JVM Deep Dive

Daniel Mitterdorfer, comSysto GmbH
@dmitterd

Topics

- JVM Overview
- Interpreter
- JIT Compiler
- Memory Management

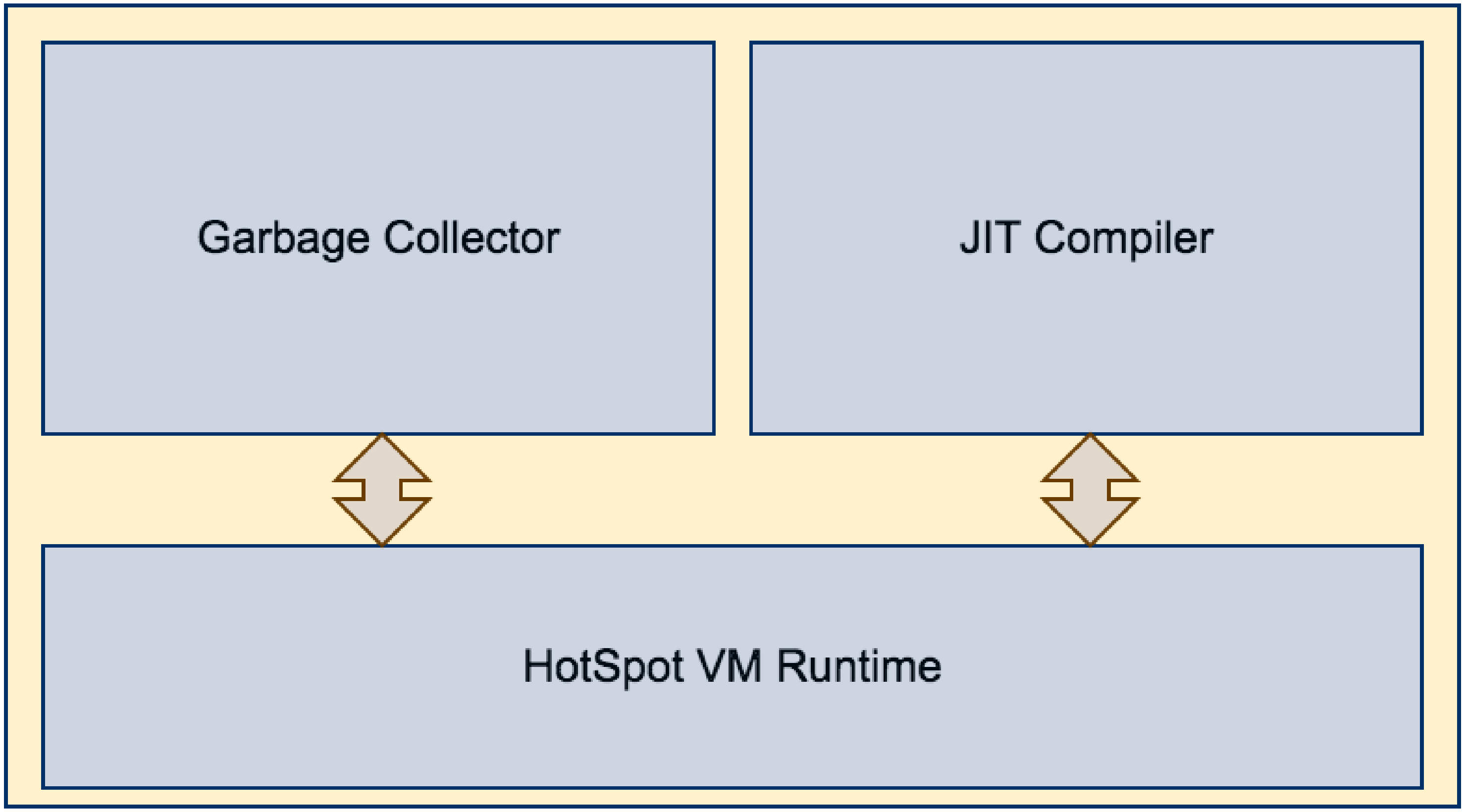
What ”is” a JVM?

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 (C4) and many other features
- J9
  Implementation by IBM
- JRockit
  Implementation by Bea. Now integrated into HotSpot.
- ...

The HotSpot JVM



Based on "Java Performance", p. 56

Let’s start simple

What happens between...

