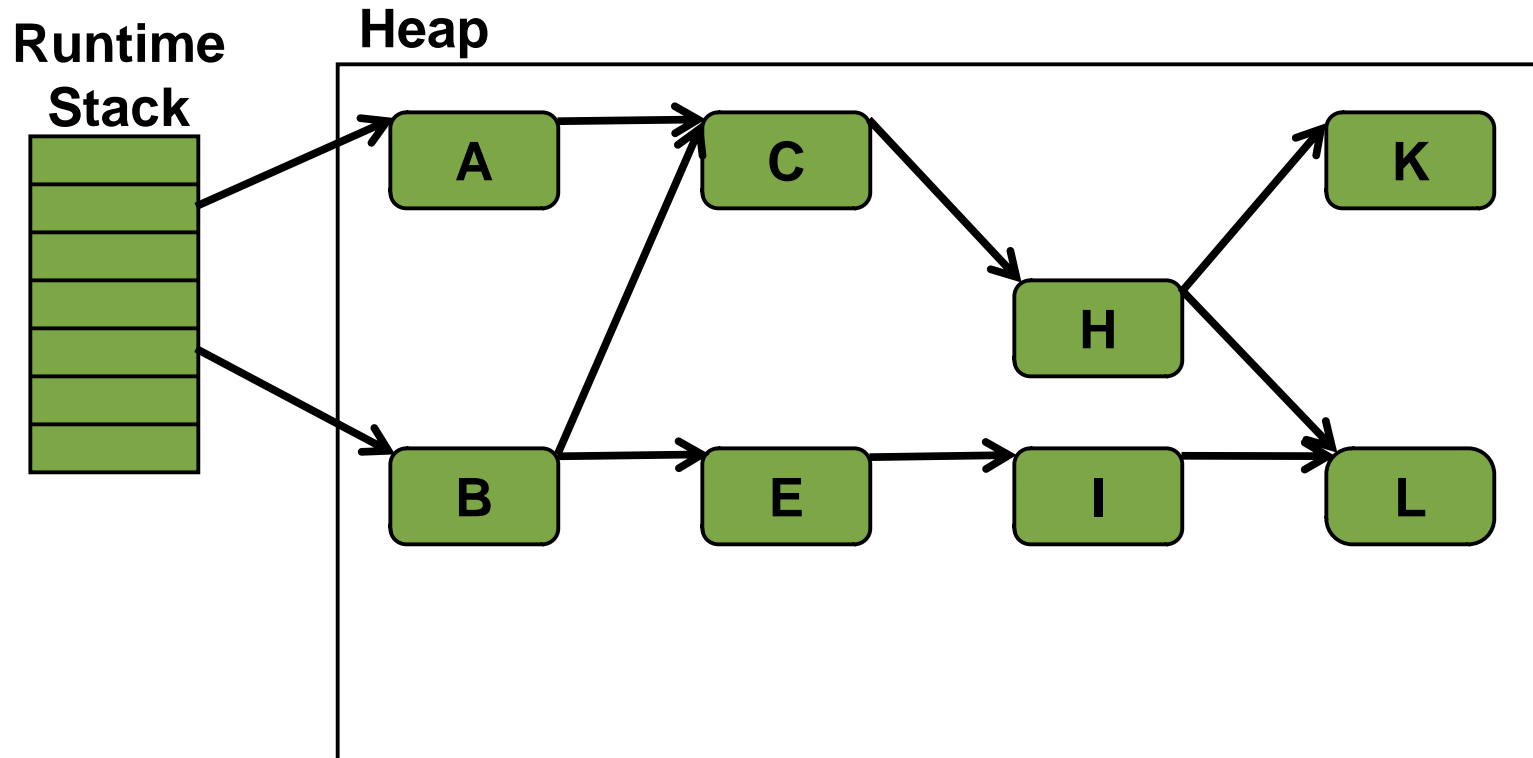


Tracing GC Example



We identified all reachable objects. We can deduce that the rest are unreachable and, therefore, dead.

Tracing GC Example

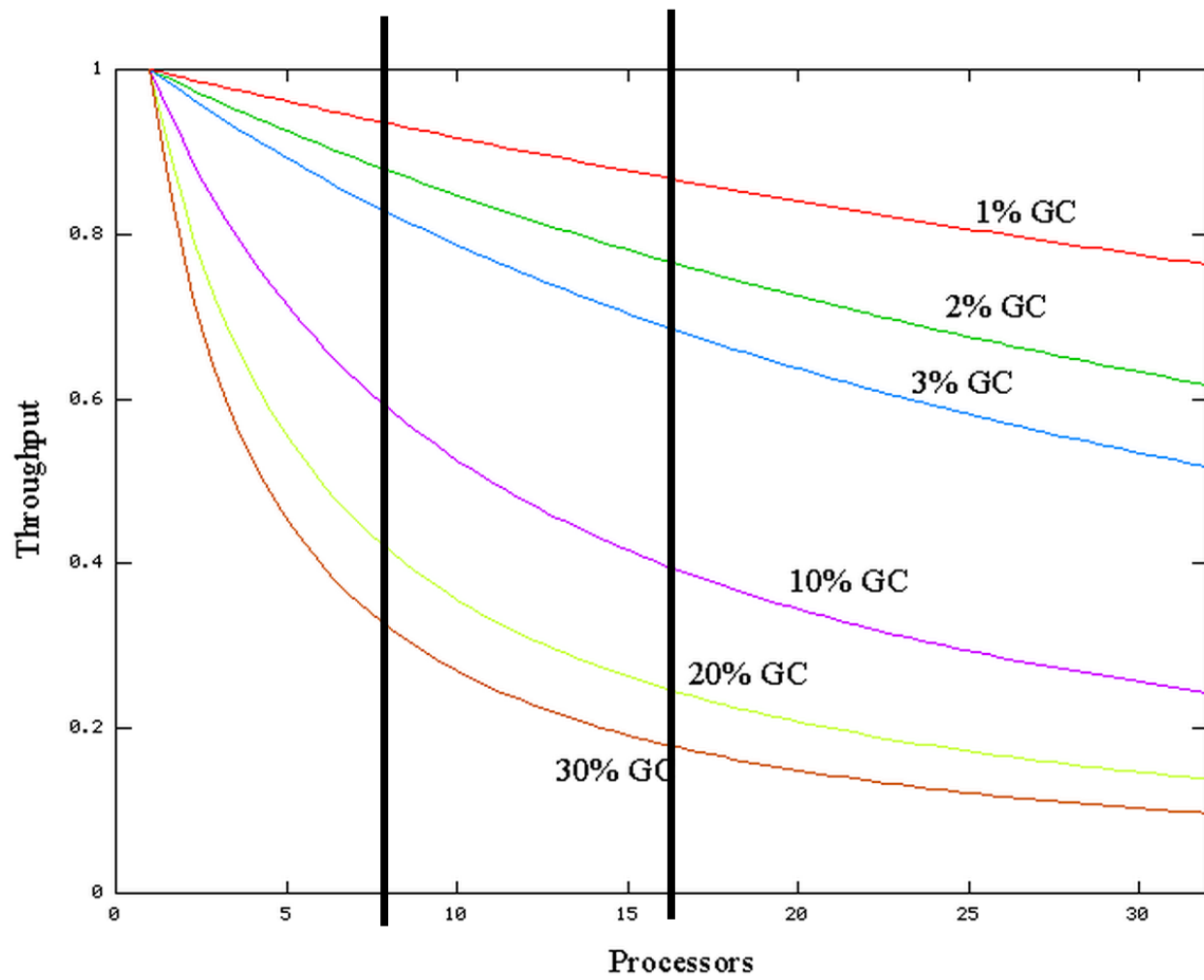


We identified all reachable objects. We can deduce that the rest are unreachable and, therefore, dead.

Garbage Collection



我真的需要花時間去
Tune GC 嗎?



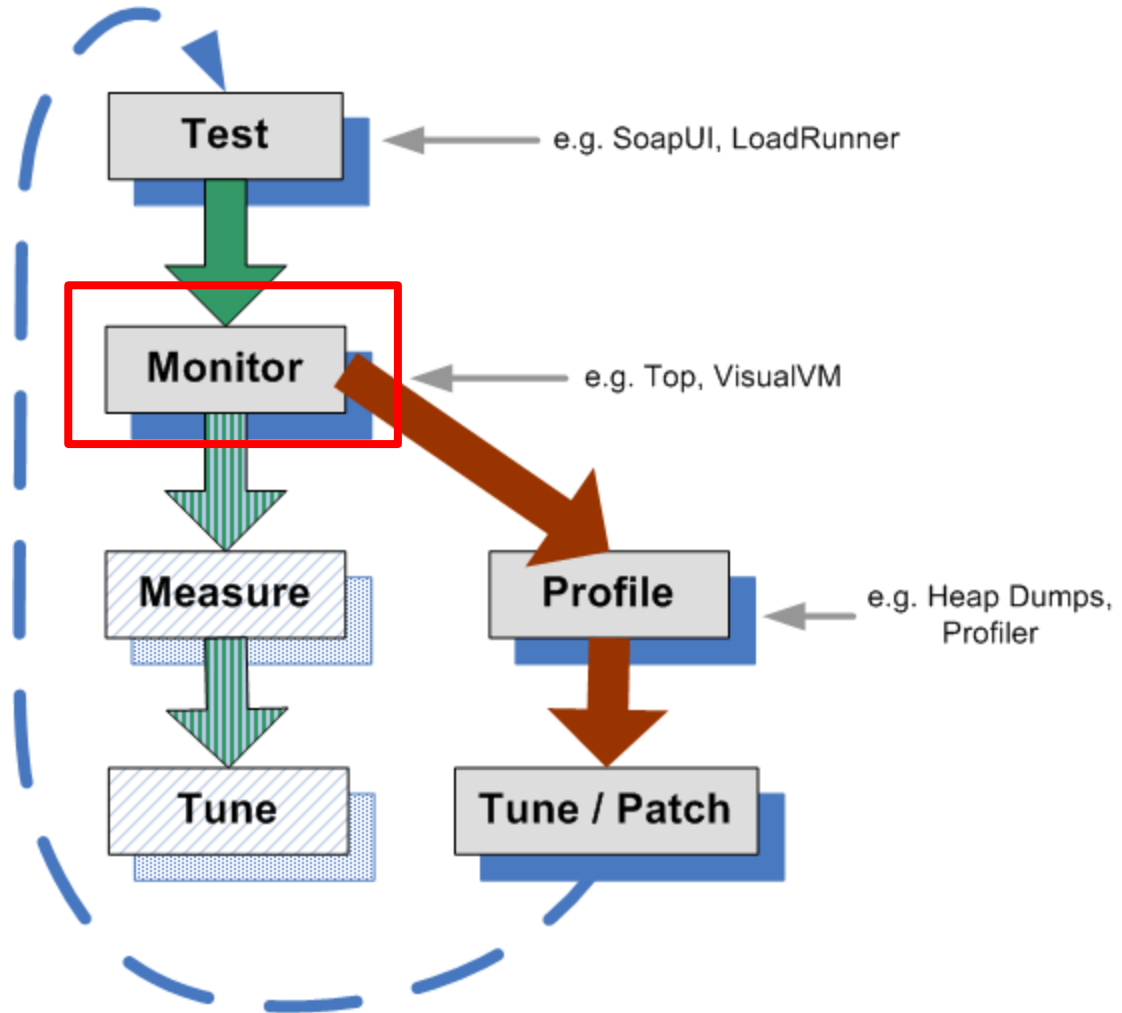


花太多時間做 GC → Performance變慢

Performance變慢 → 花太多時間做 GC
不一定!!!

JVM Performance Tuning

Performance Tuning Process



GC Logging in Production

- Don't be afraid to enable GC logging in production
 - Very helpful when diagnosing production issues
- Extremely low / non-existent overhead
 - Maybe some large files in your file system.
 - We are surprised that customers are still afraid to enable it



**If Someone doesn't enable
GC logging in production**

I shoot them!

Most Important GC Logging Parameters

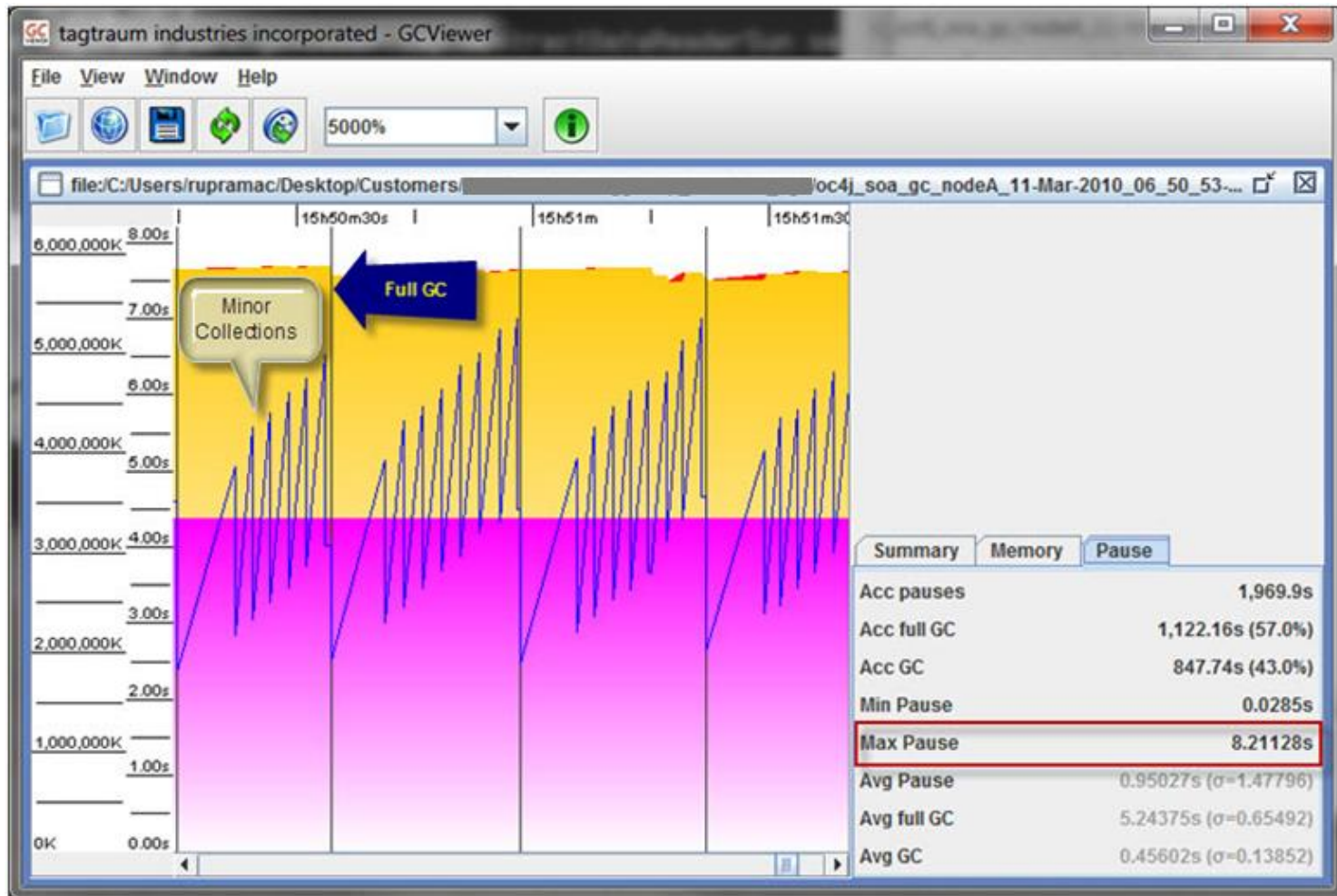
- You need at least:
 - `-XX:+PrintGCTimeStamps`
 - Add `-XX:+PrintGCDateStamps` if you must know the time
 - `-XX:+PrintGCDetails`
 - Preferred over `-verbosegc` as it's more detailed
- Also useful:
 - `-Xloggc:<file>`
 - Rotate GC Log, After Java 1.6_34 (or 1.7_2):
 - `-XX:+UseGCLogFileRotation`
 - `-XX:NumberOfGCLogFiles=5`
 - `-XX:GCLogFileSize=10M`



