Extreme Performance with Java

QCon NYC - June 2012

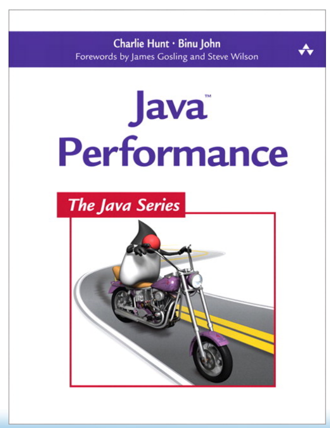
Charlie Hunt
Architect, Performance Engineering
Salesforce.com

In a Nutshell

What you need to know about a modern JVM in order to be effective at writing a low latency Java application.

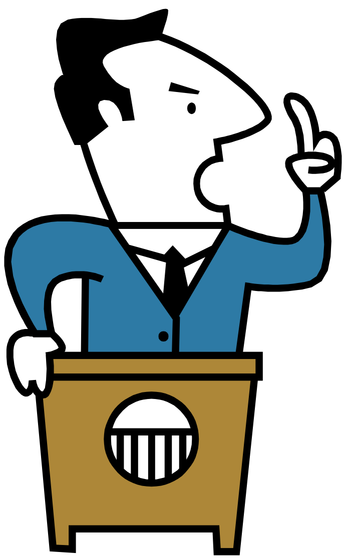
Who is this guy?

