jClarity

Hotspot Garbage Collection - The Useful Parts

Martijn Verburg (@karianna)

Session Code: 1500

Who am I?

- **aka "The Diabolical Developer"**
  - I cause trouble in the Java/JVM and F/OSS worlds
  - Especially Agile/Scrum/SC BS
- **CTO of jClarity**
  - Java Performance Tooling start-up
  - "Measure don't guess"
- **Co-lead London Java Community (LJC)**
  - Run global programmes to work on OpenJDK & Java EE
  - **Adopt-a-JSR** and **Adopt OpenJDK**
  - Community night tomorrow night!

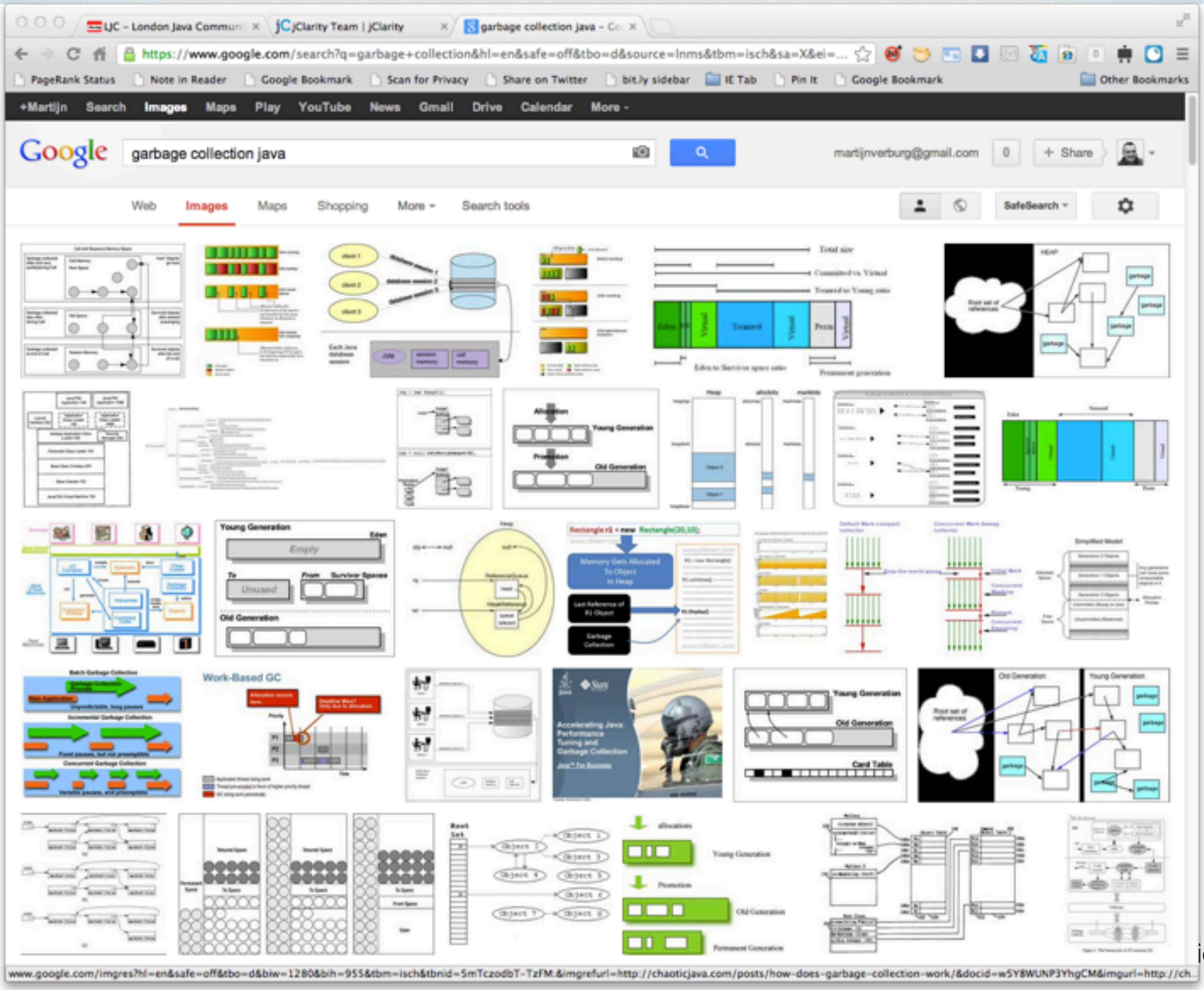
What I'm going to cover

- **Part I - Diving into the Dark (~30 min)**
  - GC Theory
  - Hotspot memory organisation and collectors
- **Break! (2 min)**
  - Our brains hurt
- **Part II - Shining a light into the Darkness (8 min)**
  - Reading GC Logs
  - Tooling and Basic Data
- **Part III - Real World Scenarios (8 min)**
  - Likely Memory Leaks
  - Premature Promotion
  - Healthy App
  - High Pausing

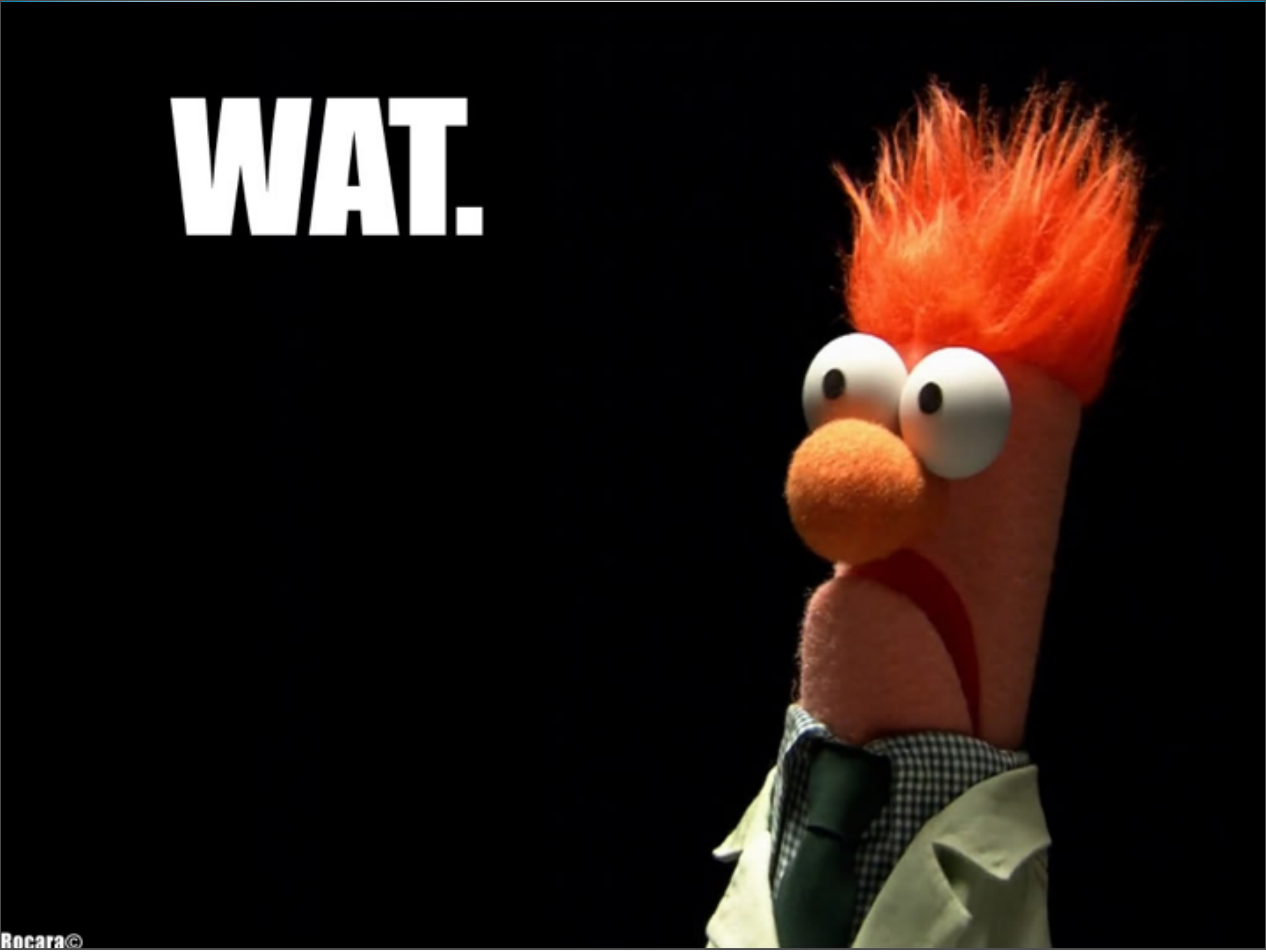
What I'm not covering

- **G1 Collector**
  - It's supported in production now
  - Not a lot of good independent empirical research on this
- **JRockit, Azul Zing, IBM J9 etc**
  - Sorry, these warrant their own talks
  - Go see Azul on level 3 though, what they do is... cool.
- **PhD level technical explanations**
  - I want you to have a working understanding
    - Reality: I'm not that smart
  - Going for that PhD? See me after

Search for Garbage Collection..



ion Code: 1500


WAT.
Rocara©

Part I - Diving into the Dark

- What is Garbage Collection (GC)?
- Hotspot Memory Organisation
- Collector Types
- Young Collectors
- Old Collectors
- Full GC

What is Garbage Collection (GC)?