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

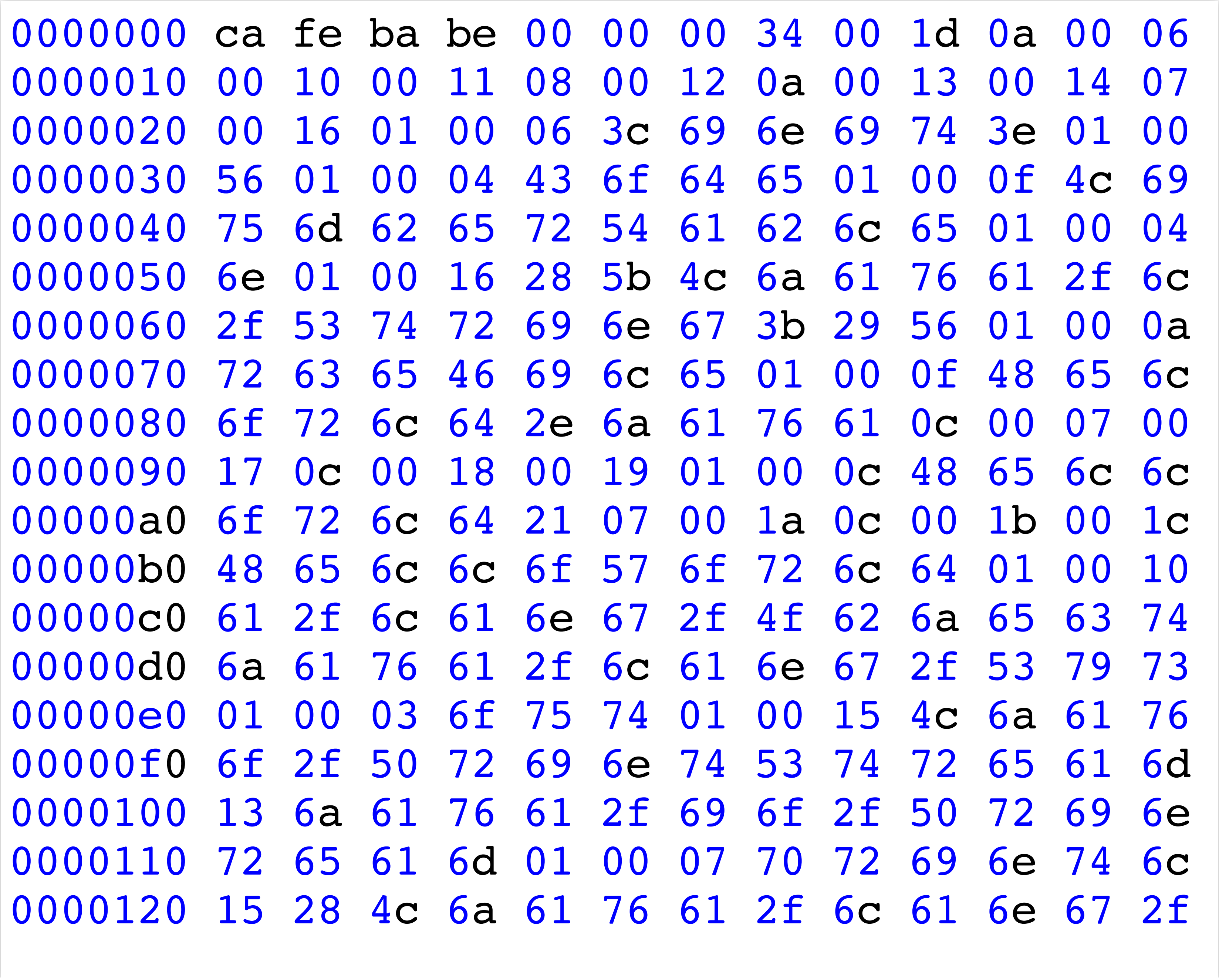
... and ...

```
Hello World!
```

”Compile”

```
javac HelloWorld.java
```

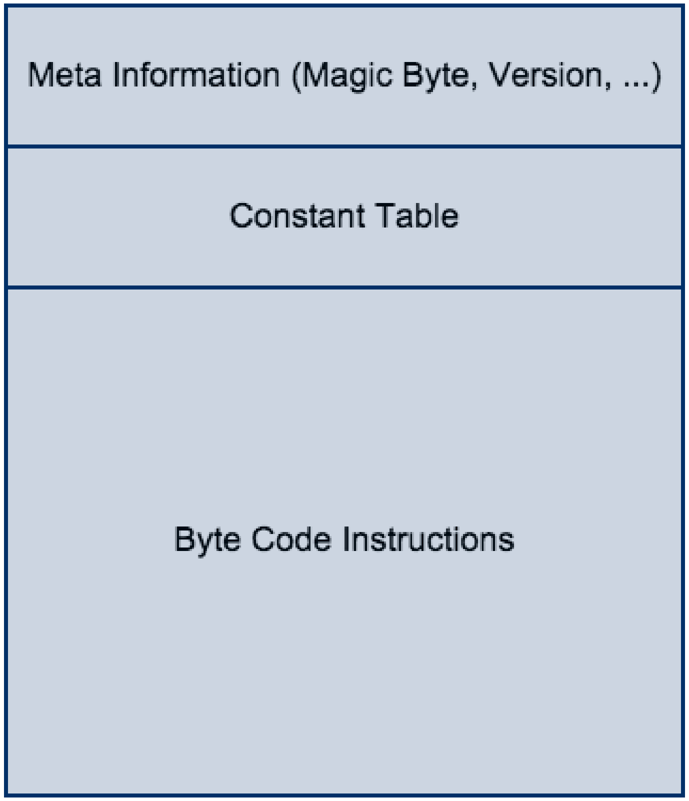
HelloWorld.class Hexdumped



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

Welcome to the Matrix

Structure of a `.class` file



Meta Information (Magic Byte, Version, ...)

Constant Table

Byte Code Instructions

Beware: This is very 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
  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
add    rsp,0x8
mov    edx,DWORD PTR [rsp]
add    rsp,0x8
add    eax,edx                 ; add parameters
movzx  ebx,BYTE PTR [r13+0x1] ; dispatch next b
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
- Inspect `.class` files with `javap`
- The interpreter is a complex beast

JIT compilation

JIT?

