Runtimes

George Burgess IV / dyreshark

VTLUUG Talk

Credentials

- i.e. Me throwing names at you hoping you'll think I'm competent
- Also a chance for you to get used to my presentation style

Worked at

- DataTactics
- Microsoft
  - OS Security
  - CLR
- [This summer] Google Research
  - For compilers/runtimes

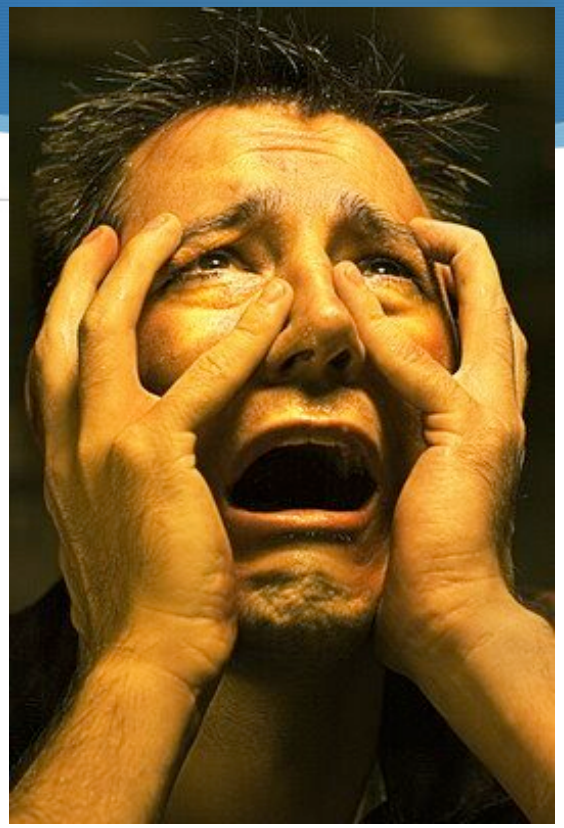
More bragging

- Worked with
  - C/C-with-classes-and-templates (6-7 years)
    - C++: 2-3 years
  - Python (5 years)
  - Java (6 years)
  - FP (Haskell, Clojure, … fanboi for ~2 years)

So what will this be about?

- Up to you
  - Three or four major sections
  - Spend more time where people are interested
  - JVM == main runtime
    - Can also talk about
      - CLR (Yup)
      - CPython (Ehh)
      - LLVM (Not-so-much)

THERE ARE SO MANY RUNTIMES. WHYYYY



- I know.
- Lots of the concepts transfer over. Trust me.

Note:

- When I say "**the** JVM" or "**the** CLR", I'm talking about Oracle's HotSpot and Microsoft's VM implementation.
  - Mono and OpenJDK mostly similar to their big-company-developed counterparts
  - I make no guarantees that what I say holds for all of them

Section I: Basics

- Quick aside…

First: What is a runtime?

- Audience?

Runtime?

Runtime

- Broadest sense is any sort of library that manages program state
  - Yes, even C has a "runtime"
  - …But we don't care about that one.
    - We're going to deal with more complex ones!
      - YAY!

So our focus will be on…

Runtimes that sit in languages at or higher-leveled than Java.

EWWW JAVA

…So, what's in one of those?

- Primitive "runtimes" are bytecode interpreters.

```
if (program[i] == ADD_BYTE) {
  stack.push(stack.pop() + stack.pop());
} else if …
```

…What else?

- They also generally have GCs (Garbage Collectors)

```
while (true) {
        new Object();
}
```

…And?

- Reflection!

```
def safeGetIter(q):
    if hasattr(q, '__iter__') and callable(q.__iter__):
        return q.__iter__()
```

Und?