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



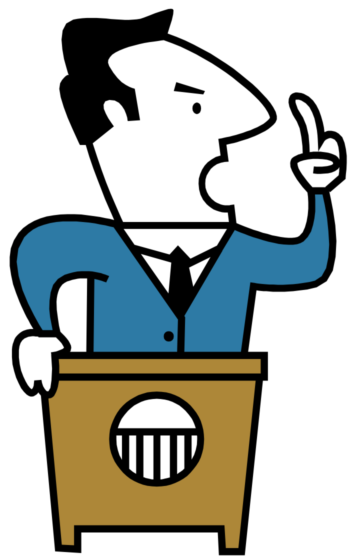
Agenda

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

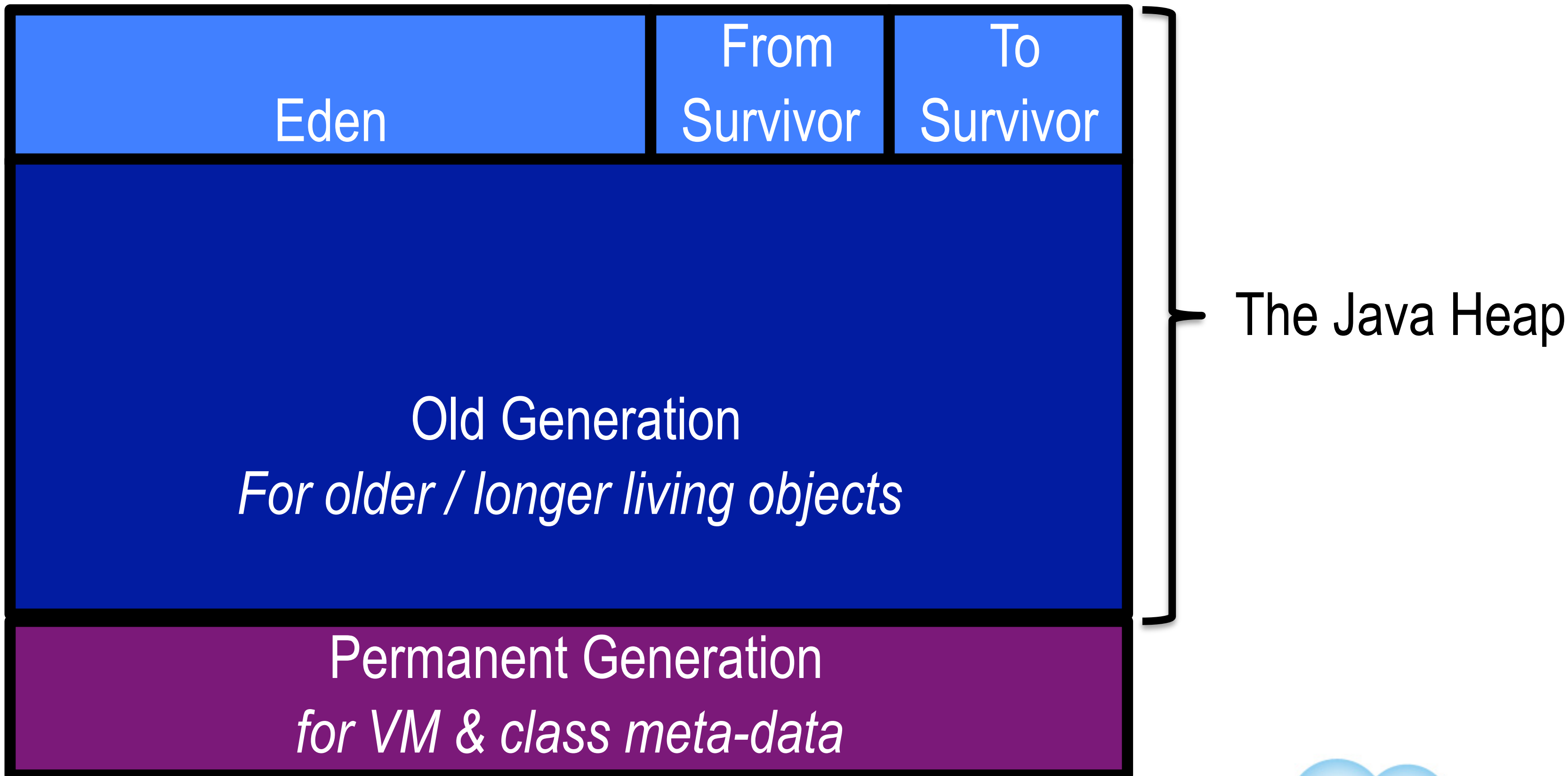


Agenda

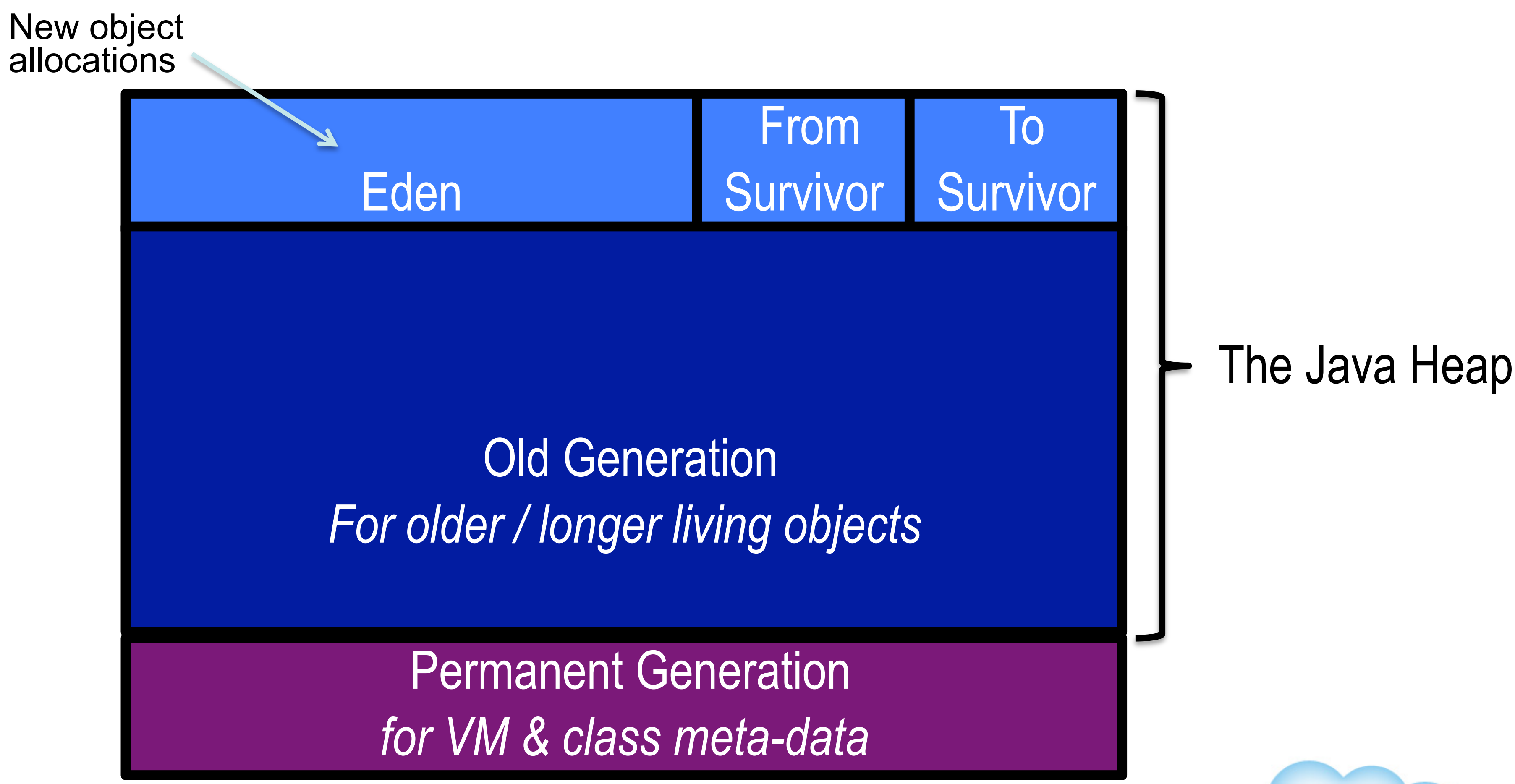
- *What you need to know about GC*
- What you need to know about JIT compilation
- Tools to help you



