Taming the Java Virtual Machine

Li Haoyi, Chicago Scala Meetup, 19 Apr 2017

Who Am I?

Previously: Dropbox Engineering

Currently: Bright Technology Services

- Data Science, Scala consultancy
- Fluent Code Explorer, www.fluentcode.com

Early contributor to Scala.js, author of Ammonite, FastParse, Scalatags, ...

Lots of work in Java, Scala, Python, JS, ...

The Fluent Code Explorer

High-performance code-explorer

- Fast, as-you-type search
- Scalable to repos of tens millions of lines of code

Single Process

- No distributed system
- Runs on a single VM

Taming the Java Virtual Machine

```
class Simple{
    public static void main(String args[]){
        String s = "Hello Java";
        int i = 123;
        System.out.println(s + 123);
    }
}
```

Taming the Java Virtual Machine

```
class Simple{
    public static void main(String args[]){
        String s = "Hello Java";
        int i = 123;
        System.out.println(s + 123);
    }
}
```

// How much memory
// does this take?

// How much memory
// does this take?

// What happens if I
// run out of memory?

// What's really
// happening here?

// What is this "JIT Compiler"
// I keep hearing about?

“Implementation Defined”?

Taming the Java Virtual Machine

Memory Layouts

Garbage Collection

Compilation

Taming the Java Virtual Machine

Memory Layouts

- `OutOfMemoryError`

Garbage Collection

- Long pauses

Compilation

- Mysterious performance issues

Taming the Java Virtual Machine

Memory Layouts

Garbage Collection

Compilation

Memory Layouts

Memory Layouts

Everything is great if you have enough

Everything is terrible if you don’t have enough

Technically implementation-defined

- in practice most people are using OpenJDK/OracleJDK

Memory Demo

Memory Layouts

Data type	Bytes
boolean	1
byte	1
short	2
int	4
long	8
float	4
double	8

Memory Layouts

Data type	Bytes
boolean	1
byte	1
short	2
int	4
long	8
float	4
double	8

Data type	Bytes
Boolean	4
Byte	4
Short	4 + 16
Int	4 + 16
Long	4 + 24
Float	4 + 16
Double	4 + 24

Memory Layouts

Data type	Bytes
boolean	1
byte	1
short	2
int	4
long	8
float	4
double	8

Data type	Bytes
Boolean	4
Byte	4
Short	4 + 16
Int	4 + 16
Long	4 + 24
Float	4 + 16
Double	4 + 24

Data type	Bytes
Array	4 + 16, rounded to next 8
Object	4 + 12 rounded to next 8

Tips for reducing memory usage

Use Arrays when dealing with large lists of primitives, instead of java.util.*

Use BitSets instead of large Arrays of booleans

Use a library to provide unboxed collections (sets, maps, etc.) of primitives:

- FastUtil: http://fastutil.di.unimi.it/
- Koloboke Collections: https://github.com/leventov/Koloboke
- Eclipse Collections: https://www.eclipse.org/collections/

Koloboke Collections

```java
Map<Integer, Integer> map = new HashMap<>();
```

- 1,000,000 items, 72.3mb

```java
Map<Integer, Integer> map = HashIntIntMaps.newMutableMap();
```

- 1,000,000 items, 16.7mb

Build your own Specialized Collections

```scala
class Aggregator[@specialized(Int, Long) T: ClassTag](initialSize: Int = 1) {
 // Can't be `private` because it makes `@specialized` behave badly
 protected[this] var data = new Array[T](initialSize)
 protected[this] var length0 = 0

 def length = length0
 def apply(i: Int) = data(i)
 def append(i: T) = {
   if (length >= data.length) {
     // Grow by 3/2 + 1 each time, same as java.util.ArrayList. Saves a bit
     // of memory over doubling each time, at the cost of more frequent
     // doublings,
     val newData = new Array[T](data.length * 3 / 2 + 1)
     System.arraycopy(data, 0, newData, 0, length)
     data = newData
   }
   data(length) = i
   length0 += 1
 }
```