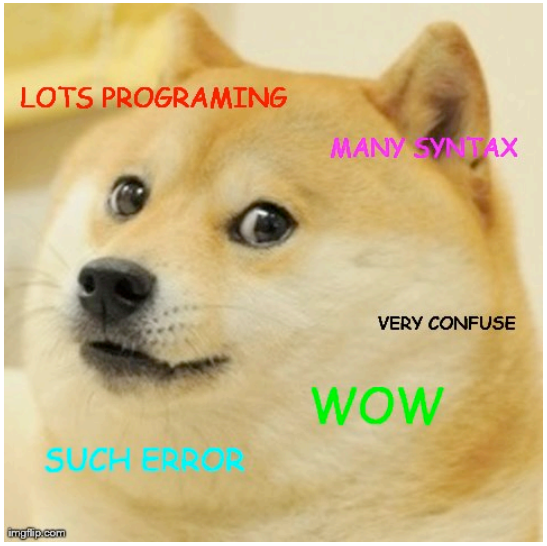
- Type-Safety

const int* arr = (const int*)”Hello, World!”; =>

CLASSCASTEXCEPTION AT

<<1 line of useful stack>>

<<100 lines of garbage>>



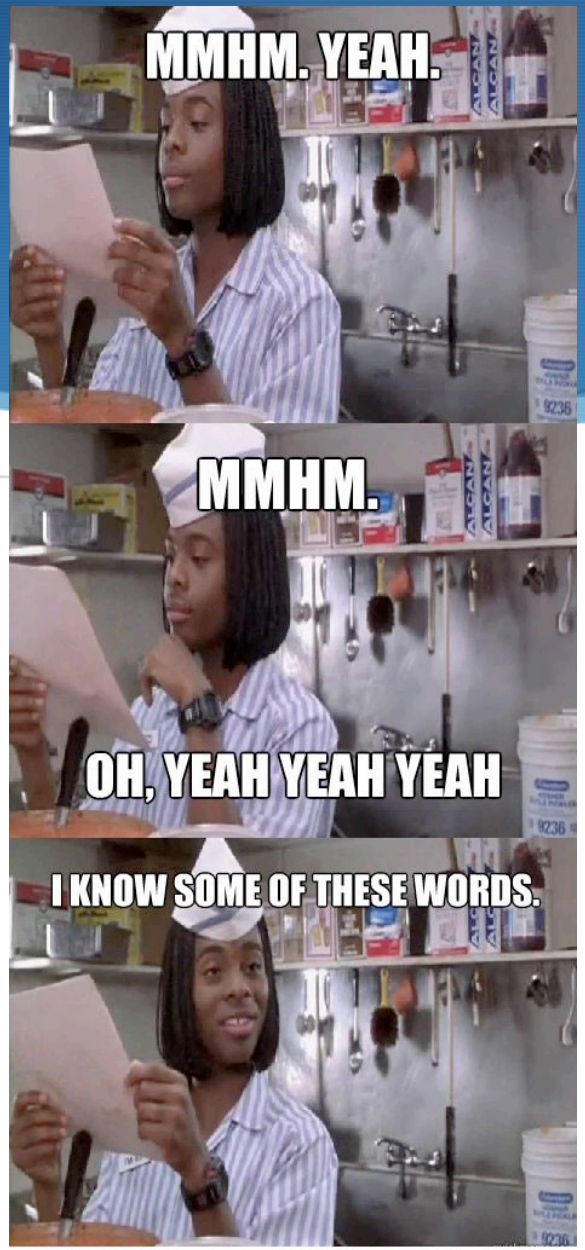
There's more?

- Array bounds-checking

```
int a[5];

int b;

a[-1] = 9001;

assert(b == 9001);
```

STOP WE GET IT

- Runtime libraries
- Code (re)loading
- Exceptions
- JIT (Just-In-Time) compiling
- Virtual dispatch everywhere
- Some level of protection over memory
- The lowest level language available is some sort of bytecode


MMHM. YEAH.
MMHM.
OH, YEAH YEAH YEAH
I KNOW SOME OF THESE WORDS.
It's time for your fun injection FakePlus.com

So that's generally what we're dealing with.

- I'm not going into all of this
  - George. What are you doing
    - This is a talk.
    - Not a 3-day long institute

I’m needy

- I want to go into three specific parts:
  - Type safety
  - GC
  - JIT

!! WARNING !!

- Each of these (sans maybe type safety) is a massive topic of its own.
  - I'll just go over them at a high level, dipping down when it gets fun.
    - Fun == "The computer does this so you can be lazy! YAY!"
  - I've literally gone on for hours about GCs before.