- **The freeing of memory that is no longer "live"**
  - Otherwise known as "collecting dead objects"
    - Which is a misnomer

- **GC is typically executed by a managed runtime**

- **Javascript, Python, Ruby, .NET CLR all have GC**

And so does Java!

- **One of the main 'selling' points in its early life**

jClarity

Session Code: 1500

Why should I care?

- **Hotspot just sorts this out doesn't it?**
- **Just set `-Xms` and `-Xmx` to be == right?**
  - Stab myself in the eye with a fork
- **A poorly tuned GC can lead to:**
  - High pause times / high % of time spent pausing
  - `OutOfMemoryError`
- **It's usually worth tuning the GC!**
  - "Cheap" performance gain
  - Especially in the short to medium term

Hotspot Java Virtual Machine

- **Hotspot is a C/C++/Assembly app**
  - Native code for different platforms
  - Roughly made up of Stack and Heap spaces
- **The Java Heap**
  - A Contiguous block of memory
  - Entire space is reserved
  - Only some space is allocated
  - Broken up into different memory pools
- **Object Creation / Removal**
  - Objects are created by application (mutator) threads
  - Objects are removed by Garbage Collection

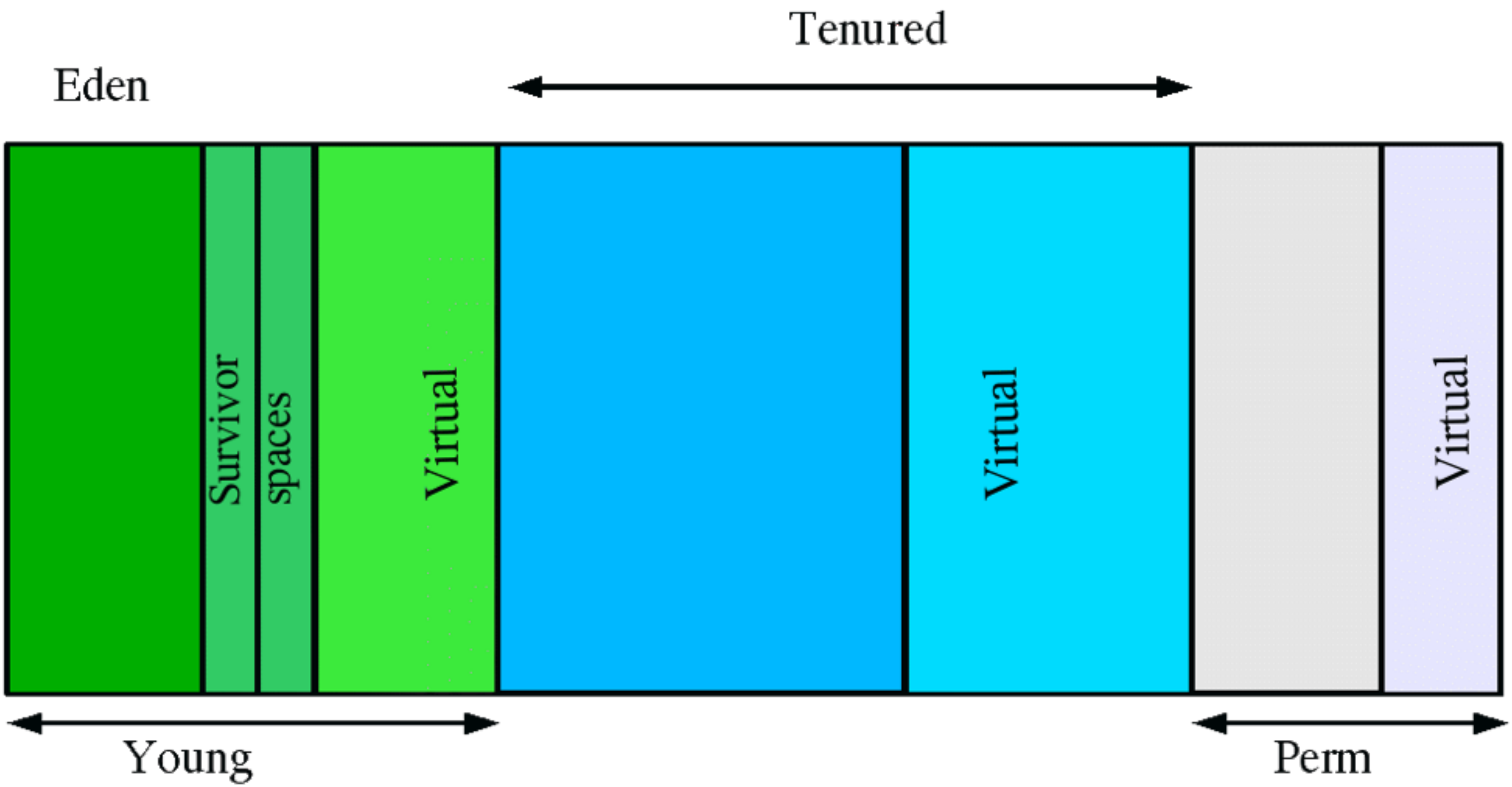
Memory Pools

- **Young Generation Pools**
  - Eden
  - Survivor 0
  - Survivor 1

- **Old Generation Pool (aka Tenured)**
  - Typically much larger than young gen pools combined

- **PermGen Pool**
  - Held separately to the rest of the Heap
  - Was intended to hold objects that last a JVM lifetime
    - Reloading and recycling of classes occurs here.
  - Going away in Java 8

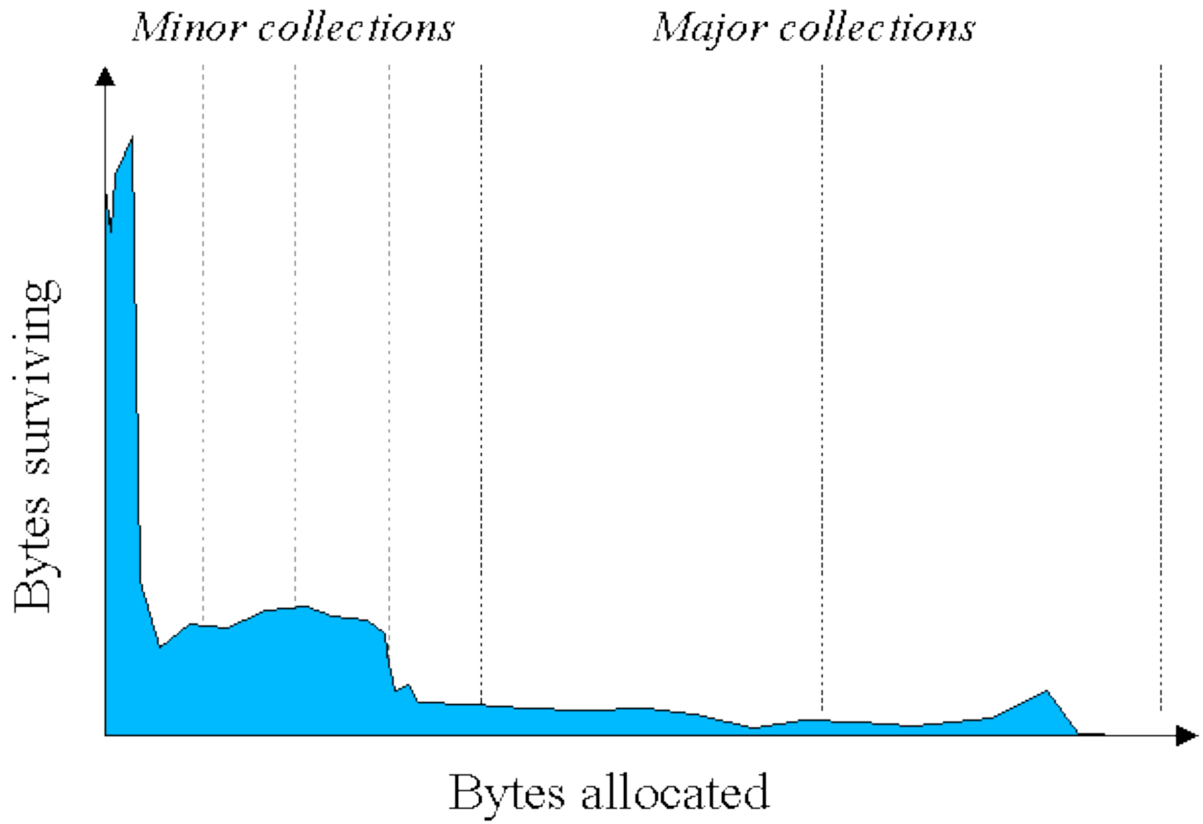
Java Heap Layout



jClarity

Session Code: 1500

Weak Generational Hypothesis



jClarity

Session Code: 1500

Only the good die young...

jClarity

Session Code: 1500

Copy

- **aka "stop-and-copy"**
  - Some literature talks about "Cheney's algorithm"
- **Used in many managed runtimes**
  - Including Hotspot
- **GC thread(s) trace from root(s) to find live objects**
- **Typically involves copying live objects**
  - From one space to another space in memory
  - The result typically looks like a move as opposed to a copy

Mark and Sweep

- **Used by many modern collectors**
  - Including Hotspot, usually for old generational collection

- **Typically 2 mandatory and 1 optional step(s)**
  1. Find live objects (***mark***)
  2. 'Delete' dead objects (***sweep***)
  3. Tidy up - optional (***compact***)