- JIT = Just In Time
- "Profile-guided" optimization
- Only hot code paths ("hot spots")

Compile Triggers

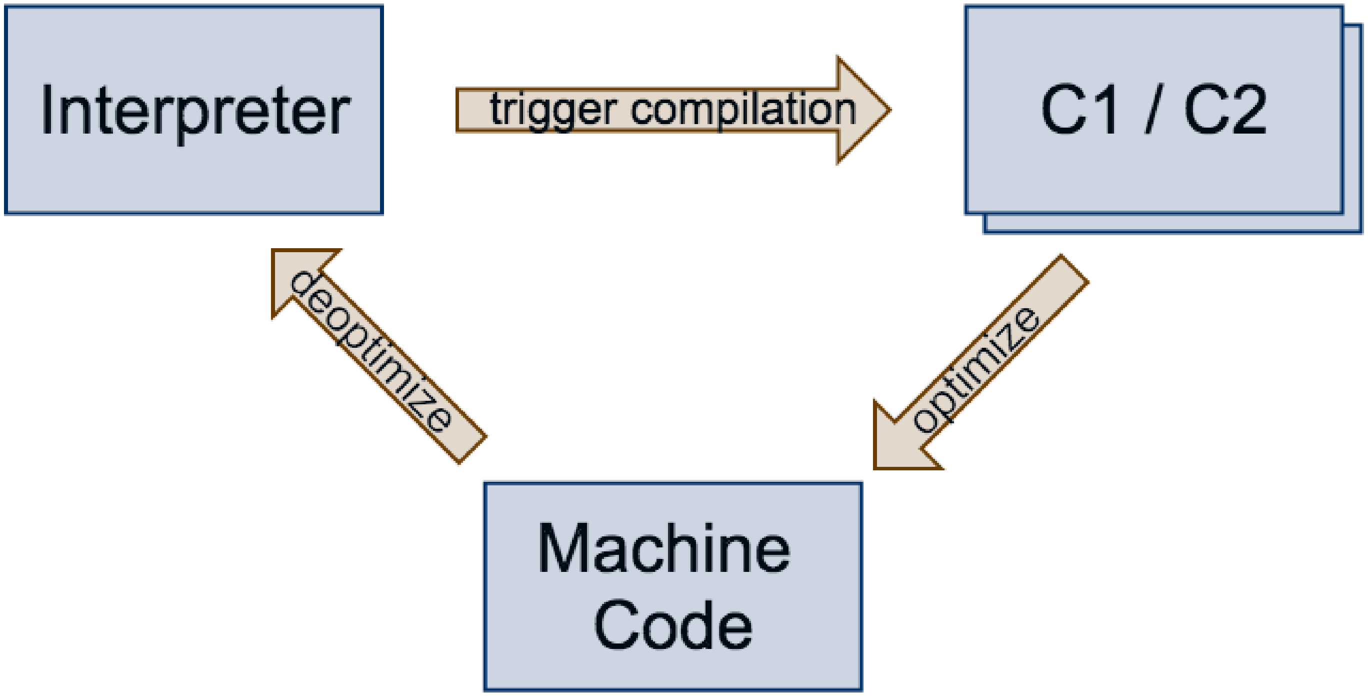
Counters in the interpreter:

- Method invocation counter
- 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`

JIT Compiler and Interpreter



Runtime Profiling

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

Optimizations

- Dead Code Elimination
- Method Inlining
- Class Hierarchy Analysis
- ...

