JVM Deep Dive

Daniel Mitterdorfer, comSysto GmbH
@dmitterd

Behold! It will get scary.

Topics

- Illusions by (J)VMs
- Interpreter
- JIT Compiler
- Memory

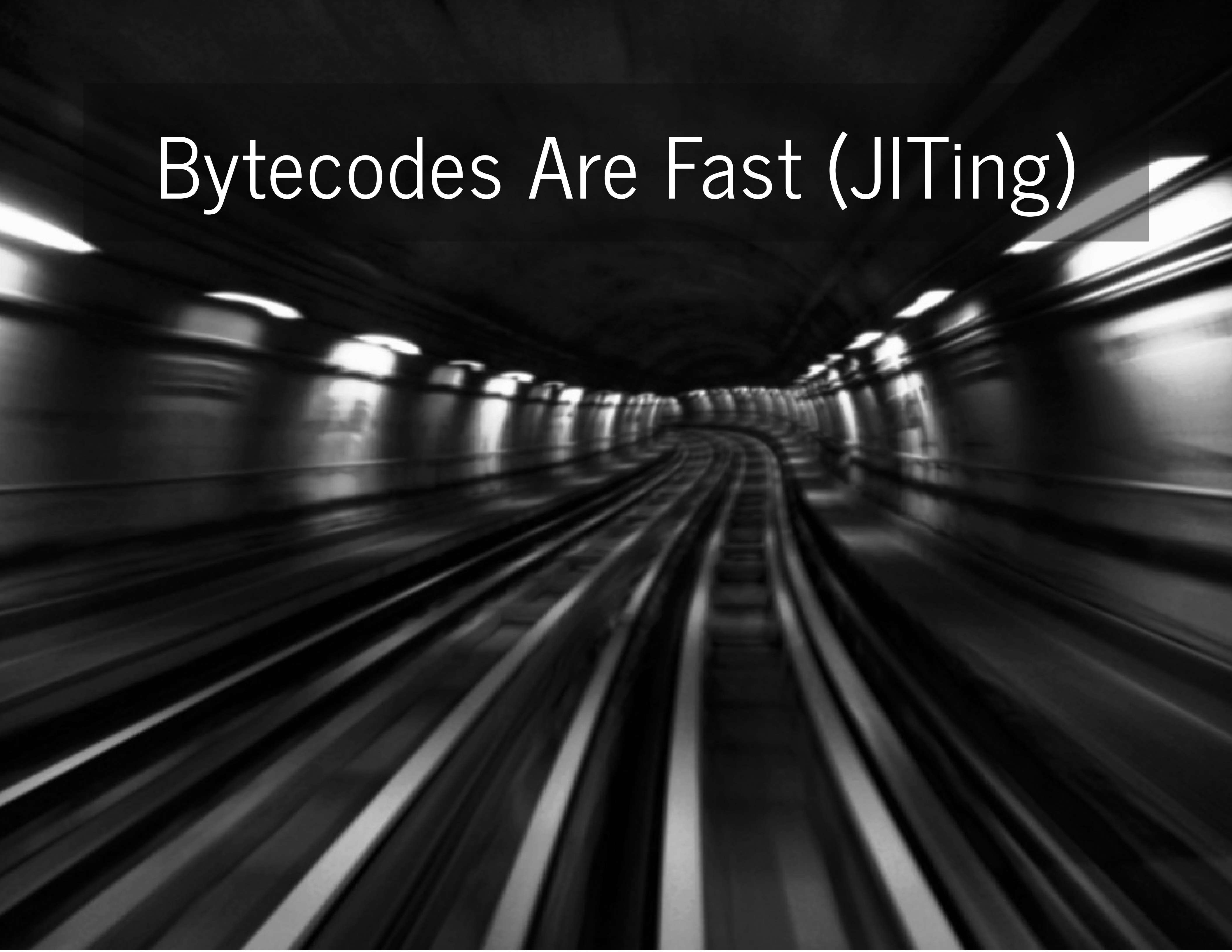
Illusions

Based on A JVM Does That???

Write Once, Run Anywhere

- One "Binary" for All Platforms
- Consistent Memory Model (Java Memory Model)
- Consistent Thread Model

Bytecodes Are Fast (JITing)



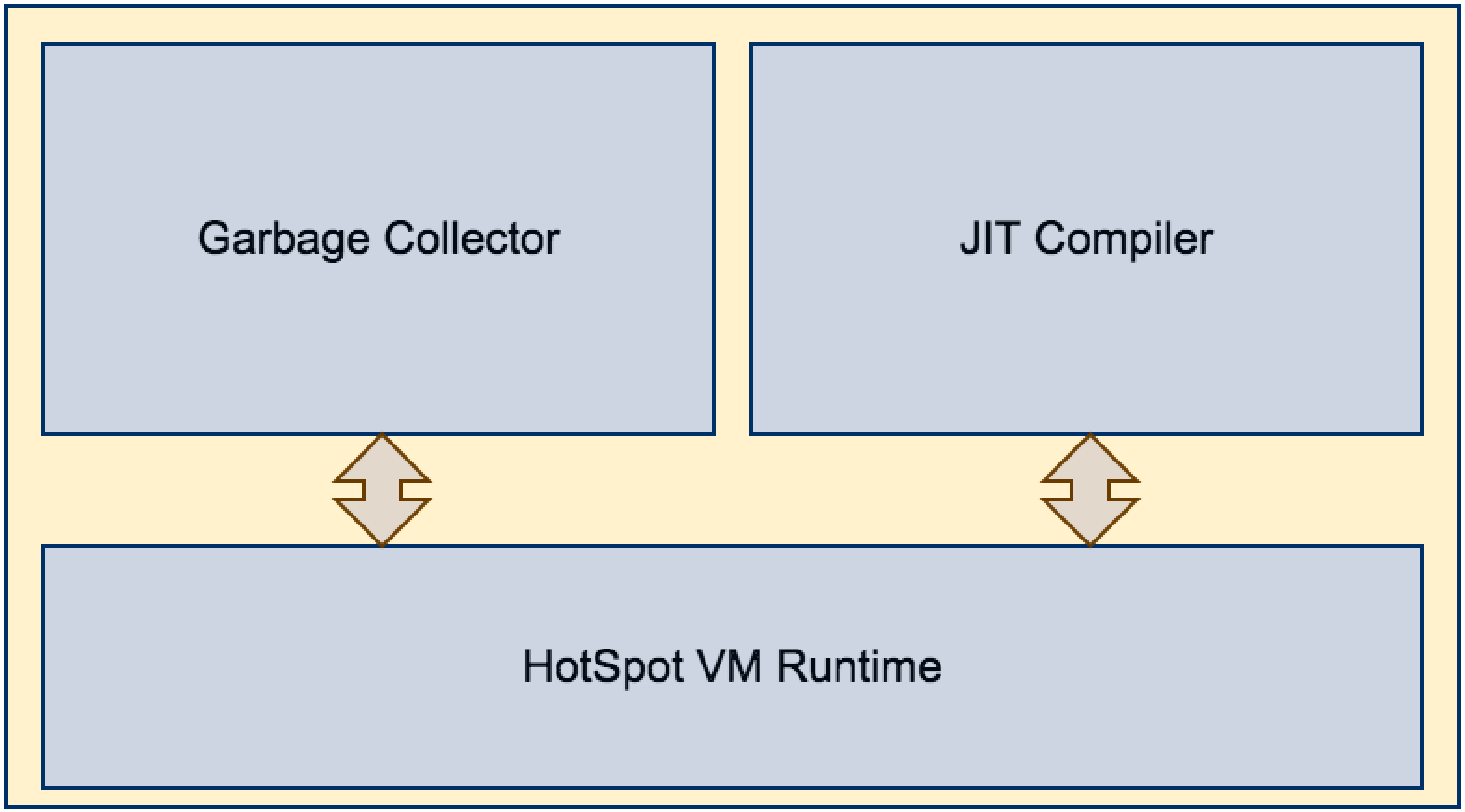
Infinite Heap (Garbage Collection)