Mark and Sweep collectors in Hotspot

- **Several Hotspot collectors use Mark and Sweep**
  - Concurrent Mark and Sweep (CMS)
  - Incremental Concurrent Mark and Sweep (iCMS)
  - MarkSweepCompact (aka Serial)
  - PS MarkSweep (aka ParallelOld)

- **So it's worth learning the theory**

Java objects

- **Java objects have Ordinary Object Pointers (OOPs)**
  - That point to an object...
  - Which points to the header

- **The header contains a `mark` bit for GC**
  - Plus other metadata (hashcodes, locking state etc)

- **When you call a constructor**
  - Space for the object is allocated

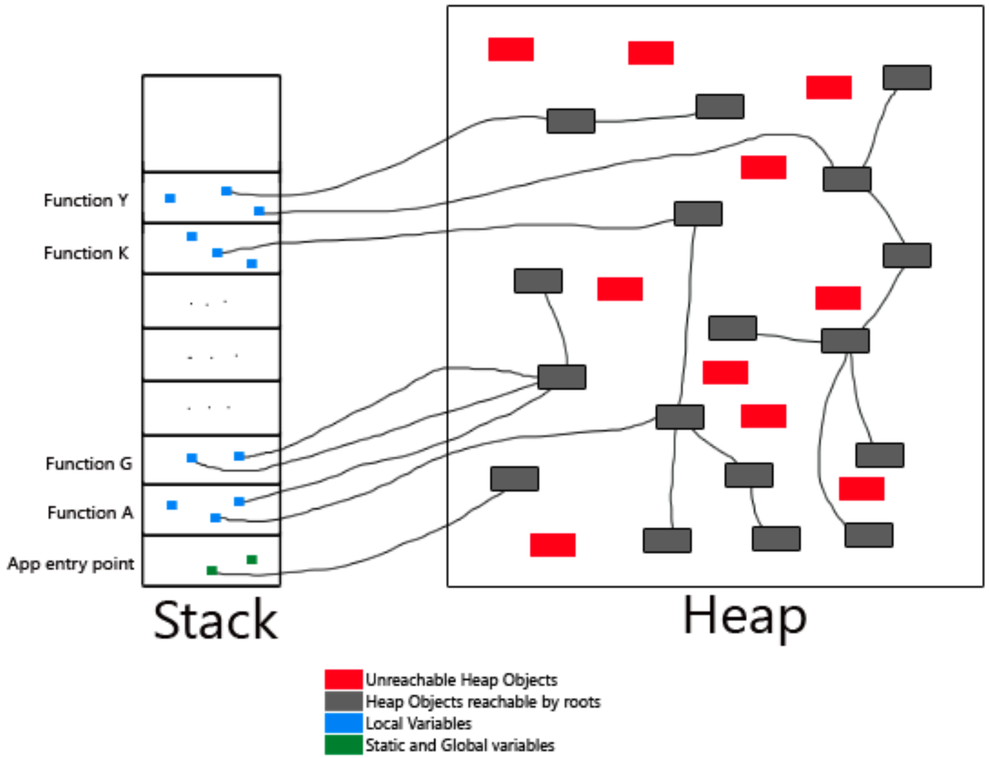
Step 1 - Clear the Mark

- **The header contains the `boolean mark` field**
  - If `true` **-->** the object is **live**

- **Step 1 - set all the `mark` fields to `false`**
  - We need to start afresh

Step 2 - Mark live objects

- **GC Roots**
  - A pointer to data in the heap that you need to keep



Step 2 - Mark live objects

- **GC Roots are made up of:**
  - Live threads
  - Objects used for synchronisation
  - JNI handles
  - The system class loaders
  - Possibly other things depending on your JVM

- **Plus one more special case...**

Step 2 - Mark live objects

- **Special case - Old Gen refs into Young Gen**
  - Treated as roots during a young collection
- **Special card table to track these**
  - Each card references an area of 512 bytes in old gen
  - If it references young gen it will have been marked as dirty
  - Dirty areas are scanned as part of the young collection
- **Conclusion - there's a lot to trace!**

Step 3 - Sweep

- **Sweep**
  - Mark space that dead objects occupy as deleted
- **Compact**
  - Not part of the normal operation of some collectors
  - Always attempted before OOME's can be thrown
  - 'Defrags' the remaining space
    - Not quite a full defrag
- **I'll cover some Java specific collectors shortly**

Heap of Fish Demo

jClarity

Session Code: 1500

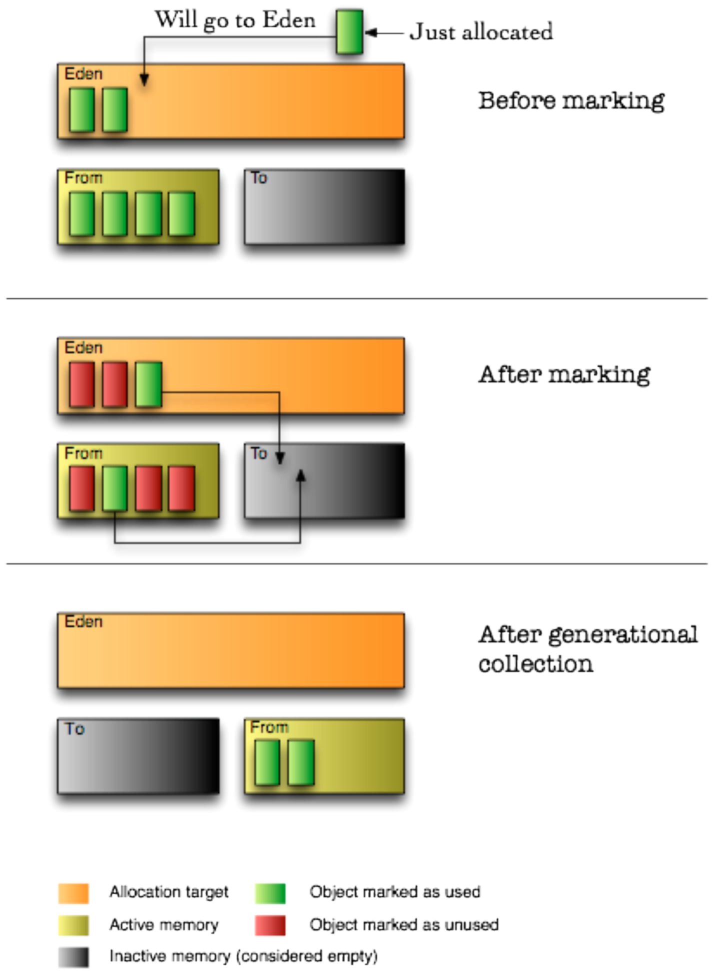
Young Generation Pools

- **Eden**
  - Where new objects should get created
  - Objects are added at the end of currently allocated block
  - Uses Thread Local Allocation Buffers (TLABs)
    - Points at end of allocated block of objects

- **Survivor 0 and Survivor 1**
  - Known as Hemispheric GC
  - Only one is active at a time
  - The other one is empty, we call it the ***target*** space

Young Generation Collectors

- **When Eden gets "full"**
  - "Full" is technically passing a threshold
  - A collector will run
- **Live objects get copied to the *target* Survivor space**
  - From Eden and active Survivor space
- **Some Live objects are promoted to Old Gen**
  - If they've survived > `tenuringThreshold` collections
  - Or if they can't fit in the ***target*** space
- **When the collector is finished**
  - A simple pointer swizzle activates the ***target*** Survivor space
  - Dead objects effectively disappear (no longer referenced)



Will go to Eden
Just allocated
Eden
From
To
Before marking
Eden
From
To
After marking
Eden
To
From
After generational collection
Allocation target
Active memory
Inactive memory (considered empty)
Object marked as used
Object marked as unused

jClarity

Young Generation Collectors

- **Most use parallel threads**
  - i.e. A multi-core machine can make your GC faster

- **I'll cover the PS Scavenge and ParNew collectors**
  - They're almost identical
  - **PS Scavenge** works with **PS MarkSweep** old gen
  - **ParNew** works with **ConcurrentMarkSweep (CMS)** old gen

- **Other young collectors:**
  - Copy (aka Serial)
  - G1

PS Scavenge / ParNew

- **aka "Throughput collectors"**
- **Number of threads is set as a ratio to # of cores**
- **They're Stop-The-World (STW) collectors**
  - They're monolithic (as opposed to incremental)
- **Each thread gets a set of GC roots**
  - They do work stealing
- **It performs an copy (aka evacuate)**
  - Surviving objects move to the newly active survivor pool

Age and Premature Promotion

- **Objects have an age**

- **Every time they survive a collection..**
  - `age++`

- **At age > `tenuringThreshold`**
  - Objects get moved (promoted) to old/tenured space
  - Default `tenuringThreshold` is `4`

- **Premature Promotion occurs when**
  - High memory pressure (high life over death ratio)
    - Eden is too small to deal with rate of new objects
  - Objects are too big to fit in Eden
  - Objects are too big to be promoted to Survivor spaces

Demo

jClarity

Session Code: 1500

Old Generation Collectors

- **Most are variations on Mark and Sweep**
- **Most use parallel threads**
  - e.g. A multi-core machine can make your GC faster
- **I'll cover PS MarkSweep & CMS**
  - CMS is often paired with the ParNew young collector
- **Other old collectors:**
  - MarkSweepCompact (aka Serial)
  - Incremental CMS (iCMS)
  - G1