Intrinsics

Hand-optimized "shortcuts" for certain JDK methods

Example: `Math#abs(double)`

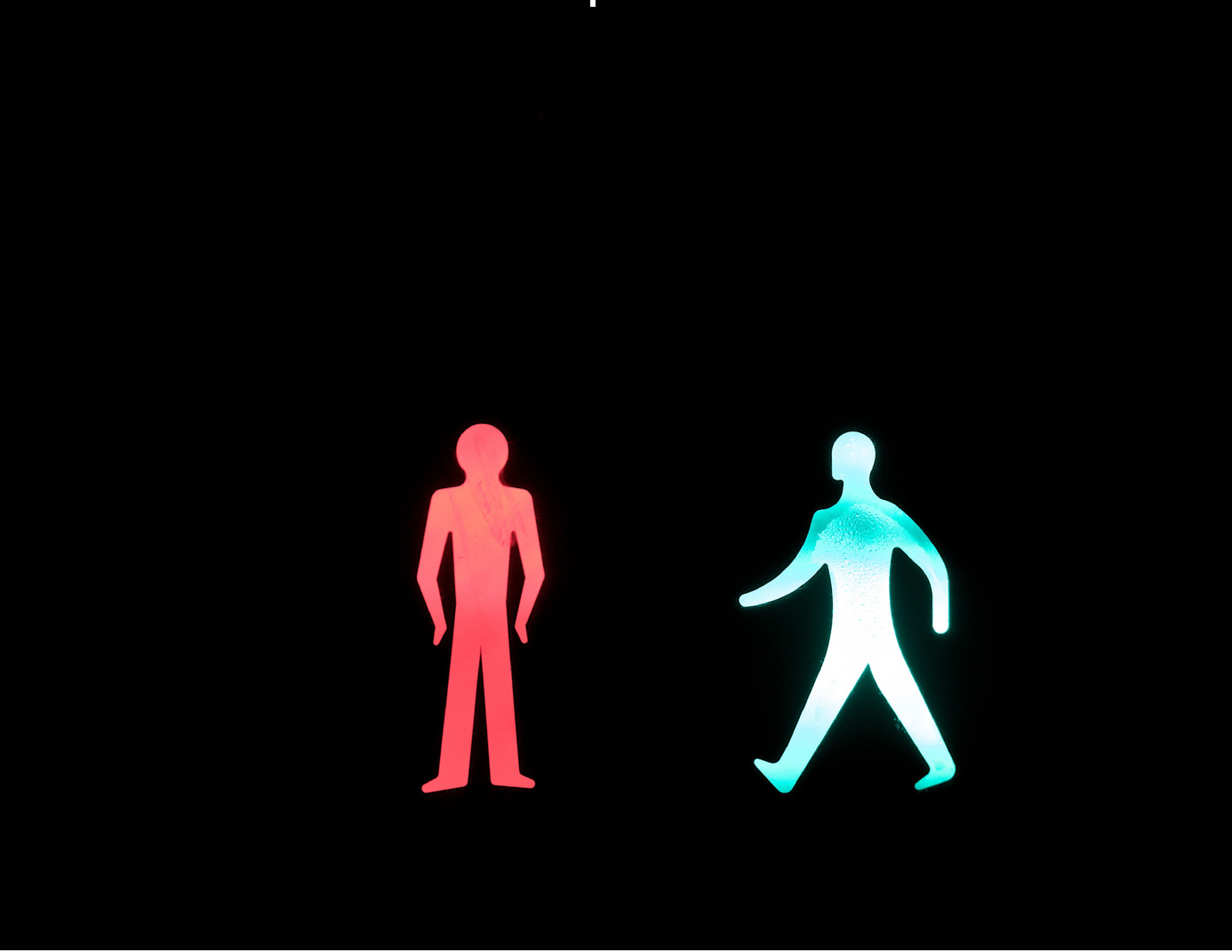
```
return (a <= 0.0D) ? 0.0D - a : a;
```

x86 Intrinsics

```
Math.abs(double)
```

↓

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



Safepoints

How to "remove" compiled machine code in a busy application?

1. Halt *every* application thread ("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

Embrace the JIT

- Use short methods (inlining)
- Use JDK methods (may use intrinsics)
- Use inheritance but take care in performance critical code