What ”is” a JVM?

The JVM is specified in The Java® Virtual Machine Specification.
There are multiple implementations:

- HotSpot
  JVM reference implementation; part of OpenJDK and Oracle JDK
- Azul Zing
  Commercial performance optimized JVM based on HotSpot with a low-pause GC (called C4) and many other features
- J9
  Implementation by IBM
- JRockit
  Implementation by Bea. Now integrated into HotSpot.
- ...

Internal Structure of the Hot-Spot JVM



Based on "Java Performance", p. 56

Let’s start simple

What happens between...

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

... and ...

```
Hello World!
```

”Compile”

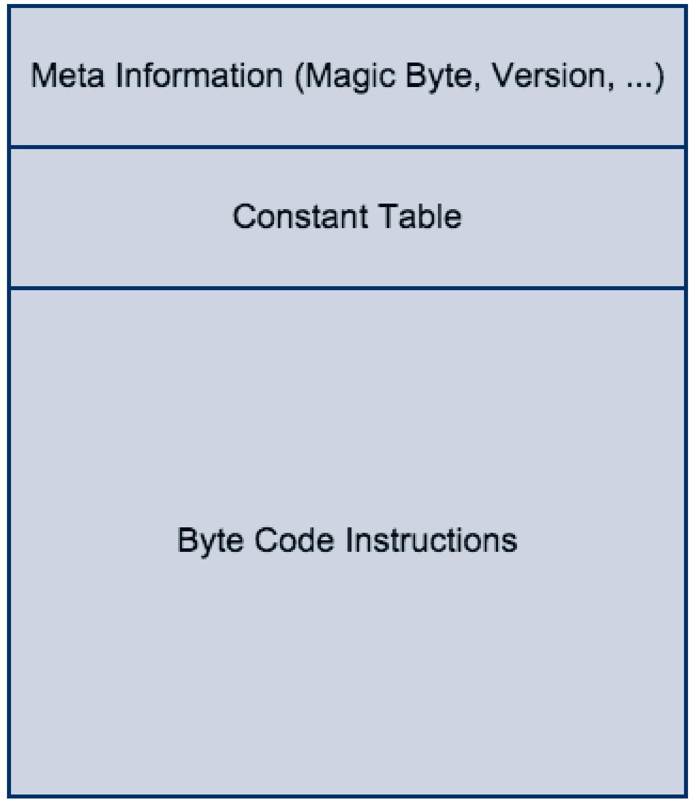
```
javac HelloWorld.java
```

HelloWorld.class Hexdumped

```
0000000 ca fe ba be 00 00 00 34 00 1d 0a 00 06 00 0f 0
0000010 00 10 00 11 08 00 12 0a 00 13 00 14 07 00 15 0
0000020 00 16 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 2
0000030 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4
0000040 75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 6
0000050 6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 6
0000060 2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 7
0000070 72 63 65 46 69 6c 65 01 00 0f 48 65 6c 6c 6f 5
0000080 6f 72 6c 64 2e 6a 61 76 61 0c 00 07 00 08 07 0
0000090 17 0c 00 18 00 19 01 00 0c 48 65 6c 6c 6f 20 5
00000a0 6f 72 6c 64 21 07 00 1a 0c 00 1b 00 1c 01 00 0
00000b0 48 65 6c 6c 6f 57 6f 72 6c 64 01 00 10 6a 61 7
00000c0 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 01 00 1
00000d0 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6
00000e0 01 00 03 6f 75 74 01 00 15 4c 6a 61 76 61 2f 6
00000f0 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 01 0
0000100 13 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 7
0000110 72 65 61 6d 01 00 07 70 72 69 6e 74 6c 6e 01 0
0000120 15 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 7
0000130 69 6e 67 3b 29 56 00 21 00 05 00 06 00 00 00 0
0000140 00 02 00 01 00 07 00 08 00 01 00 09 00 00 00 1
0000150 00 01 00 01 00 00 00 05 2a b7 00 01 b1 00 00 0
0000160 01 00 0a 00 00 00 06 00 01 00 00 00 01 00 09 0
0000170 0b 00 0c 00 01 00 09 00 00 00 25 00 02 00 01 0
0000180 00 00 09 b2 00 02 12 03 b6 00 04 b1 00 00 00 0
0000190 00 0a 00 00 00 0a 00 02 00 00 00 03 00 08 00 0
00001a0 00 01 00 0d 00 00 00 02 00 0e
00001aa
```

Welcome to the Matrix

Structure of a `.class` file



Beware: This is almost criminally simplified.