PS MarkSweep

- **aka "ParallelOld"**
  - Often paired with PS Scavenge for young gen
- **Parallel GC threads get sections to look after**
  - Usual Mark and Sweep occur
- **Special Compact phase takes place**
  - low occupancy sections get merged
  - e.g. A compact / defrag operation

CMS Old Gen Collector

- **Only runs when Tenured is about to get full**
  - Tunable as to what 'about to get full' means
- **Attempts to share CPU with application**
  - About a 50/50 ratio as a default
  - Application can keep working whilst GC is taking place
- **It's a partial Stop-The-World (STW) collector**
  - It has 6 phases
    - 2 STW
    - 4 Concurrent
- **It does not compact unless it fails..**

CMS Phases

- **Phase 1 - Initial Mark (STW)**
  - Marks objects adjacent to GC roots
- **Phase 2 - Mark (Concurrent)**
  - Completes depth first marking
- **Phase 3 - Pre Clean (Concurrent)**
  - Retraces the updated objects, finds dirty cards
- **Phase 4 - Re Mark / Re Scan (STW)**
  - Hopefully a smaller graph traversal over dirty paths
- **Phase 5/6 - Concurrent Sweep and Reset**
  - Sweep out dead objects and reset any data structures

Concurrent Mode Failure (CMF)

- **Occurs when CMS can't complete 'in time'**
  - 'In time' meaning that tenured has filled up
- **GC subsystem reverts to a Full GC at this point**
  - Basically ouch

Promotion Failure

- **Occurs when objects can't be promoted into Tenured**
  - Often due to the Swiss Cheese nature of Old Gen
    - Because CMS does not compact

- **This will almost always happen.... eventually**

- **Triggers a Full GC**
  - Which compacts old space
  - No more Swiss Cheese! For a short while...

Full GC

- **Can be triggered by a number of causes**
  - A CMF from the CMS Collector
  - Promotion Failure
  - When tenured gets above a threshold
  - `System.gc()`
  - Remote `System.gc()` via RMI

- **Runs a full STW collection**
  - Over Young and Old generational spaces
  - Compacts as well

Special Case: OOME

Session Code: 1500

Special Case: OOME

- 98%+ time is spent in GC
- < 2% of Heap is freed in a collection
- Allocating an object larger than heap
- Sometimes when the JVM can't spawn a new Thread

Part II - Shining a light into the dark

- Collector Flags ahoy
- Reading CMS Log records
- Tooling and basic data

'Mandatory' Flags

- **`-verbose:gc`**
  - Get me some GC output
- **`-Xloggc:<pathtofile>`**
  - Path to the log output, make sure you've got disk space
- **`-XX:+PrintGCDetails`**
  - Minimum information for tools to help
  - Replace `-verbose:gc` with this
- **`-XX:+PrintTenuringDistribution`**
  - Premature promotion information
- **`-XX:+PrintGCApplicationStoppedTime`**

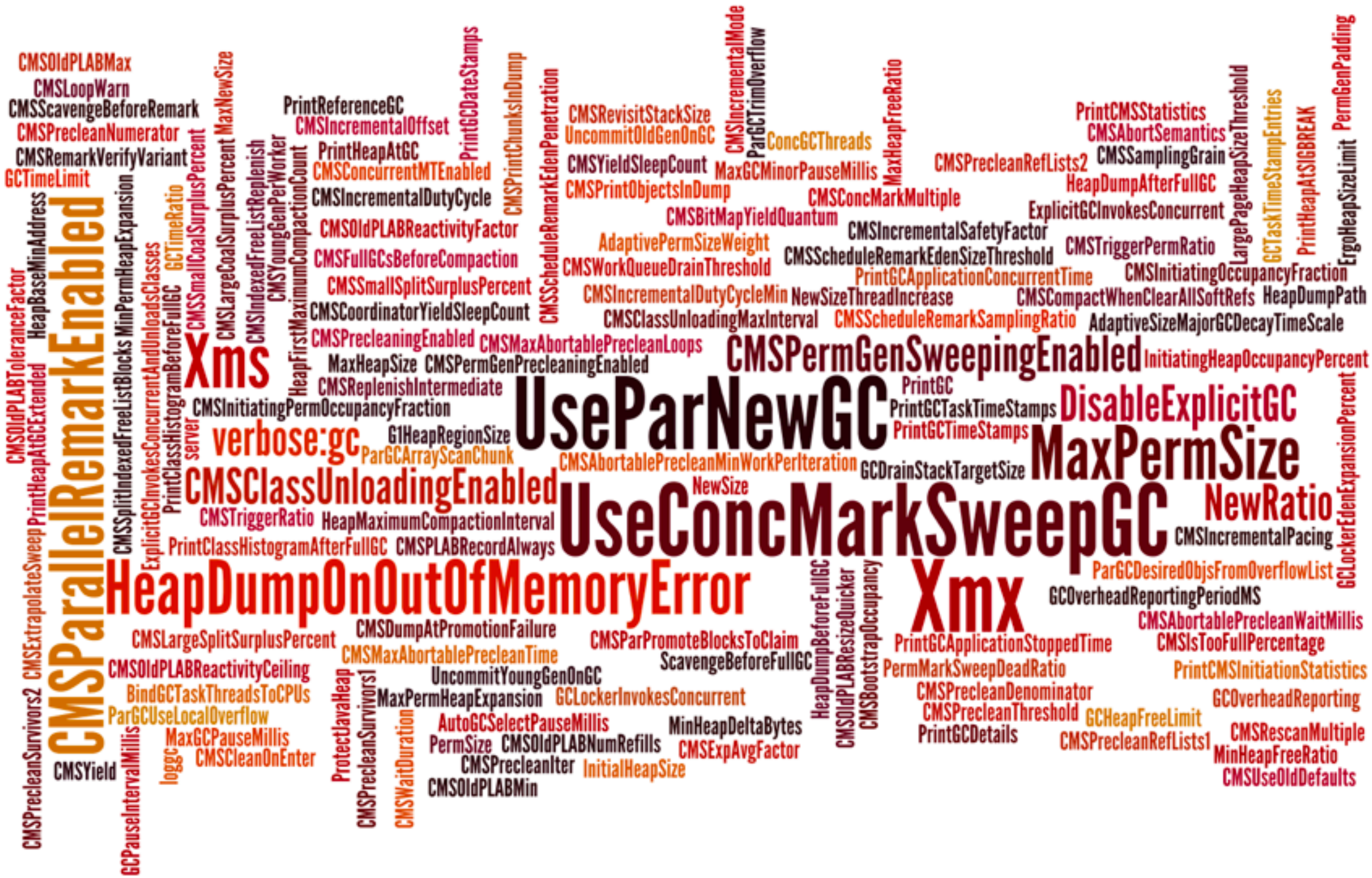
Basic Heap Sizing Flags

- **`-Xms<size>`**
  - Set the minimum size reserved for the heap
- **`-Xmx<size>`**
  - Set the maximum size reserved for the heap
- **`-XX:MaxPermSize=<size>`**
  - Set the maximum size of your perm gen
  - Good for Spring apps and App servers

Other Flags

- `-XX:NewRatio=N`
- `-XX:NewSize=N`
- `-XX:MaxNewSize=N`
- `-XX:MaxHeapFreeRatio`
- `-XX:MinHeapFreeRatio`
- `-XX:SurvivorRatio=N`
- `-XX:MaxTenuringThreshold=N`
- .....