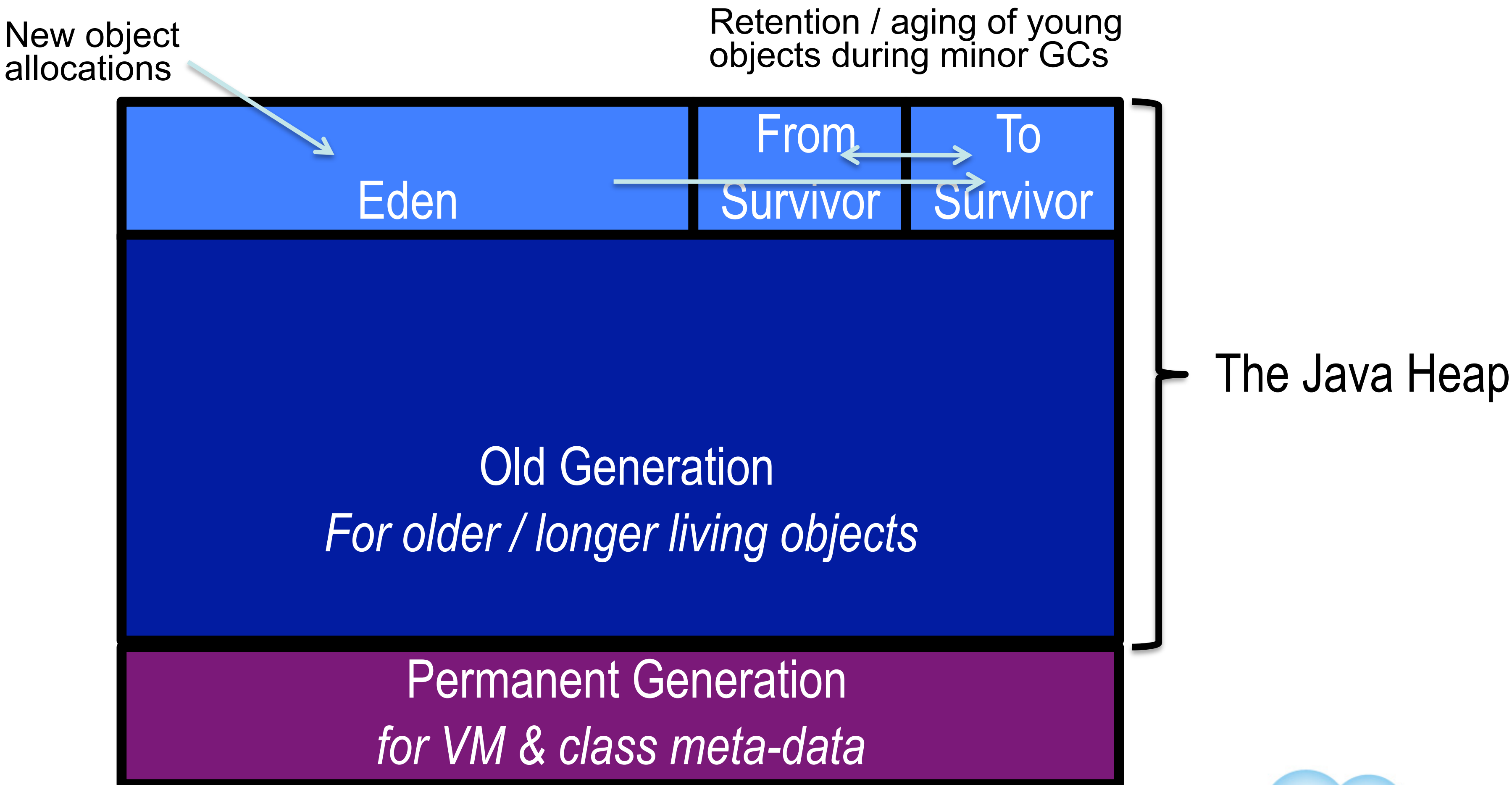
Java HotSpot VM Heap Layout



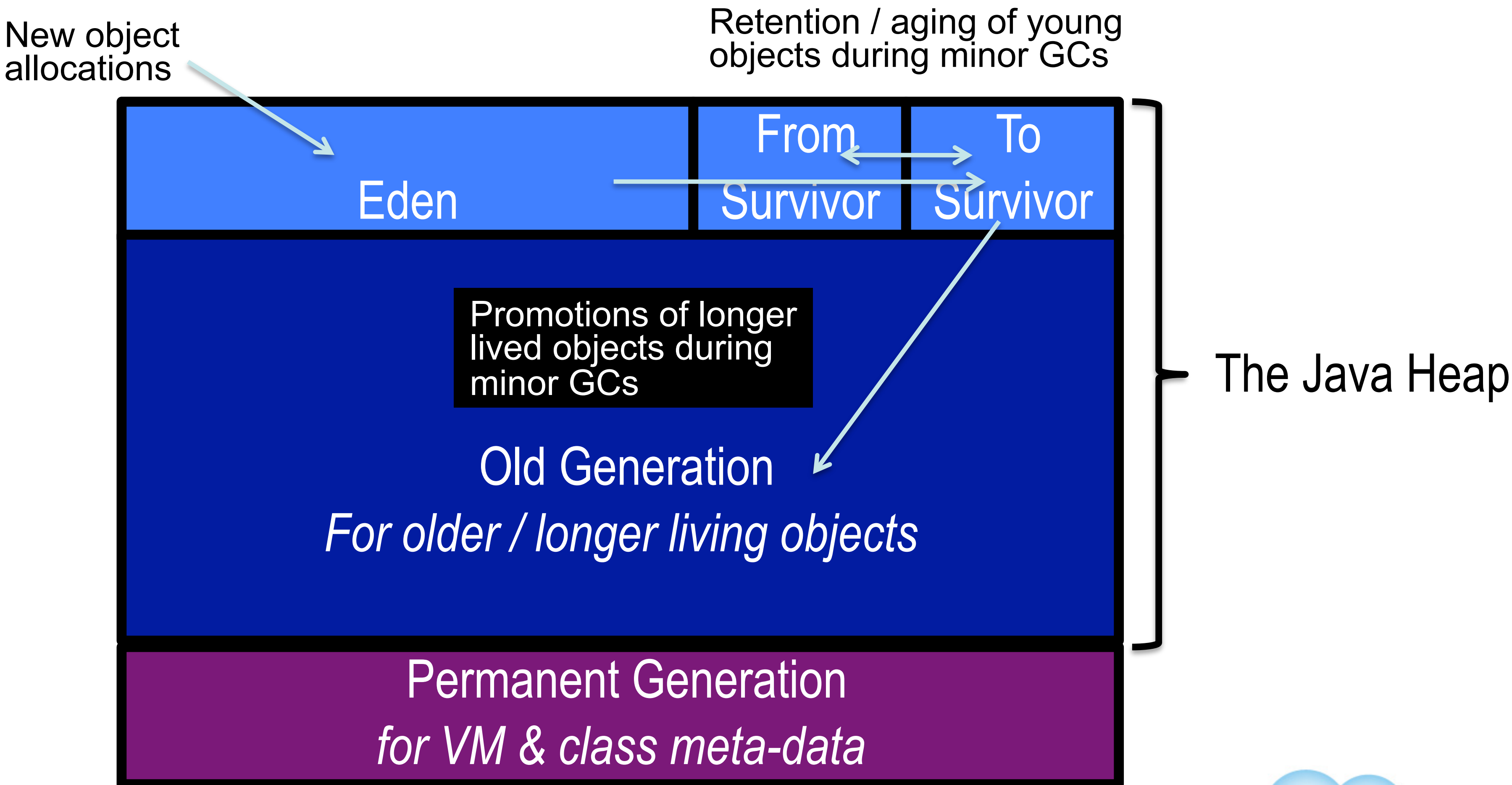
Java HotSpot VM Heap Layout



Java HotSpot VM Heap Layout



Java HotSpot VM Heap Layout



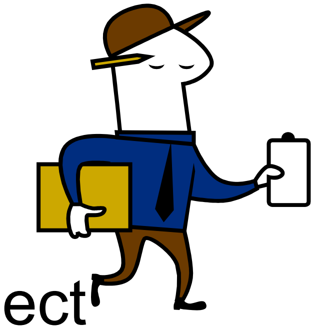
Important Concepts (1 of 4)

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

Important Concepts (1 of 4)

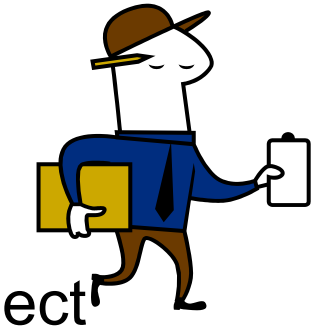
- Frequency of minor GC is dictated by
  - Application object allocation rate
  - Size of the eden space
- Frequency of object promotion into old generation is dictated by
  - Frequency of minor GCs (how quickly objects age)
  - Size of the survivor spaces (large enough to age effectively)
    - Ideally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

Important Concepts (2 of 4)



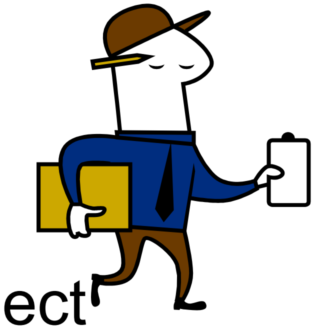
- Object retention impacts latency more than object allocation