Taming the Java Virtual Machine

Memory Layouts

Garbage Collection

Compilation

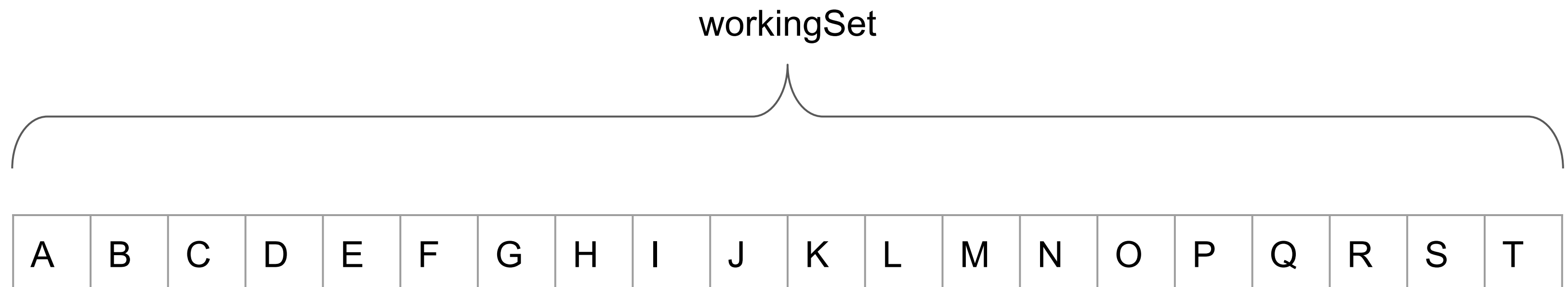
Garbage Collection

Easy to take for granted

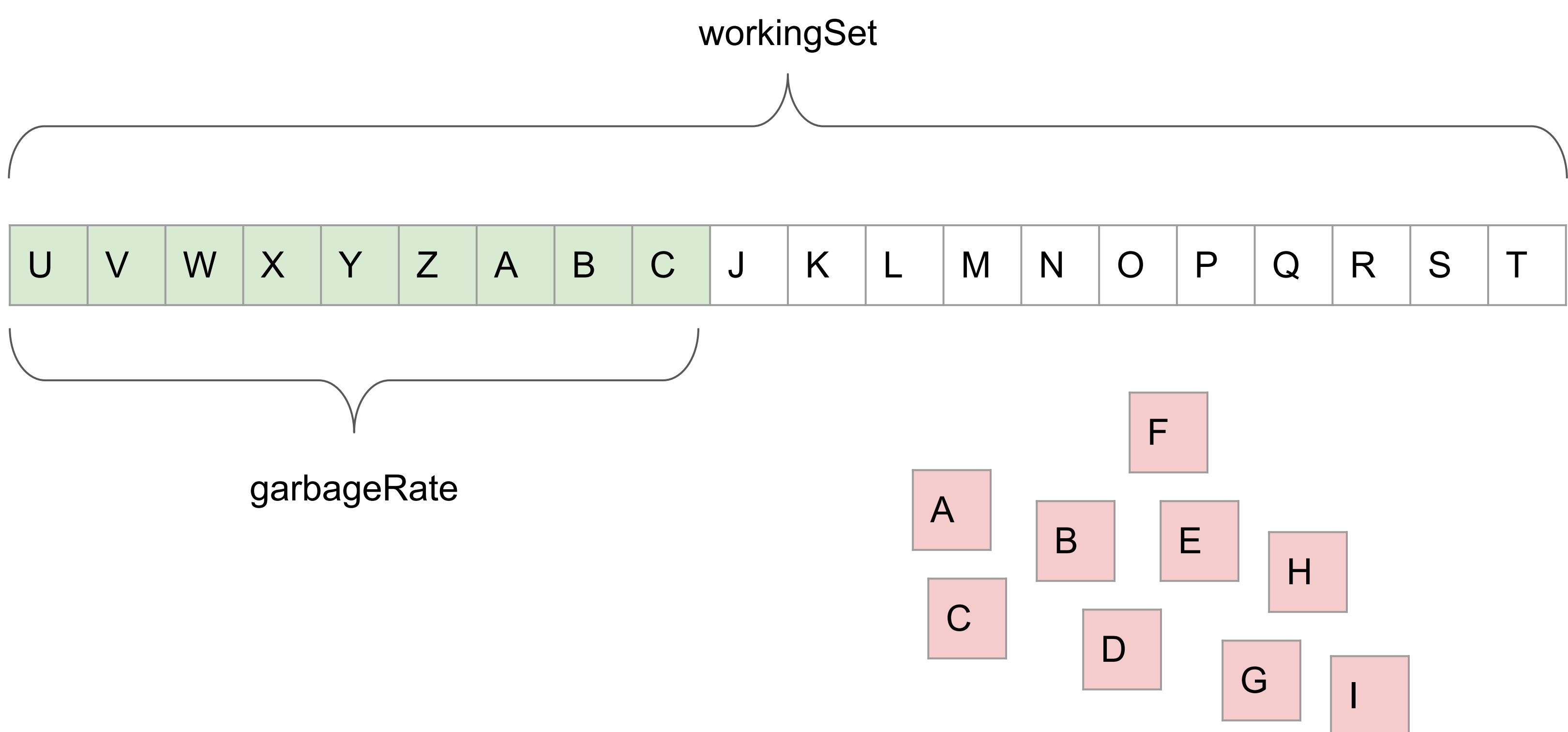
In theory “invisible” to the logic of your application