More Flags than your Deity



Copyright Frank Pavageau

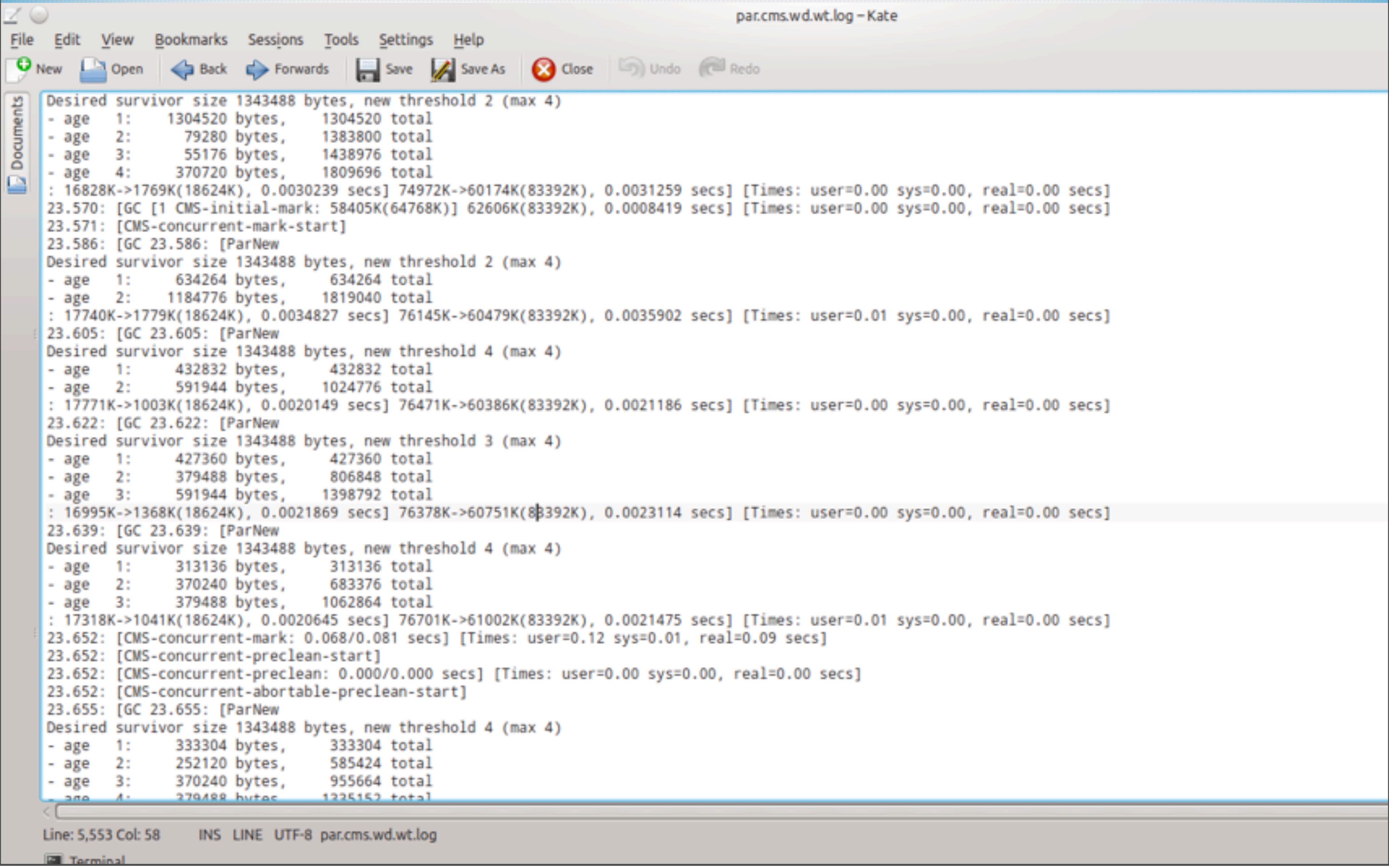
jClarity

Session Code: 1500

Why Log Files?

- **Log file can be post processed**
- **Log files contain more information**
  - Than runtime MXBeans
- **Runtime MXBeans impact the running application**
  - Causing it's own GC problems!

Raw GC Log File



par.cms.wd.wt.log – Kate

File Edit View Bookmarks Sessions Tools Settings Help

New Open Back Forwards Save Save As Close Undo Redo

```
Desired survivor size 1343488 bytes, new threshold 2 (max 4)
- age   1:    1304520 bytes,    1304520 total
- age   2:      79280 bytes,    1383800 total
- age   3:      55176 bytes,    1438976 total
- age   4:     370720 bytes,    1809696 total
: 16828K->1769K(18624K), 0.0030239 secs] 74972K->60174K(83392K), 0.0031259 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
23.570: [GC [1 CMS-initial-mark: 58405K(64768K)] 62606K(83392K), 0.0008419 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
23.571: [CMS-concurrent-mark-start]
23.586: [GC 23.586: [ParNew
Desired survivor size 1343488 bytes, new threshold 2 (max 4)
- age   1:     634264 bytes,     634264 total
- age   2:    1184776 bytes,    1819040 total
: 17740K->1779K(18624K), 0.0034827 secs] 76145K->60479K(83392K), 0.0035902 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
23.605: [GC 23.605: [ParNew
Desired survivor size 1343488 bytes, new threshold 4 (max 4)
- age   1:     432832 bytes,     432832 total
- age   2:     591944 bytes,    1024776 total
: 17771K->1003K(18624K), 0.0020149 secs] 76471K->60386K(83392K), 0.0021186 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
23.622: [GC 23.622: [ParNew
Desired survivor size 1343488 bytes, new threshold 3 (max 4)
- age   1:     427360 bytes,     427360 total
- age   2:     379488 bytes,     806848 total
- age   3:     591944 bytes,    1398792 total
: 16995K->1368K(18624K), 0.0021869 secs] 76378K->60751K(83392K), 0.0023114 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
23.639: [GC 23.639: [ParNew
Desired survivor size 1343488 bytes, new threshold 4 (max 4)
- age   1:     313136 bytes,     313136 total
- age   2:     370240 bytes,     683376 total
- age   3:     379488 bytes,    1062864 total
: 17318K->1041K(18624K), 0.0020645 secs] 76701K->61002K(83392K), 0.0021475 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
23.652: [CMS-concurrent-mark: 0.068/0.081 secs] [Times: user=0.12 sys=0.01, real=0.09 secs]
23.652: [CMS-concurrent-preclean-start]
23.652: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
23.652: [CMS-concurrent-abortable-preclean-start]
23.655: [GC 23.655: [ParNew
Desired survivor size 1343488 bytes, new threshold 4 (max 4)
- age   1:     333304 bytes,     333304 total
- age   2:     252120 bytes,     585424 total
- age   3:     370240 bytes,     955664 total
- age   4:     379488 bytes,    1335152 total
```

Line: 5,553 Col: 58 INS LINE UTF-8 par.cms.wd.wt.log

Terminal

WAT

e: 1500

General Format

from->to(total size)
i.e:
16963K->884K(18624K)

ocupancy(size)
i.e:
62606K(83392K)

Young Gen Collection Part I

```
14.896: [GC 14.896: [ParNew
Desired survivor size 1343488 bytes, new threshold 4 (max 4)
- age   1:     181872 bytes,     181872 total
- age   2:     374976 bytes,     556848 total
- age   3:     216304 bytes,     773152 total
- age   4:     129048 bytes,     902200 total
: 16963K->884K(18624K), 0.0017349 secs] 66634K->50555K(81280K), 0.0018305 secs]
```

Tenuring information

Young Size

Young Occupancy before and after

Young Gen Collection Part II

```
14.896: [GC 14.896: [ParNew
Desired survivor size 1343488 bytes, new threshold 4 (max 4)
- age   1:     181872 bytes,     181872 total
- age   2:     374976 bytes,     556848 total
- age   3:     216304 bytes,     773152 total
- age   4:     129048 bytes,     902200 total
: 16963K->884K(18624K), 0.0017349 secs] 66634K->50555K(81280K), 0.0018305 secs]
```

Pause

Heap Occupancy Before and after

Heap Size

CMS Initial Mark

Tenured Occupancy Tenured Size Pause

```
40.146: [GC [1 CMS-initial-mark: 26386K(786432K)] 26404K(1048384K), 0.0074495 secs]
```

Heap Occupancy Heap Size

All CMS

12.986: [GC [1 CMS-initial-mark: 33532K(62656K)] 49652K(81280K),
0.0014191 secs]

12.987: [CMS-concurrent-mark-start]
13.071: [CMS-concurrent-mark: 0.068/0.084 secs]

13.071: [CMS-concurrent-preclean-start]
13.075: [CMS-concurrent-preclean: 0.001/0.004 secs]

13.077: [GC[YG occupancy: 3081 K (18624 K)]13.077: [Rescan (parallel) ,
0.0009121 secs]13.078: [weak refs processing, 0.0000365 secs]
[1 CMS-remark: 35949K(62656K)] 39030K(81280K), 0.0010300 secs]

13.078: [CMS-concurrent-sweep-start]
13.097: [CMS-concurrent-sweep: 0.016/0.019 secs]

13.264: [CMS-concurrent-reset-start]
13.266: [CMS-concurrent-reset: 0.001/0.001 secs]

Tenured Occupancy
Tenured Size
Young occupancy
Young Size
Heap Occupancy
Heap Size
Pause Time

Tooling

- **HPJMeter (Google it)**
  - Solid, but no longer supported / enhanced
- **GCViewer (http://www.tagtraum.com/gcviewer.html)**
  - Has rudimentary G1 support
- **GarbageCat (http://code.google.com/a/eclipselabs.org/p/garbagecat/)**
  - Best name
- **IBM GCMV (http://www.ibm.com/developerworks/java/jdk/tools/gcmv/)**
  - J9 support
- **jClarity Censum (http://www.jclartity.com/products/censum)**
  - The prettiest and most useful, but we're biased!