Important Concepts (2 of 4)

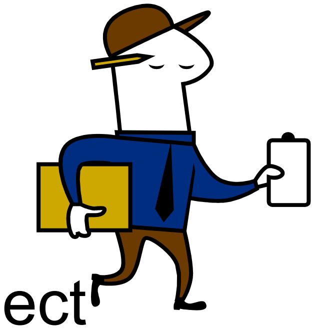


- Object retention impacts latency more than object allocation
  - In other words, the longer an object lives, the greater the impact on latency

Important Concepts (2 of 4)



- Object retention impacts latency more than object allocation
  - In other words, the longer an object lives, the greater the impact on latency
  - 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 impacts latency more than object allocation
  - In other words, the longer an object lives, the greater the impact on latency
  - 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

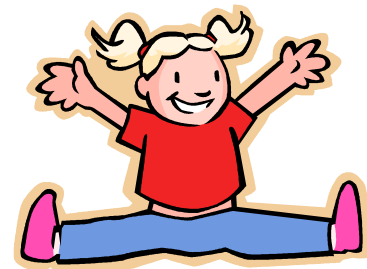
Important Concepts (4 of 4)

- 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:+UseParallel[Old]GC)
  - Parallel GC offers the fastest minor GC times
  - If you can avoid full GCs, you'll likely achieve the best throughput, smallest footprint and lowest latency

Advice on choosing a GC

- Start with Parallel GC (-XX:+UseParallel[Old]GC)
  - Parallel GC offers the fastest minor GC times
  - If you can avoid full GCs, you'll likely achieve the best throughput, smallest footprint 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 DirectByteBuffer.java)

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 between 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)

- Consider the following:

```
class MyImpl extends ClassWithFinalizer {
    private byte[] buffer = new byte[1024 * 1024 * 2];
    ....
```

- What's the object retention consequences if ClassWithFinalizer has a finalizer?

Subtle Object Retention (1 of 2)

- Consider the following:

```
class MyImpl extends ClassWithFinalizer {
    private byte[] buffer = new byte[1024 * 1024 * 2];
    ....
```

- What's the object retention consequences if ClassWithFinalizer has a finalizer?
  - At least 2 GC cycles to free the byte[] buffer
- How to lower the object retention?

Subtle Object Retention (1 of 2)

- Consider the following:

```
class MyImpl extends ClassWithFinalizer {
    private byte[] buffer = new byte[1024 * 1024 * 2];
    ....
```

- What's the object retention consequences if ClassWithFinalizer has a finalizer?
  - At least 2 GC cycles to free the byte[] buffer
- How to lower the object retention?

```
class MyImpl {
    private ClassWithFinalier classWithFinalizer;
    private byte[] buffer = new byte[1024 * 1024 * 2];
```

Subtle Object Retention (2 of 2)

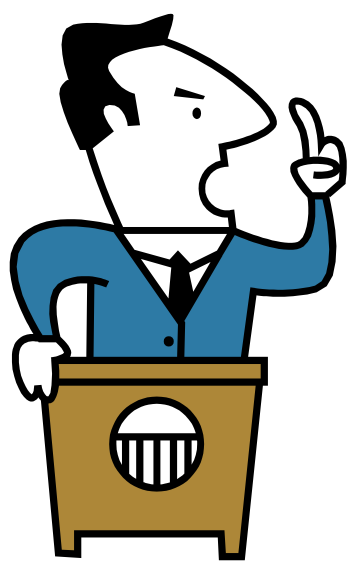
- What about inner classes?