In practice can have huge impact on its runtime characteristics

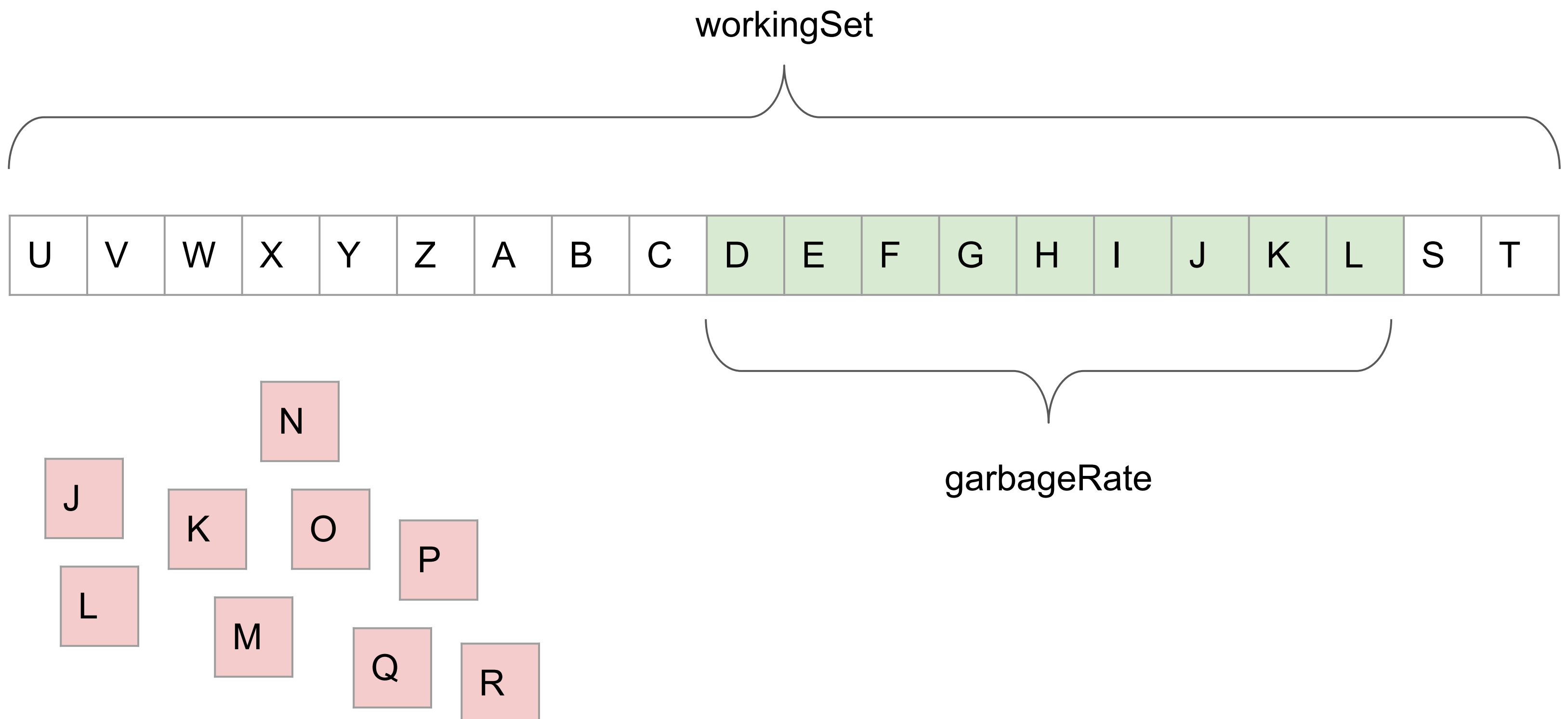
Garbage Collection Demo



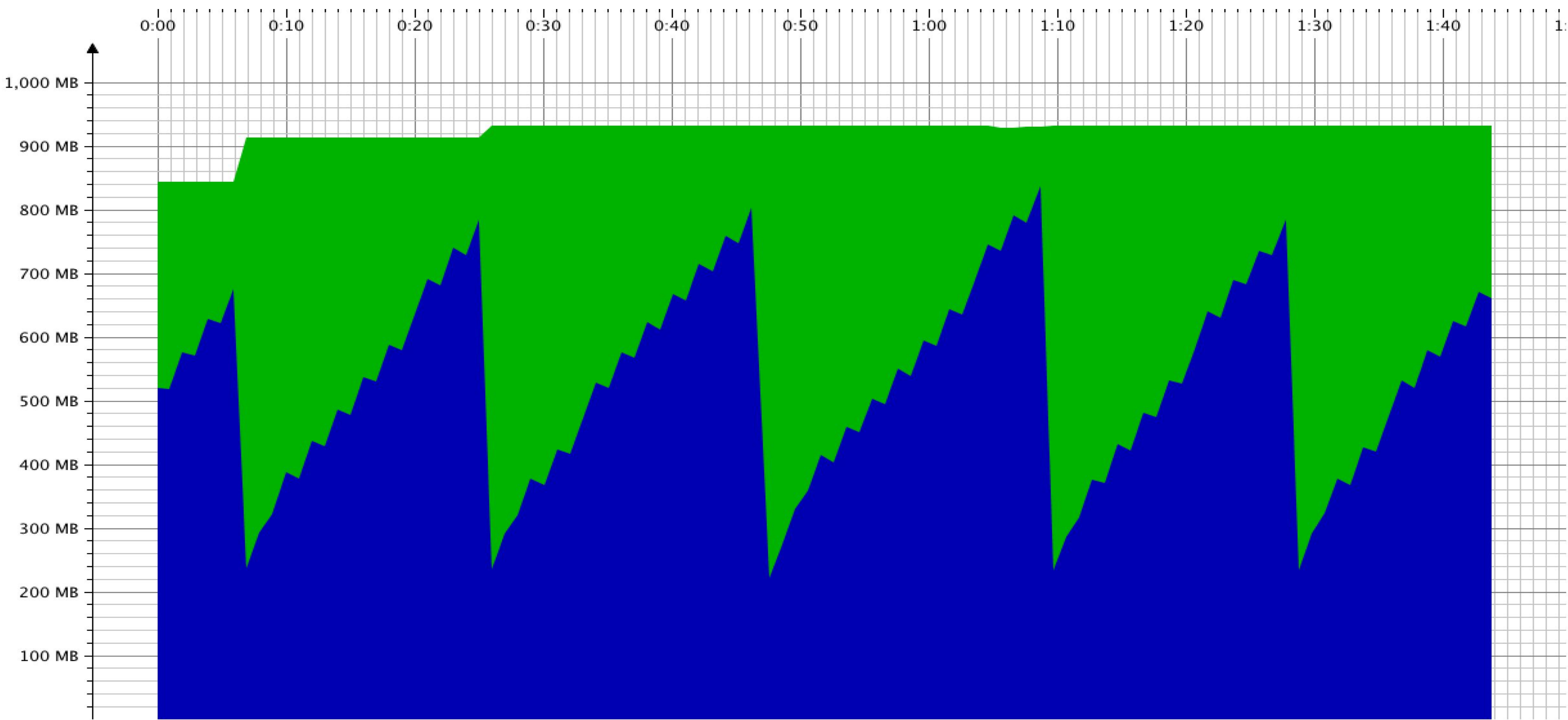
Garbage Collection Demo



Garbage Collection Demo



Parallel GC (Default)



Parallel GC: garbage doesn’t affect pause times

Live Set\Garbage Rate	1,600	6,400	25,600
100,000	17ms	17ms	20ms
200,000	30ms	31ms	30ms
400,000	362ms	355ms	356ms
800,000	757ms	677ms	663ms
1,600,000	1651ms	1879ms	1627ms

