The Fundamentals of JVM Tuning

Charlie Hunt
Architect, Performance Engineering
Salesforce.com

In a Nutshell

What you need to know about a modern JVM to be effective at tuning it …

In a Nutshell

What you need to know about a modern JVM to be effective at tuning it … and …

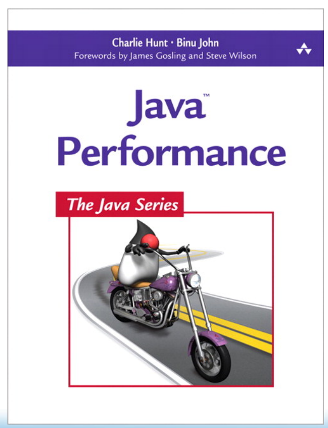
In a Nutshell

What you need to know about a modern JVM to be effective at tuning it … and …

What you need to know about a modern JVM to realize good performance when writing Java code

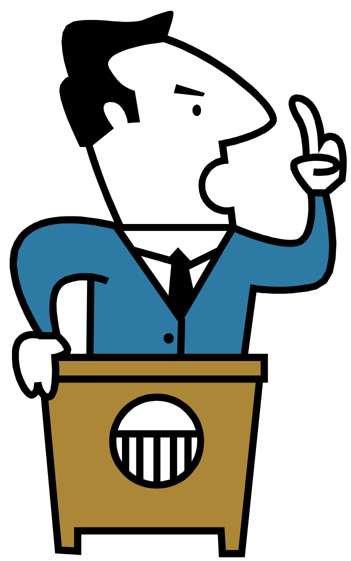
Who is this guy?

- Charlie Hunt
  - Architect of Performance Engineering at Salesforce.com
  - Former Java HotSpot VM Performance Architect at Oracle
  - 20+ years of (general) performance experience
  - 14 years of Java performance experience
  - Lead author of ***Java Performance***, published Sept. 2011



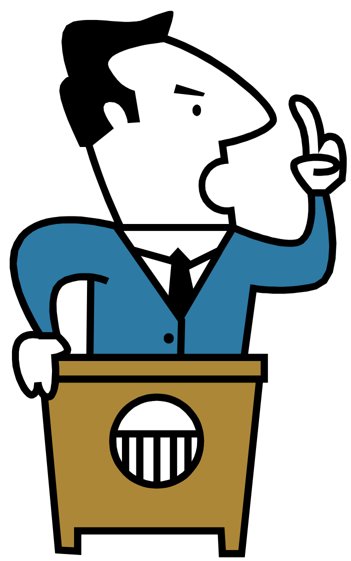
Agenda

- What you need to know about GC
- What you need to know about JIT compilation
- Tools to help you