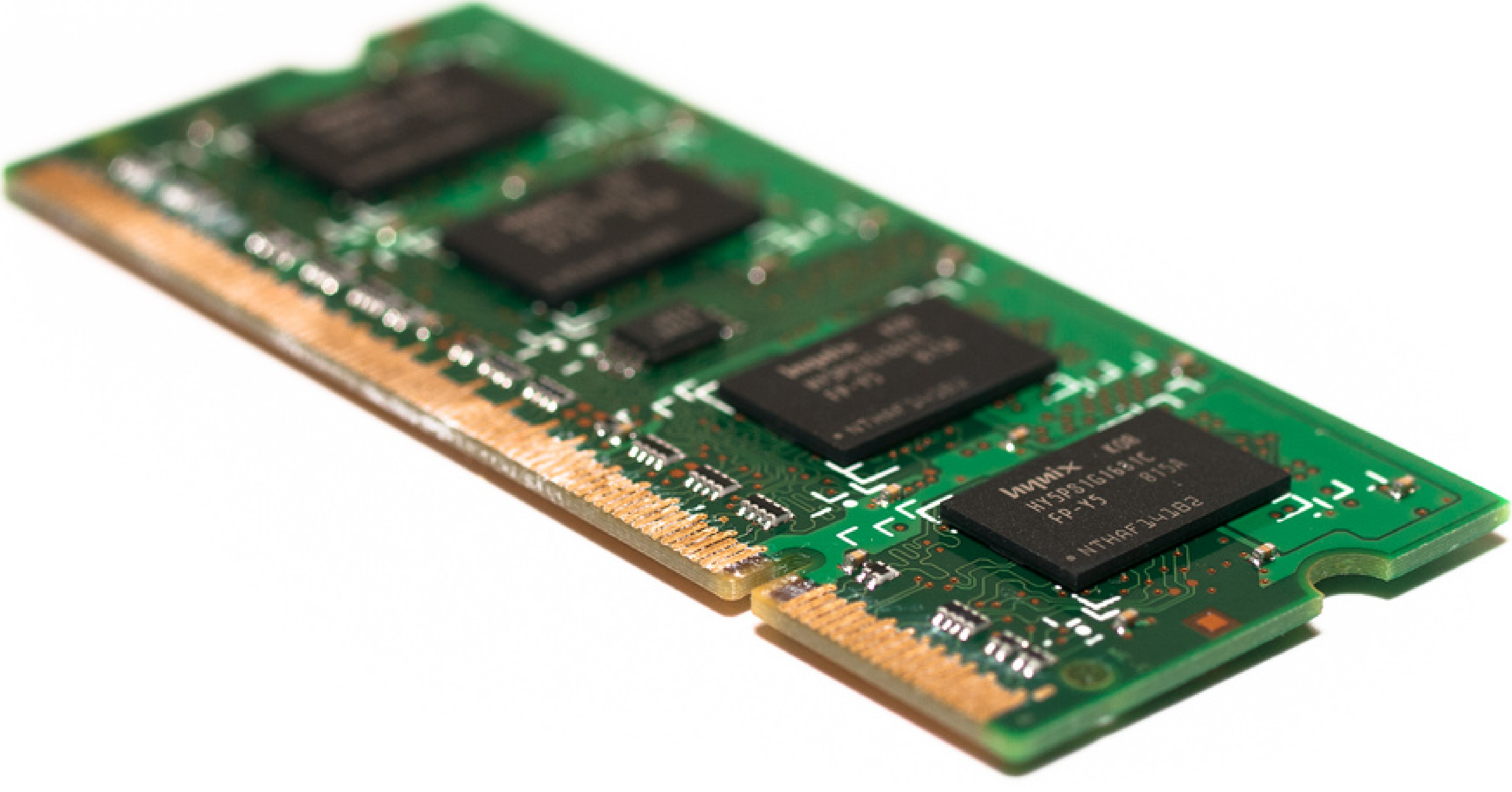
Inspecting Compilation

- Use `-XX:+PrintCompilation`
- Use JITWatch

Take Aways

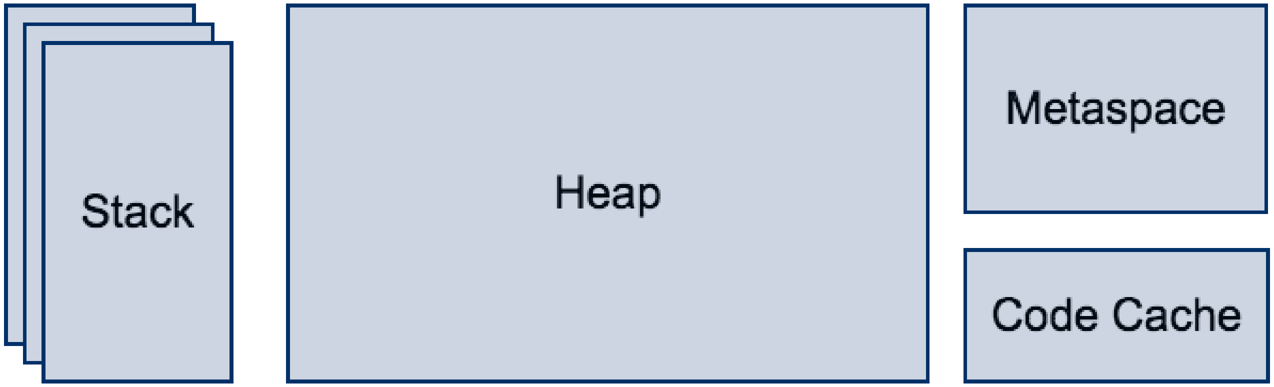
- JIT compilation makes Java code fast
- JIT compilation relies on runtime information
- Cooperation needed between runtime, interpreter and JIT compiler

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

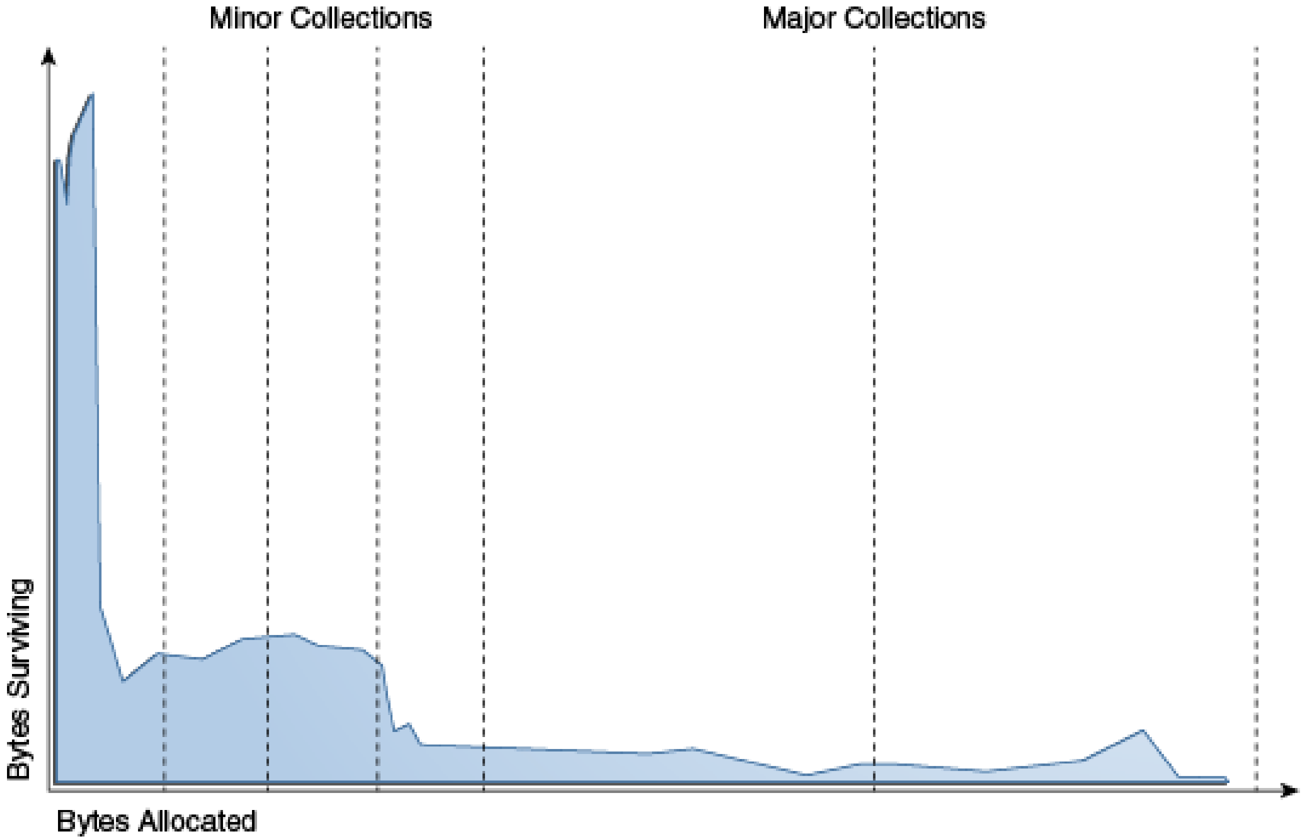
1. `Object x = new Object();`
2. There is no step 2