JVM Performance Tuning

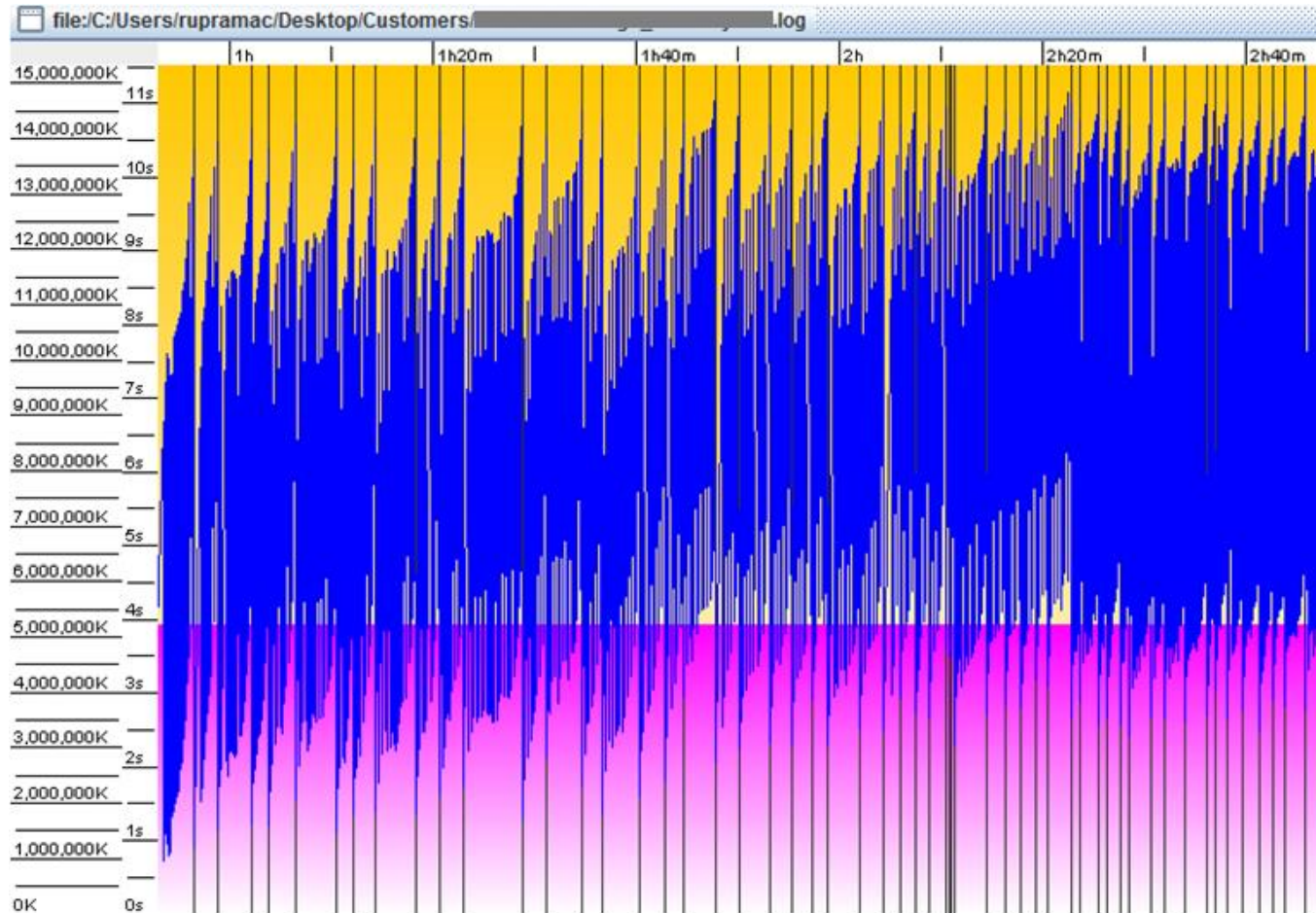
GCViewer – Offline analysis of GC logs

<https://github.com/chewiebug/GCViewer>



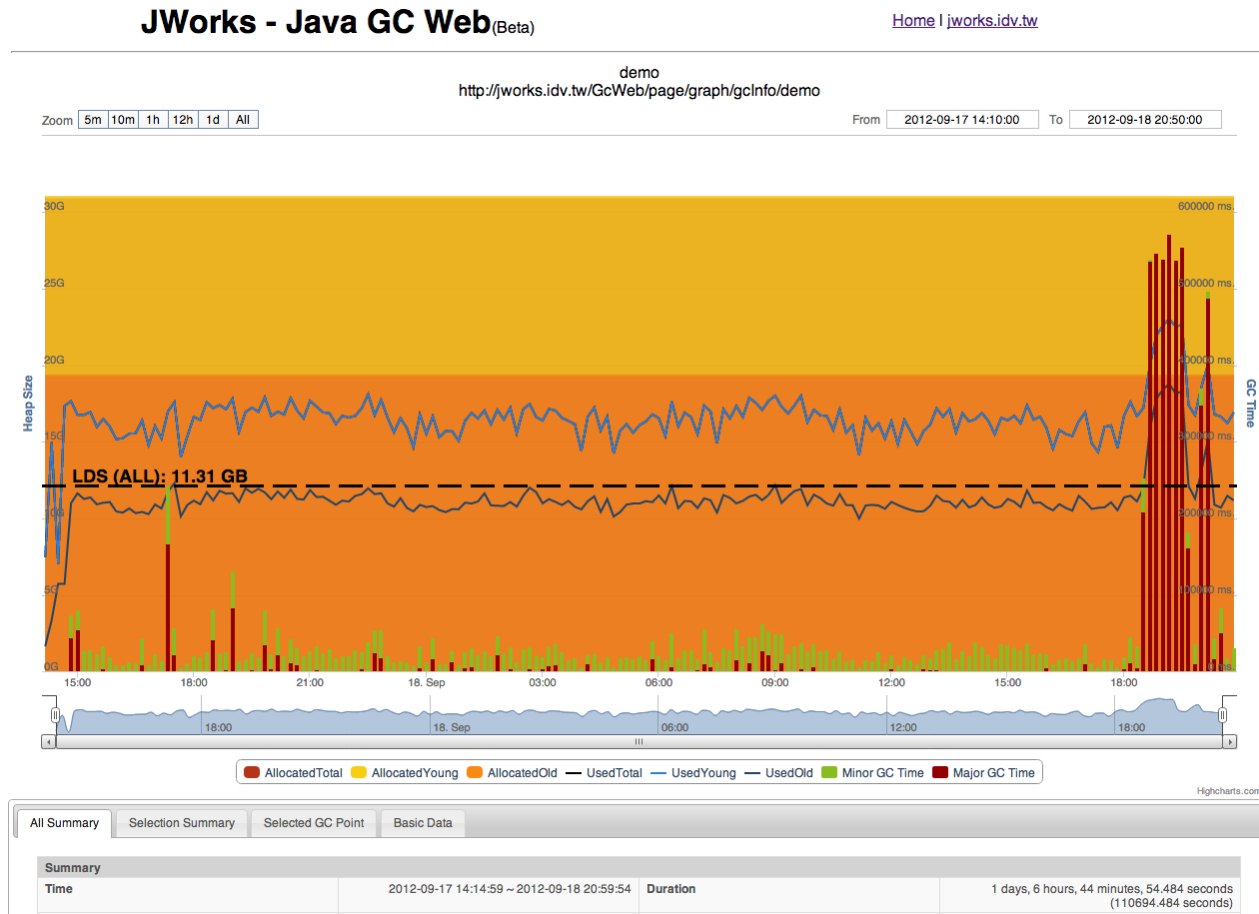
JVM Performance Tuning

GCViewer – Memory Leak Pattern



JWorks GC Web (Beta)

<http://jworks.idv.tw/GcWeb/>





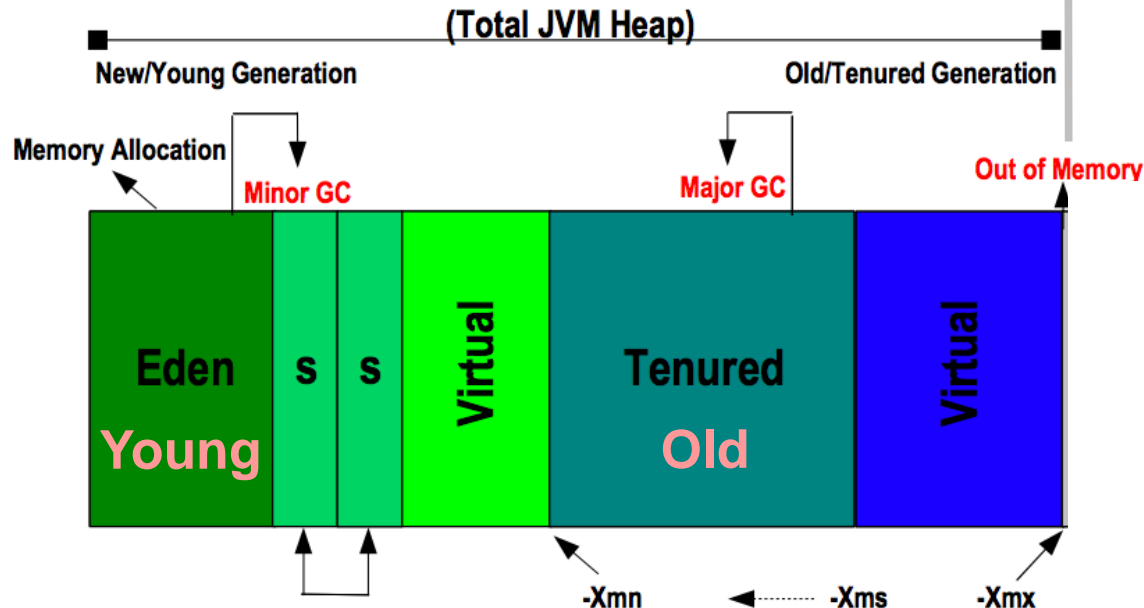
Java Heap Structure and GC

Hotspot JVM Heap Layout



Total memory, for 32bits, max = $2^{32} = 4\text{GB}$.

In windows is 2GB, so max JVM Heap is about 1.6GB ~ 1.8GB



Young Generation

- Eden where new objects get instantiated
- 2 Survivor Spaces to hold live objects during minor GC

Old Generation

- For tenured objects

Permanent Generation

- JVM meta data

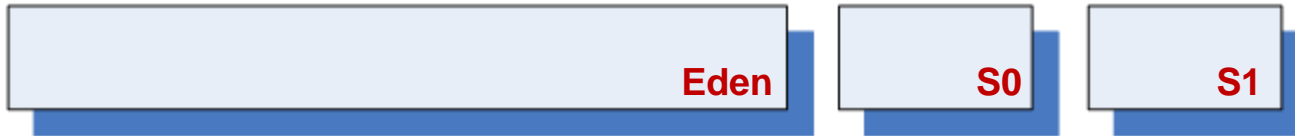
Constraints of 32 Bits JVM

- Normal Windows (2GB process memory)
 - ~1.5GB possible Java heap memory
- Windows started with /3GB (3GB process memory)
 - ~2.8GB possible Java heap memory (2.6 or 2.7 GB recommended with WLS)
- Linux, large process support enabled (3GB process memory)
 - ~2,8GB possible Java heap memory