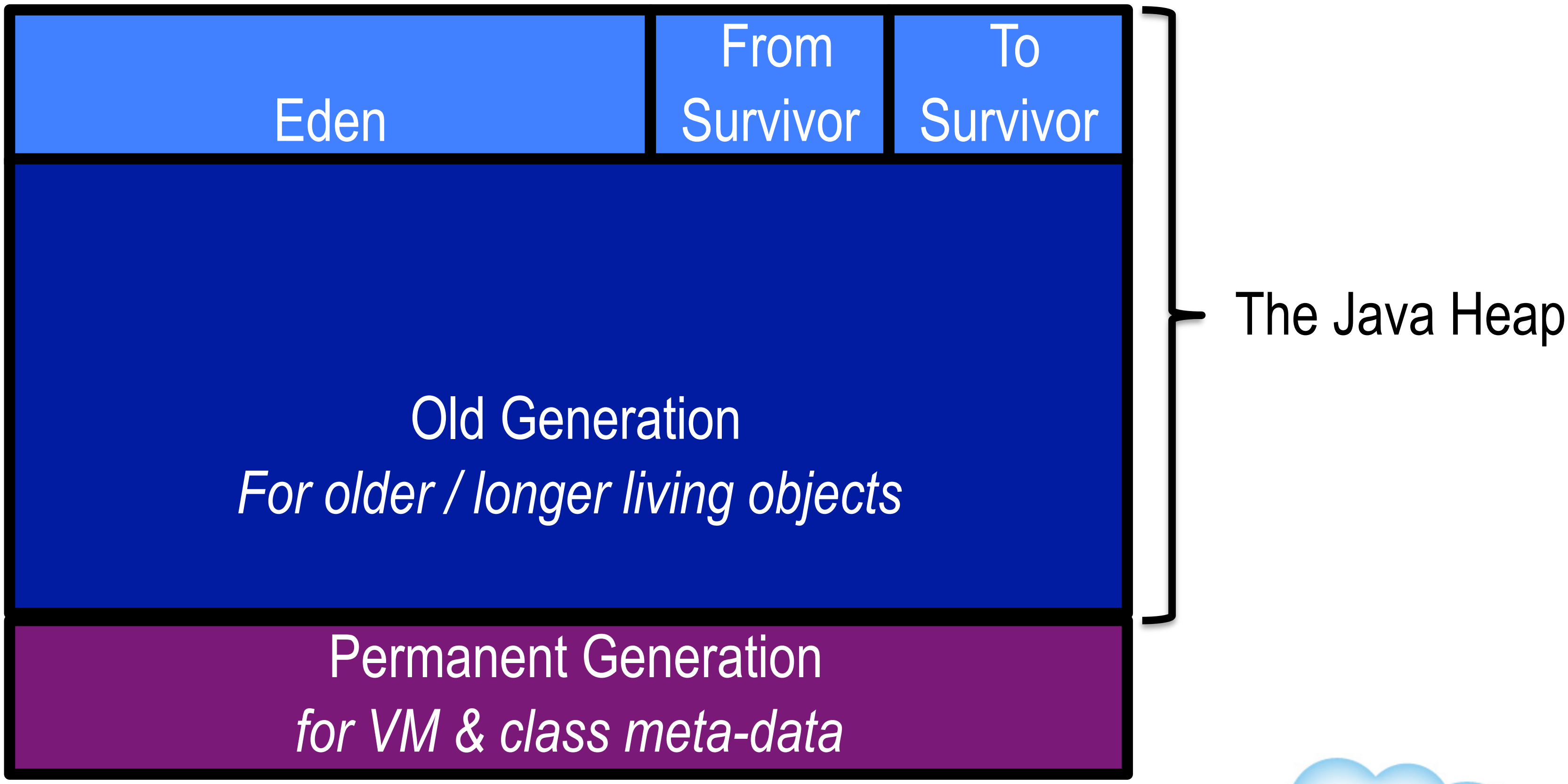


Agenda

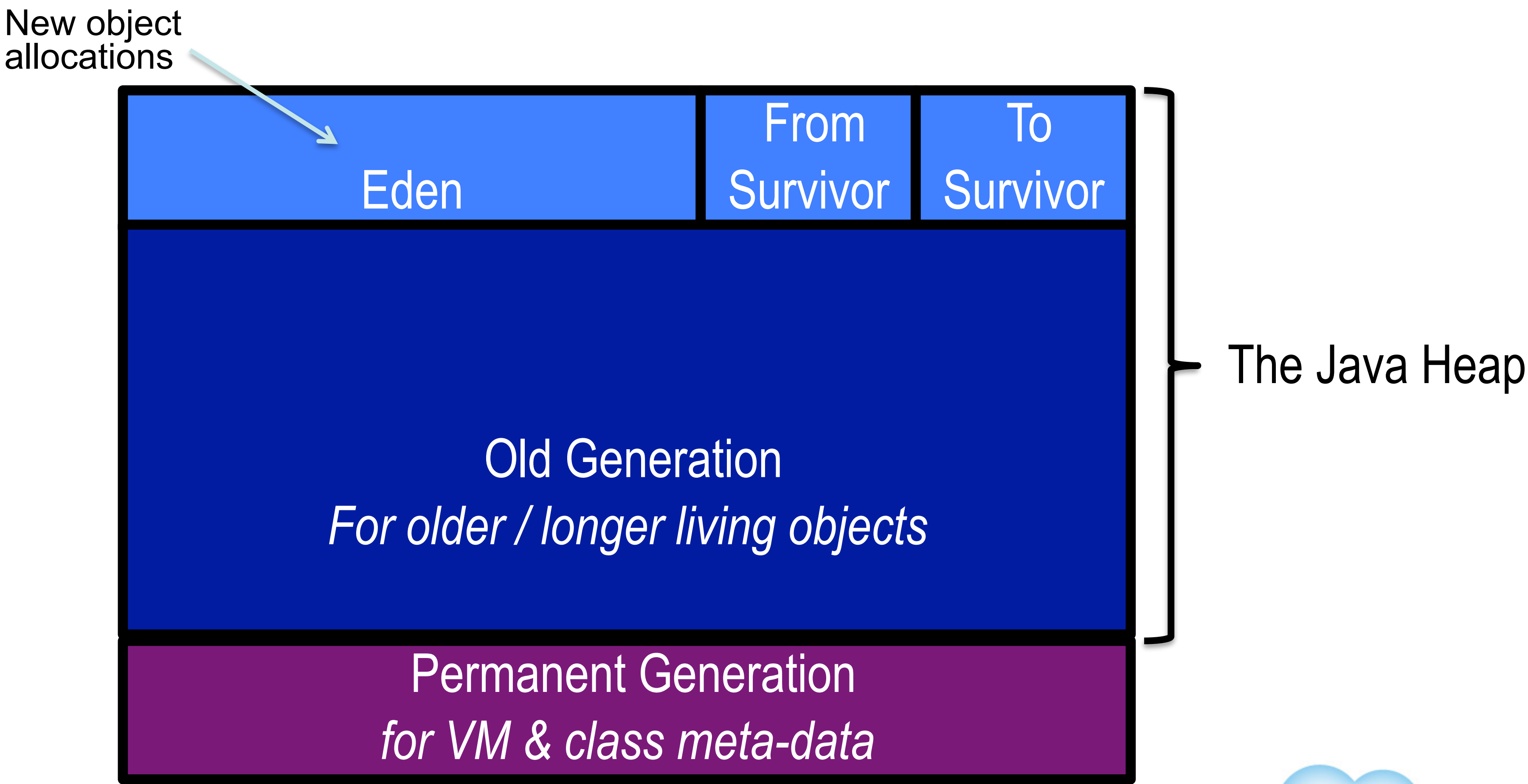
- *What you need to know about GC*
- What you need to know about JIT compilation
- Tools to help you