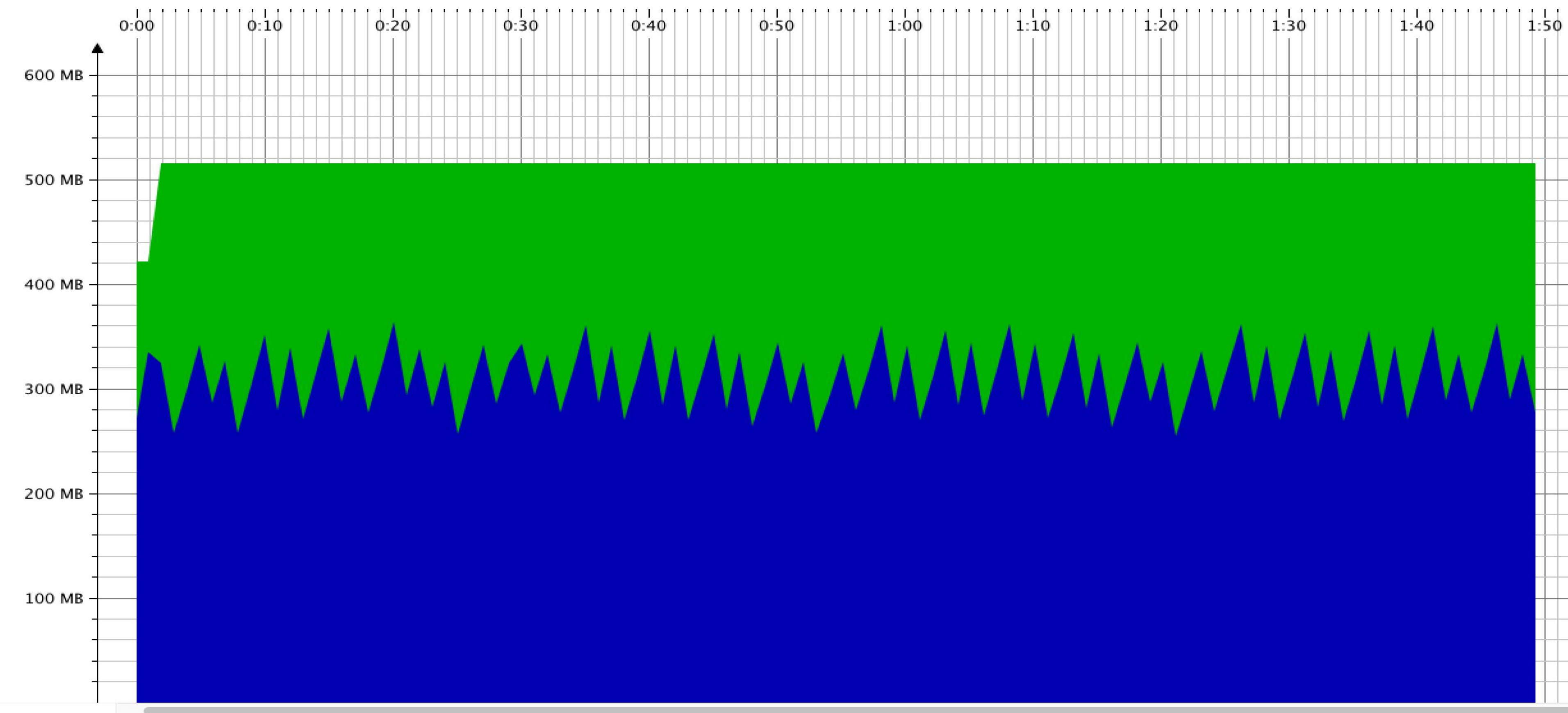
Parallel GC

- **Pause times** (mostly) proportional to **working set**
  - Garbage load doesn’t matter!
  - ~1 millisecond per megabyte of working set
- **Frequency of pauses** proportional to **garbage load**, inversely proportional to **total memory**
- **Will use as much heap as it can**!
  - Even if it doesn’t “need” all of it

Creating less garbage doesn't reduce pause times!

Working Set	Garbage Rate	Maximum Pause	
400,000	200	345ms	
400,000	800	351ms	
400,000	3,200	389ms	
400,000	12,800	388ms	

Concurrent Mark & Sweep



Concurrent Mark & Sweep

Live Set\Garbage Rate	1,600	6,400	25,600
100,000	26ms	31ms	34ms
200,000	33ms	37ms	43ms
400,000	43ms	61ms	91ms
800,000	44ms	*281ms	720ms
1,600,000	1311ms	1405ms	1403ms