Garbage Collector Tradeoffs

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

Weak Generational Hypothesis

Most objects survive for only a short period of time



Source

Weak Generational Hypothesis

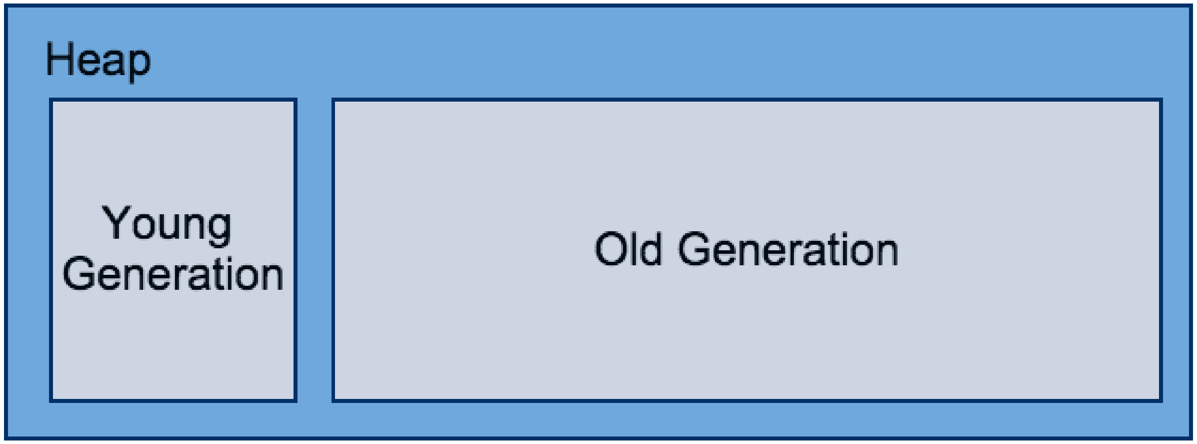
Most GC algorithms are based on this assumption

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

Result: Increased GC performance

Heap Layout

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



Common Algorithms

Serial GC

- `-XX:+UseSerialGC`
- Client applications with small heaps (<< 1 GB)