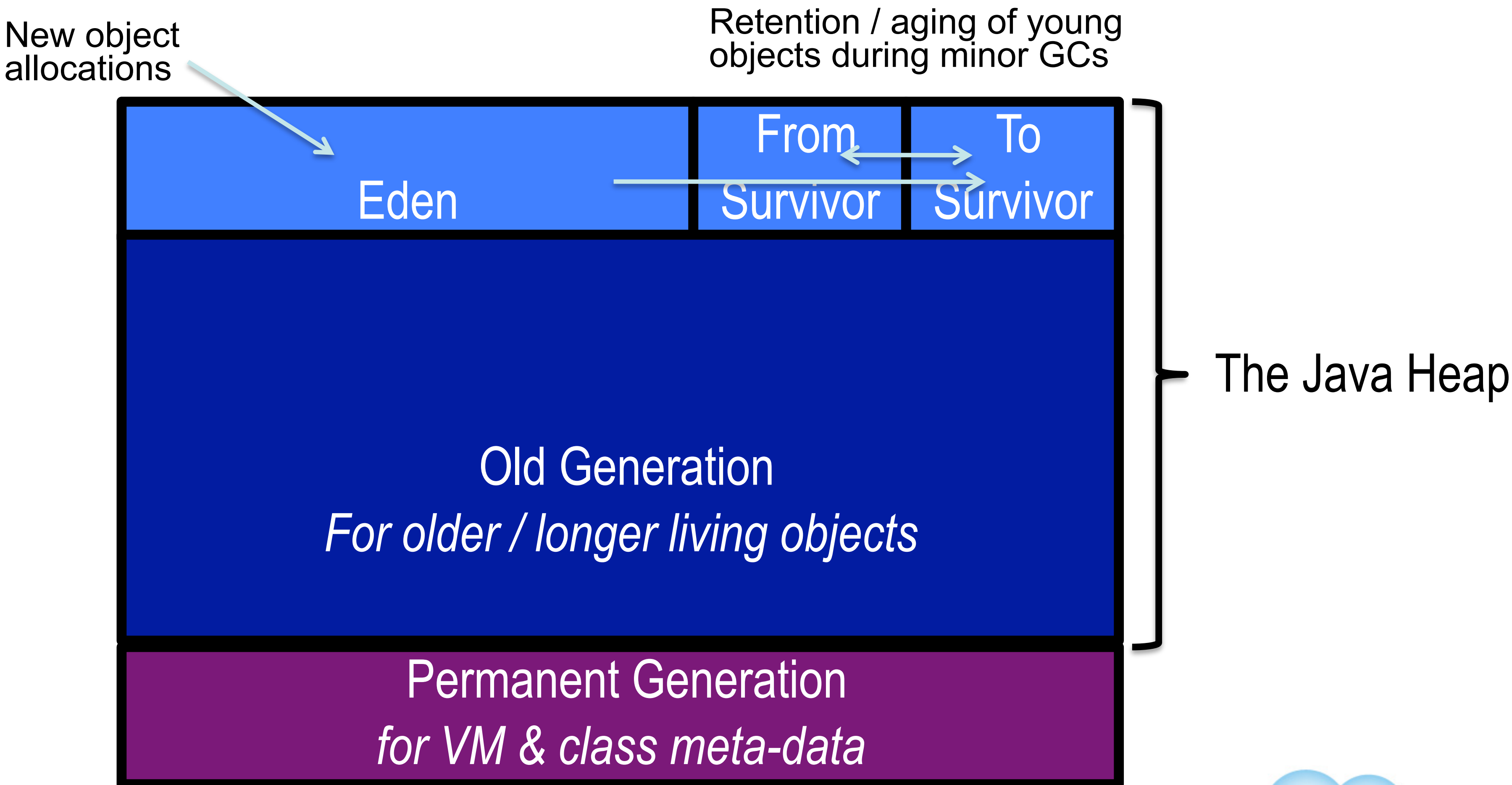
Java HotSpot VM Heap Layout



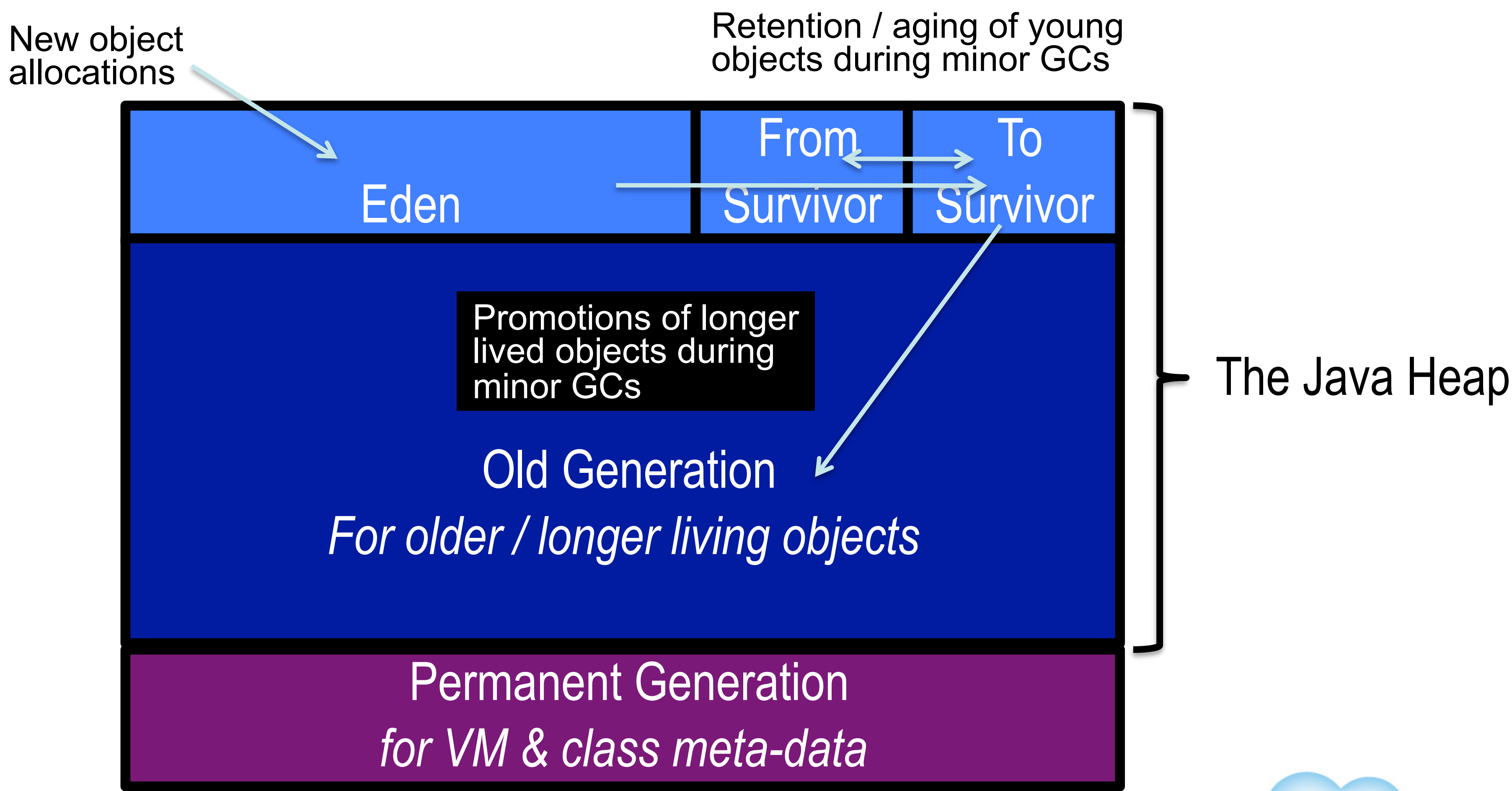
Java HotSpot VM Heap Layout



Java HotSpot VM Heap Layout



Java HotSpot VM Heap Layout



Important Concepts (1 of 4)

- Frequency of minor GC is dictated by
  - Application object allocation rate
  - Size of the eden space

Important Concepts (1 of 4)

- Frequency of minor GC is dictated by
  - Application object allocation rate
  - Size of the eden space
- Frequency of object promotion into old generation is dictated by
  - Frequency of minor GCs (how quickly objects age)
  - Size of the survivor spaces (large enough to age effectively)
    - Ideally promote as little as possible (more on this a bit later)

Important Concepts (2 of 4)

- Object retention can degrade performance more than object allocation

Important Concepts (2 of 4)

- Object retention can degrade performance more than object allocation
  - In other words, the longer an object lives, the greater the impact on throughput, latency and footprint

Important Concepts (2 of 4)

- Object retention can degrade performance more than object allocation
  - In other words, the longer an object lives, the greater the impact on throughput, latency and footprint
  - Objects retained for a longer period of time
    - Occupy available space in survivor spaces
    - May get promoted to old generation sooner than desired
    - May cause other retained objects to get promoted earlier

Important Concepts (2 of 4)

- Object retention can degrade performance more than object allocation
  - In other words, the longer an object lives, the greater the impact on throughput, latency and footprint
  - Objects retained for a longer period of time
    - Occupy available space in survivor spaces
    - May get promoted to old generation sooner than desired
    - May cause other retained objects to get promoted earlier
  - GC only visits live objects
  - GC duration is a function of the number of live objects and object graph complexity

Important Concepts (3 of 4)

- Object allocation is very cheap!
  - 10 CPU instructions in common case

Important Concepts (3 of 4)

- Object allocation is very cheap!
  - 10 CPU instructions in common case
- Reclamation of new objects is also very cheap!
  - Remember, only live objects are visited in a GC

Important Concepts (3 of 4)

- Object allocation is very cheap!
  - 10 CPU instructions in common case
- Reclamation of new objects is also very cheap!
  - Remember, only live objects are visited in a GC
- Don’t be afraid to allocate short lived objects
  - … especially for immediate results

Important Concepts (3 of 4)