!!! DANGER !!!

- You need to pay attention to the first part to get how the second/third work. You can ignore me after the first part.

!!!! CRITICAL !!!!

- This space intentionally left blank

Let's go

- Type safety

But before that

- Exercise time!

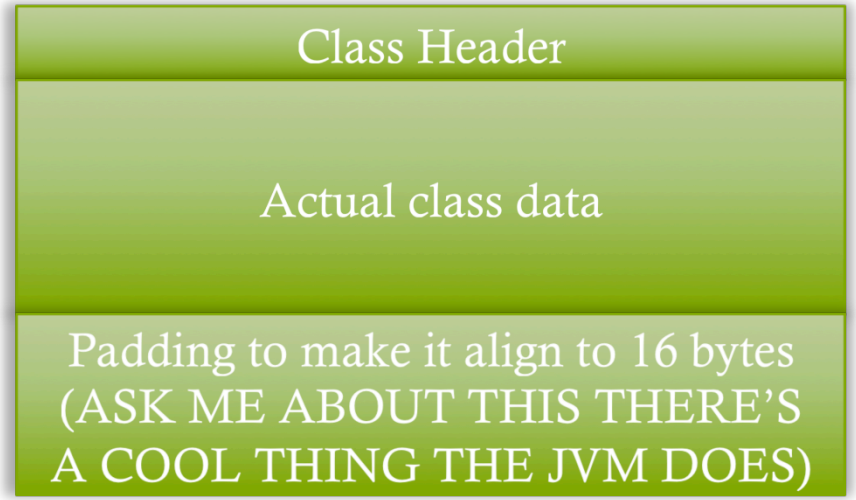
```
Object o = new Integer(1);

System.out.println(o.toString());
```

…Does it call Object.toString or Integer.toString?

Integer!

- Yup. It calls Integer.toString.
- How does it know to do that?

Layout of a class when you call `new`



…Header?

- A magic number to identify the class
  - Pointer or index into global array
    - Gets you to a struct that has all of the class info
      - Superclass, interfaces, method pointers, …
        - LOOK HOW FAR I CAN NEST THESE
          - WOO HOO TABS

So what does this mean?

- For any given object, you can “cheaply” look up just about anything about it

What else does this mean?

- Every time you're not dealing with a concrete class with 0 children, you **have** to do virtual dispatch
  - i.e. instead of `call 0x1234`, you have to:
    - (Sometimes) load address of methods array
    - Load index in methods array
    - Call the result of that load

…So method calls are more expensive.

Especially when it comes to interfaces!

- Wat. Why.

Exercise!

- Say I had:

```
class Foo extends Object {
        public String toString() { return “Foo”; }
}
```

…What would the methods array look like?

<Insert title here>

Object

Method Index	Address
0 (toString)	0x100
1 (wait)	0x200
2 (hashCode)	0x300
…	

Foo

Method Index	Address
0 (toString)	0x400
1 (wait)	0x200
2 (hashCode)	0x300
…	

…So

- That's exactly how it's done
- Calling toString is effectively
  - call classPtr->methods[0];
    - Index not guaranteed to be 0.
    - But it will be constant throughout the program's execution.

We **can't** do this with interfaces though.