HPJMeter - Summary

File View Help

Summary | Heap Usage After GC | Duration | Cumulative Allocation | Creation Rate | User-Defined | Multiple User-Defined

Heap Capacity

	Eden	Survivor	Old	Perm	Total
Initial Capacity	N/A	N/A	N/A	N/A	57.375 (MB)
Final Capacity	N/A	N/A	783.625 (MB)	20.75 (MB)	933.375 (MB)
Peak Capacity	N/A	N/A	783.625 (MB)	20.75 (MB)	933.375 (MB)
Peak Usage of Capacity	N/A	N/A	100%	16.049%	100%

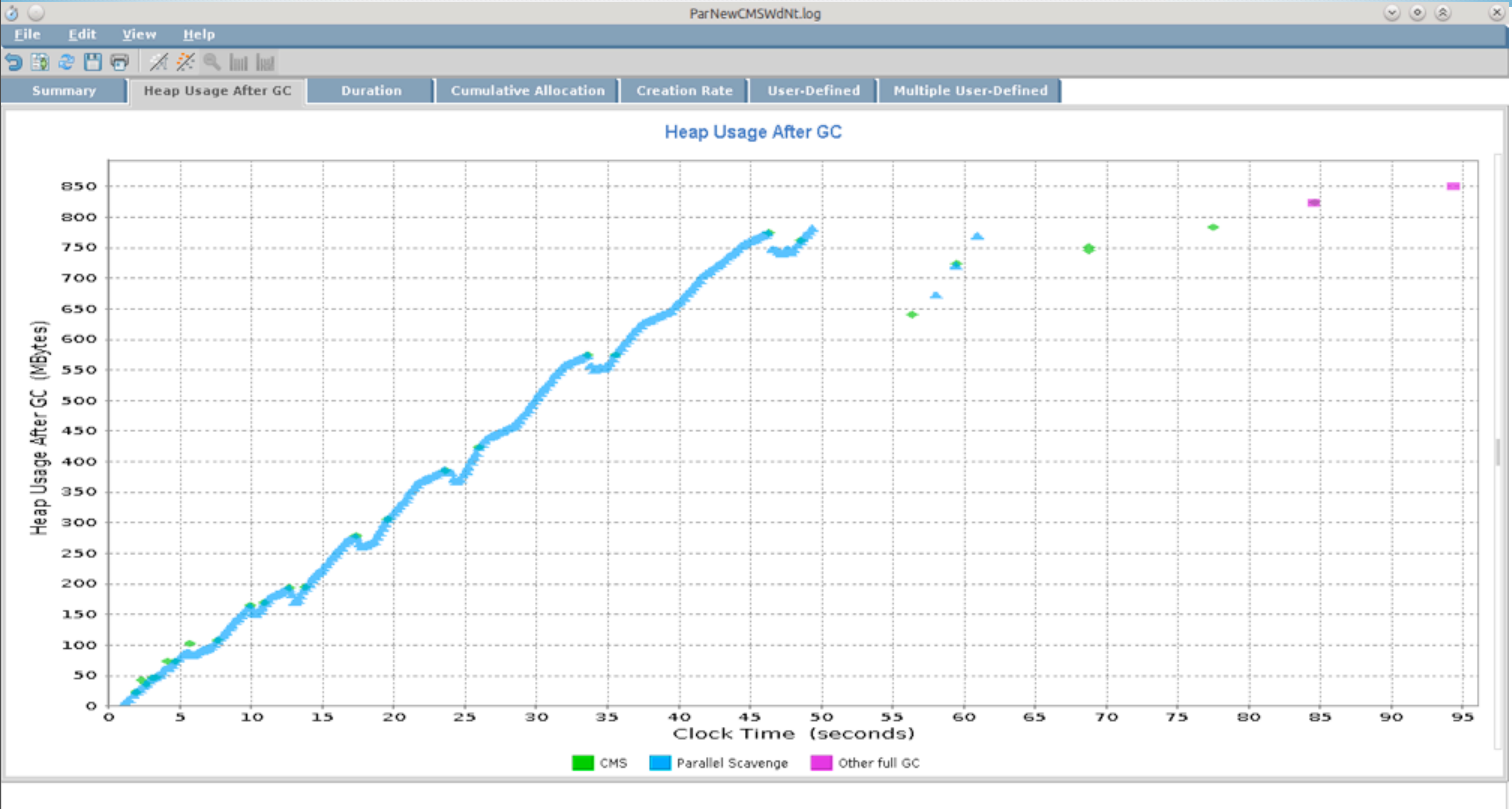
GC Activity Summary

	Last occurrence (s)	Count	Average interval (s)	Average duration (s)	Average rate of collection
CMS	84.54 (s)	27	3.181 (s)	0.81 (s)	0 (B/s)
Parallel Scavenge	60.839 (s)	226	0.266 (s)	0.075 (s)	163.802 (MB/s)
Other full GC	94.216 (s)	2	9.797 (s)	7.183 (s)	13.349 (MB/s)

Overall Statistics

Name	Value	Name	Value
Duration of the measurement	96.08 (s)	Time spent in GC	53.165 (s)
Total bytes allocated	3.798 (GB)	Percentage of time in GC	55.335%
Number of GC events	255	Time spent in Full GC	14.366 (s)
Average bytes allocated per GC	15.251 (MB)	Percentage of time in Full GC	14.952%
Avg. ideal allocation rate	90.625 (MB/s)	Avg. allocation rate	40.478 (MB/s)
Residual bytes	850.562 (MB)		

HPJMeter - Heap Usage After GC



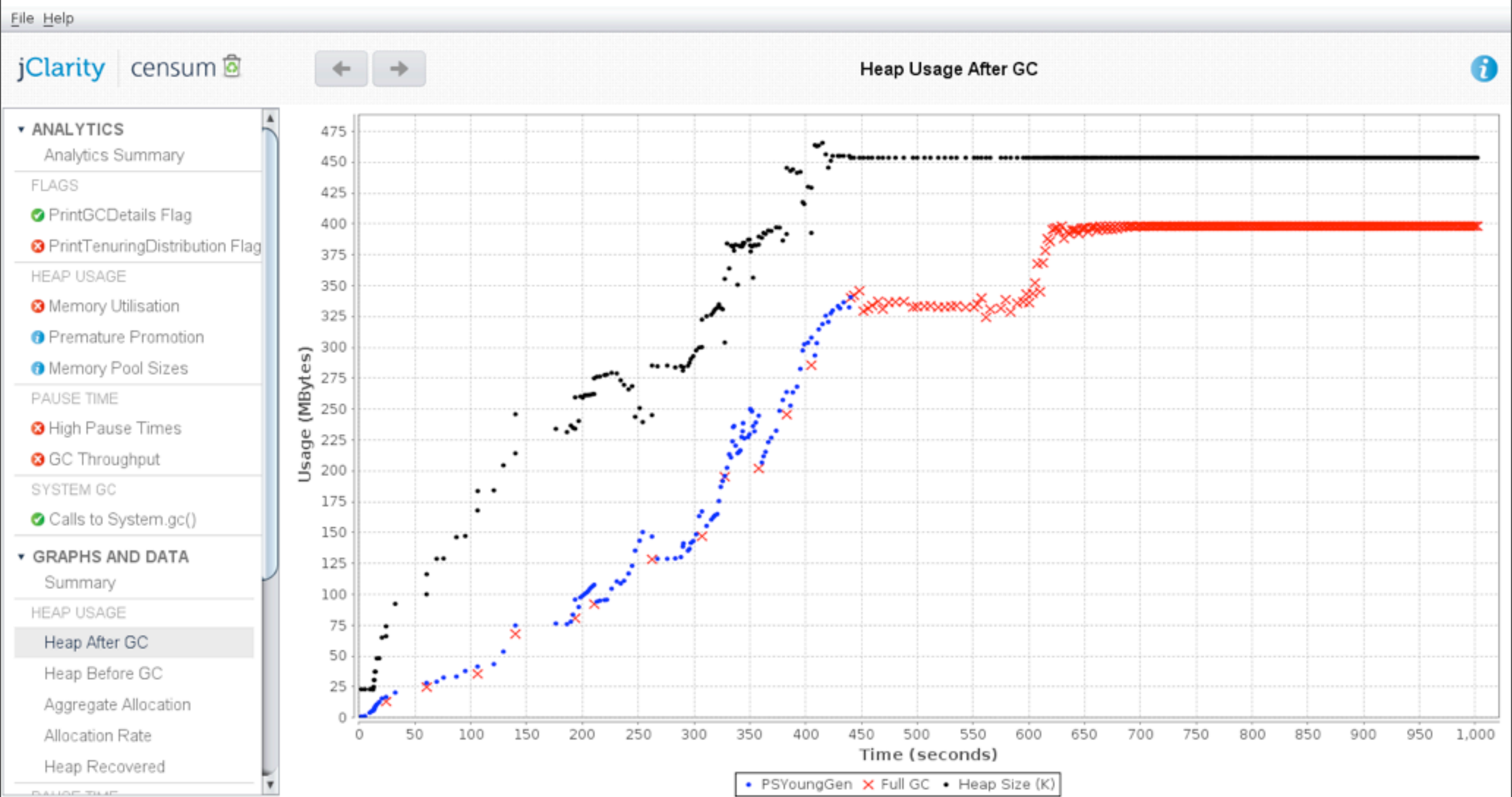
jClarity

Session Code: 1500

Part III - Scenarios

- Possible Memory Leak(s)
- Premature Promotion
- Healthy Application
- High percentage of time spent pausing