- Object allocation is very cheap!
  - 10 CPU instructions in common case
- Reclamation of new objects is also very cheap!
  - Remember, only live objects are visited in a GC
- Don’t be afraid to allocate short lived objects
  - … especially for immediate results
- GCs love small immutable objects and short-lived objects
  - … especially those that seldom survive a minor GC

Important Concepts (4 of 4)

- But, don't go overboard

Important Concepts (4 of 4)

- But, don't go overboard
  - Don't do "needless" allocations

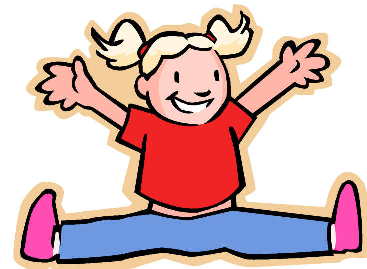
Important Concepts (4 of 4)

- But, don't go overboard
  - Don't do "needless" allocations
  - … more frequent allocations means more frequent GCs
  - … more frequent GCs imply faster object aging
  - … faster promotions
  - … more frequent needs for possibly either; concurrent old generation collection, or old generation compaction (i.e. full GC) … or some kind of disruptive GC activity

Important Concepts (4 of 4)

- But, don’t go overboard
  - Don’t do “needless” allocations
  - … more frequent allocations means more frequent GCs
  - … more frequent GCs imply faster object aging
  - … faster promotions
  - … more frequent needs for possibly either; concurrent old generation collection, or old generation compaction (i.e. full GC) … or some kind of disruptive GC activity
- It is better to use short-lived immutable objects than long-lived mutable objects

Ideal Situation



- After application initialization phase, only experience minor GCs and old generation growth is negligible
  - Ideally, never experience need for old generation collection
  - Minor GCs are (generally) the fastest GC

Advice on choosing a GC

- Start with Parallel GC (-XX:+UseParallelOldGC)
  - Parallel GC offers the fastest minor GC times
  - If you can avoid full GCs, you'll likely achieve the best throughput and lowest latency

Advice on choosing a GC

- Start with Parallel GC (-XX:+UseParallelOldGC)
  - Parallel GC offers the fastest minor GC times
  - If you can avoid full GCs, you'll likely achieve the best throughput and lowest latency
- Move to CMS or G1 if needed (for old gen collections)
  - CMS minor GC times are slower due to promotion into "free lists"
  - CMS full GC avoided via old generation concurrent collection
  - G1 minor GC times are slower due to "remembered set" overhead
  - G1 full GC avoided via concurrent collection and fragmentation avoided by "partial" old generation collection

GC Friendly Programming (1 of 3)

- Large objects
  - Expensive (in terms of time & CPU instructions) to allocate
  - Expensive to initialize (remember Java Spec ... object zeroing)