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

Agenda

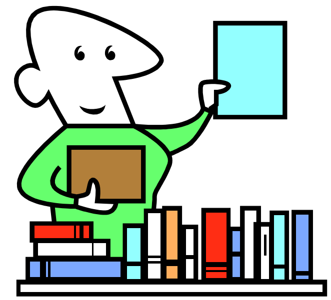
- What you need to know about GC
- *What you need to know about JIT compilation*
- Tools to help you



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 keep information around to “recover” which (may) limit 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!

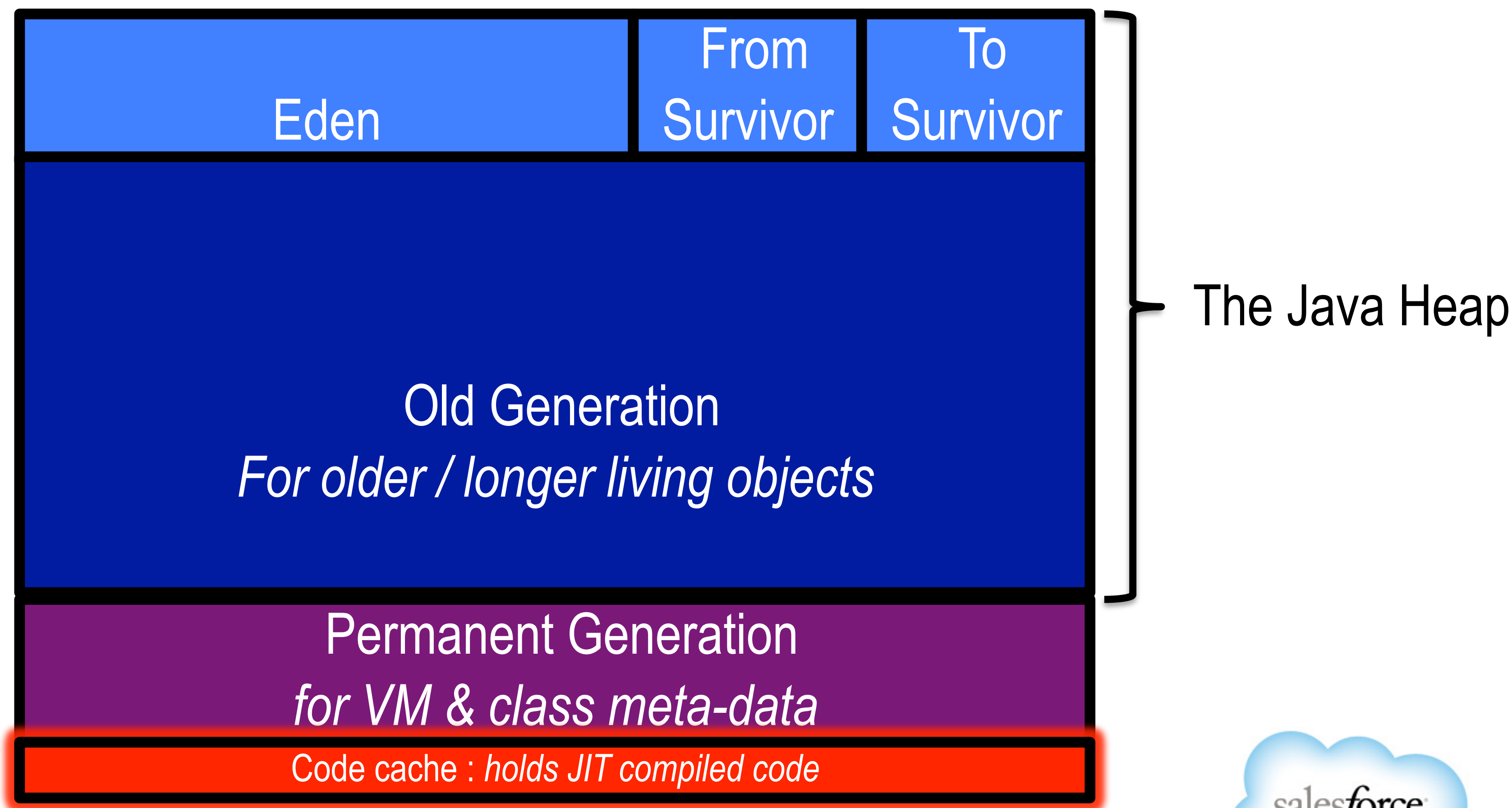
Inlining and Virtualization, Completing Forces