Hotspot Internals

Generations & Object Lifecycle

Young Generation



Old Generation

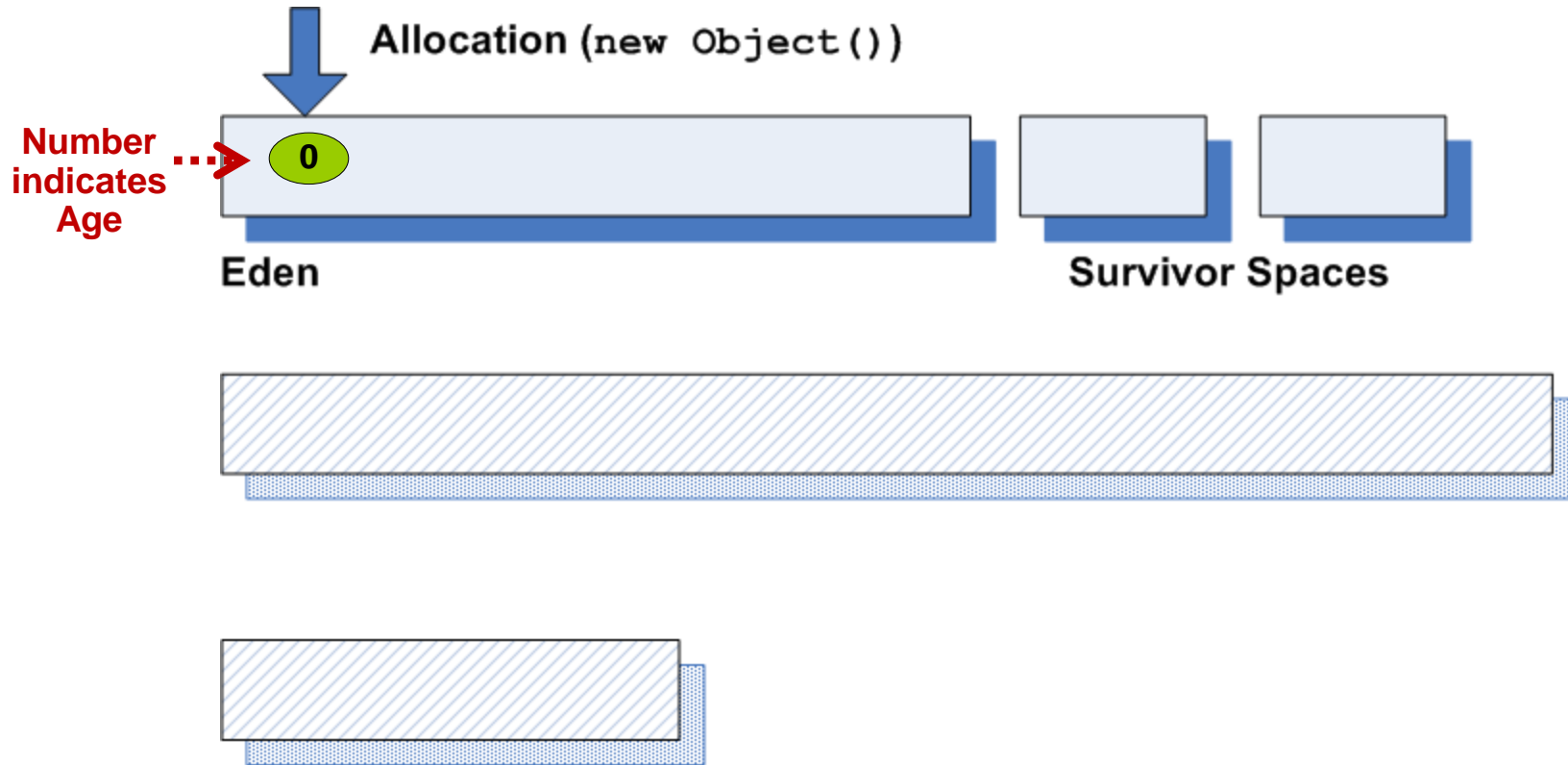


Permanent Generation



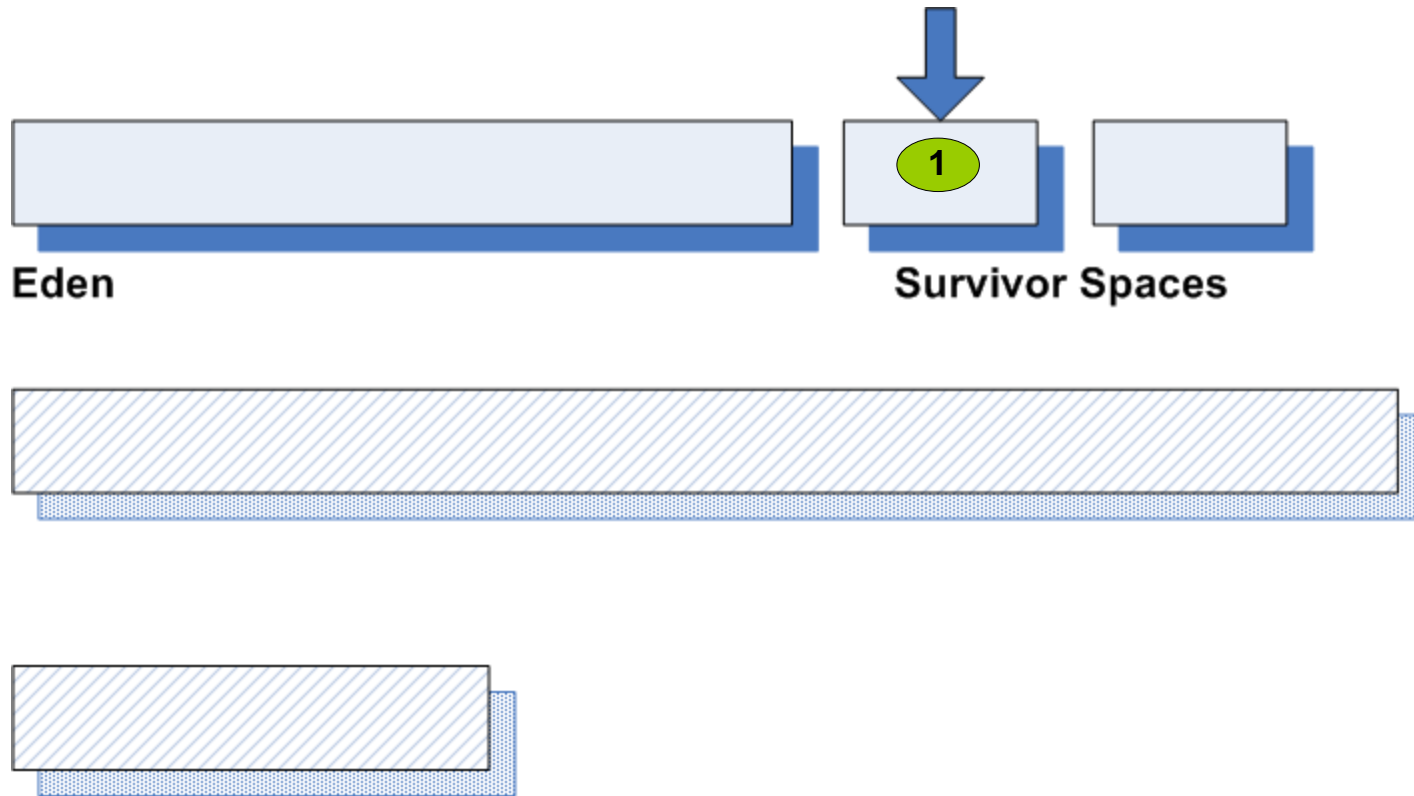
Hotspot Internals

Object lifecycle



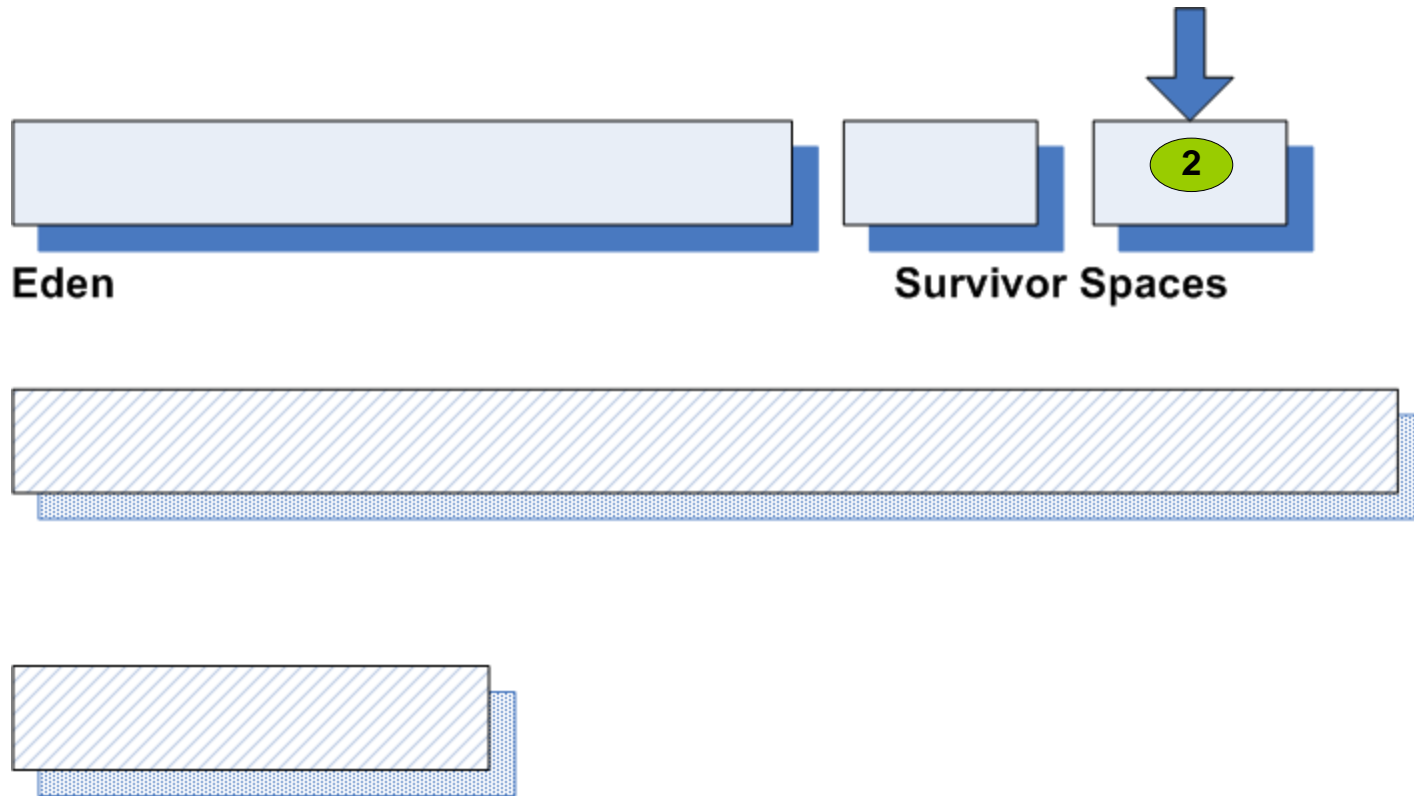
Hotspot Internals

Object lifecycle



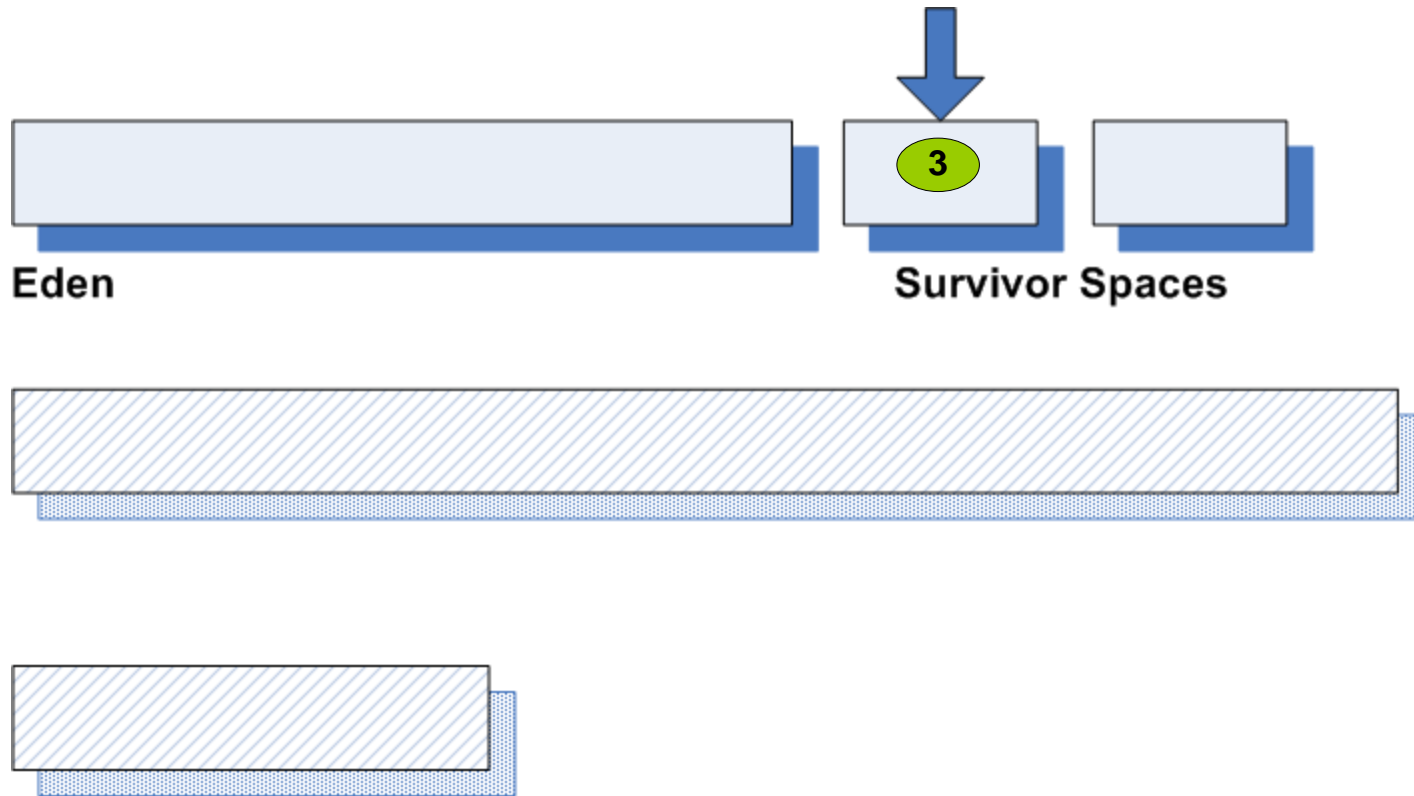
Hotspot Internals

Object lifecycle



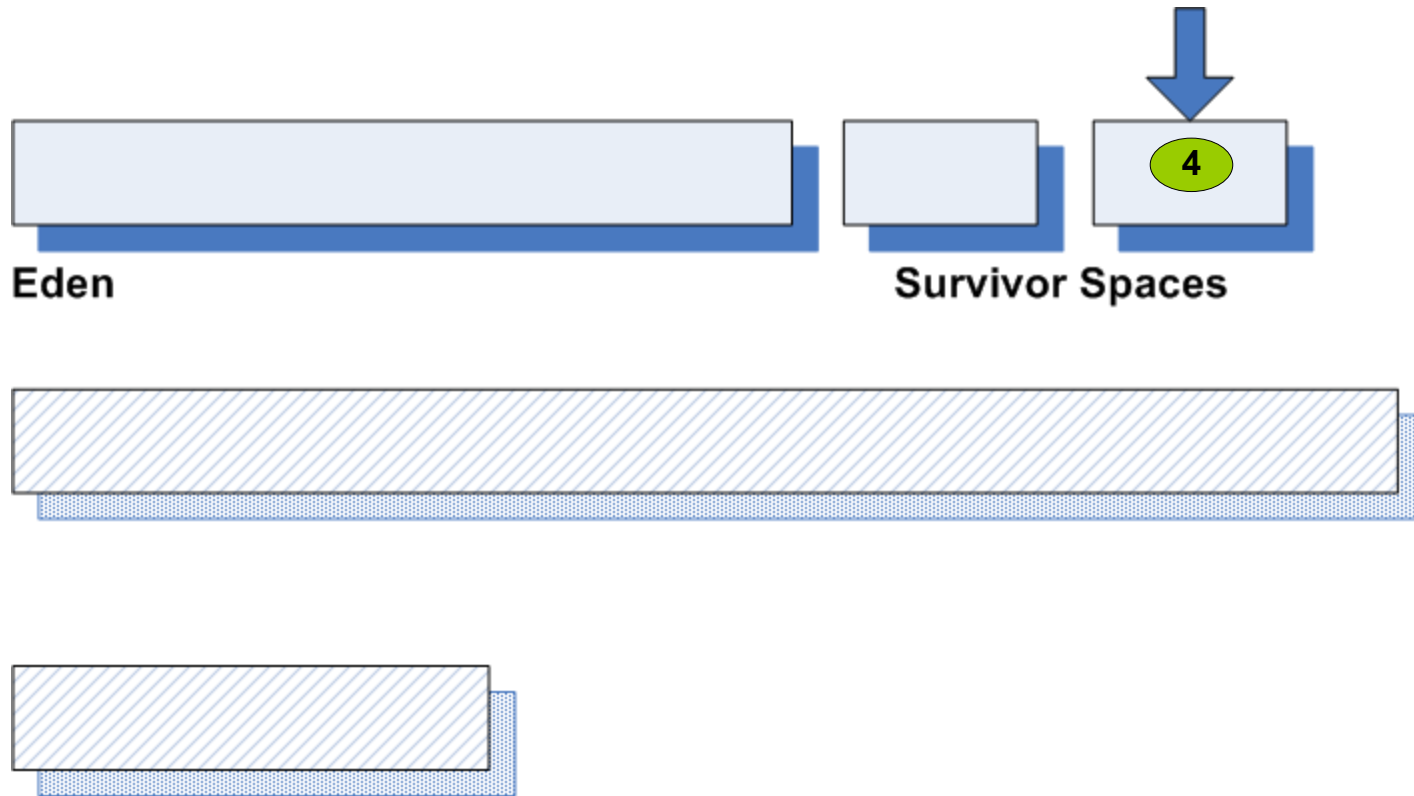
Hotspot Internals

Object lifecycle



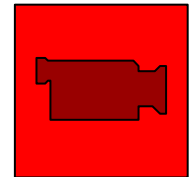
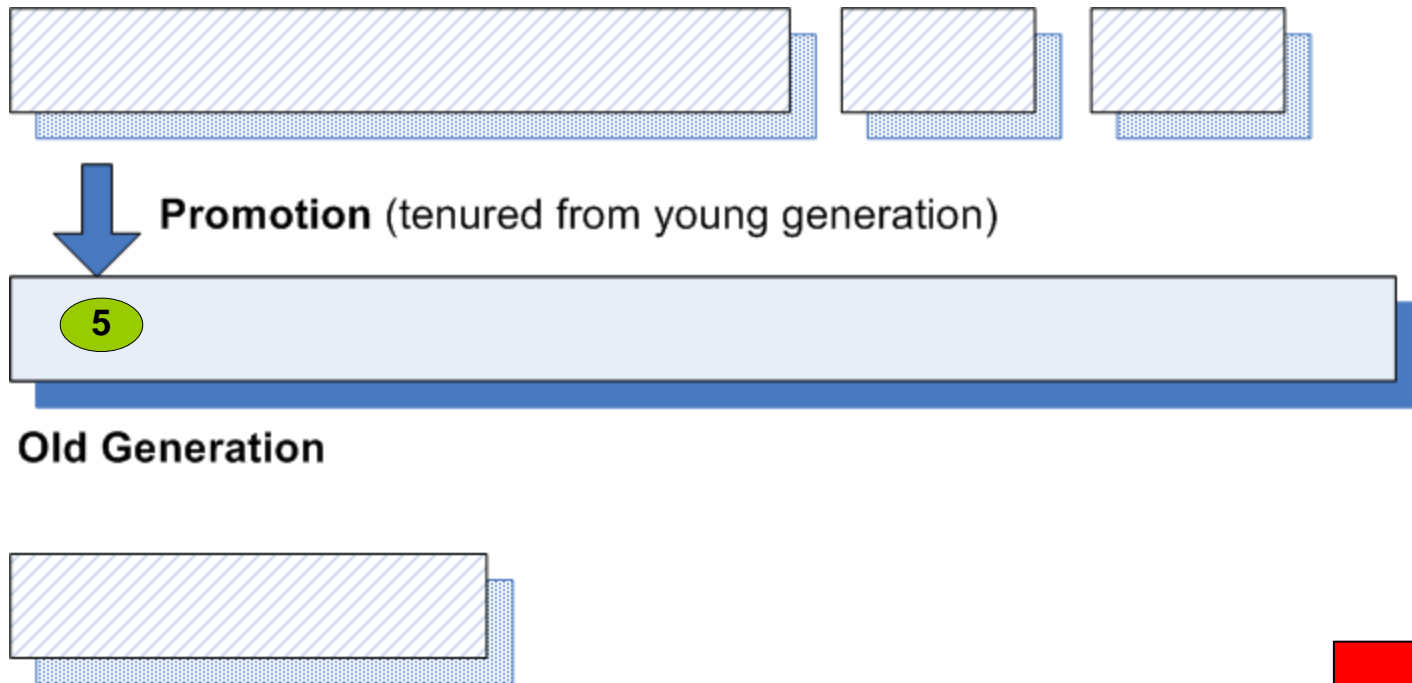
Hotspot Internals

Object lifecycle



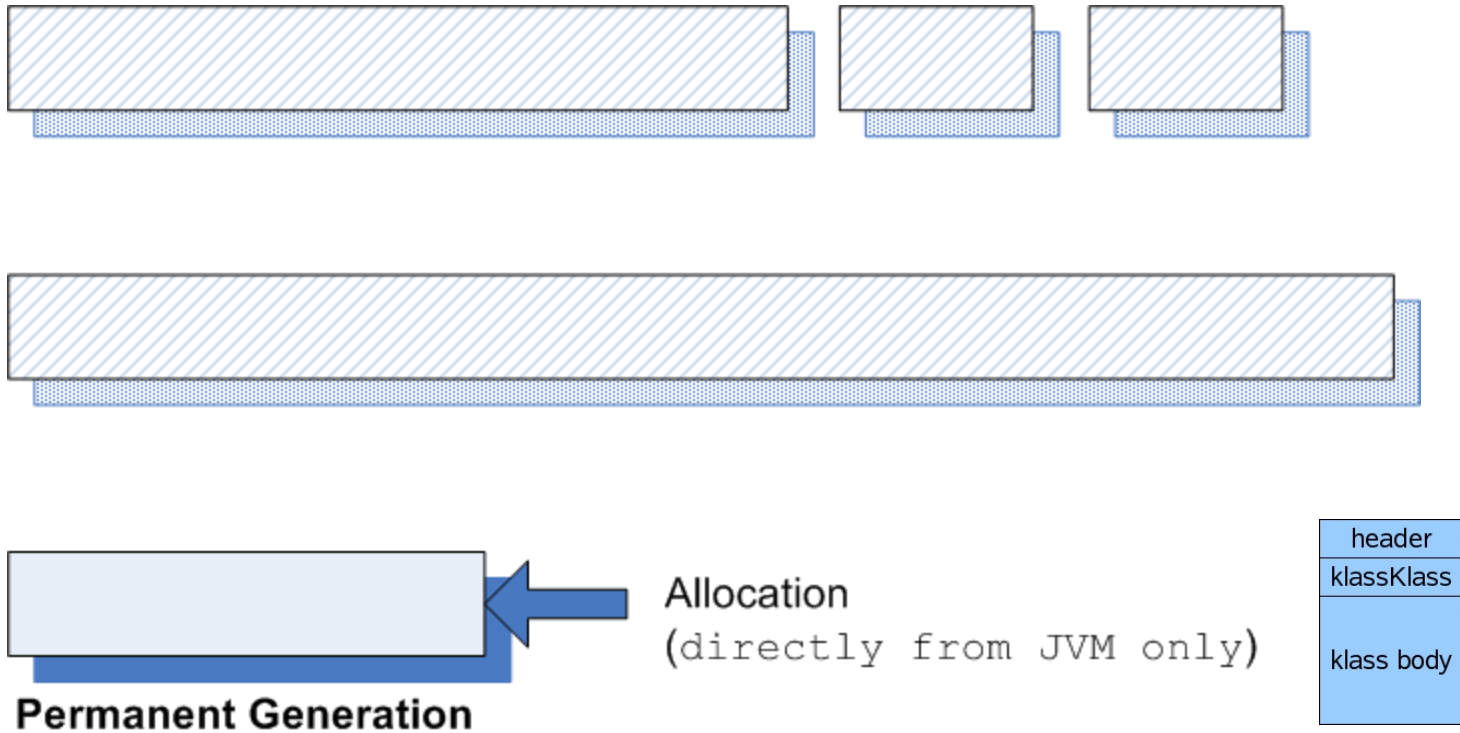
Hotspot Internals

Object lifecycle



Hotspot Internals

Object lifecycle





Hotspot GC Tuning

```
graph TD; A[Footprint (Heap Size)] --> B[Throughput]; A --> C[Latencies];
```