GC Friendly Programming (1 of 3)

- Large objects
  - Expensive (in terms of time & CPU instructions) to allocate
  - Expensive to initialize (remember Java Spec ... object zeroing)
- Large objects of different sizes can cause Java heap fragmentation
  - A challenge for CMS, not so much so with ParallelGC or G1

GC Friendly Programming (1 of 3)

- Large objects
  - Expensive (in terms of time & CPU instructions) to allocate
  - Expensive to initialize (remember Java Spec ... Object zeroing)
- Large objects of different sizes can cause Java heap fragmentation
  - A challenge for CMS, not so much so with ParallelGC or G1
- Advice,
  - Avoid large object allocations if you can
    - Especially frequent large object allocations during application "steady state"

GC Friendly Programming (2 of 3)

- Data Structure Re-sizing
  - Avoid re-sizing of array backed collections / containers
  - Use the constructor with an explicit size for the backing array

GC Friendly Programming (2 of 3)

- Data Structure Re-sizing
  - Avoid re-sizing of array backed collections / containers
  - Use the constructor with an explicit size for the backing array
- Re-sizing leads to unnecessary object allocation
  - Also contributes to Java heap fragmentation

GC Friendly Programming (2 of 3)

- Data Structure Re-sizing
  - Avoid re-sizing of array backed collections / containers
  - Use the constructor with an explicit size for the backing array
- Re-sizing leads to unnecessary object allocation
  - Also contributes to Java heap fragmentation
- Object pooling potential issues
  - Contributes to number of live objects visited during a GC
    - Remember GC duration is a function of live objects
  - Access to the pool requires some kind of locking
    - Frequent pool access may become a scalability issue

GC Friendly Programming (3 of 3)

- Finalizers

GC Friendly Programming (3 of 3)

- Finalizers
  - PPP-lleeeaa-ssseee don't do it!

GC Friendly Programming (3 of 3)

- Finalizers
  - PPP-lleeeaa-ssseee don't do it!
  - Requires at least 2 GCs cycles and GC cycles are slower
  - If possible, add a method to explicitly free resources when done with an object
    - Can’t explicitly free resources?
    - Use Reference Objects as an alternative
      - See JDK’s DirectByteBuffer.java implementation for an example use

GC Friendly Programming (3 of 3)

- SoftReferences

GC Friendly Programming (3 of 3)

- SoftReferences
  - PPP-lleeeaa-ssseee don't do it!

GC Friendly Programming (3 of 3)

- SoftReferences
  - PPP-lleeeaa-ssseee don't do it!
- Referent is cleared by GC
  - JVM GC's implementation determines how aggressive they are cleared
    - In other words, the JVM GC's implementation really dictates the degree of object retention
    - Remember the relationship to object retention
      - Higher object retention, longer GC pause times
      - Higher object retention, more frequent GC pauses

GC Friendly Programming (3 of 3)

- SoftReferences
  - PPP-lleeeaa-ssseee don't do it!
- Referent is cleared by GC
  - JVM GC's implementation determines how aggressive they are cleared
    - In other words, the JVM GC's implementation really dictates the degree of object retention
    - Remember the relationship between object retention
      - Higher object retention, longer GC pause times
      - Higher object retention, more frequent GC pauses
- IMO, SoftReferences == bad idea!

Subtle Object Retention (1 of 2)

```
class ClassWithFinalizer {
    protected void finalize() { // do some cleanup }
}
class MyClass extends ClassWithFinalizer {
    private byte[] buffer = new byte[1024 * 1024 * 2];
    …
```

- Object retention consequences of MyClass?

Subtle Object Retention (1 of 2)

```
class ClassWithFinalizer {
    protected void finalize() { // do some cleanup }
}
class MyClass extends ClassWithFinalizer {
    private byte[] buffer = new byte[1024 * 1024 * 2];
    …
```

- Object retention consequences of MyClass?
  - At least 2 GC cycles to free the byte[] buffer
- How to lower the object retention?

Subtle Object Retention (1 of 2)

```
class ClassWithFinalizer {
    protected void finalize() { // do some cleanup }
}
class MyClass extends ClassWithFinalizer {
    private byte[] buffer = new byte[1024 * 1024 * 2];
    …
```

- Object retention consequences of MyClass?
  - At least 2 GC cycles to free the byte[] buffer
- How to lower the object retention?

```
class MyClass {
    private ClassWithFinalizer classWithFinalizer;
    private byte[] buffer = new byte[1024 * 1024 * 2];
```

Subtle Object Retention (2 of 2)

- What about inner classes?

Subtle Object Retention (2 of 2)

- What about inner classes?
  - Remember that inner classes have an implicit reference to the outer instance
- Potentially can increase object retention
- Again, increased object retention … more live objects at GC time … increased GC duration

Fundamentals - Minor GCs

- Minor GC Frequency – How often they occur
  - Dictated by object allocation rate and size of eden space
  - Higher allocation rate or smaller eden $\Rightarrow$ more frequent minor GC
  - Lower allocation rate or larger eden $\Rightarrow$ less frequent minor GC

Fundamentals - Minor GCs

- Minor GC Frequency – How often they occur
  - Dictated by object allocation rate and size of eden space
  - Higher allocation rate or smaller eden ⇒ more frequent minor GC
  - Lower allocation rate or larger eden ⇒ less frequent minor GC
- Minor GC Pause Time
  - Dictated (mostly) by # of live objects
  - Some deviations of course, number of reference objects, object graph structure, number of promotions to old gen

Fundamentals – Full GC Frequency

- Full GC Frequency – How often they occur
  - For Parallel GC (and Serial GC)
    - Dictated by promotion rate and size of old generation space
    - Higher promotion rate or smaller old gen ⇒ more frequent full GC
    - Lower promotion rate or larger old gen ⇒ less frequent full GC