Demo

```
javap -verbose -c HelloWorld.class
```

The JVM: A stack-based machine

```
int sum = op0 + op1;
```

↓

```
20: iload_1
21: iload_2
22: iadd
23: istore_3
```

Bytecode Execution: Straightforward

```
//pseudocode
for(;;) {
  current_byte_code = read_byte_code_at(program_counte
  switch(current_byte_code) {
    case iadd: handle_iadd(); break;
    case iload_1: handle_iload_1(); break;
    // ...
  }
}
```

Bytecode Execution: Faster

1. Generate assembler code at startup for each bytecode
2. Execute generated code for each bytecode

Better optimized for current hardware, no more bytecode dispatching in C++

Example: Generated code for `iadd`

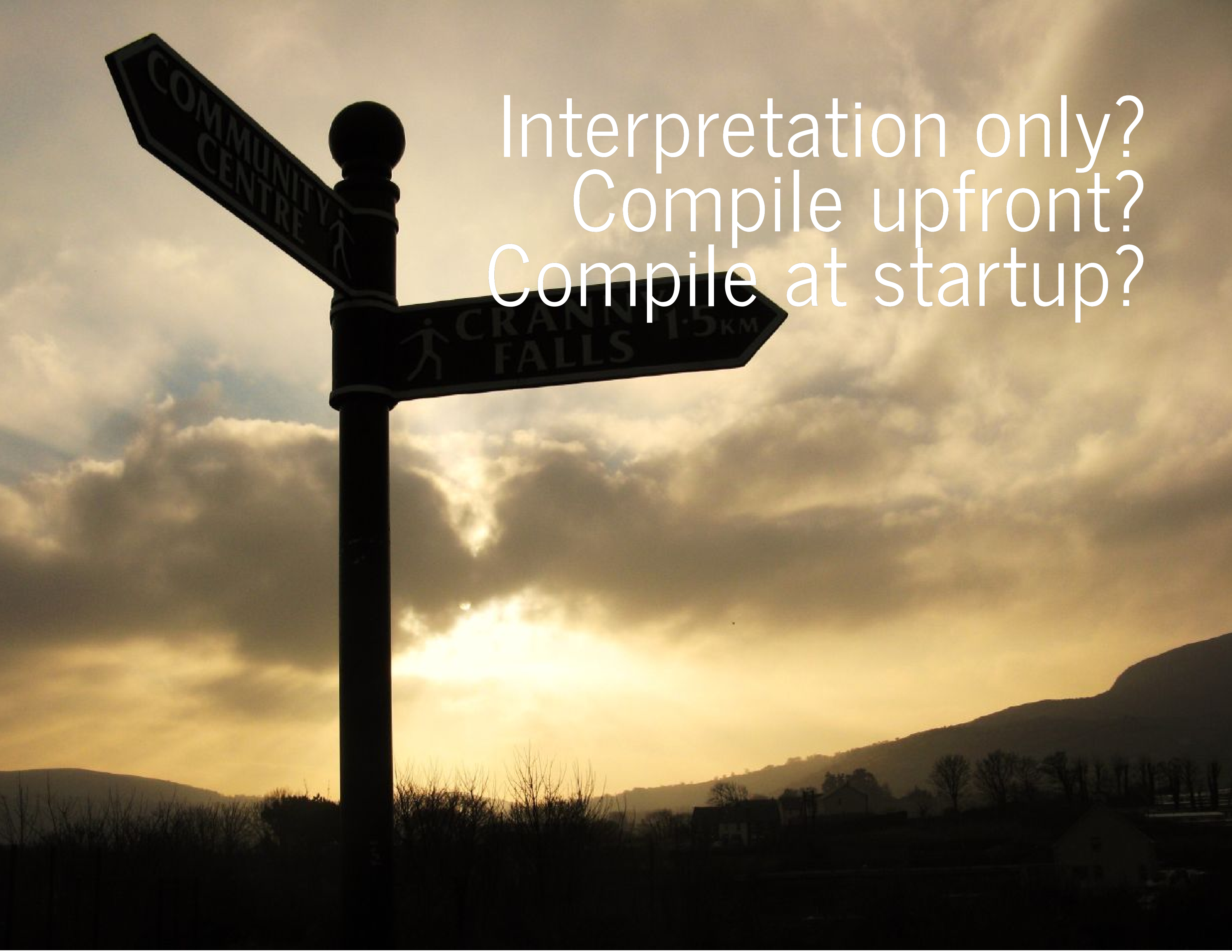
```
mov    eax,DWORD PTR [rsp]      ; take parameters from s
add    rsp,0x8
mov    edx,DWORD PTR [rsp]
add    rsp,0x8
add    eax,edx                  ; add parameters
movzx  ebx,BYTE PTR [r13+0x1]   ; dispatch next byte cod
inc    r13
movabs r10,0x109c72270
jmp    QWORD PTR [r10+rbx*8]
```

Slightly simplified

Take Aways

- `javac` produces `.class` files which reflect the Java code
- `.class` files contain platform independent byte codes
- Look at byte codes with `javap`
- The interpreter is a complex beast

JIT compilation


Interpretation only?
Compile upfront?
Compile at startup?
COMMUNITY CENTRE
FALLS
1·5KM

JIT Compilation

- Just In Time
- "Profile-guided" optimization
- Compile only hot code paths ("hot spots")

Triggering a Compilation

Based on interpreter events. Overflow of:

- Method invocation counter (methods)
- Backedge counter (loop invocations)

JIT Compilation Strategies

- Client Compiler (C1)
  Faster startup, less compilation overhead, less optimizations
- Server Compiler (C2)
  Takes time, more aggressive optimizations
- Tiered Compilation
  First compile with C1, then with C2. Active by default, deactivate with `-XX:-TieredCompilation`

Runtime Profiling

- Invariants: Loaded classes
- Statistics: Branches taken
- ...

Common Optimizations

- Dead Code Elimination
- Method Inlining
- Class Hierarchy Analysis
- Lock elision/coarsening
- Loop transformations

... and many more

Intrinsics

Hand-optimized "shortcuts" for certain Java methods

Example: `Math#abs(double)`

```
return (a <= 0.0D) ? 0.0D - a : a;
```

`Math#abs(double)` as Bytecode

```
0: dload_0
1: dconst_0
2: dcmpg
3: ifgt 12
6: dconst_0
7: dload_0
8: dsub
9: goto 13
12: dload_0
13: dreturn
```