Footprint
(Heap Size)

Throughput

Latencies


```
graph TD; A[Footprint (Heap Size)] --> B[Throughput]; A --> C[Latencies];
```

Footprint
(Heap Size)

Throughput

Latencies

Hotspot Tuning

Sizing Heap

- Young Generation size determines
 - **Frequency** of minor GC
 - **Number of objects** reclaimed in minor GC
- Old Generation Size
 - Should hold application's **steady-state** live data size
 - Try to **minimize frequency** of major GC's
- JVM footprint should not exceed physical memory
 - Max of 80-90% RAM (leave room for OS)
- **Thumb Rule:** Try to maximize objects reclaimed in young gen. **Minimize Full GC frequency**

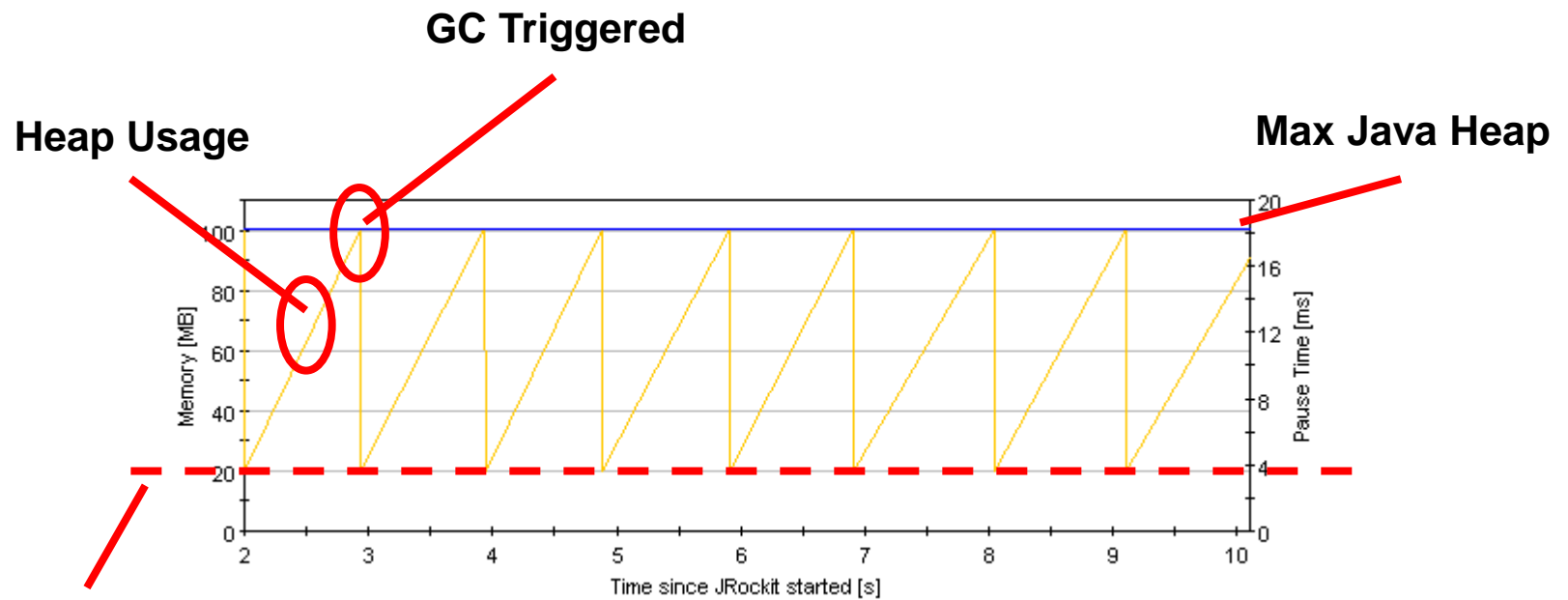
Calculate Live Data Size (LDS)

- Make sure you get full GCs during steady-state
- Induce a couple if necessary
 - JConsole / VisualVM
 - Connect to the JVM, click on the “Perform GC” button
 - jmap
 - `jmap -histo:live <pid>`
 - Can even introduce a thread that does regular `System.gc()`'s
 - Just for this data gathering purpose, OK? Remove it before deployment. :-)

Calculate Live Data Size (ii)

- From the GC log you will get
 - Approximation of the Live Data Size (LDS)
 - It is the heap occupancy after each full GC
 - Approximation of max perm gen size
 - It is the perm gen occupancy after each full GC
 - Worst-case latency scenario due to the full GCs
- GC log example:

```
170.517: [Full GC
         [PSYoungGen: 10128K->0K(163392K) ]
         [ParOldGen: 348898K->161378K(350208K) ]
                   359026K->161378K(513600K)
         [PSPermGen:    5799K->5798K(16384K) ],
                                0.3224960 secs]
```



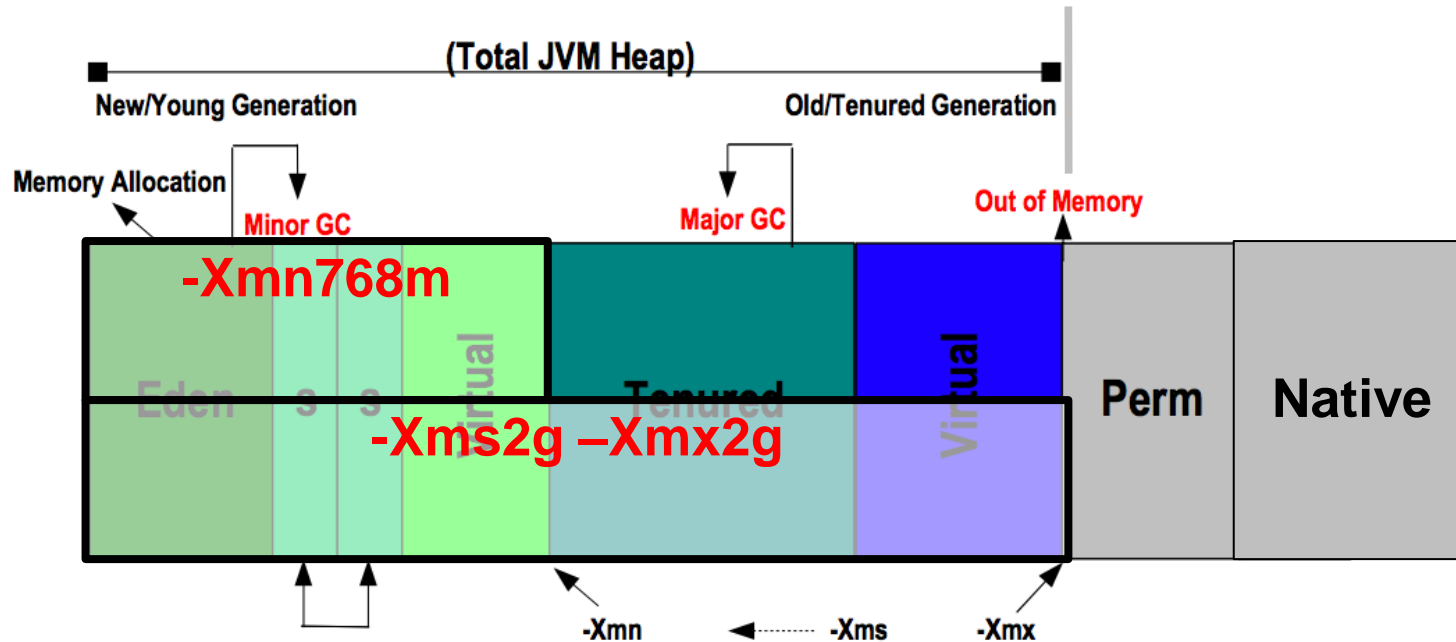
**Size of
"long lived objects"**

Initial Heap Configuration

- You can now make an informed decision on choosing a reasonable heap size
 - Rule of thumb
 - Set -Xms and -Xmx to **3x** to **4x** LDS
 - Set both -XX:PermSize and -XX:MaxPermSize to around 1.2x to 1.5x the max perm gen size
- Set the generation sizes accordingly
 - Rule of thumb
 - Young gen should be around 1x to 1.5x LDS
 - Old gen should be around 2x to 3x LDS
 - e.g., young gen should be around 1/3-1/4 of the heap size
- e.g., For LDS of 512m : -Xmn768m -Xms2g -Xmx2g

Hotspot JVM Heap Layout

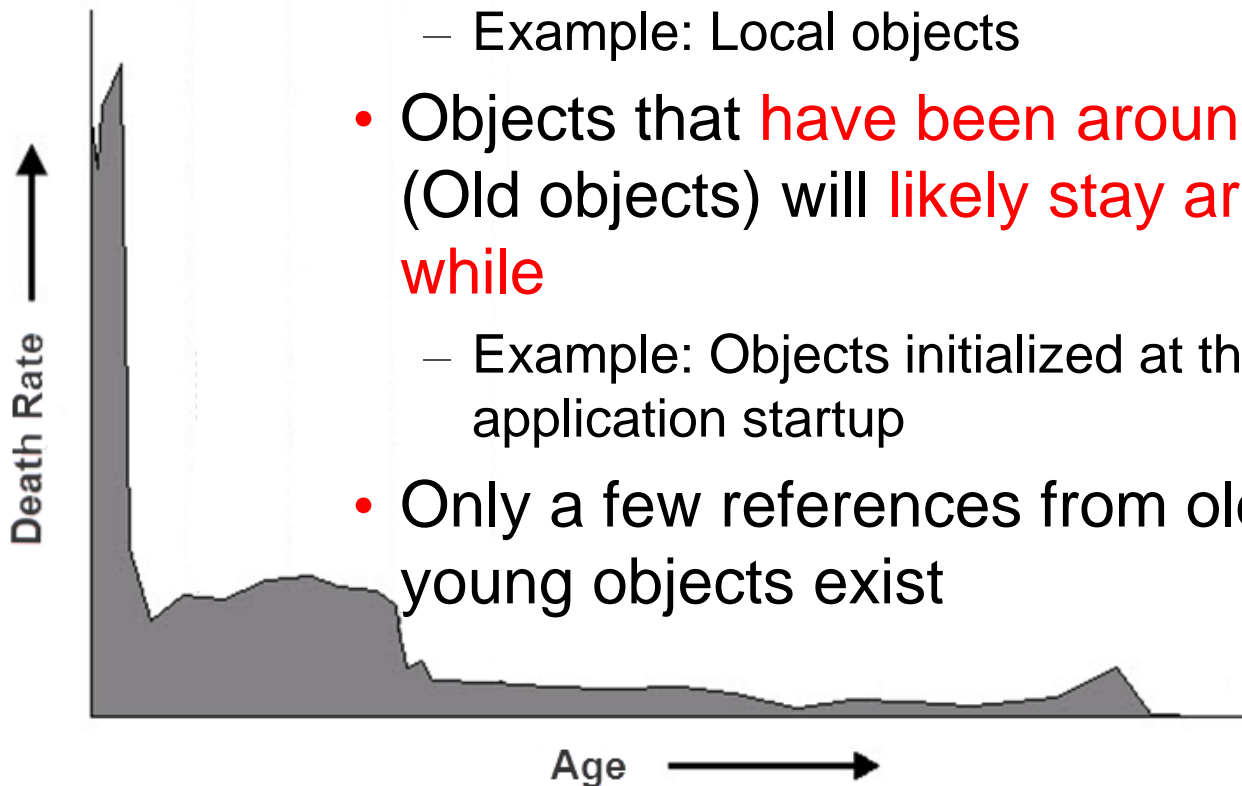
For LDS of 512m : -Xmn768m -Xms2g -Xmx2g



GC Fundamentals

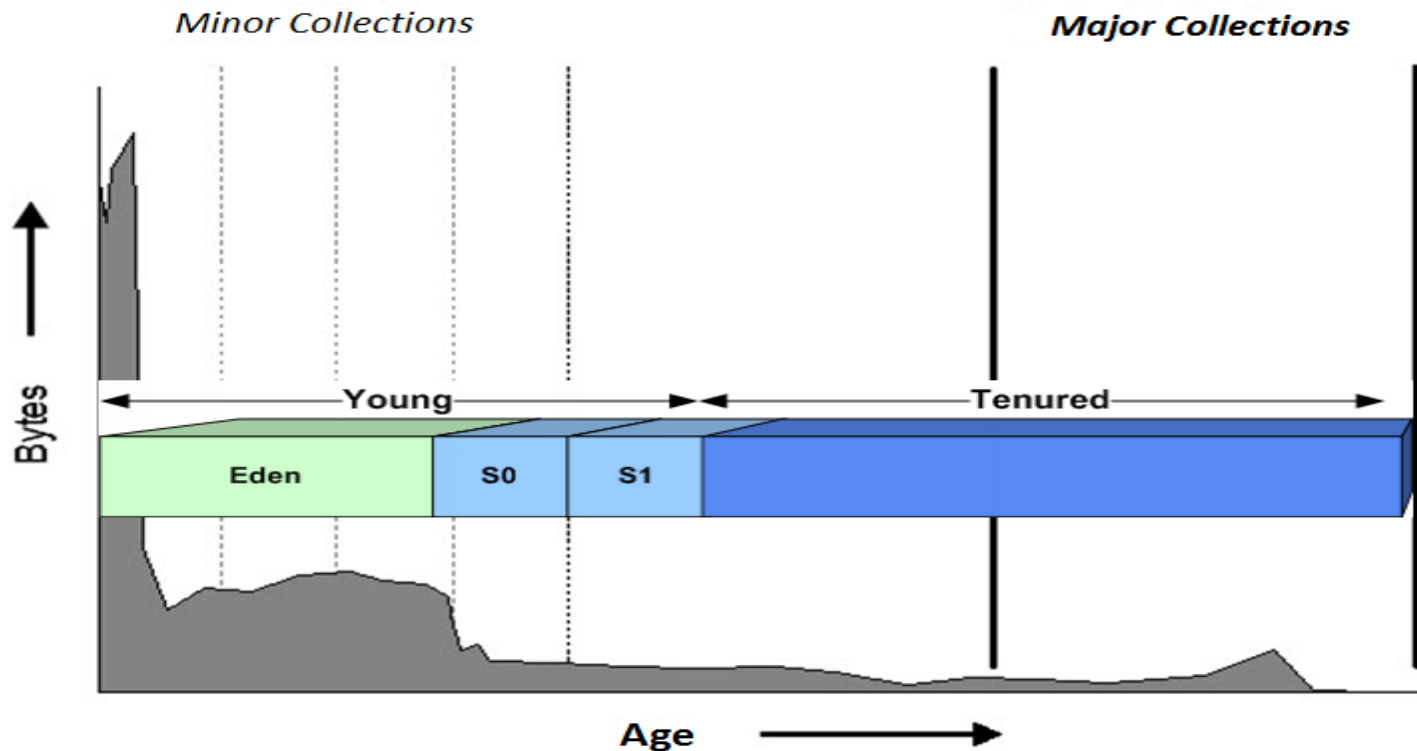
Garbage Distribution – Objects die young

- Typical object (Young object) is most likely to die shortly after it was created
 - Example: Local objects
- Objects that have been around for a while (Old objects) will likely stay around for a while
 - Example: Objects initialized at the time of application startup
- Only a few references from old objects to young objects exist



GC Fundamentals

Garbage Distribution - Generational Collection



- Buckets ideally correspond to distribution curve
- Collect each bucket with most efficient algorithm
- Size buckets for best GC performance