Fundamentals – Full GC Frequency

- Full GC Frequency – How often they occur
  - For Parallel GC (and Serial GC)
    - Dictated by promotion rate and size of old generation space
    - Higher promotion rate or smaller old gen ⇒ more frequent full GC
    - Lower promotion rate or larger old gen ⇒ less frequent full GC
  - For CMS & G1 – a bit more complex!
    - Dictated by promotion rate, time to execute a concurrent cycle and when the concurrent cycle is initiated – potential for "losing the race"
      - Some differences between CMS & G1 concurrent cycles
    - Also for CMS, also dictated by frequency of old gen fragmentation, a situation that requires old gen compaction via a full GC
      - G1 has shown to combat fragmentation very well

Fundamentals – Concurrent Cycle Frequency

- For CMS & G1, Concurrent Cycle Frequency
  - Dictated by the promotion rate, size of old gen and when concurrent cycle is initiated (a heap occupancy threshold)
    - CMS initiating threshold is a percent of old gen occupancy
    - G1 initiating threshold is a percent of the entire Java heap, not just old gen occupancy
  - Remember concurrent cycles execute at the same time as your application taking CPU from your application

Fundamentals – Full GC Pause Time

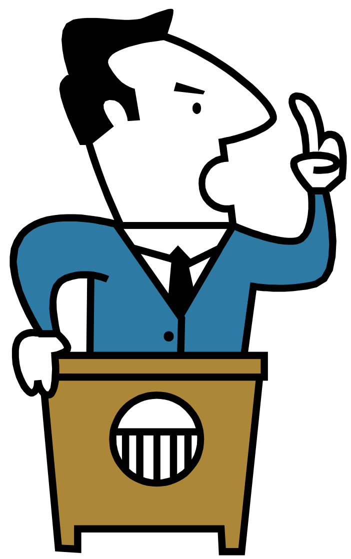
- For Parallel GC (or Serial GC)
  - Dictated (mostly) by # of live objects
  - Some deviations of course, number of reference objects, object graph structure, etc

Fundamentals – Full GC Pause Time

- For CMS or G1
  - Almost always a very lengthy pause
    - Expect a much longer pause than Parallel Old GC's full GC
  - Single threaded
    - CMS – in reaction to a promotion failure; "losing the race" (concurrent cycle did not finish in time) or fragmentation (old generation requires compaction)
    - G1 – in reaction to there not being enough space available to evacuate live objects to an available region "to-space overflow"
  - May have to "undo" reference updates due to promotion failure or to-space overflow – a time consuming operation

Agenda

- What you need to know about GC
- *What you need to know about JIT compilation*
- Tools to help you



Important Concepts

- Optimization decisions are made based on
  - Classes that have been loaded and code paths executed
  - JIT compiler does not have full knowledge of entire program
  - Only knows what has been classloaded and code paths executed

Important Concepts

- Optimization decisions are made based on
  - Classes that have been loaded and code paths executed
  - JIT compiler does not have full knowledge of entire program
  - Only knows what has been classloaded and code paths executed
  - Hence, optimization decisions makes assumptions about how a program has been executing – it knows nothing about what has not been classloaded or executed

Important Concepts

- Optimization decisions are made based on
  - Classes that have been loaded and code paths executed
  - JIT compiler does not have full knowledge of entire program
  - Only knows what has been classloaded and code paths executed
  - Hence, optimization decisions makes assumptions about how a program has been executing – it knows nothing about what has not been classloaded or executed
  - Assumptions may turn out (later) to be wrong … it must be to "recover" which (may) limit the type(s) of optimization(s)
  - New classloading or code path … possible de-opt/re-opt

Inlining and Virtualization, Completing Forces

- Greatest optimization impact realized from “method inlining”
  - Virtualized methods are the biggest barrier to inlining
    - Good news … JIT compiler can de-virtualize methods if it only sees 1 implementation of a virtualized method … effectively makes it a mono-morphic call

Inlining and Virtualization, Completing Forces

- Greatest optimization impact realized from “method inlining”
  - Virtualized methods are the biggest barrier to inlining
    - Good news … JIT compiler can de-virtualize methods if it only sees 1 implementation of a virtualized method … effectively makes it a mono-morphic call
    - Bad news … if JIT compiler later discovers an additional implementation it must de-optimize, re-optimize for 2nd implementation … now we have a bi-morphic call
    - This type of de-opt & re-opt will likely lead to lesser peak performance, especially true when / if you get to the 3rd implementation because now its a mega-morphic call

Inlining and Virtualization, Completing Forces

- Important point(s)
  - Discovery of additional implementations of virtualized methods will slow down your application
  - A mega-morphic call can limit or inhibit inlining capabilities

Inlining and Virtualization, Completing Forces

- Important point(s)
  - Discovery of additional implementations of virtualized methods will slow down your application
  - A mega-morphic call can limit or inhibit inlining capabilities
- How ‘bout writing “JIT Compiler Friendly Code” ?

Inlining and Virtualization, Completing Forces

- Important point(s)
  - Discovery of additional implementations of virtualized methods will slow down your application
  - A mega-morphic call can limit or inhibit inlining capabilities
- How ‘bout writing “JIT Compiler Friendly Code” ?
  - Ahh, that's a premature optimization!