Concurrent Mark & Sweep

- Lower throughput than the Parallel GC

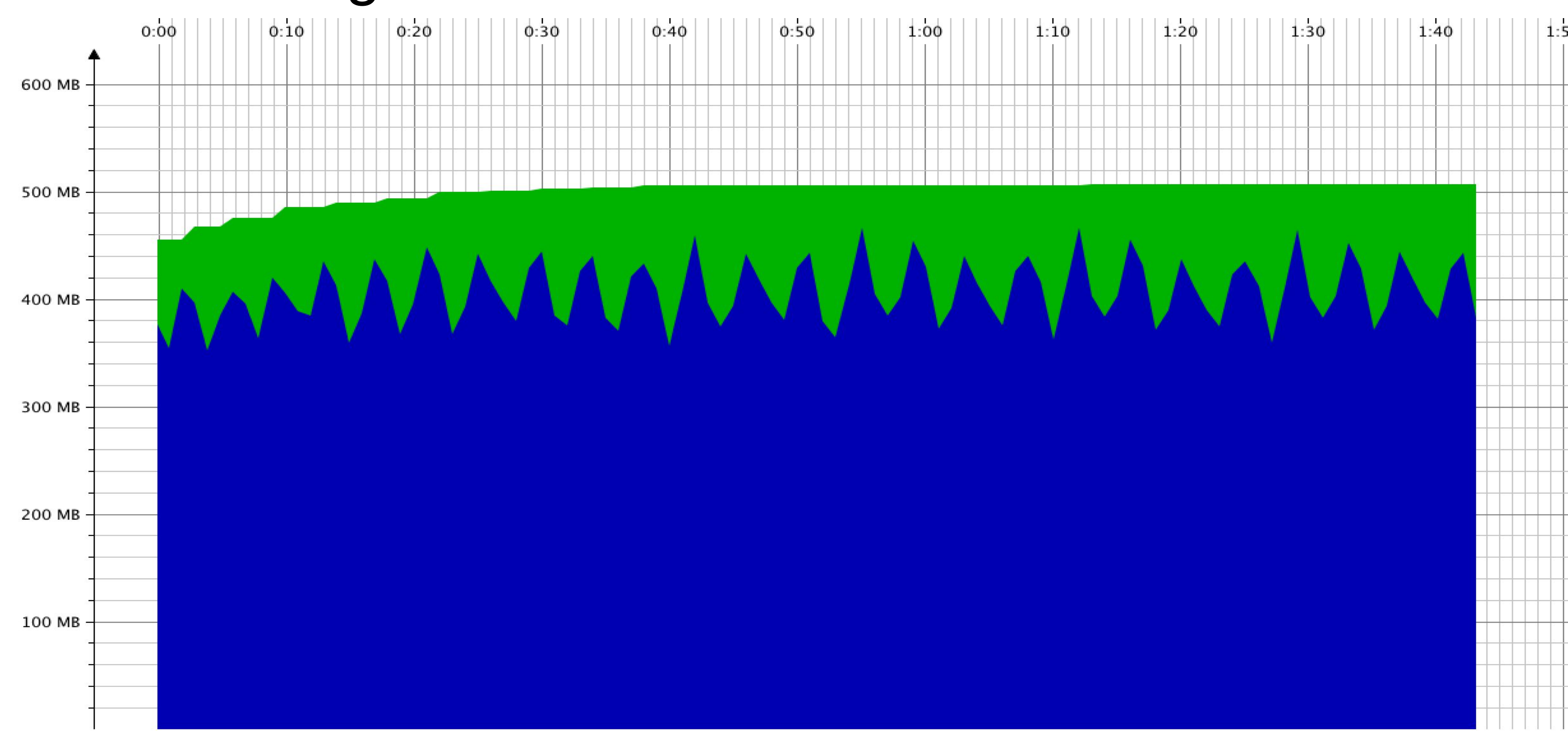
- Pause times around 30-50ms

- Doesn’t grow the heap unnecessarily
  - % of time spent collecting not dependent on heap size

CMS: less garbage *does* reduce pause times

Working Set	Garbage Rate	Maximum Pause	
400,000	200	51ms	
400,000	800	41ms	
400,000	3,200	44ms	
400,000	12,800	105ms	

G1 “Garbage First”



G1 "Garbage First"

Live Set\Garbage Rate	1,600	6,400	25,600
100,000	21ms	15ms	18ms
200,000	29ms	30ms	32ms
400,000	43ms	45ms	48ms
800,000	29ms	842ms	757ms
1,600,000	1564ms	1324ms	1374ms

G1 “Garbage First”

- Basically a better version of CMS GC

- Better support for larger heaps, more throughput

- Might become the default in Java 9

GC Comparisons

CMS	1,600	6,400	25,600
100,000	26ms	31ms	34ms
200,000	33ms	37ms	43ms
400,000	43ms	61ms	91ms
800,000	44ms	*281ms	720ms
1,600,000	1311ms	1405ms	1403ms

Parallel	1,600	6,400	25,600
100,000	17ms	17ms	20ms
200,000	30ms	31ms	30ms
400,000	362ms	355ms	356ms
800,000	757ms	677ms	663ms
1,600,000	1651ms	1879ms	1627ms

G1	1,600	6,400	25,600
100,000	21ms	15ms	18ms
200,000	29ms	30ms	32ms
400,000	43ms	45ms	48ms
800,000	29ms	842ms	757ms
1,600,000	1564ms	1324ms	1374ms

The Generational Hypothesis

- Most objects are either very-short-lived or very-long-lived

- Most GCs optimize for these cases

- If your code matches this profile, the GC is a lot happier

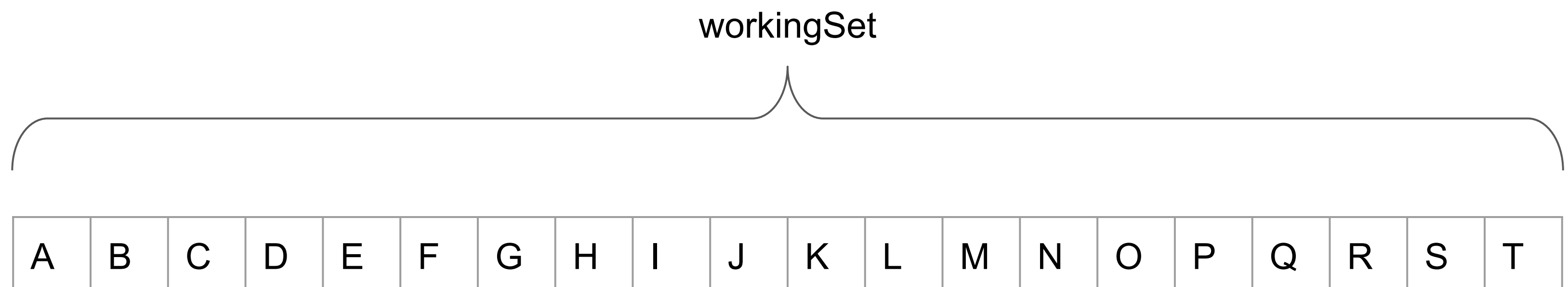
The Generational Hypothesis

For the HotSpot Java VM, the memory pools for serial garbage collection are the following.