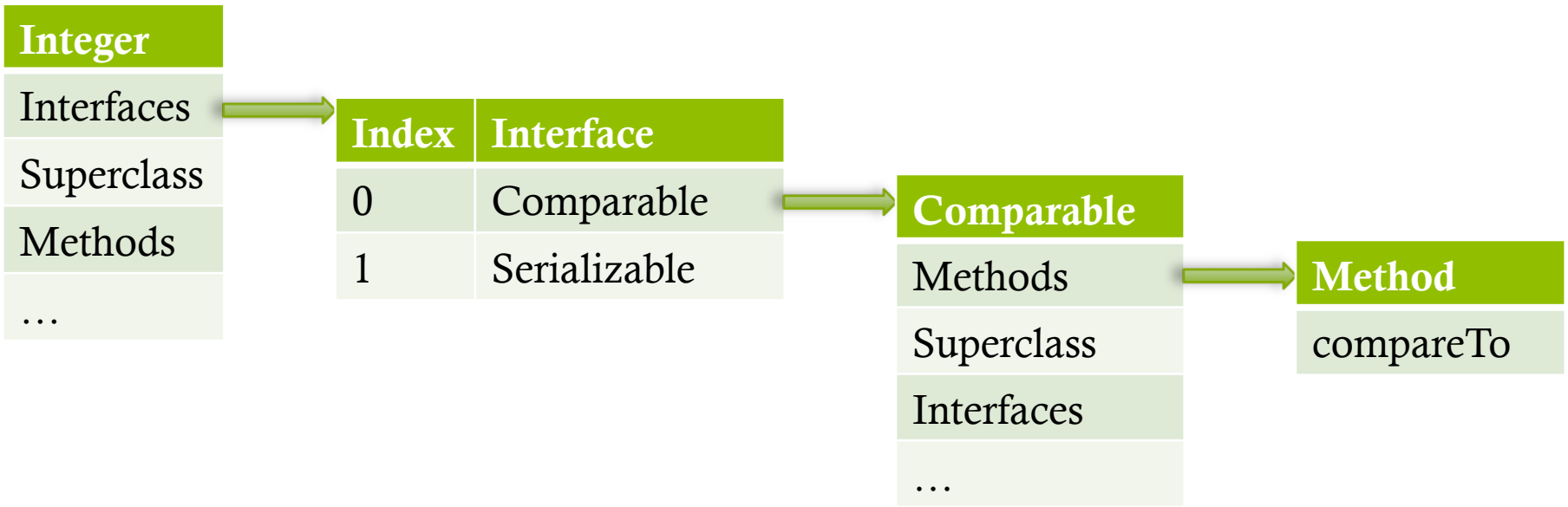
- Can implement 30 interfaces
  - How would we guarantee that IFoo.bar is **always** index N in an array?
    - …Give each interface its own methods array

…So for interface dispatch

- We need to go through the interfaces array
  - Find the interface we're looking for (linear/logarithmic time WRT number of interfaces)
    - Constant matters more than Big-Oh*
  - THEN load the methods from THAT interface
  - And go at our offset
  - And call that.

Visualization

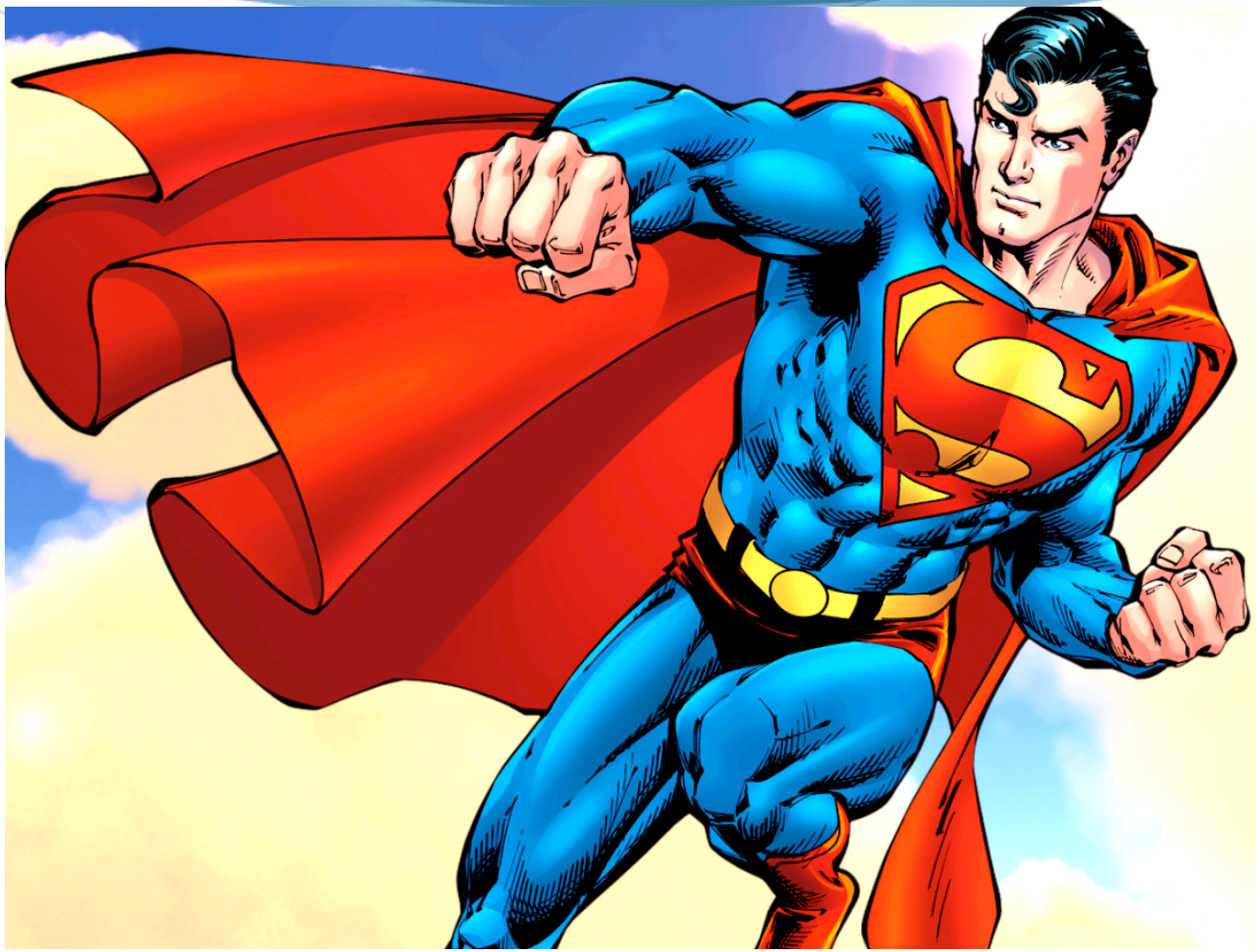
```
Comparable<Integer> i = new Integer(1);

i.compareTo(4);
```