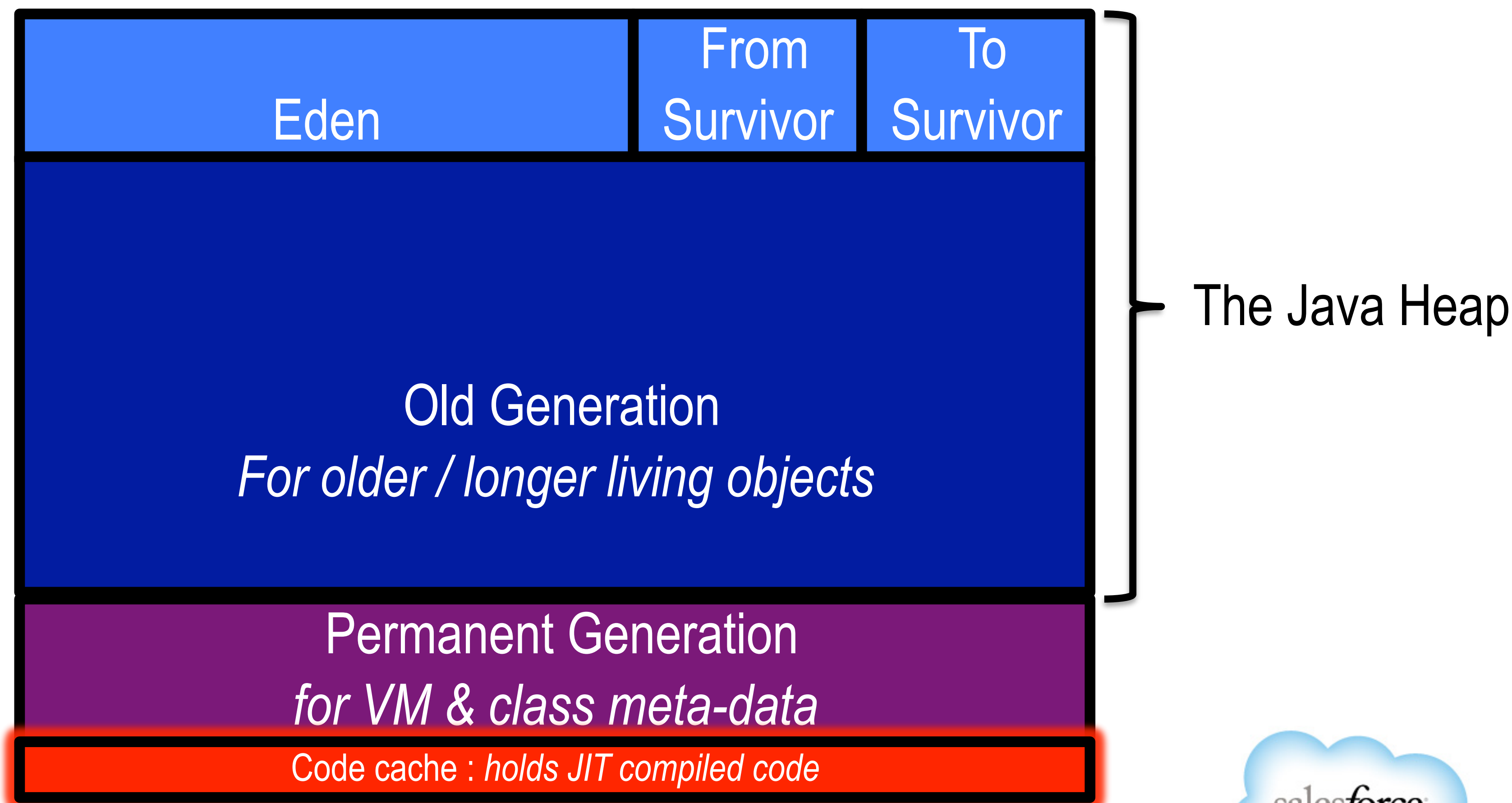
Inlining and Virtualization, Completing Forces

- Important point(s)
  - Discovery of additional implementations of virtualized methods will slow down your application
  - A mega-morphic call can limit or inhibit inlining capabilities
- How ‘bout writing “JIT Compiler Friendly Code” ?
  - Ahh, that's a premature optimization!
- Advice?

Inlining and Virtualization, Completing Forces

- Important point(s)
  - Discovery of additional implementations of virtualized methods will slow down your application
  - A mega-morphic call can limit or inhibit inlining capabilities
- How 'bout writing "JIT Compiler Friendly Code" ?
  - Ahh, that's a premature optimization!
- Advice?
  - Write code in its most natural form, let the JIT compiler agonize over how to best optimize it
  - Use tools to identify the problem areas and make code changes as necessary

Code cache, the "hidden space"



Code cache

- Default size is 48 megabytes for HotSpot Server JVM
  - Increased to 96 megabytes for Java 8
  - 32 megabytes in HotSpot Client JVMs
- Prior to Java 7, if you run out of code cache space
  - JVM prints a warning message:

    "CodeCache is full. Compiler has been disabled."

    "Try increasing the code cache size using -XX:ReservedCodeCacheSize="
- Common symptom … application mysteriously slows down after its been running for a lengthy period of time
  - Generally, more likely to see on (large) enterprise class apps

Code cache

- How to monitor code cache space
  - Can’t periodically look at code cache space occupancy with monitoring tools such as JConsole
  - JIT compiler will continue to mark code that’s no longer valid, but will not re-initiate new compilations, i.e. -XX:+PrintCompilation shows “made not entrant” and “made zombie”, but not new activations
    - So, code cache could look like it has available space via JConsole when in reality it is exhausted – can be very misleading!