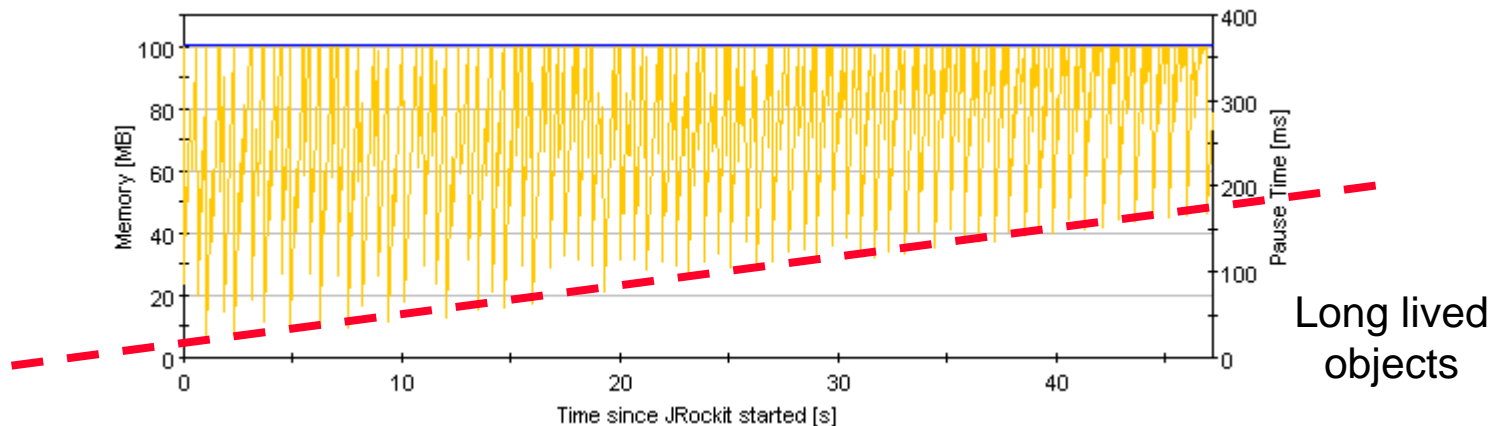
JVM Heap Tuning

- Young generation sizing is second most important
 - -XX:NewSize=<n> -XX:MaxNewSize=<n>
 - Fixed Size young generation and Fixed Size Heap can provide better performance (less Full GC)
 - Set NewSize = MaxNewSize
 - Or, -Xmn<n> is a convenient option
 - More scalable flag is -XX:NewRatio=<r>

http://blogs.sun.com/jonthecollector/entry/the_second_most_important_gc

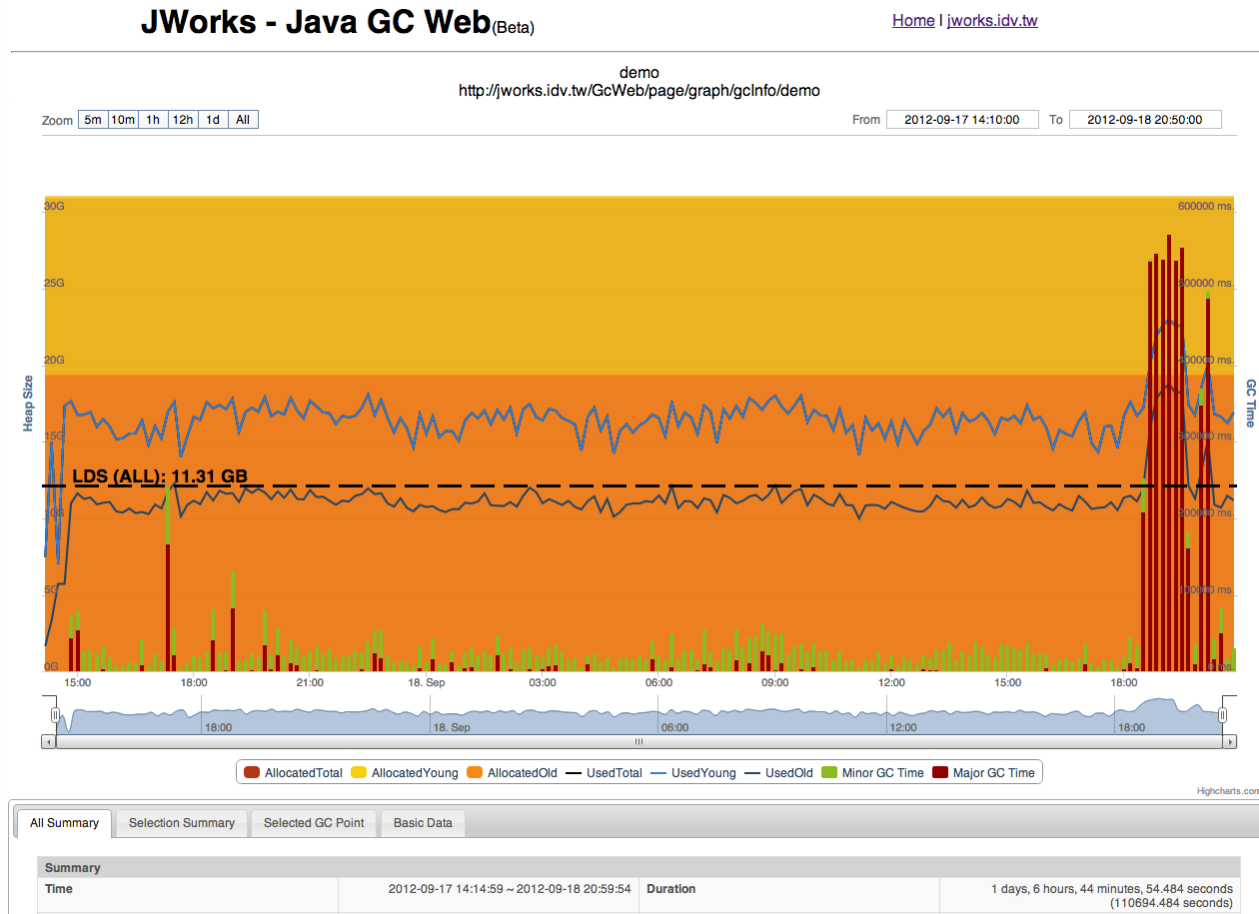
Memory Leak

- How do you know there is memory leak?
 - OOM (OutOfMemory) after a while (a few hours/a few days)
 - Memory used for long lived objects increases over time
 - Usually not a JVM problem, but a problem in application



JWorks GC Web (Beta)

<http://jworks.idv.tw/GcWeb/>



Types of Object Reference

- Strong references
 - Normal
- SoftReference
 - Used for cache
 - GC'ed any time after there are no strong references to the referent, but is typically retained until memory is low.
 - Try to use `-XX:SoftRefLRUPolicyMSPerMB=0`
- WeakReference
 - Used for cache
 - GC'ed any time after there are no strong or soft references to the referent.
- PhantomReference
 - Used for housekeeping (without using `finalize`, don't use `finalize`)

Footprint
(Heap Size)



Throughput



Latencies

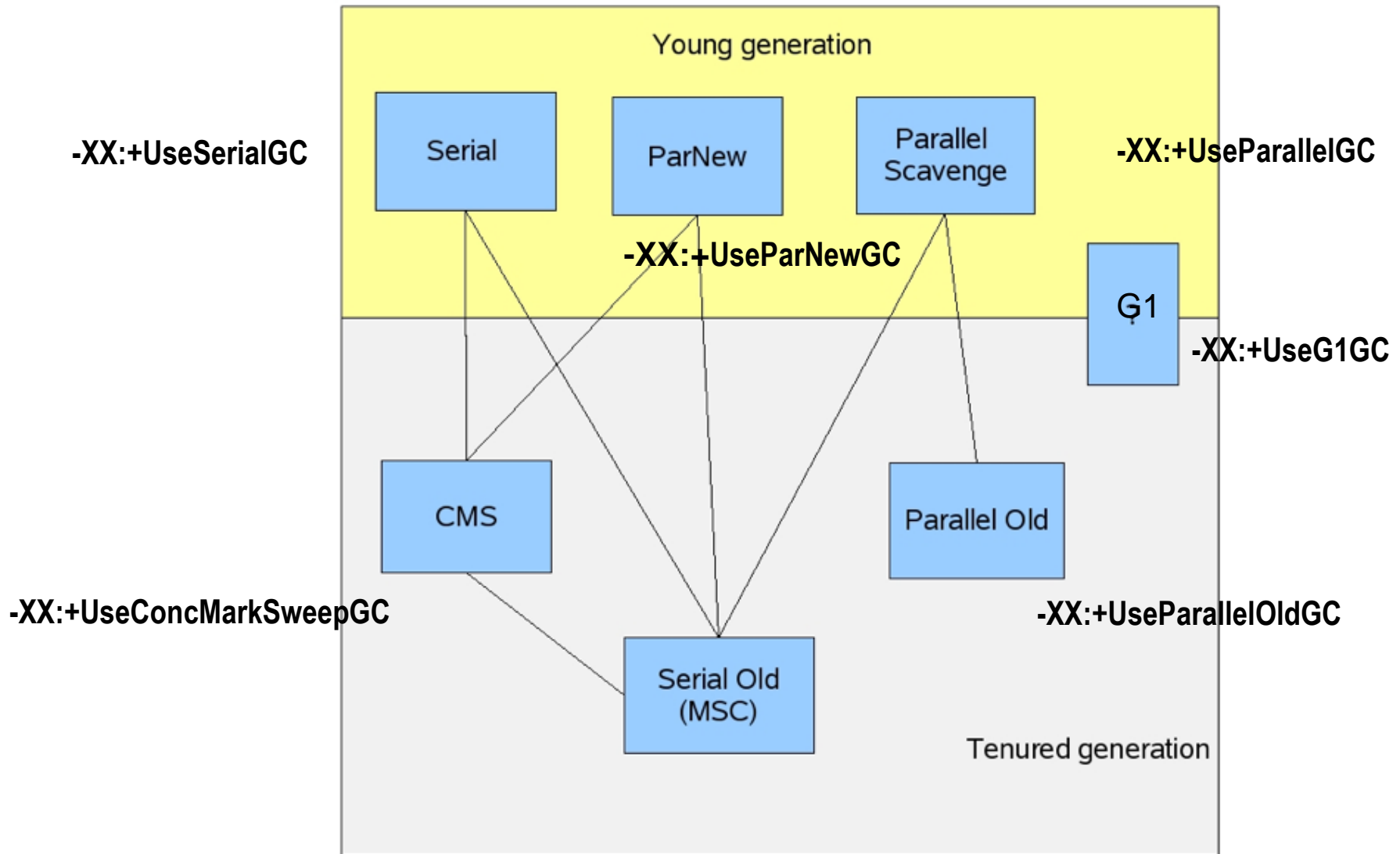
Throughput vs. Latency

- For most applications, GC overhead is small
 - 2% – 5%
- **Throughput GCs**
 - Move most work **to** GC pauses
 - Application threads do as little as possible
 - **Least overall GC overhead**
- **Low Latency (Pause) GCs**
 - Move work **out** of GC pauses
 - Application threads do more work
 - Bookkeeping for GC more expensive
 - **More overall GC overhead**

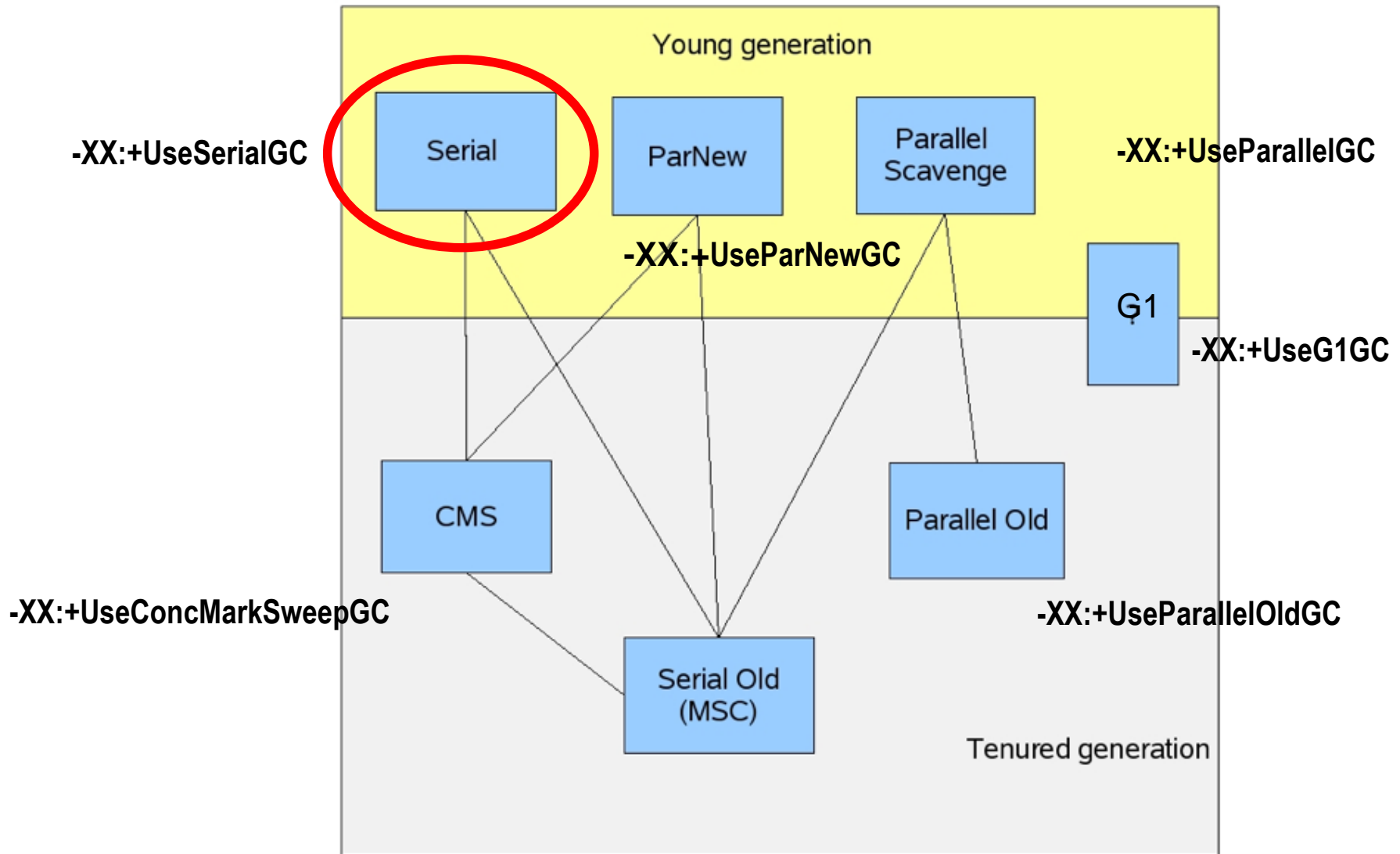
Application Requirement

- Different applications have different requirements
 - Higher Throughput:
 - Web application: pauses during garbage collection may be tolerable, or simply obscured by network latencies
 - Batch processing
 - Lower Latencies:
 - Interactive graphics application