x86 Intrinsics

`Math.abs(double)`

↓

```
andpd $dst, [0x7fffffffffffffff]
```

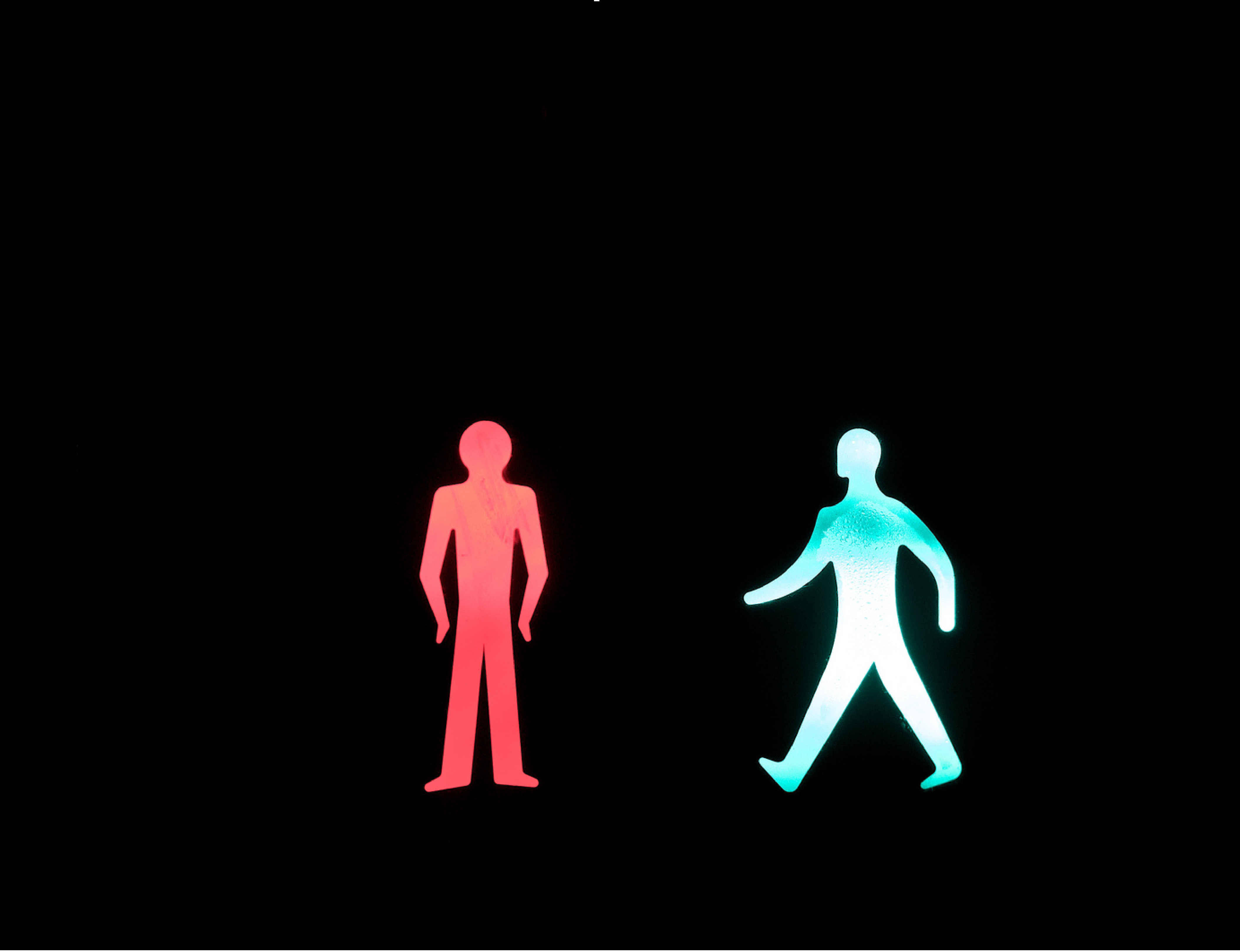
JIT Compilation Strategy

- Optimize aggressively based on current runtime profile
- Deoptimization: Revert to interpretation on violated assumptions

Constant back and forth between interpreter and JIT compiler

Some Reasons for Deoptimization

- Unexpected `null` encountered
- Method is too old



Safepoints

How to "remove" compiled machine code given that multiple threads are constantly in flight?

1. Halt *every* application thread in the JVM ("safepoint")
2. Replace machine code with interpreted code

Safepoints

Safepoints are used for different tasks in the JVM, for example:

- Garbage Collection
- Thread Dumps
- Deadlock Detection
- Revocation of Biased Locking

Embrace the JIT

- Use short methods for readability (inlining)
- Use standard library methods (may use intrinsics)
- Use inheritance but take care in performance critical code