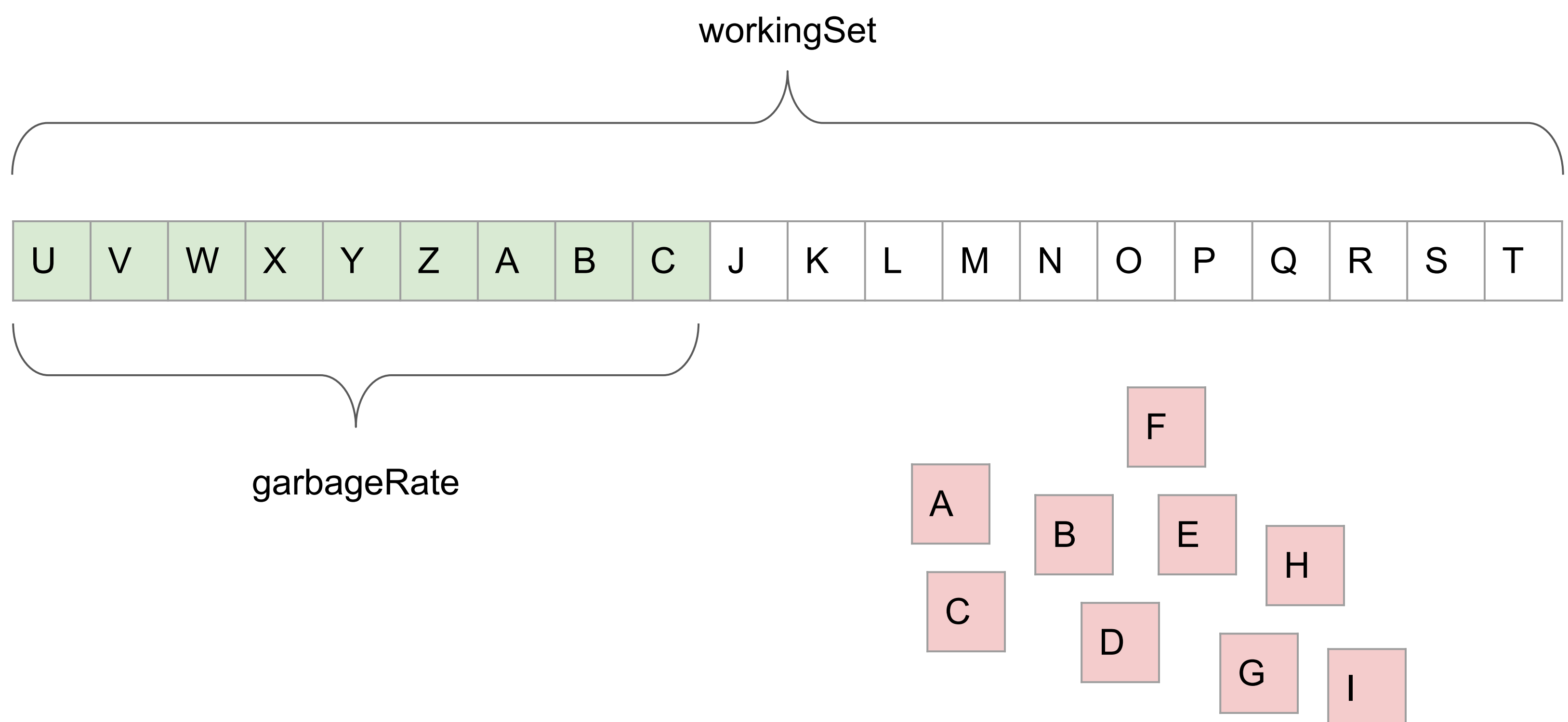
- ***Eden Space**: The pool from which memory is initially allocated for most objects.*
- ***Survivor Space**: The pool containing objects that have survived the garbage collection of the Eden space.*
- ***Tenured Generation**: The pool containing objects that have existed for some time in the survivor space.*

http://stackoverflow.com/questions/2129044/java-heap-terminology-young-old-and-permanent-generations