HotSpot Garbage Collectors in Java SE 6



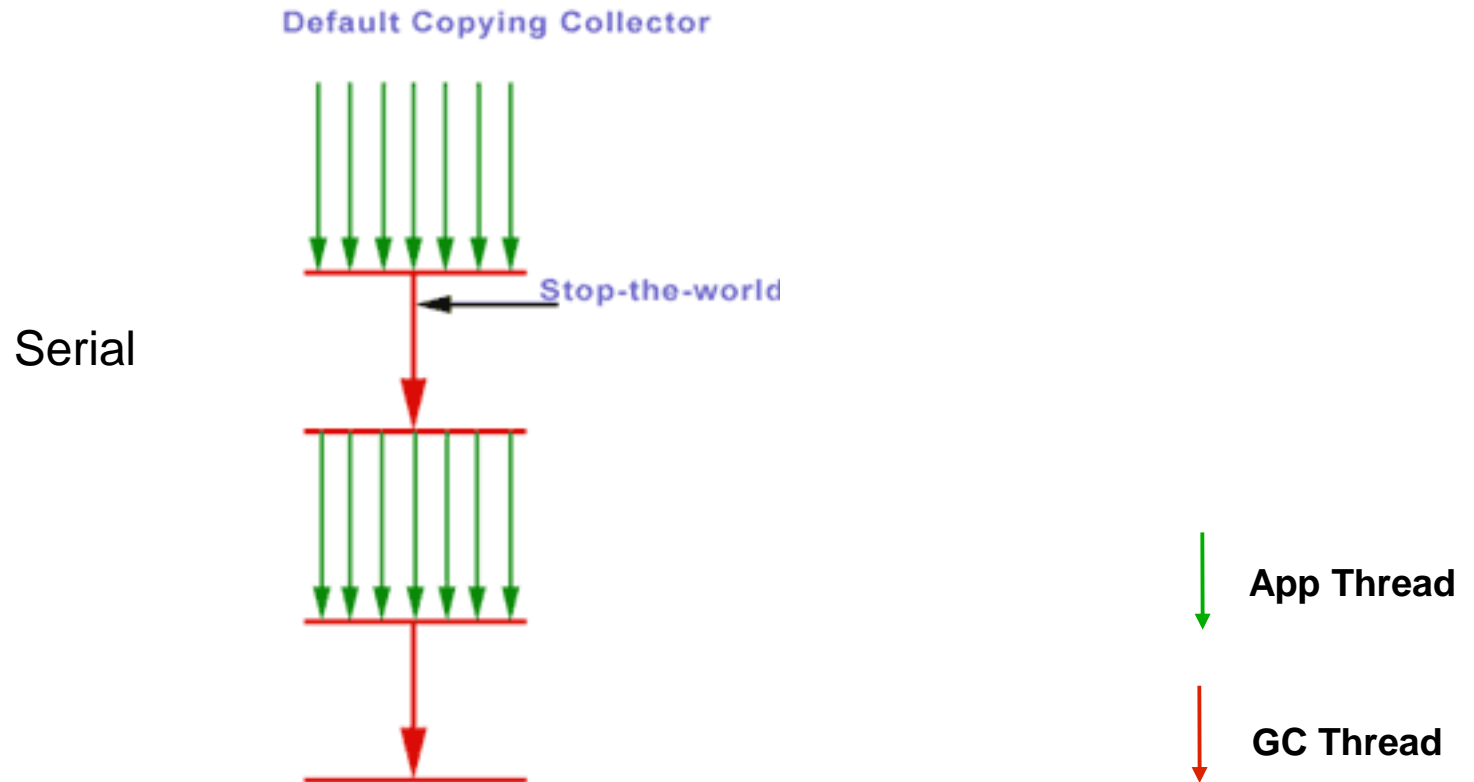
HotSpot Garbage Collectors in Java SE 6



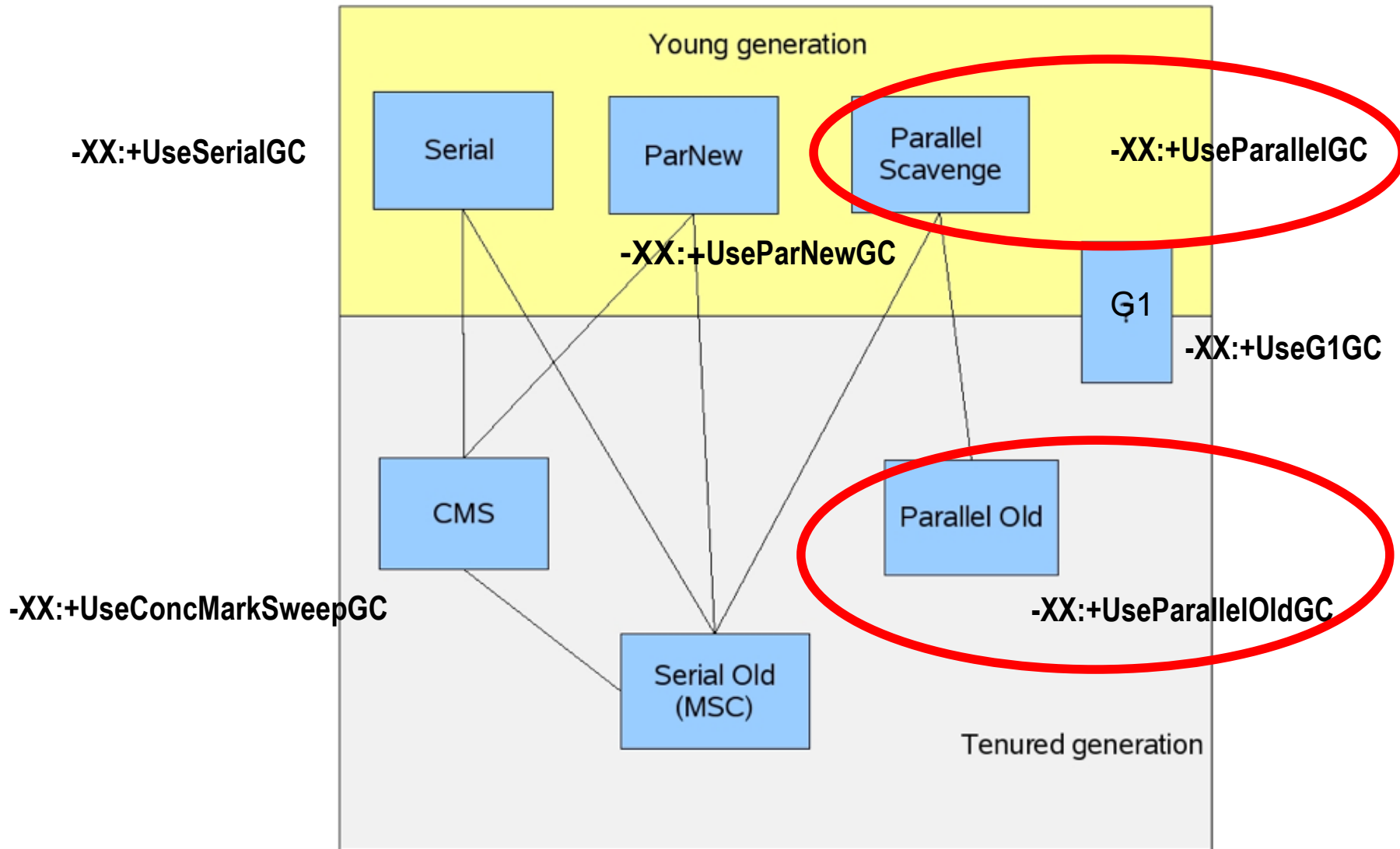
Selecting Garbage Collectors

- Serial Collector (**default for -client**) (-XX:+UseSerialGC)
 - Works well in **single processor** configuration
 - Suitable for applications with **small data sets**
 - Co-locating multiple instances of WLS(JVM's)

Serial Collector



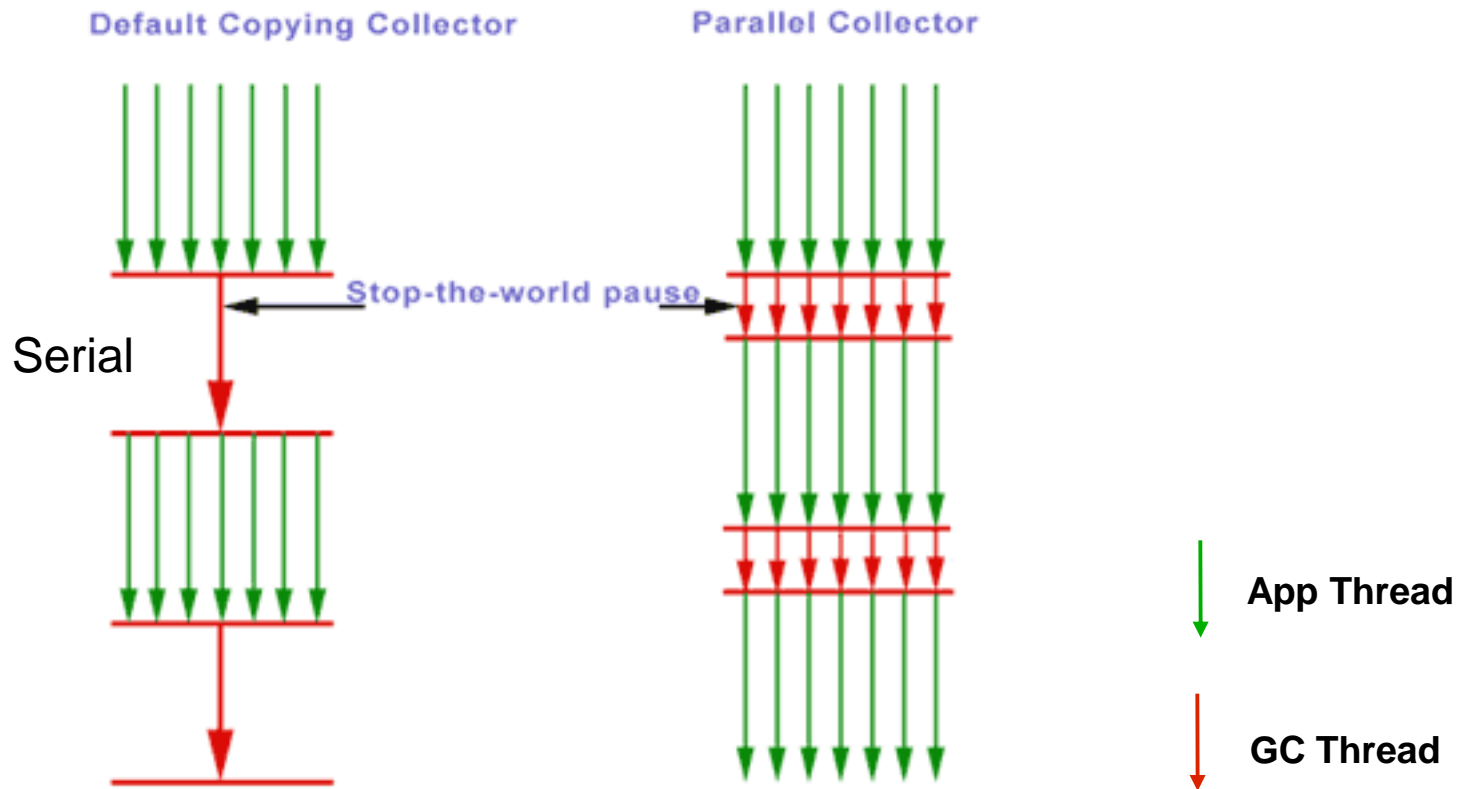
HotSpot Garbage Collectors in Java SE 6



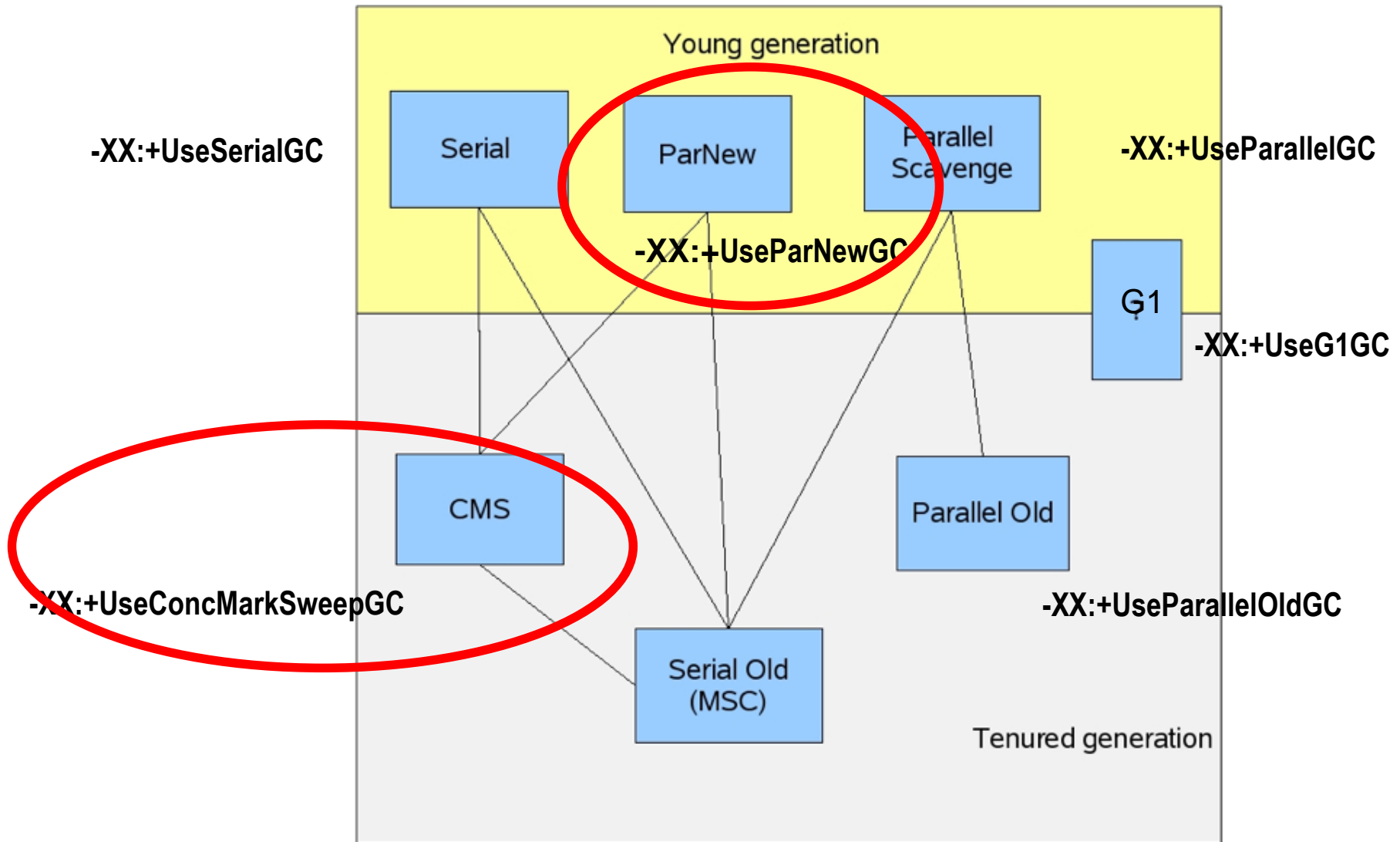
Selecting Garbage Collectors

- **Parallel Scavenge Collector (Throughput, -XX:+UseParallelGC)**
 - Perform minor collections in parallel to improve performance on multi-core and multi-threaded CPU's
 - Designed for applications with medium to large data sets
 - Control GC Threads using -XX:ParallelGCThreads=<n>
 - By default, <n> is set to total number of “CPU’s”
 - In JDK 5 Update 6, parallel compaction for old generation was added (-XX:+UseParallelOldGC)
 - Minimize garbage collection impact on throughput
 - **Parallel compaction is now default in JDK 6**
 - -XX:+UseParallelOldGC

Serial vs Parallel Collector



HotSpot Garbage Collectors in Java SE 6

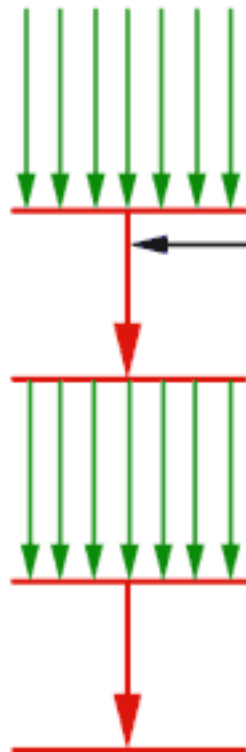


Selecting Garbage Collectors

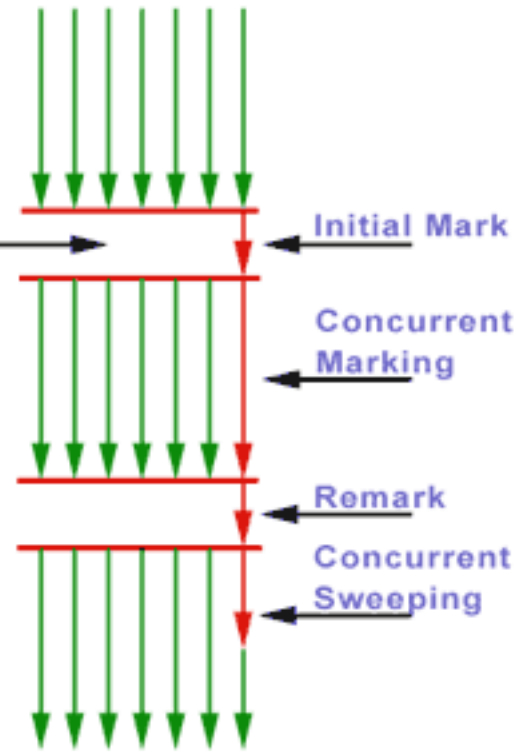
- Low Latencies (Pause) Collector
(-XX:+UseConcMarkSweepGC)
 - Execute the application concurrently during Full GC
 - Constant response time on multi-CPU servers
 - To improve throughput, use the following for young gen
 - -XX:+UseParNewGC (and -XX:ParallelGCThreads=<n>)

CMS Collector

Default Mark-compact collector

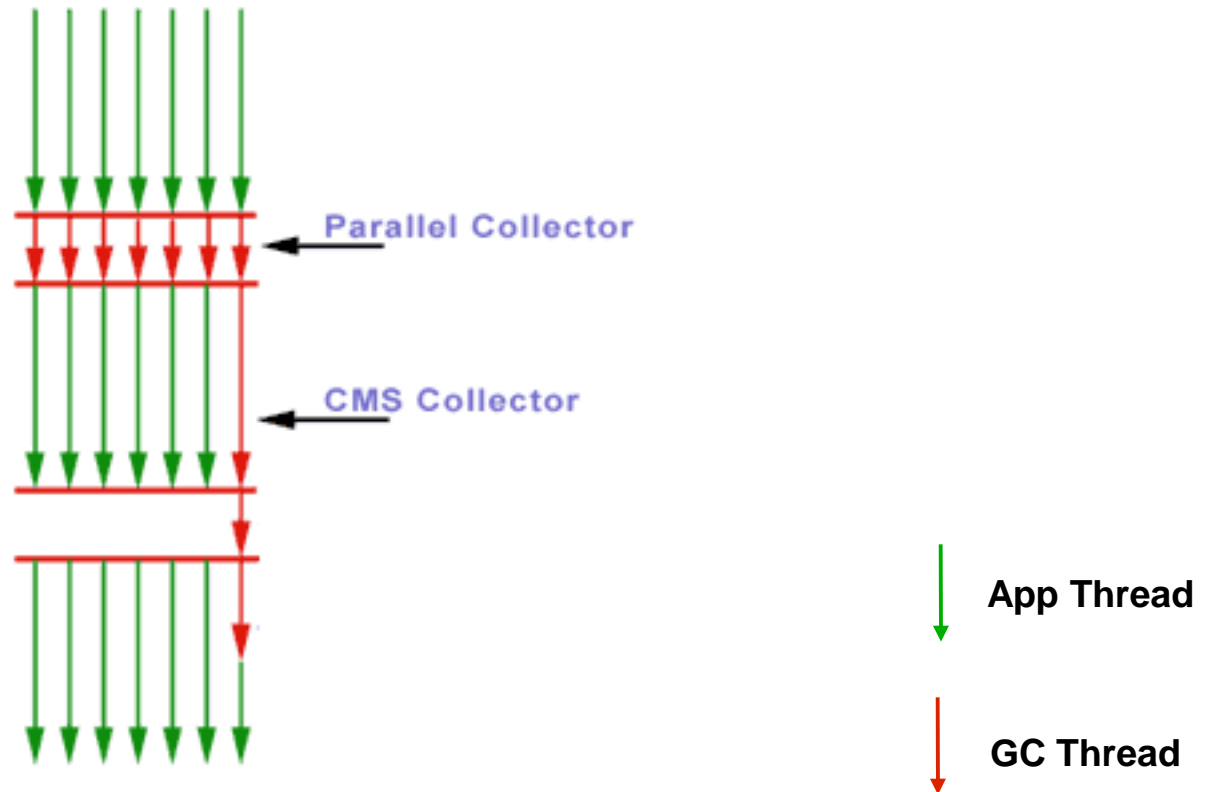


Concurrent Mark-Sweep collector



App Thread
GC Thread

CMS Collector with ParNewGC



Hotspot JDK Heap Option Summary

Generation	Low Pause Collectors		Throughput Collectors		Heap Sizes
	1 CPU	2+ CPUs	1 CPU	2+ CPUs	
Young	Serial Copying Collector (default)	Parallel Copying Collector -XX:+UseParNewGC	Serial Copying Collector (default)	Parallel Scavenge Collector -XX:+UseParallelGC	-XX:NewSize -XX:MaxNewSize -XX:SurvivorRatio
Old	Mark-Compact Collector (default)	Concurrent Collector - XX:+UseConcMarkSweepGC	Mark-Compact Collector (default)	Parallel Old -XX:+UseParallelOldGC	-Xms -Xmx
Permanent	Can be turned off with -Xnocompact (use with care)				-XX:PermSize -XX:MaxPermSize