Inspecting Compilation

- Use `-XX:+PrintCompilation`
- Use JIT Watch

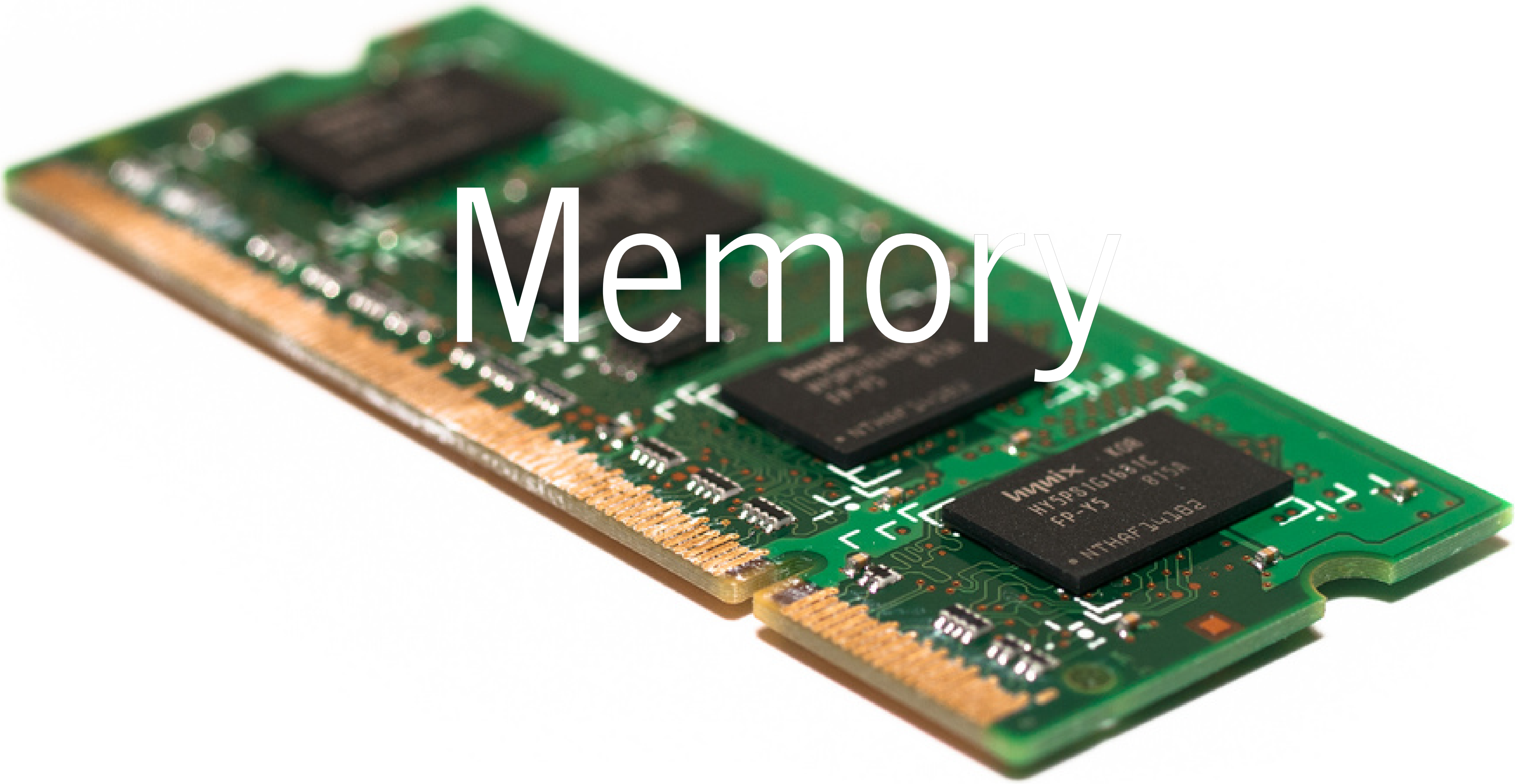
Demo

Intrinsics demo

Take Aways

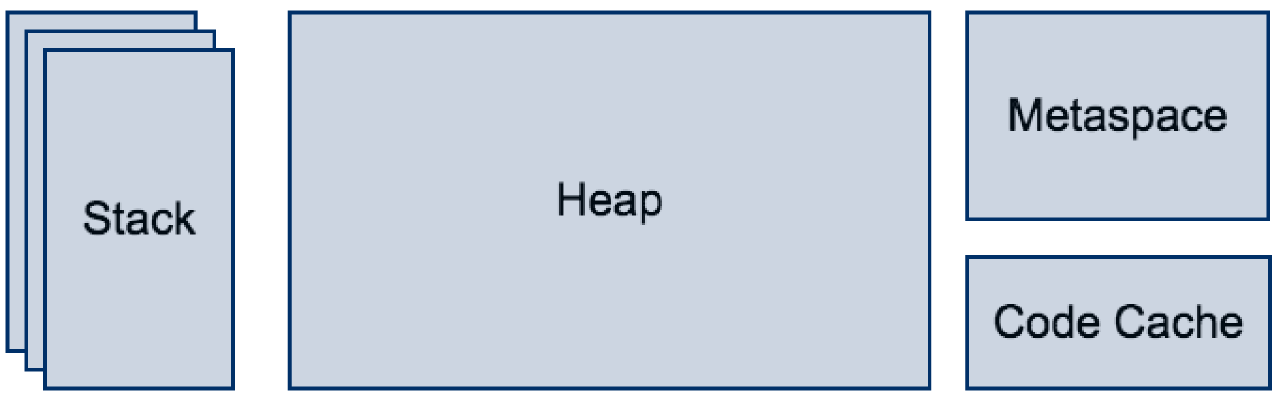
- JIT compilation makes Java code fast
- JIT compilation relies on runtime information
- Cooperation needed between runtime, interpreter and JIT compiler

Memory



Memory Regions

- Stack
  Each Java thread has its own stack
- Heap
  One heap for each Java process
- Metaspace (Java 8+)
  contains class data; native memory, grows unlimited by default
- Code Cache
  contains JIT compiled code