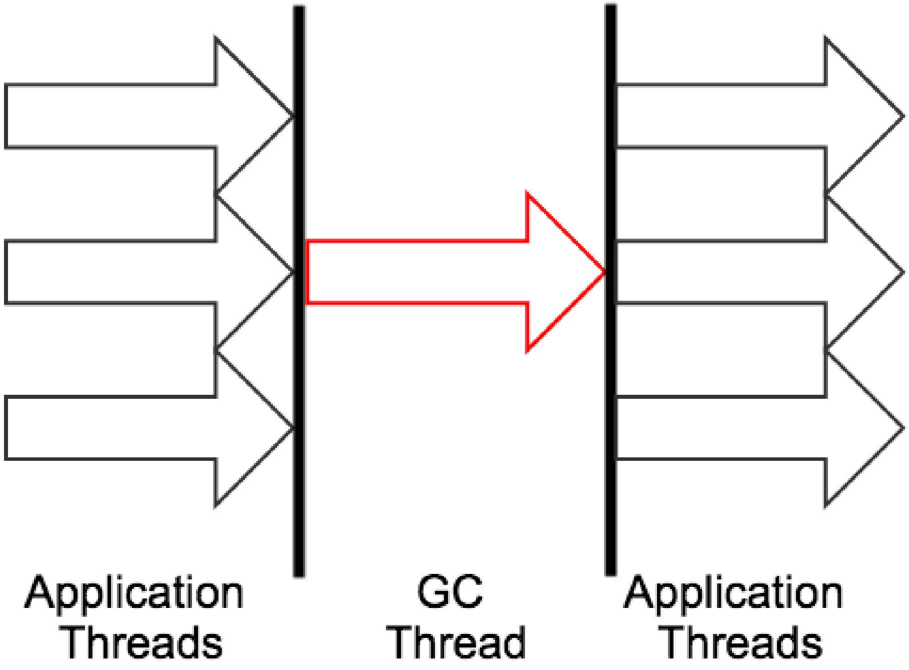


Image based on "Java Performance", page 86

Parallel GC / Parallel Old GC

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

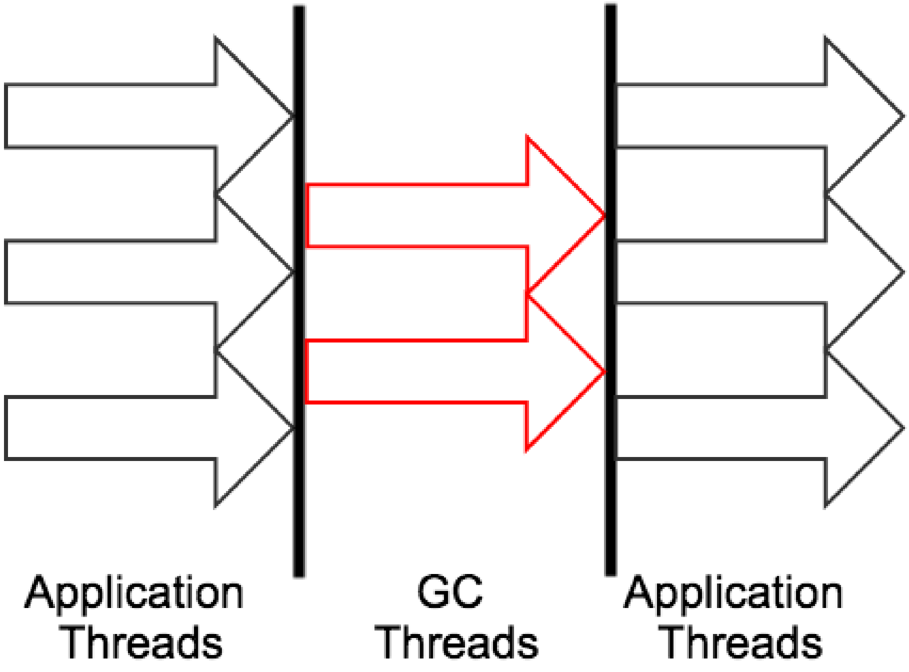


Image based on "Java Performance", page 86

Concurrent Mark-Sweep (CMS)

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

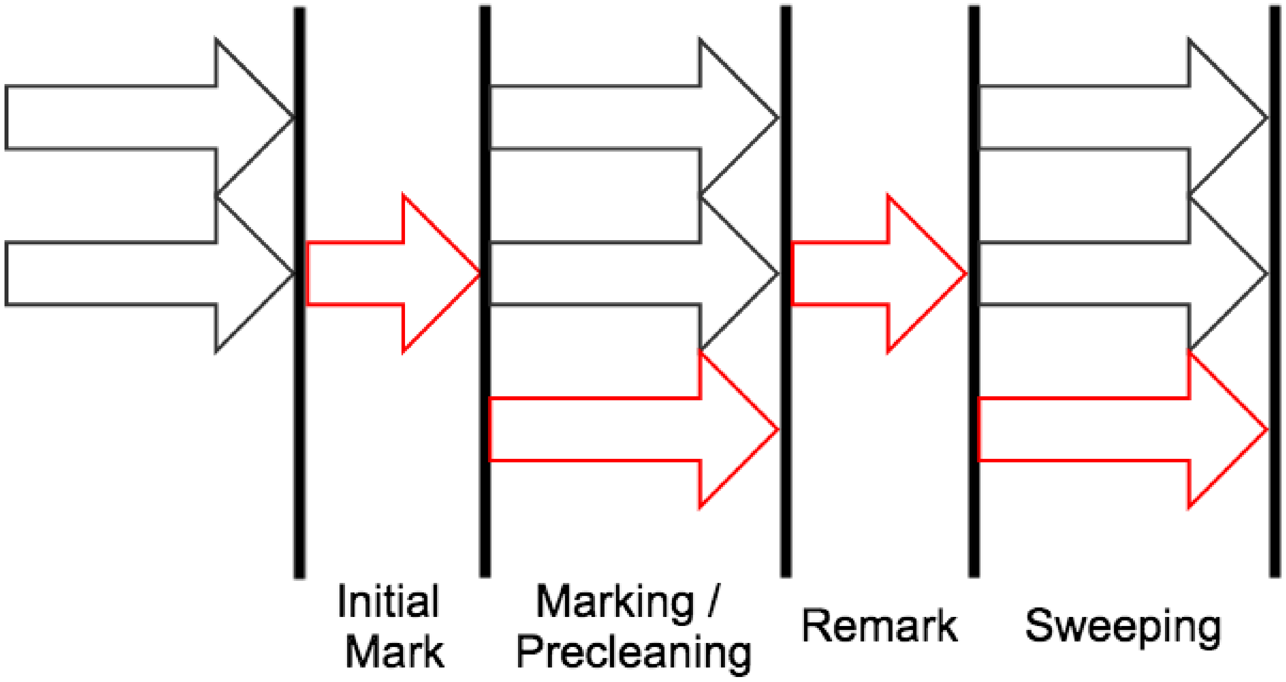


Image based on "Java Performance", page 88

Garbage First (G1)

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

Other GC Algorithms

Very large heaps (> 100 GB)

- Shenandoah (OpenJDK): Currently in alpha
- C4 (Azul Zing)

Which GC am I using?

```
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal -version | grep -E "Use.*GC.*true"
```

GC Tuning

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

GC Tuning

Starting point:

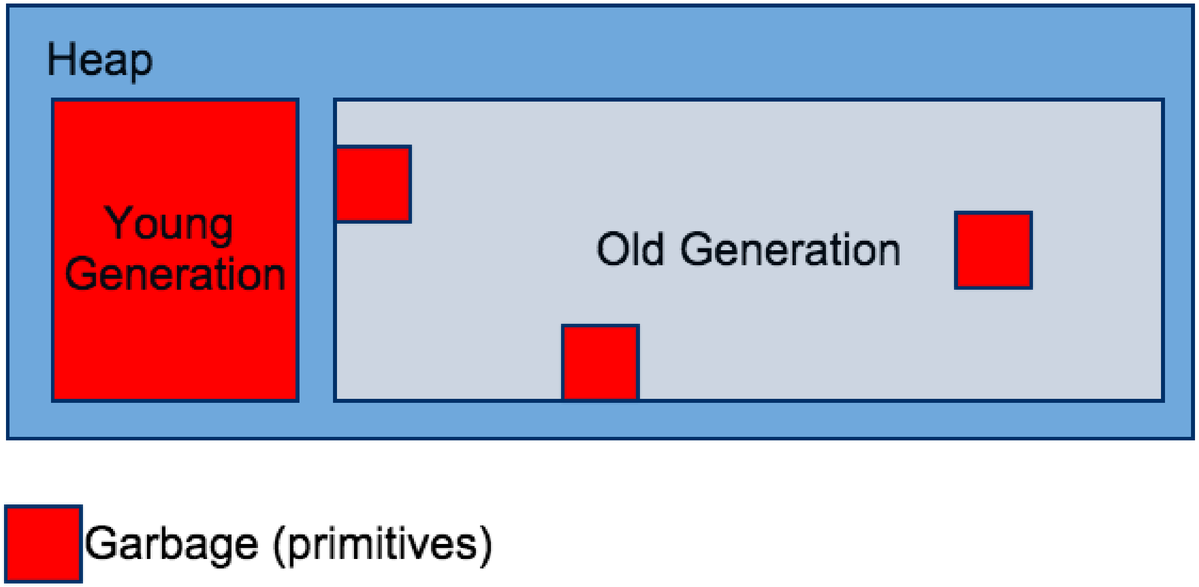
```
-Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGC
```

Use tools like GCViewer for analysis

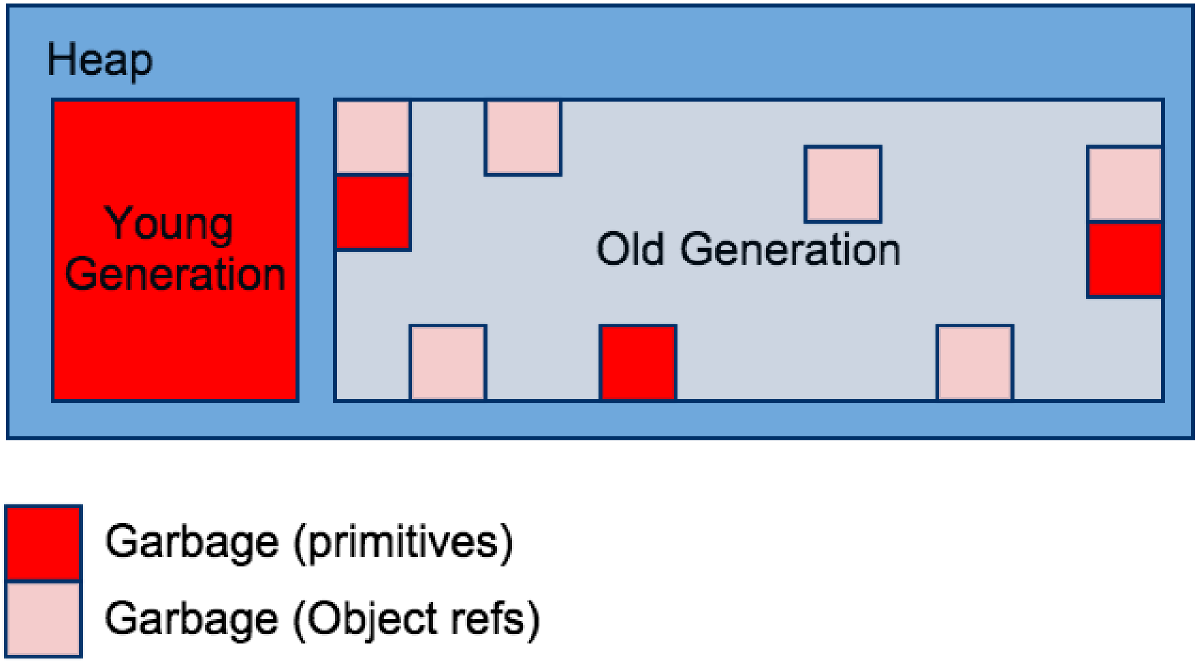
Demo: Inspecting the GC

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

Getting started yourself

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

Slides

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

Q & A

Image Credit

- -Hydra- by arvalis (License: cc by-nc-nd 3.0)
- Movie-Matrix-wallpaper by Tony Werman (License: cc by 2.0)
- Jet Dragsters by J. Michael Raby (License: cc by-nc-nd 2.0)
- 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.