New G1 Collector as of WLS10.3.1/JDK 6 Update 14

G1 Collector

- The Garbage-First Collector
 - Since JDK 6u14, officially supported as of JDK 7u4
 - -XX:+UseG1GC
 - Future CMS Replacement
 - Server “Style” low latency collector
 - Parallel
 - Concurrent
 - Generational
 - Good Throughput
 - Compacting
 - Improved ease-of-use
 - Predictable (Soft Real-Time)

G1 Collector

- The Garbage-First Collector

- Since JDK 6u14, officially supported as of JDK 7u4
- -XX:+UseG1GC
- Future CMS Replacement
- Server “Style” low latency collector
 - Parallel
 - Concurrent
- Generational
- Good Throughput
- **Compacting**
- **Improved ease-of-use**
- **Predictable (Soft Real-Time)**

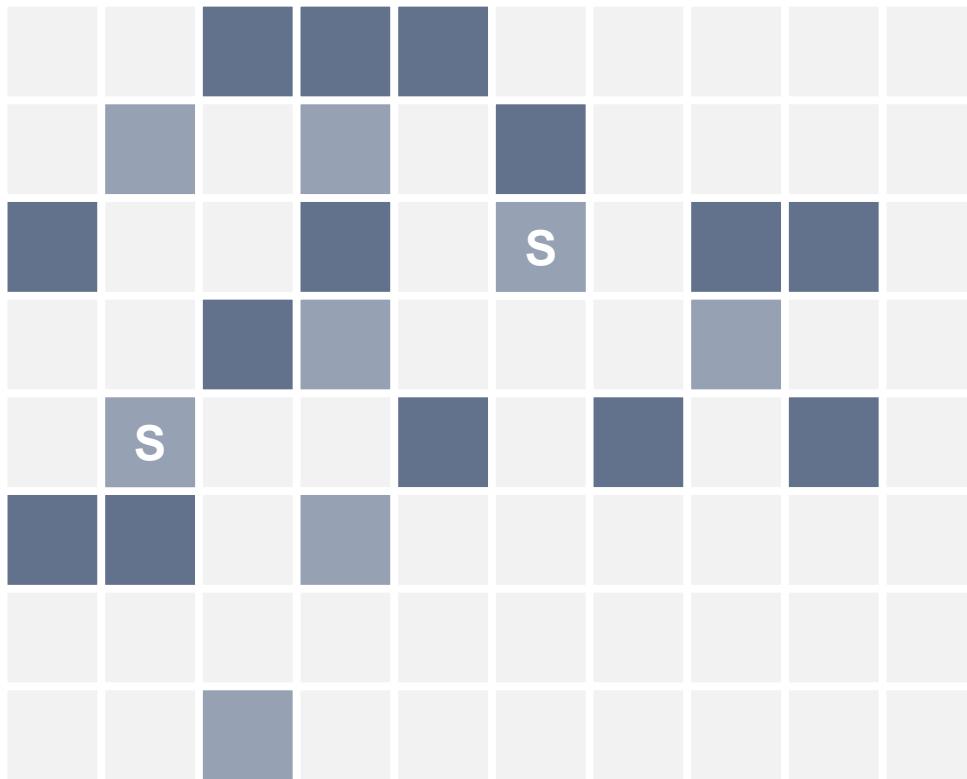
**Main differences
between
Garbage-First
and CMS**

G1 Collector: Parallelism & Concurrency



G1 – Garbage First

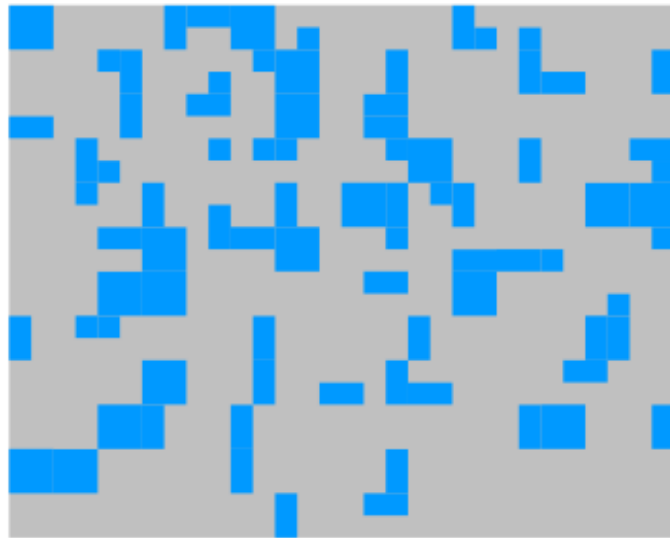
Heap Layout



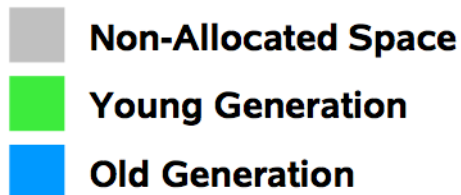
Region Type



CMS vs G1 Collectors



CMS

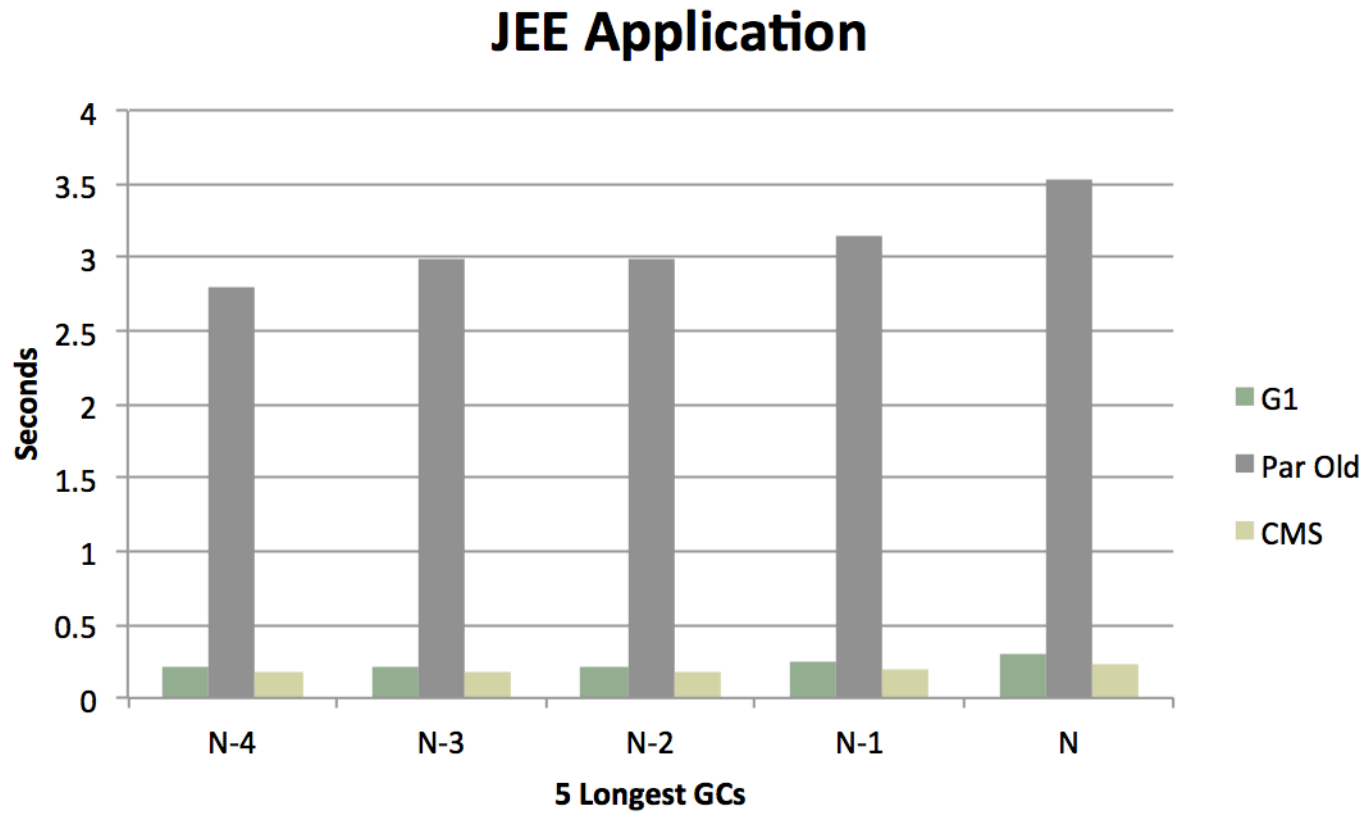


G1

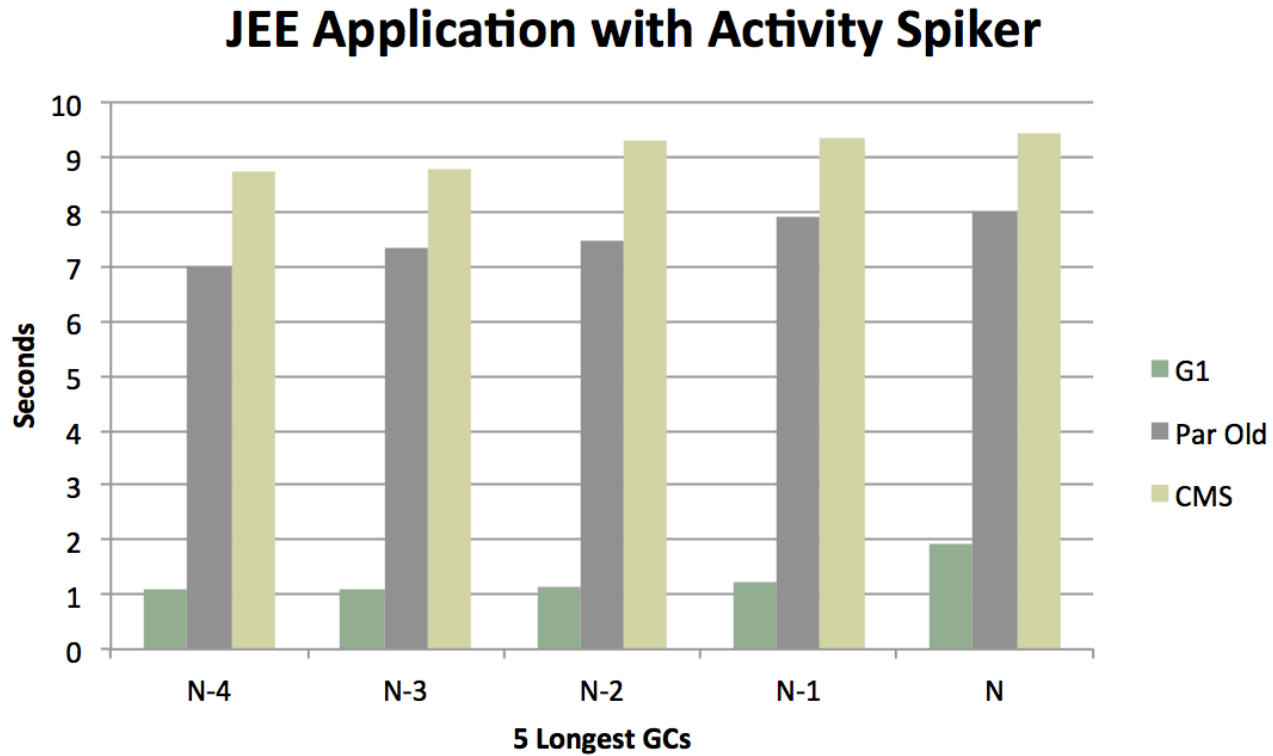


- Heap split into regions
- Young generation (A set of regions)
- Old generation (A set of regions)

GC Comparison

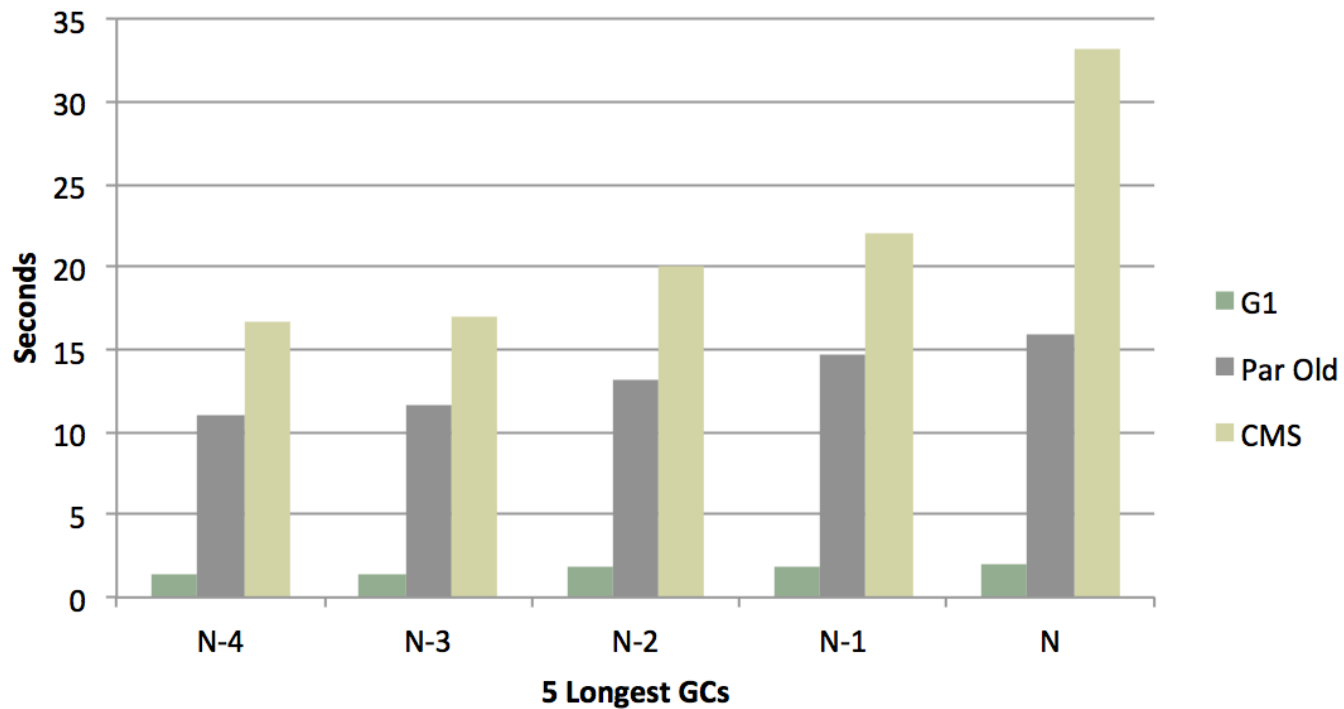


GC Comparison



GC Comparison

JEE Application with Heap Fragmentation



G1 – Pros and Cons

- Pros
 - Only one (concurrent) task needs to act on the whole heap
 - Nice since heaps are getting larger and larger
 - Parallel young and mixed collections
 - Nice since machines tend to have more and more cores
 - Has a lot of flexibility and built in heuristics for choosing young generation and the collection set
 - The only collector in Hotspot that is being actively developed
- Cons
 - Heuristics in place but not fine tuned yet
 - Has a larger memory footprint
 - More concurrent work compared to other collectors

G1 - Targeted Use Cases

- Large heaps with limited GC latencies
 - Typically ~6GB or larger heaps
- Applications that have
 - Varied object allocation rate
 - Undesired long GC or compaction pauses (greater than 0.5s) and want to lower down the pauses time
- Tuning:
 - Garbage First Garbage Collector Tuning
<http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html>
 - Deep Dive into G1 Garbage Collector
<http://www.infoq.com/presentations/java-g1>

G1 - Currently Not Targeted Use Cases

- Applications with very low latency requirements
 - Continue to use Deterministic GC or CMS
- Maximum throughput with no latency requirements
 - Continue to use Throughput collector or Parallel/ParallelOld
- Applications that continuously load and unload classes



References

- Step-by-Step: Garbage Collection Tuning in the Java HotSpot™ Virtual Machine – JavaOne 2010
- The Garbage Collection Mythbusters – JavaOne 2010
- GC Tuning for the HotSpot JVM – JavaOne 2009
- http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html

Hotspot Resources

- **Java Performance** — Charlie Hunt, Binu John
[Hotspot Tuning Book](#)
- **HotSpot GC Tuning Guide**
for Java SE 6
<http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>
- **Oracle University Training**
[Java SE Performance Tuning Ed 1](#)
- **Memory Management in the Java HotSpot VM**
<http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>
- **Java 6 Performance Whitepaper**
http://java.sun.com/performance/reference/whitepapers/6_performance.html





Java (JRockit) Mission Control

Java SE Advanced

- Both HotSpot and JRockit VMs under a single license
- JRockit VM brings to the table
 - Intuitive, user-friendly tooling for monitoring, diagnosing and tuning a Java environment.
 - Dynamic Memory Leak Detection
- Enterprise JRE features i.e. ability to turn auto update off, and usage tracking.