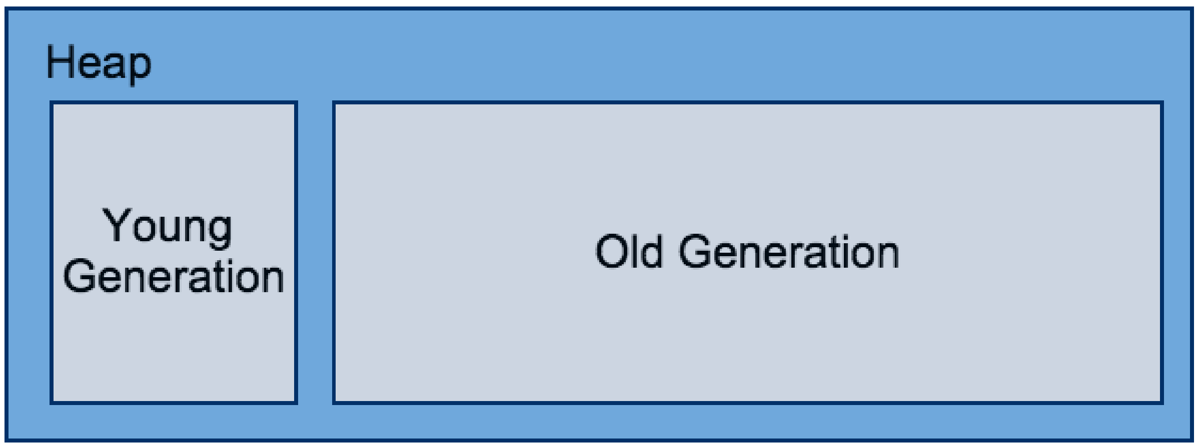
Garbage Collectors

Memory Management on the JVM

1. `Object x = new Object();`
2. There is no step 2

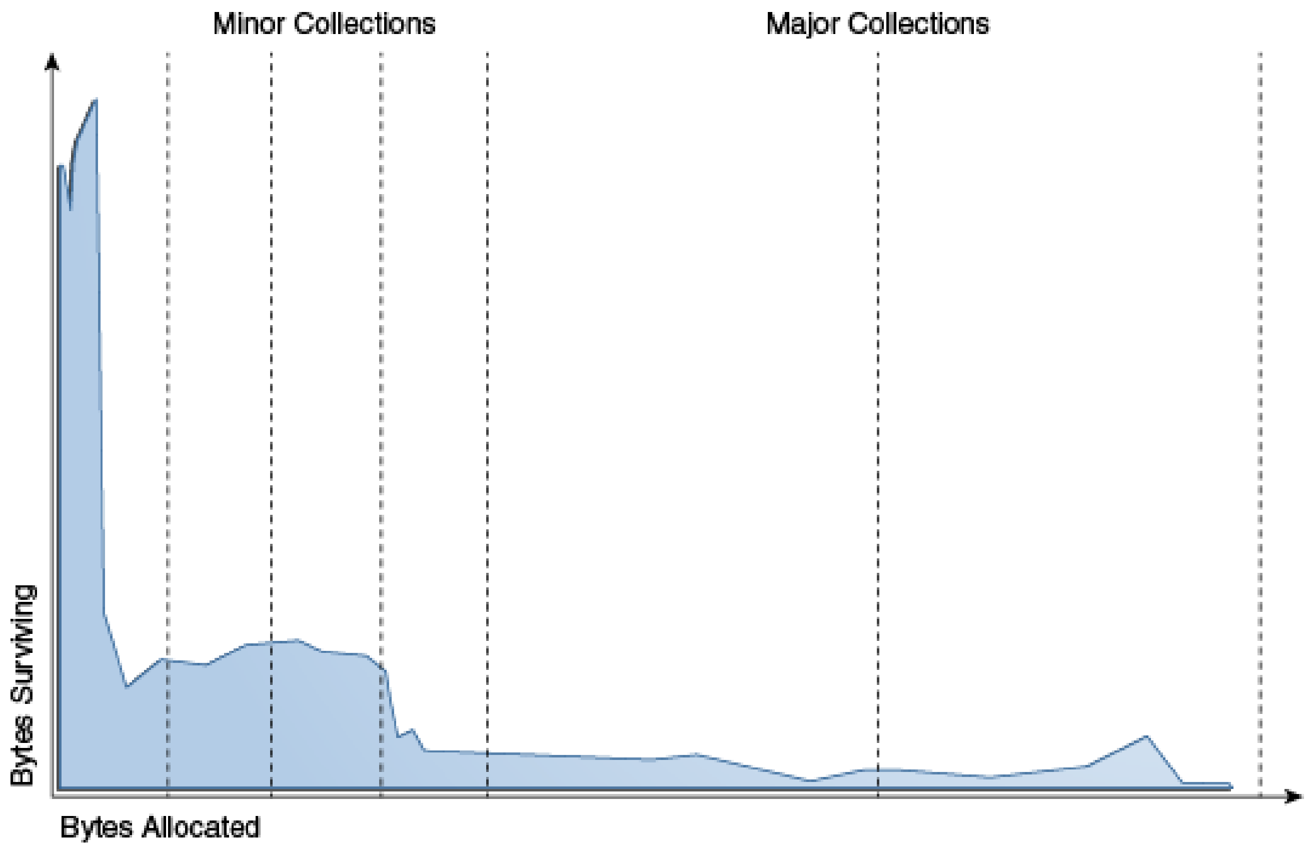
Heap Layout

- Young Generation
  Contains newly instantiated objects
- Old Generation (also: Tenured Generation)
  Contains older objects that survived multiple garbage collections



Weak Generational Hypothesis

Most objects survive for only a short period of time



Source

Weak Generational Hypothesis

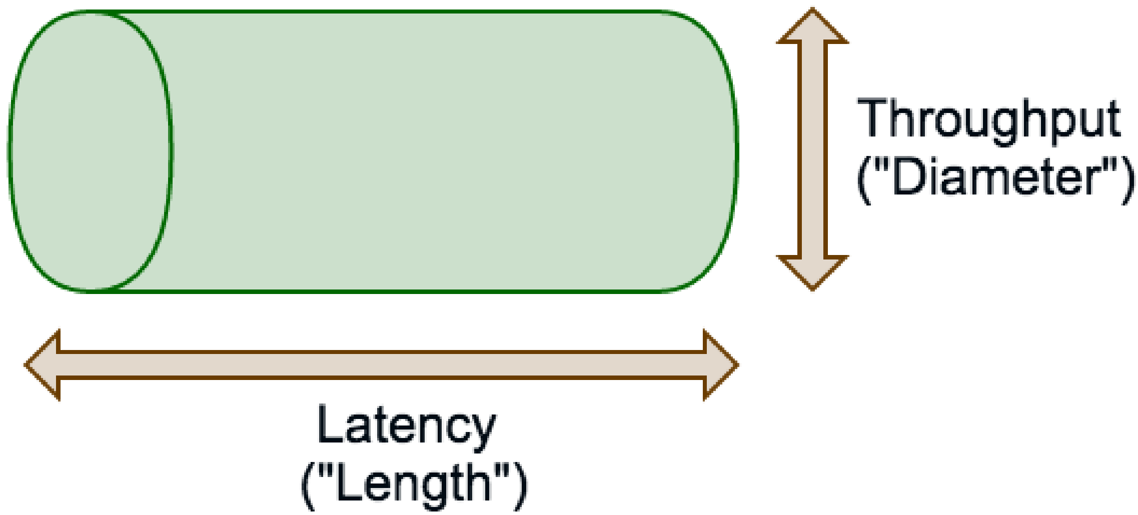
Most GC algorithms on the JVM are based on this assumption

- Split the heap into "generations"
- Collect generations separately

Result: Increased GC performance

Latency vs. Throughput

Consider a pipe:



Garbage Collector Tradeoffs

Different algorithms have tradeoffs typically in those areas:

- Latency
  Human-facing systems need fast response times
- Throughput
  Batch processing systems need more throughput
- Memory
  Waste as little as possible

Garbage Collection (GC) Algorithms

- Serial
- Parallel / Parallel Old
- Concurrent Mark-Sweep (CMS)
- Garbage First (G1)
- Shenandoah (Alpha version)
- C4 (Zing only)

Serial GC

- `-XX:+UseSerialGC`
- Mostly for client applications with small heaps (<< 1 GB)

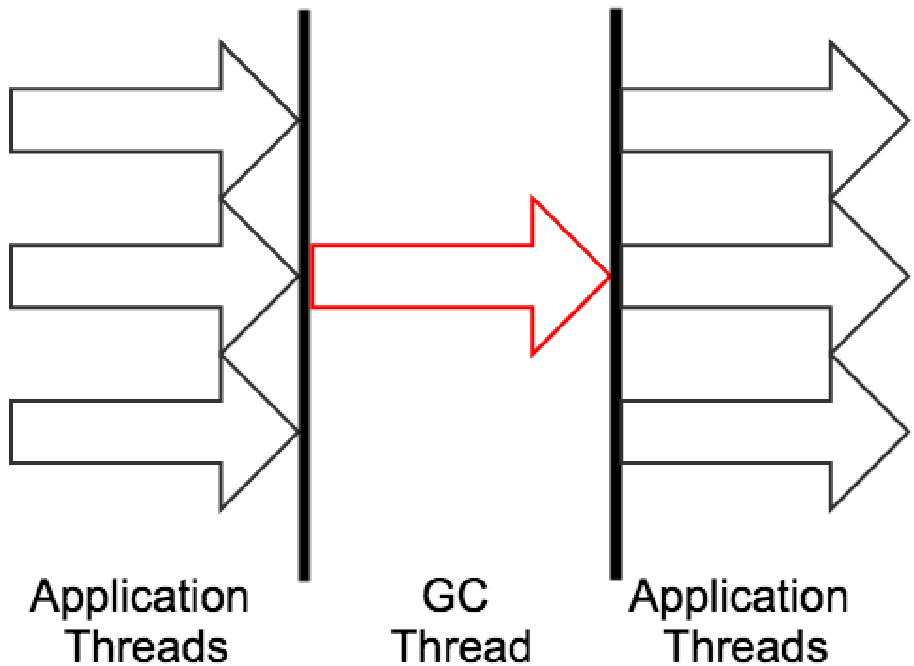


Image based on "Java Performance", page 86

Parallel GC / Parallel Old GC

- `-XX:+UseParallelGC` (Young Generation)
- `-XX:+UseParallelOldGC` (Old Generation)
- High throughput, higher pause times