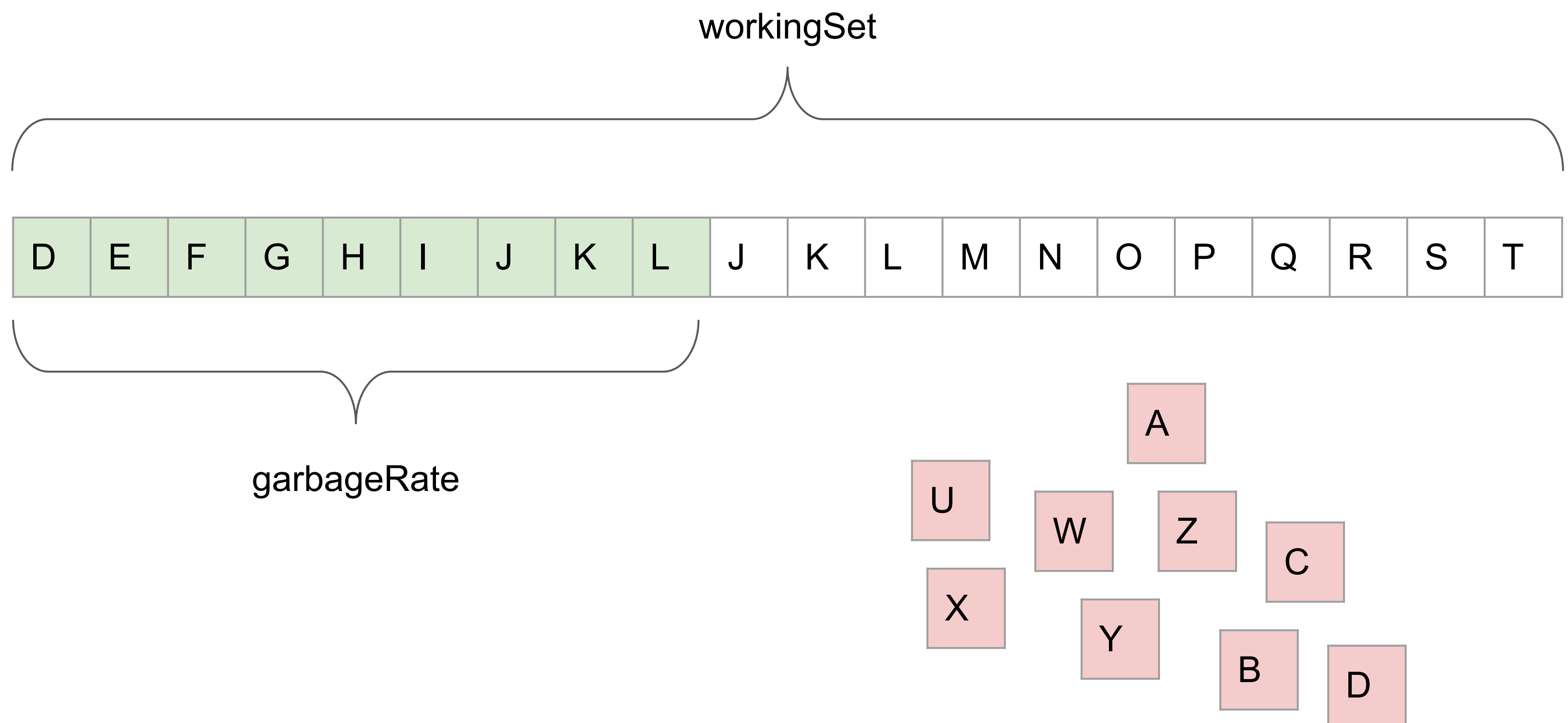
Generation Garbage Collection



Generation Garbage Collection



Generation Garbage Collection



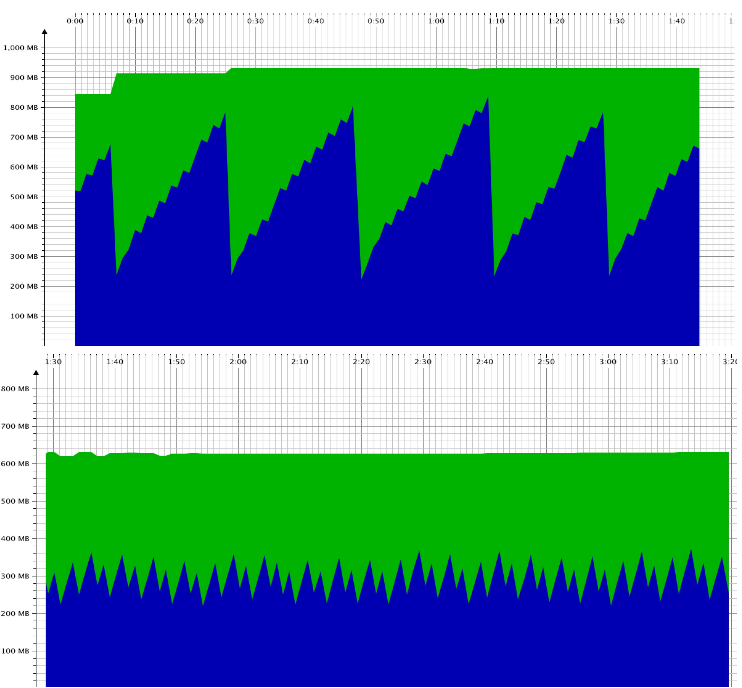
GC Demo

Parallel GC, Generational

Parallel GC, Generational

- Non-generational workload: 500ms pauses

- Generational workload: 2ms pauses



Garbage Collection Takeaways

The default GC *will* use up all your memory and *will* result in pauses.

Creating less garbage reduces frequency of GC pauses, but not their length

To reduce the length of GC pauses

- Reduce the size of the *working set*
- Try to ensure garbage is short-lived

Worth trying out different garbage collectors if you are seeing unwanted pauses

Taming the Java Virtual Machine

Memory Layouts

Garbage Collection

Compilation

Compilation

Javac compiles Java to Byte Code, at compile time

JVM JIT compiles Byte Code to Assembly, at runtime

Bytecode

Bytecode

```
public static void init(){
    previous = Runtime.getRuntime().totalMemory() -
               Runtime.getRuntime().freeMemory();
}
```

Bytecode

```
public static void init();
  Code:
     0: invokestatic  #2 // Method java/lang/Runtime.getRuntime:()Ljava/lang/Runtime;
     3: invokevirtual #3 // Method java/lang/Runtime.totalMemory:()J
     6: invokestatic  #2 // Method java/lang/Runtime.getRuntime:()Ljava/lang/Runtime;
     9: invokevirtual #4 // Method java/lang/Runtime.freeMemory:()J
    12: lsub
    13: putstatic     #5 // Field previous:J
    16: return
```

String Construction

```
String s1 = "" + input
```

```
String s2 = String.valueOf(input);
```

Stringify Demo

String Construction

```
String s1 = "" + input

11: new           #7  // class StringBuilder
14: dup
15: invokespecial #8  // StringBuilder."<init>":()V
18: ldc           #9  // String
20: invokevirtual #10 //
StringBuilder.append:(LString;)LStringBuilder;
23: iload_1
24: invokevirtual #11 //
StringBuilder.append:(I)LStringBuilder;
27: invokevirtual #12 //
StringBuilder.toString:()LString;
```

```
String s2 = String.valueOf(input);
32: invokestatic  #13 // String.valueOf:(I)String;
```

Switch Demo

Integer Switch

```
switch((int)i){
    case 0:
        println("Hello");
        break;
    case 1:
        println("World");
}
```

```
13: lookupswitch  { // 2
                0: 40
                1: 48
          default: 53
     }
40: ldc           #7 // String Hello
42: invokestatic  #8 // println:(Ljava/lang/String;)V
45: goto          53
48: ldc           #9 // String World
50: invokestatic  #8 // println:(Ljava/lang/String;)V
```