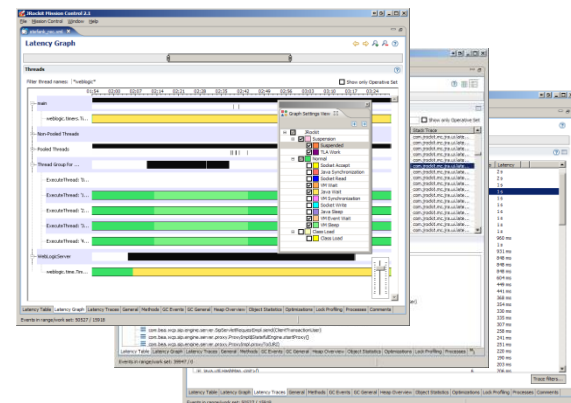
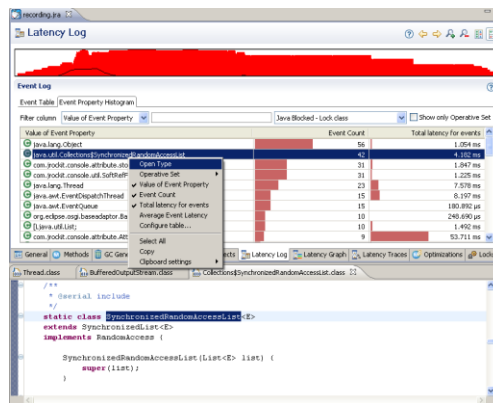
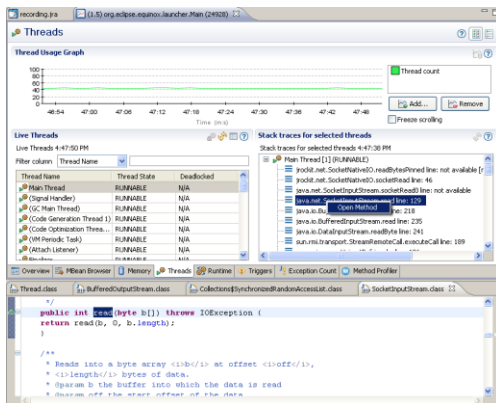
Java SE Advanced

- * Oracle's JRE
- * JRockit JDKs
- * **Java(JRockit) Mission Control**
- * Monitoring and Management features (including APIs for EM integration)

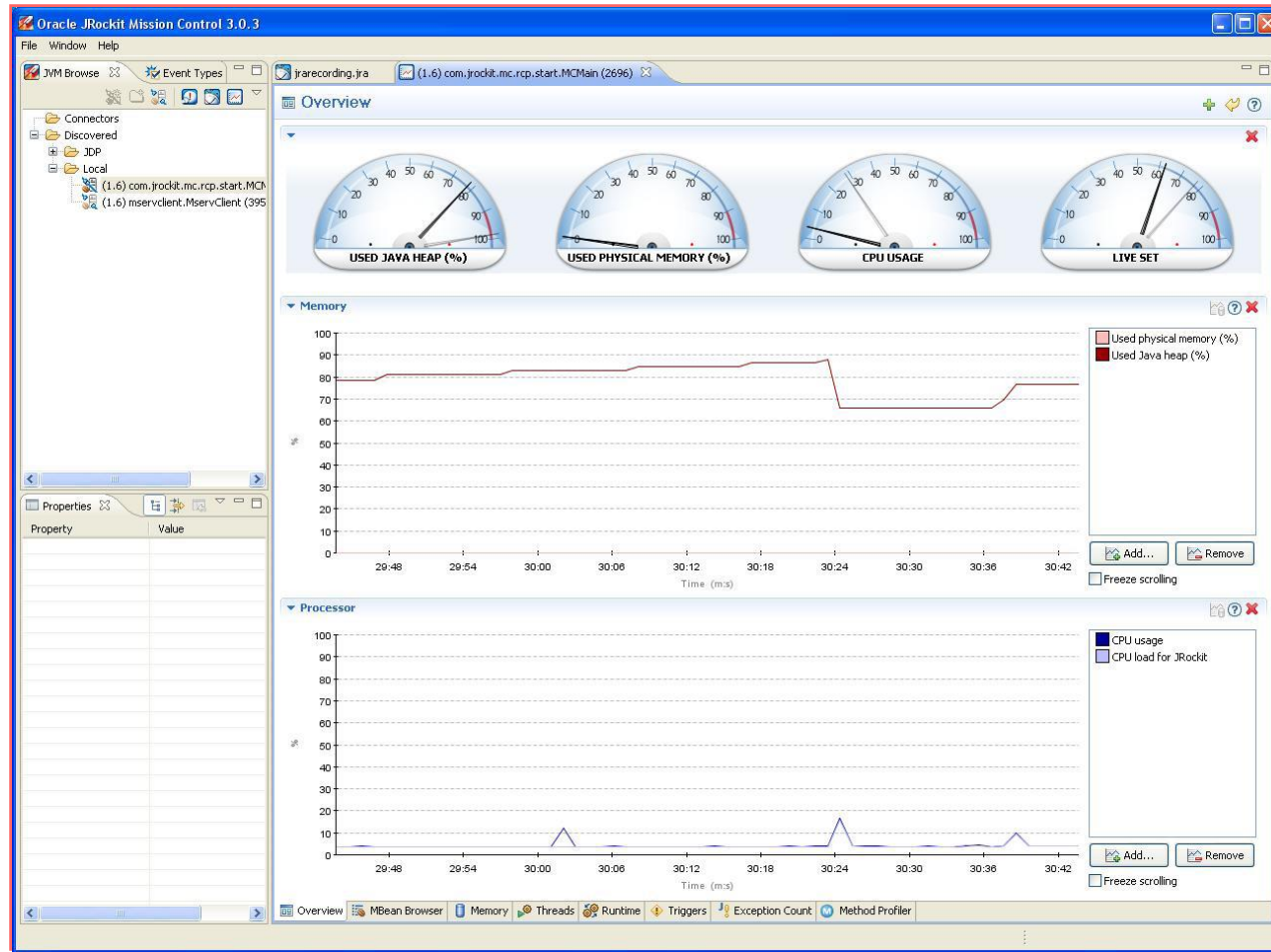
Java (JRockit) Mission Control

After JDK1.7.0_40

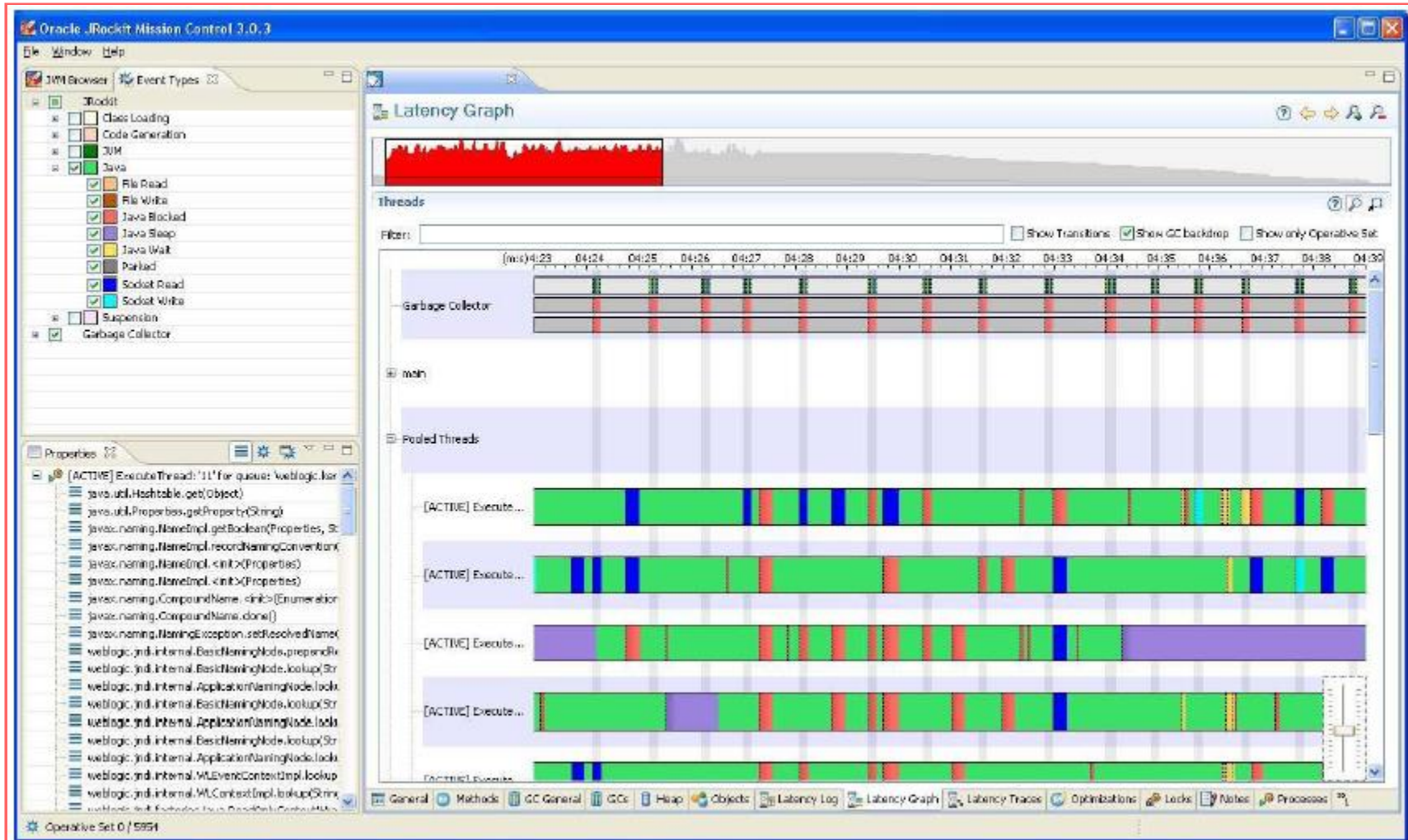
- Monitor health and performance in production
- A suite of powerful tools for
 - Performance Tuning
 - Diagnostics
 - Support
 - Internal survey: 25% more efficient with JRMC



Java(JRockit) Mission Control



Java(JRockit) Mission Control



Java(JRockit) Mission Control

Profiling - Detect Hot Methods

Mission Control - recording11.jra [Mission Control\recording11.jra] - Eclipse SDK

File Edit Navigate Search Project Run Window Help

JVM Browser

- Connectors
- Discovered
- JDP
- Local
- [null] 5884 (5,884)
- This Mission Control

HotMethods.java recording11.jra

Hot Methods

The methods where the application spent the most time executing. Total number of samples are 29,789.

Filter column Method

Method	Percent
java.lang.Integer.equals(Object)	52.5...
java.util.LinkedList.contains(Object)	47.1...
Initiator.countSimilar(Initiator)	0.08%
Initiator.initiate(int)	0.04%
jrockit.vm.Allocator.allocObject(i...	0.03%
java.util.LinkedList.add(Object)	0.02%
jrockit.vm.Allocator.allocInCurre...	0.02%
java.util.LinkedList\$ListItr.hasNex...	0.01%
jrockit.vm.Allocator.allocSlowCas...	0.01%
java.util.LinkedList\$ListItr.next()	0.01%
jrockit.vm.Allocator.allocObjectO...	0.01%
java.lang.Integer.valueOf(int)	0.01%
java.util.LinkedList\$Entry.<init>(O...	0.01%
java.util.LinkedList.add(Object)	0.01%
Initiator.initiate(int)	0.01%
java.util.LinkedList\$ListItr.hasNex...	0.01%
java.lang.Integer.<init>(int)	0.00%
jrockit.vm.Allocator.allocSlowCas...	0.00%
jrockit.vm.Allocator.allocSlowCas...	0.00%
sun.management.counter.perf.Pe...	0.00%
bea.jrockit.management.Garbage...	0.00%

Predecessors

Predecessors for java.lang.Integer.equals(Object)

- 100.00% [10,361] java.lang.Integer.equals(Object)
- 100.00% java.util.LinkedList.contains(Object)
- 100.00% Initiator.countSimilar(Initiator)

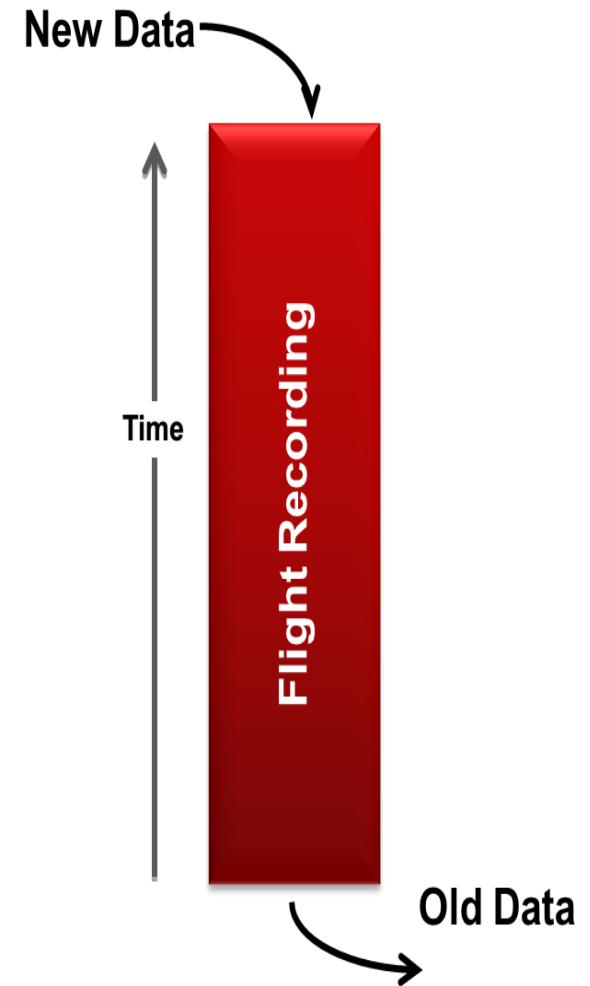
Successors

Successors for java.lang.Integer.equals(Object)

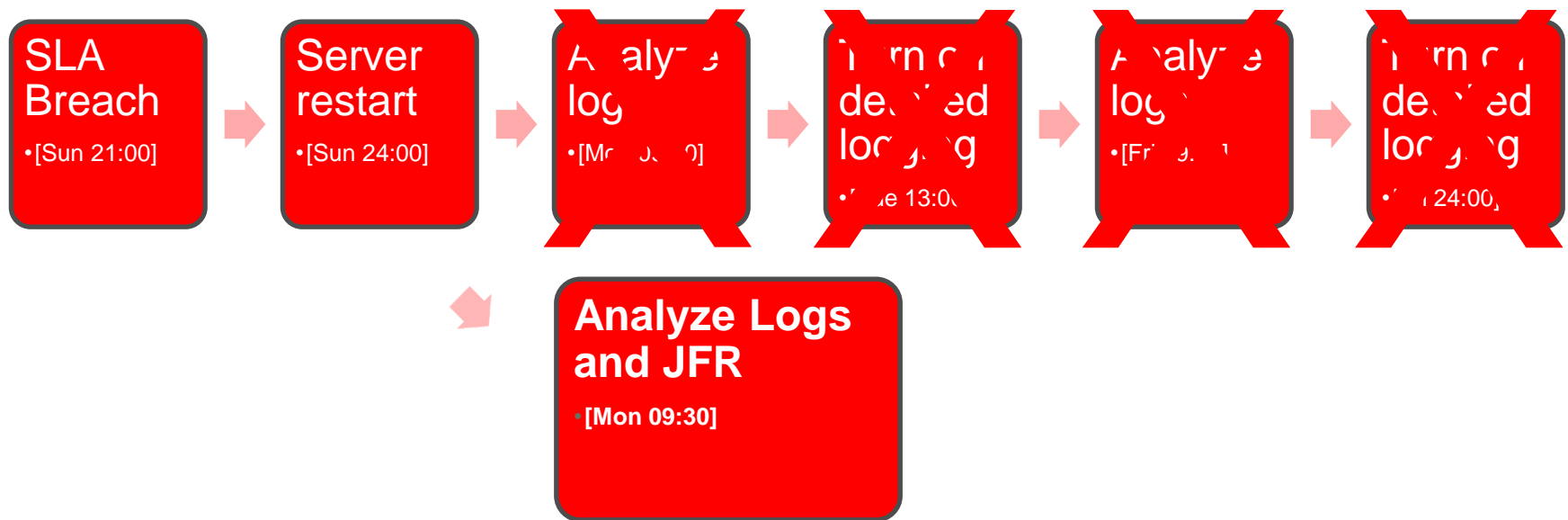
- [0] java.lang.Integer.equals(Object) (OPT)

Overview Hot Methods Optimizations

Oracle Java(JRockit) Flight Recorder



The Problem: The "Intermittent Cycle"



How to use?

- Java 7 update 40 above
- Use following options to start the JMV which you want to monitor:
 - -XX:+UnlockCommercialFeatures
 - -XX:+FlightRecorder
 - -XX:FlightRecorderOptions=defaultrecording=true,
dumponexit=true,dumponexitpath=/tmp/dumponexit.jfr”
- Use Java Mission Control GUI (jmc) to analyze the result

Hardware and Software

ORACLE®

Engineered to Work Together



ORACLE®

Thank you!