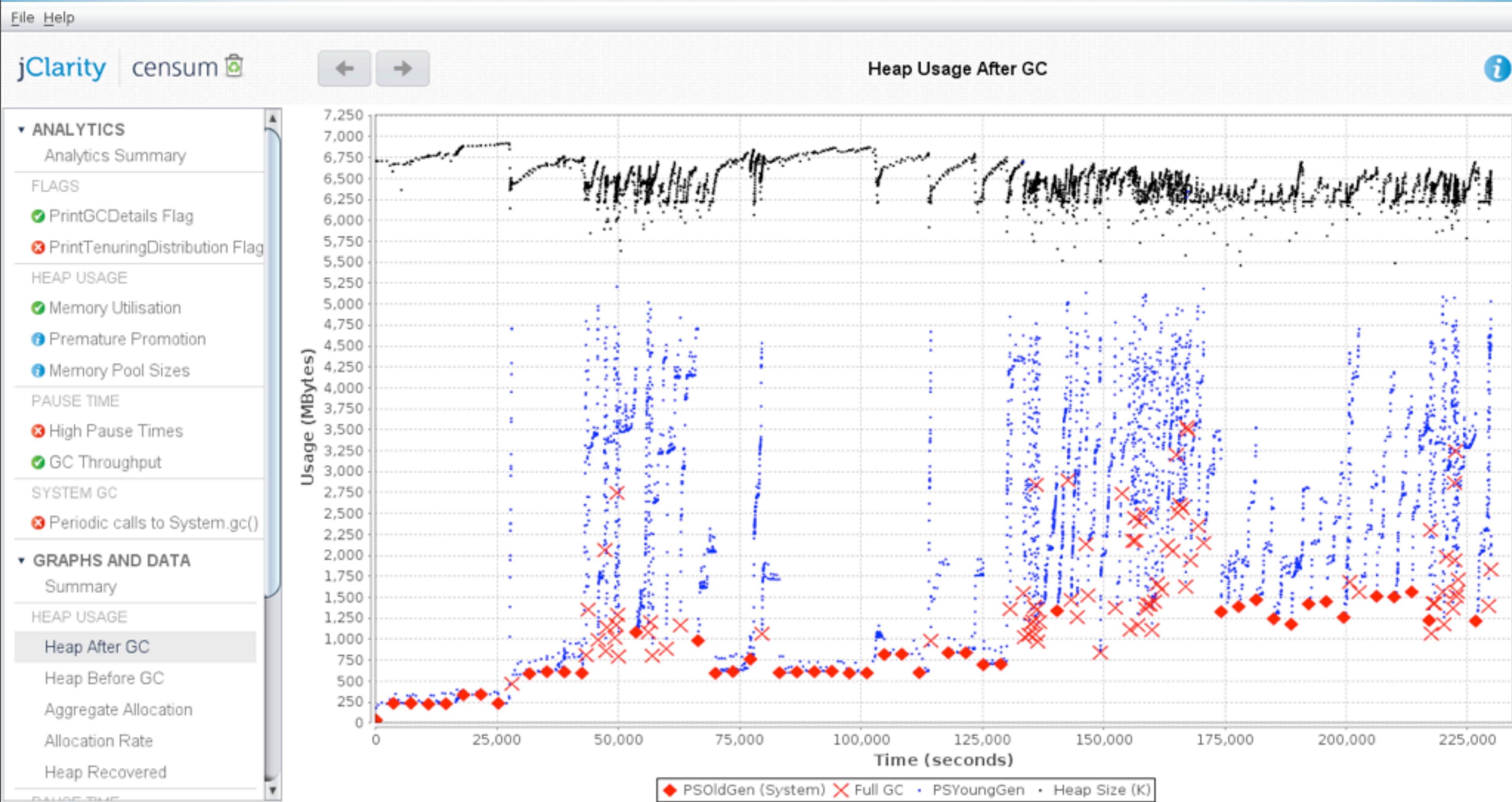
A Memory Leak



jClarity

Session Code: 1500

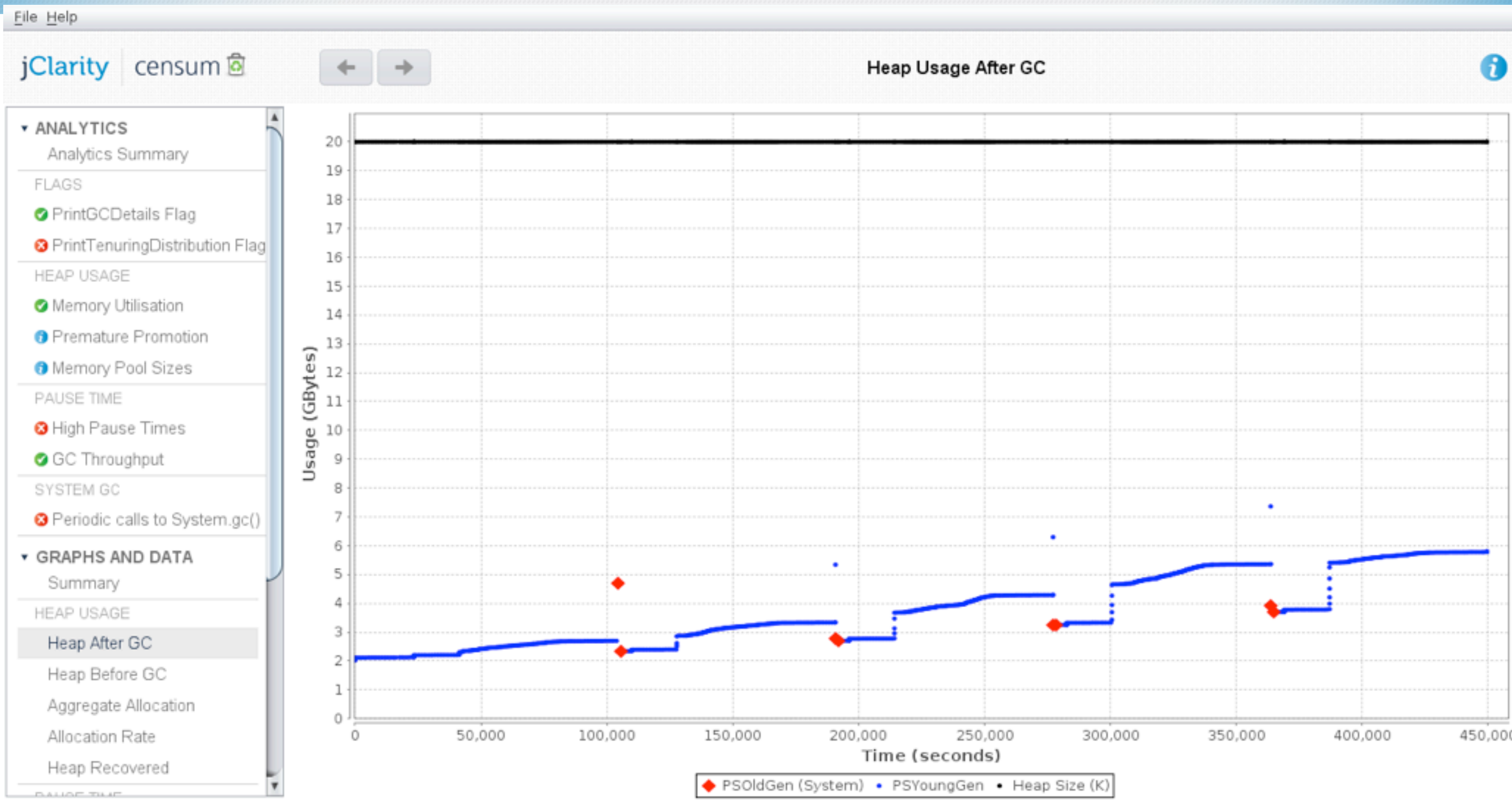
A Possible Memory Leak - I



jClarity

Session Code: 1500

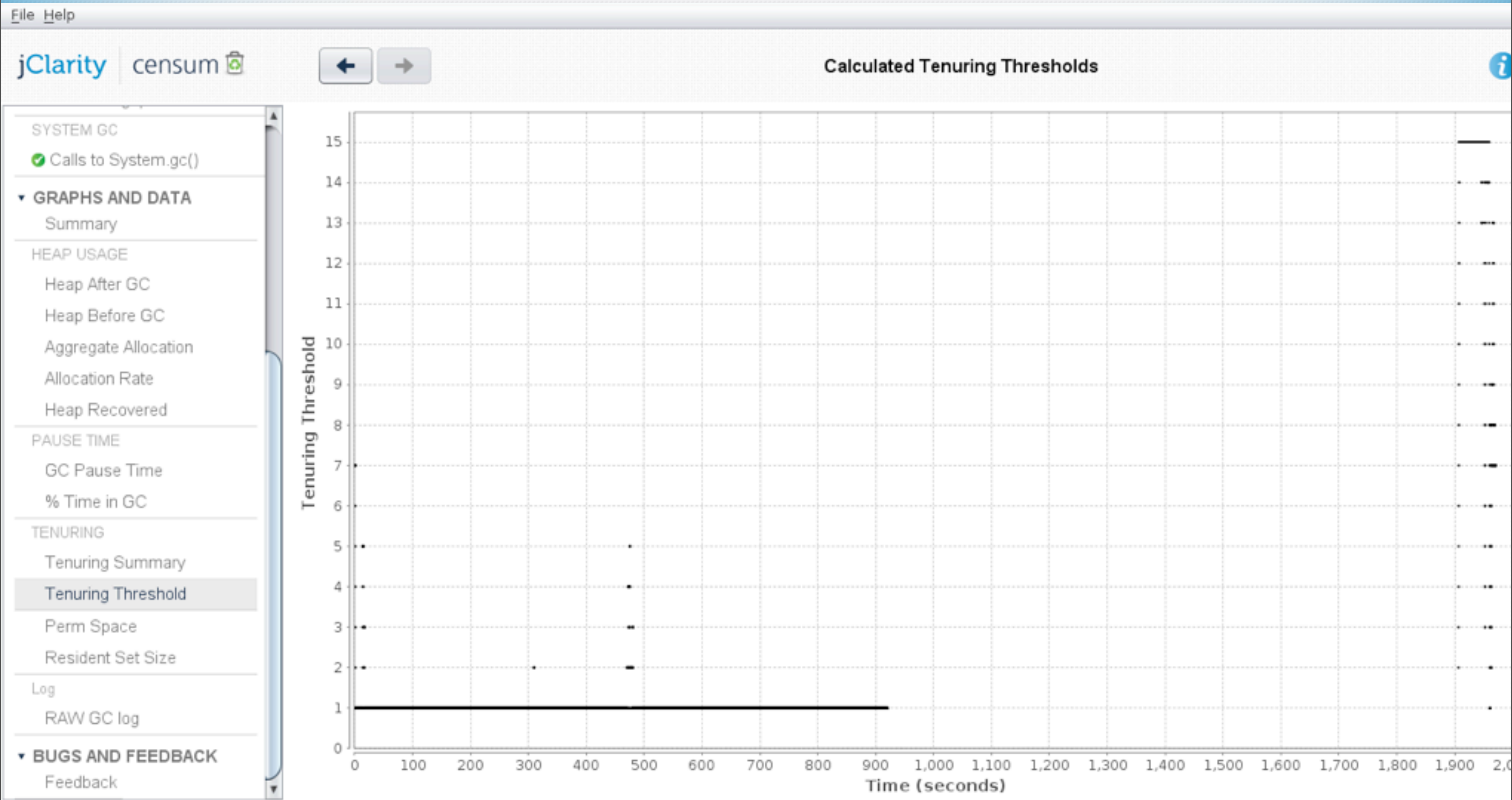
A Possible Memory Leak - II



jClarity

Session Code: 1500

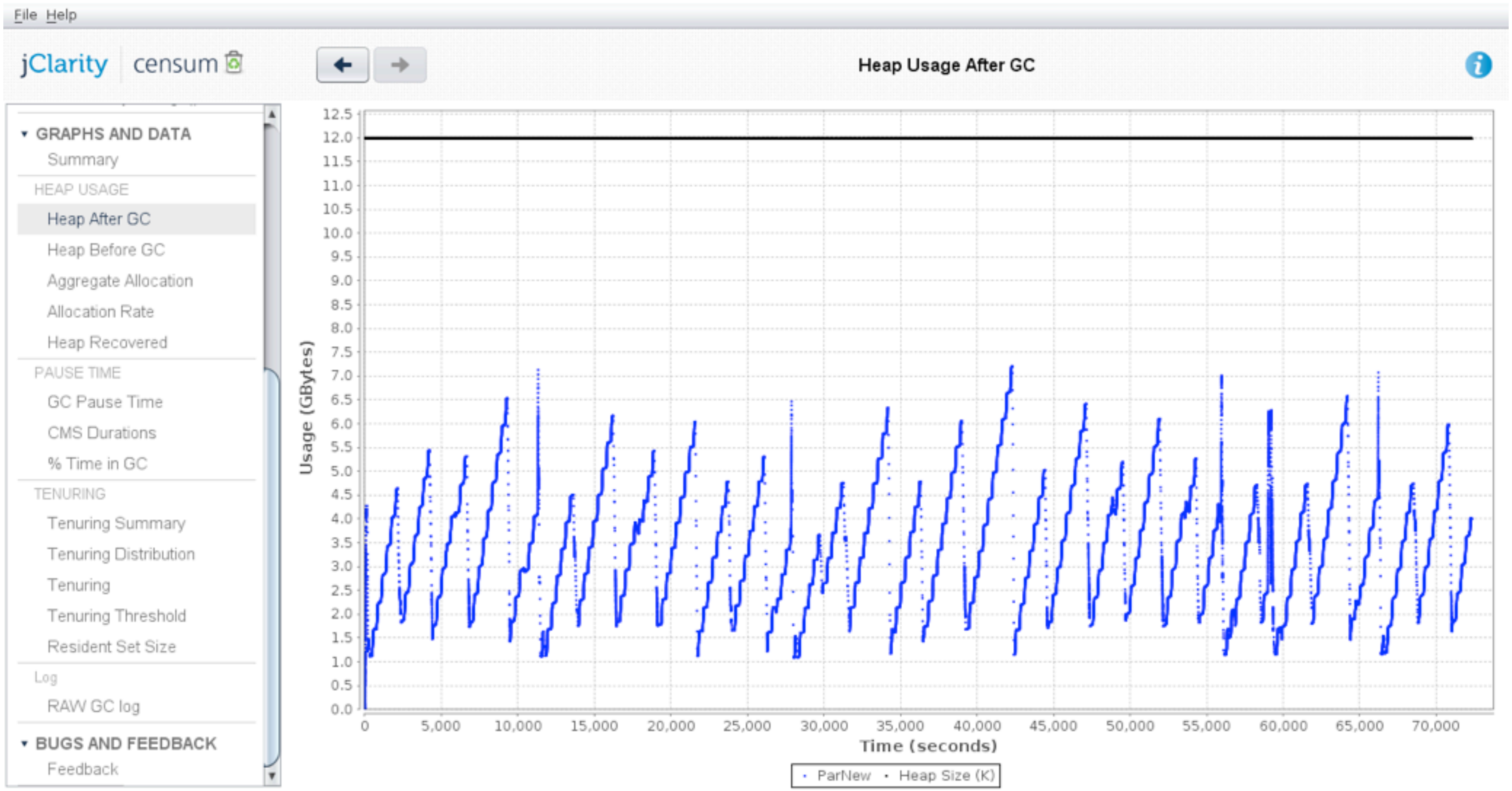
Premature Promotion



jClarity

Session Code: 1500

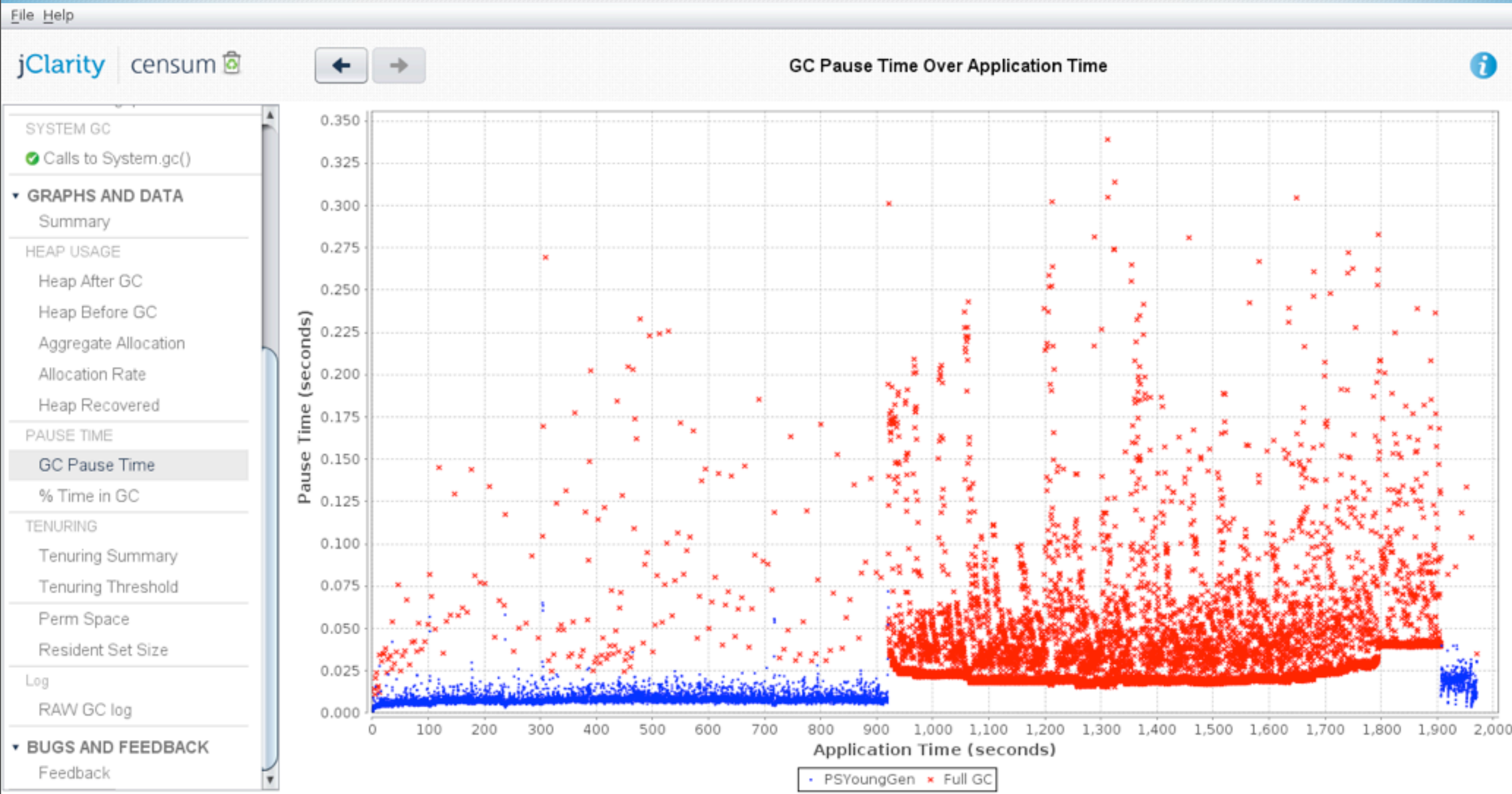
Healthy Application



jClarity

Session Code: 1500

High Percentage of time Paused



jClarity

Session Code: 1500

Summary

- **You need to understand some basic GC theory**

- **You want most objects to die young, in young gen**

- **Turn on GC logging!**
  - Reading raw log files is hard
  - Use tooling!

- **Use tools to help you tweak**
  - "Measure, don't guess"

jClarity

jClarity

Join our performance community

http://www.jclarity.com

Martijn Verburg (@karianna)