String Switch

```
switch((String)s){
    case "0":
        println("Hello S");
        break;
    case "1":
        println("World S");
        break;
}
```

```
74: invokevirtual #11                // Method java/lang/String.hashCode:()I
77: lookupswitch  { // 2
              48: 104
              49: 119
         default: 131
    }
104: aload_3
105: ldc           #12               // String 0
107: invokevirtual #13               // Method
java/lang/String.equals:(Ljava/lang/Object;)Z
110: ifeq          131
113: iconst_0
114: istore        4
116: goto          131
119: aload_3
120: ldc           #14               // String 1
122: invokevirtual #13               // Method
java/lang/String.equals:(Ljava/lang/Object;)Z
125: ifeq          131
128: iconst_1
```

String Switch

```
74: invokevirtual #11    // String.hashCode:()I
77: lookupswitch  { // 2
              48: 104
              49: 119
         default: 131
    }
104: aload_3
105: ldc           #12   // String 0
107: invokevirtual #13   // String.equals:(Object;)Z
110: ifeq          131
113: iconst_0
114: istore        4
116: goto          131
119: aload_3
120: ldc           #14  // String 1
```

```
122: invokevirtual #13  // String.equals:(Object;)Z
125: ifeq          131
128: iconst_1
129: istore        4
131: iload         4
133: lookupswitch  { // 2
               0: 160
               1: 168
         default: 173
    }
160: ldc           #15  // String Hello S
162: invokestatic  #8   // Method println:(String;)V
165: goto          173
168: ldc           #16  // String World S
170: invokestatic  #8   // Method println:(String;)V
```

Why Read Bytecode?

Understand what your code compiles to

- Understanding performance characteristics

Debugging frameworks that muck with bytecode

- AspectJ
- Javassist

Working with non-Java languages (Scala, Clojure, Groovy, …)

- These all speak Bytecode

Assembly

The JIT compiler is not a black box

You can see the actual assembly that gets run

https://www.ashishpaliwal.com/blog/2013/05/jvm-how-to-see-assembly-code-for-your-java-program/

Assembly

```
for(int i = 0; i < count; i += 1){
    items[i] = new int[2];
}
```

```
0x00000001121d2c52: mov    %rbx,0x8(%rax,%rsi,8)
0x00000001121d2c57: dec    %rsi
0x00000001121d2c5a: jne    0x00000001121d2c52  ;*newarray
                                               ; - Memory::main@22 (line 22)
```

JIT Demo

Why Read Assembly?

Next level of “Truth” underneath the bytecode

What is *actually* getting run on my processor?

java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly

Polymorphism

```
interface Hello{
    int get();
}
class HelloOne implements Hello{
    public int get(){
        return 1;
    }
}
class HelloTwo implements Hello{
    public int get(){
        return 2;
    }
}
```

```
for(int j = 0; j < 100; j++){
    for(int i = 0; i < count; i++){
        evenTotal += input[i].get();
    }
}
```

Polymorphism Demo

Polymorphism

```
interface Hello{
    int get();
}
class HelloOne implements Hello{
    public int get(){
        return 1;
    }
}
class HelloTwo implements Hello{
    public int get(){
        return 2;
    }
}
```

```
for(int j = 0; j < 100; j++){
    for(int i = 0; i < count; i++){
        evenTotal += input[i].get();
    }
}
```

# of subclasses	Time Taken
1	1595ms
2	2234ms
3	4533ms
4	4460ms

Polymorphism

```
for(int j = 0; j < 100; j++){
    for(int i = 0; i < count; i++){
        evenTotal += input[i].get();
    }
}
```

Polymorphism: Bytecode

```
for(int j = 0; j < 100; j++){
    for(int i = 0; i < count; i++){
        evenTotal += input[i].get();
    }
}
```

```
166: iload           13
168: iload           4
170: if_icmpge       195
173: lload           10
175: aload           5
177: iload           13
179: aaload
180: invokeinterface #24,  1       // Hello.get:()I
185: i2l
186: ladd
187: lstore          10
189: iinc            13, 1
192: goto            166
195: iinc            12, 1
198: goto            156
```

Polymorphism: 2 subclasses

```
0x000000010b2859a0: mov    0x8(%r12,%r9,8),%r11d  ;*invokeinterface get
                                                  ; -Polymorphism::main@157 (line 49)
                                                  ; implicit exception: dispatches to
0x000000010b285b2f
0x000000010b2859a5: movslq %r10d,%r10
0x000000010b2859a8: add    %r14,%r10              ;*ladd
                                                  ; -Polymorphism::main@163 (line 49)

0x000000010b2859ab: cmp    $0xf800c0bc,%r11d      ;   {metadata('HelloOne')}
0x000000010b2859b2: je     0x000000010b285960
0x000000010b2859b4: cmp    $0xf800c105,%r11d      ;   {metadata('HelloTwo')}
0x000000010b2859bb: jne    0x000000010b285a32
```

Polymorphism: 3 subclasses

```
0x00000001095841eb: nop
0x00000001095841ec: nop
0x00000001095841ed: movabs $0xffffffffffffffff,%rax
0x00000001095841f7: callq  0x00000001094b0220  ; OopMap{[248]=Oop off=4732}
                                                ;*invokeinterface get
                                                ; - Polymorphism::main@180 (line 49)
                                                ;   {virtual_call}
0x00000001095841fc: movslq %eax,%rax
0x00000001095841ff: mov    0xe8(%rsp),%rdx
0x0000000109584207: add    %rdx,%rax
0x000000010958420a: mov    0xf0(%rsp),%ecx
```

Compilation Takeaways

- Dumb code runs faster

- Even if it compiles to the exact same bytecode!

Taming the Java Virtual Machine

Taming the Java Virtual Machine

Memory Layouts

Garbage Collection

Compilation

Taming the Java Virtual Machine

Memory Layouts

- `OutOfMemoryError`

Garbage Collection

- Long pauses

Compilation

- Mysterious performance issues

Taming the Java Virtual Machine

```java
class Simple{
    public static void main(String args[]){
        String s = "Hello Java";
        int i = 123;
        System.out.println(s + 123);
    }
}
```

Taming the Java Virtual Machine

Li Haoyi, Chicago Scala Meetup, 19 Apr 2017