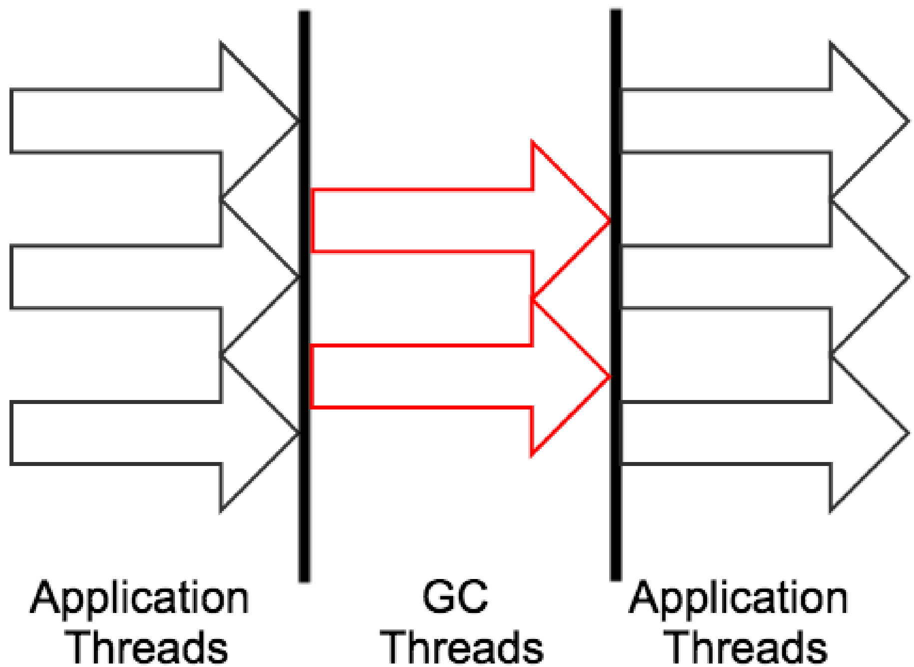


Image based on "Java Performance", page 86

Concurrent Mark-Sweep (CMS)

- `-XX:+UseConcMarkSweepGC`
- Affects only the old generation
- Less throughput, smaller pause times

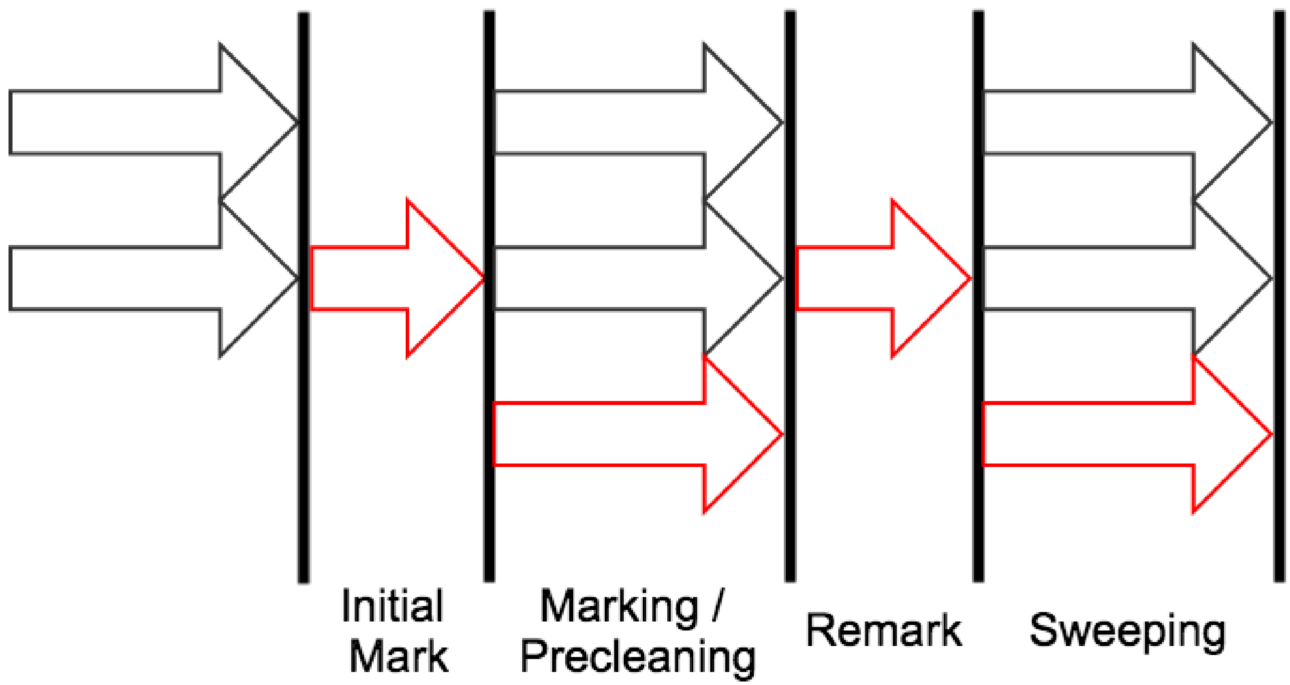


Image based on "Java Performance", page 88

Garbage First (G1)

- `-XX:+UseG1GC`
- Vastly different heap layout. Intended for large heaps (>> 8 GB)
- Less throughput, smaller pause times

Other GC Algorithms

For very large heaps of around 100 GB and more:

- Shenandoah (Red Hat)
- C4 (Azul): By far lowest pause times of all GCs for large heaps

GC Tuning

- Know your application's behavior and SLAs
- Turn the least amount of knobs (70+ GC related JVM flags)
- Performance mantra: Measure, measure, measure