- 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 figure out 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
  - 32 megabytes for HotSpot Client JVM
- 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 enterprise class apps

Code cache

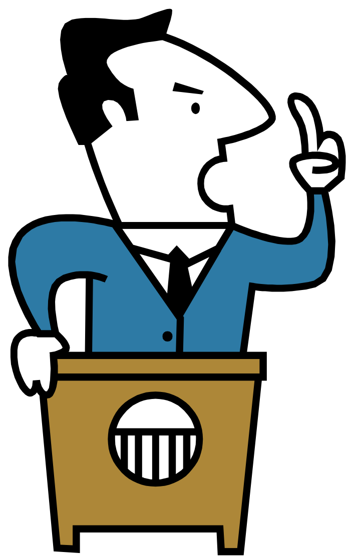
- How to monitor code cache space
  - Can’t merely periodically look at code cache space occupancy in JConsole
  - JIT compiler will throw out code that’s no longer valid, but will not re-initiate new compilations, i.e. -XX:+PrintCompilation shows “made not entrant” and “made zombie”, but not new activations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

Code cache

- Advice
  - Profile app with profiler that also profiles the JVM
    - Look for high JVM Interpreter CPU time
  - Check log files for log message saying code cache is full
  - Use -XX:+UseCodeCacheFlushing on recent Java 6 and Java 7 Update releases
    - 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 +UseCodeCacheFlusing & increase ReservedCodeCacheSize

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
- Online mode, while application is running
  - VisualGC plug-in for VisualVM (found in JDK’s bin directory, launched as 'jvisualvm')
- VisualVM or Eclipse MAT for unnecessary object allocation and object retention

JIT Compilation Analysis Tools

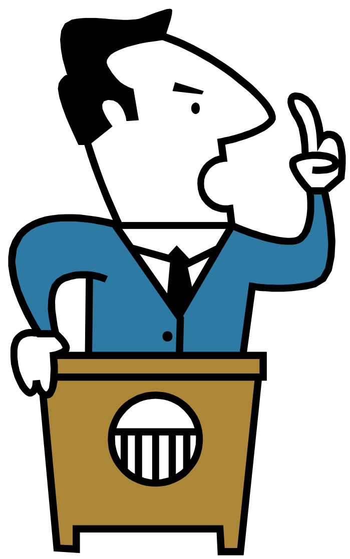
- Command line tools
  - -XX:+PrintOptoAssembly
    - Requires “debug JVM”, can be built from OpenJDK sources
    - Offers the ability to see generated assembly code with Java code
    - Lots of output to digest
  - -XX:+LogCompilation
    - Must add -XX:+UnlockDiagnosticVMOptions, but “debug JVM” not required
    - Produces XML file that shows the path of JIT compiler optimizations
    - Very, very difficult 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, port to Linux fairly recent, some issues observed on Linux x64)
    - See generated JIT compiler code embedded with Java source
    - Free download (search for "Studio Performance Analyzer")
    - Also a method profiler, lock profiler and profile by CPU hardware counter
  - 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. Much of the GC related material, especially the “GC friendly”, is material originally drafted by Tony and John
- And thanks to Tom Rodriguez and Vladimir Kozlov for sharing their HotSpot JIT compiler expertise and advice

Additional Reading Material

- *Java Performance*. Hunt, John. 2012
  - High level overview of how the Java HotSpot VM works including both JIT compiler and GC along with many other “goodies”
- *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)

Thank you!

ADM

Adaptive Delivery Methodology