Code cache

- Advice
  - Profile app with profiler that also profiles the internals of the JVM
    - Look for high JVM Interpreter CPU time
  - Check log files for log message saying code cache is full
  - Use -XX:+UseCodeCacheFlushing (Java 6u* releases & later)
    - Will evict least recently used code from code cache
    - Possible for compiler thread to cycle (optimize, throw away, optimize, throw away), but that's better than disabled compilation
  - Best option, increase -XX:ReservedCodeCacheSize, or do both +UseCodeCacheFlushing & increase ReservedCodeCacheSize

Code cache

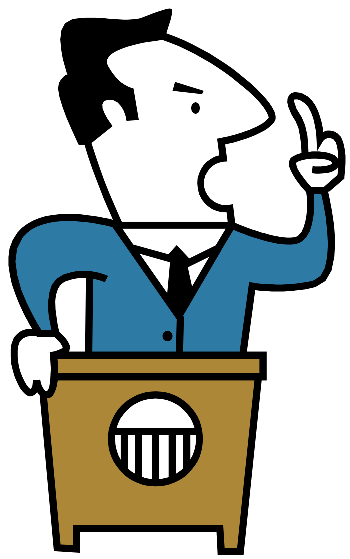
- Java 7 and forward
  - -XX:+UseCodeCacheFlushing is on by default
    - But, “flushing” may be an intrusive operation for the JIT compiler if there are a lot of additional demands made on it, i.e. new activations, code invalidations
    - May need to tune -XX:CodeCacheMinimumFreeSpace and -XX:MinCodeCacheFlushingInterval

Code cache

- Java 7 and forward
  - -XX:+UseCodeCacheFlushing is on by default
    - But, “flushing” may be an intrusive operation for the JIT compiler if there are a lot of additional demands made on it, i.e. new activations, code invalidations
    - May need to tune -XX:CodeCacheMinimumFreeSpace and -XX:MinCodeCacheFlushingInterval
  - Advice
    - Profile with a profiler that also profiles JVM internals and look for high amounts of CPU used in code cache flushing
    - Best option, increase -XX:ReservedCodeCacheSize, tune code cache flushing as a secondary activity

Agenda

- What you need to know about GC
- What you need to know about JIT compilation
- *Tools to help you*



GC Analysis Tools

- Offline mode, after the fact
  - GCHisto or GCViewer (search for “GCHisto” or “chewiebug GCViewer”) – both are GC log visualizers
  - Recommend -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps or -XX:+PrintGCDateStamps JVM command line options

GC Analysis Tools

- Offline mode, after the fact
  - GCHisto or GCViewer (search for “GCHisto” or “chewiebug GCViewer”) – both are GC log visualizers
  - Recommend -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps or -XX:+PrintGCDateStamps JVM command line options
- Online mode, while application is running
  - VisualGC plug-in for VisualVM (found in JDK’s bin directory, launched as 'jvisualvm’ – then install VisualGC plug-in)

GC Analysis Tools

- Offline mode, after the fact
  - GCHisto or GCViewer (search for “GCHisto” or “chewiebug GCViewer”) – both are GC log visualizers
  - Recommend -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps or -XX:+PrintGCDateStamps JVM command line options
- Online mode, while application is running
  - VisualGC plug-in for VisualVM (found in JDK’s bin directory, launched as 'jvisualvm’ – then install VisualGC plug-in)
- VisualVM or Eclipse MAT for unnecessary object allocation and object retention

JIT Compilation Analysis Tools

- Command line tools
  - -XX:+PrintOptoAssembly
    - Requires “fastdebug JVM”, can be built from OpenJDK sources
    - Offers the ability to see generated assembly code with Java code
    - Lots of output to digest

JIT Compilation Analysis Tools

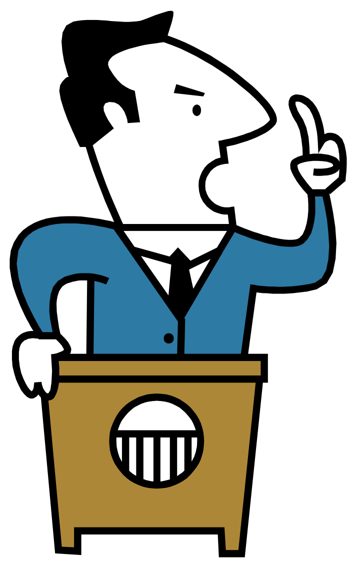
- Command line tools
  - -XX:+PrintOptoAssembly
    - Requires “fastdebug JVM”, can be built from OpenJDK sources
    - Offers the ability to see generated assembly code with Java code
    - Lots of output to digest
  - -XX:+LogCompilation
    - Must add -XX:+UnlockDiagnosticVMOptions, but “fastdebug JVM” not required
    - Produces XML file that shows the path of JIT compiler optimizations
    - Non-trivial to read and understand
    - Search for “HotSpot JVM LogCompilation” for more details

JIT Compilation Analysis Tools

- GUI Tools
  - Oracle Solaris Studio Performance Analyzer (my favorite)
    - Works with both Solaris and Linux (x86/x64 & SPARC)
    - Better experience on Solaris (more mature, ported to Linux a couple years ago, and no CPU microstate info on Linux)
    - See generated JIT compiler code embedded with Java source
    - Free download (search for “Studio Performance Analyzer”)
    - Excellent method profiler, lock profiler and hardware counter profiler (i.e. CPU cache misses, TLB misses, instructions retired, etc.)
  - Similar tools
    - Intel VTune
    - AMD CodeAnalyst

Agenda

- What you need to know about GC
- What you need to know about JIT compilation
- Tools to help you



Acknowledgments

- Special thanks to Tony Printezis and John Coomes. Some of the GC related material, especially the "GC friendly", is material originally drafted by Tony & John [1]
- And thanks to Tom Rodriguez and Vladimir Kozlov for sharing their HotSpot JIT compiler expertise and advice

[1]: *Garbage Collection Friendly Programming*. Printezis, Coomes, 2007 JavaOne Conference, San Francisco, CA

Additional Reading Material

- *Java Performance*. Hunt, John. 2011
  - High level overview of how the Java HotSpot VM works including both JIT compiler and GC along with “step by step” JVM tuning
- *The Garbage Collection Handbook*. Jones, Hosking, Moss. 2012
  - Just about anything and everything you’d ever want to know about GCs, (used in any programming language)
- *Sea of Nodes Compilation Approach*. Chang. 2010, http://www.masonchang.com/blog/2010/8/9/sea-of-nodes-compilation-approach.html
  - A summary of the compilation approach used by Java HotSpot VM’s server (JIT) compiler

Thank you!