GC Tuning

Starting point:

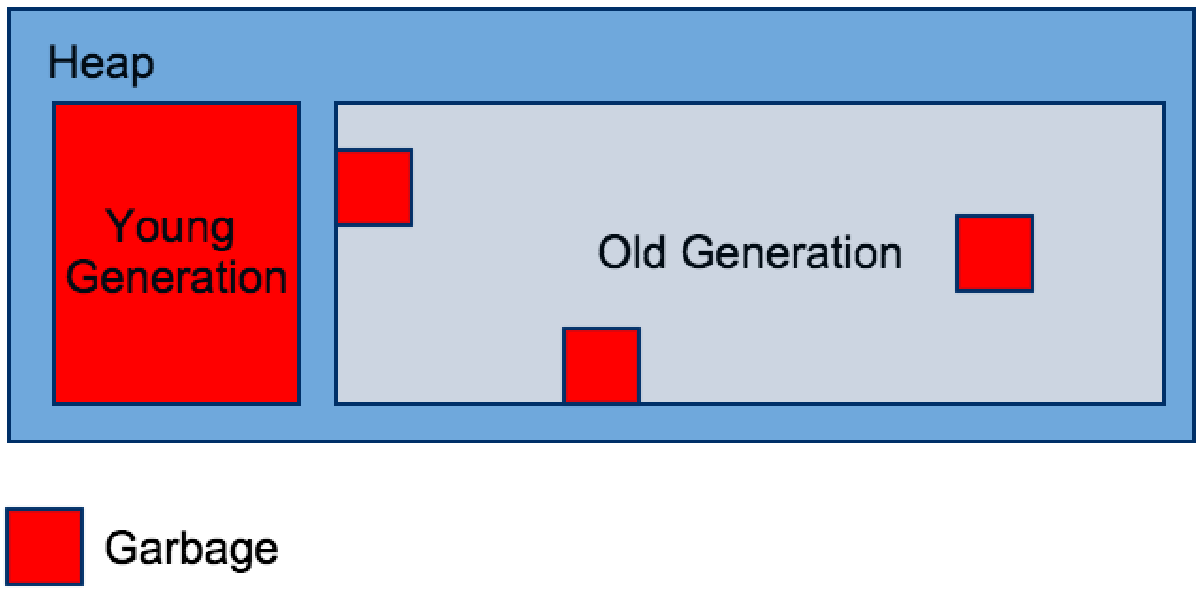
```
-Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCDateSta
```

Use tools like GCViewer for analysis

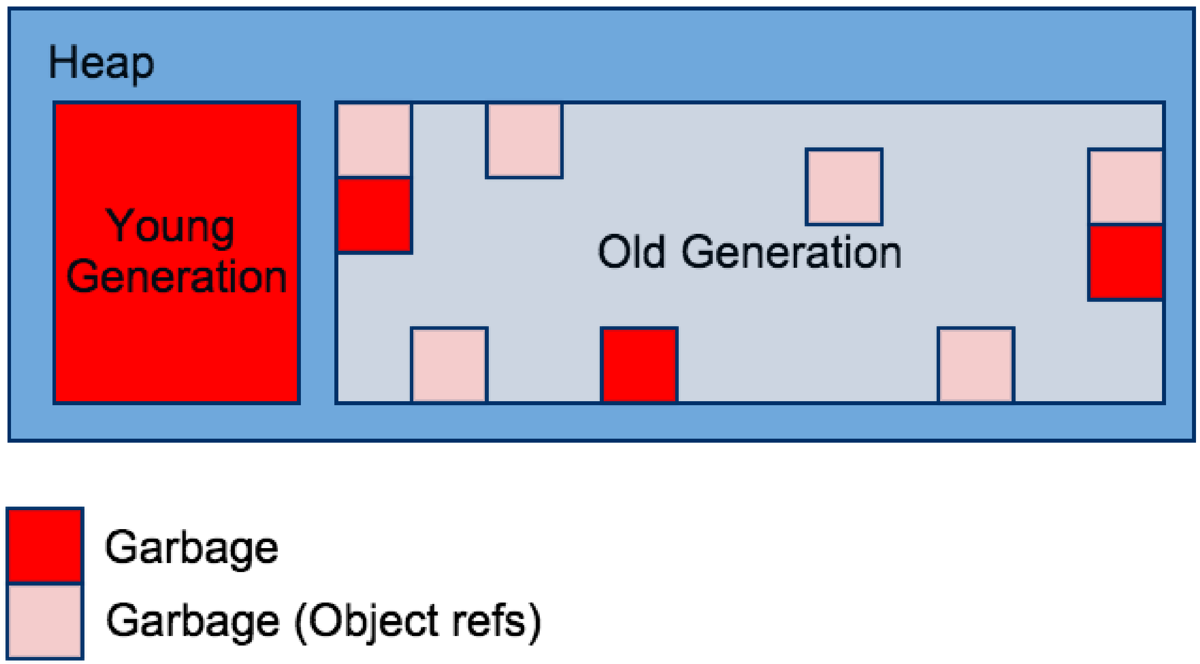
Demo: Inspecting the GC

Based on MinorGC demo by Gil Tene

Demo: Mostly Young-Gen Garbage



Demo: Mostly Young-Gen Garbage + 5% Object Refs



Take Aways

- GC helps with memory management
- Different algorithms - Know their characteristics

What we haven't seen

- Class loading
- JMX and Production Monitoring
- Memory Model
- Thread Model
- ...

Getting started yourself

Download the OpenJDK source code at http://openjdk.java.net and dive in!

Slides

http://bit.ly/jvm-deep-dive-ljug

Q & A

Image Credit

- -Hydra- by arvalis (License: cc by-nc-nd 3.0)
- Escher ladder - escalera de Escher by rromer (License: cc by-nc-sa 2.0)
- nowhere fast... by Mikel (License: cc by-nc-sa 2.0)
- Endless by Maurice (License: cc by-nc-sa 2.0)
- Movie-Matrix-wallpaper by Tony Werman (License: cc by 2.0)
- Jet Dragsters by J. Michael Raby (License: cc by-nc-nd 2.0)
- Signpost by JMC Photos (License: cc by-nc-nd 2.0)
- Stop! Go! by Nana B Agyei (License: cc by 2.0)
- 1GB DDR3 Memory Module by William Warby (License: cc by 2.0)
- _DSC8852 by Rusty Stewart (License: cc by nc nd 2.0)
- Night mechanic by Ali Bindawood (License: by-nd)

None of the pictures have been modified or altered.