Yes. We just did multiple comparisons, multiple loads, etc…

- ALL TO CALL A SINGLE METHOD.
  - WHY.
  - THIS IS TERRIBLE.
  - WHYYYYYYYYYYYYYYYYYYYYY

Enter: JITs



Questions? Comments? Stretch break?

Welcome to part 2. Where things get *really* fun.

DISCLAIMER

- Very few people can guarantee when lots of optimizations will happen
  - I'm not one of them
- Profile then "optimize".
  - But you can keep some of these in mind and mention them in code reviews
    - Makes you look smart and stuff

What is a JIT?

- Just-In-Time compiler.
  - It turns bytecode (what Java/Python/C#/etc compile to) to machine code (x86, ARM, …) at runtime
    - Why not just compile ahead of time (AOT)?
      - This is done in some cases
      - Has advantages
      - Has disadvantages
      - Interested? Ask. Else, we'll keep going.

First off, how are they helpful?

- They turn

```
if (program[i] == ADD_BYTE) {
  stack.push(stack.pop() + stack.pop());
} else if …
```

- Into

```
addl 4(%ebp), 8(%ebp)
```

…So?

- See JITTest.java

Wao.

- Yes. Wao.


BUKIDNON
LANAO
DEL NORTE
LANAO
DEL SUR
WAO
Illana Bay
COTABATO
MAGUINDANAO

How do they do this? Does compiling *really* help that much

- Short answer: no.
- Long answer: It's a combination of compiling and optimizing.

Okay, okay, hold up. Why the "warmup" of 3,000 loops?

- This. Is. HOTSPOT!
  - Java won't JIT a method until it's been run a lot
    - Otherwise startup time suffers
      - JITing is EXPENSIVE.
        - ^ Moar? Ask about that

Okay… So what does JITing do, in terms of optimizations?

- A LOT.
  - Actually, most of the things GCC does
    - But not everything
    - But a lot more too
      - What. But isn't C fast?
        - Yes.

So how does Java do MORE? It's so SLOW.

- 2-4x slower than C in "the normal case" (i.e. enterprise apps).
- The language speaks at a higher level, which means more assumptions can be made.*
  - It **knows** you can only access inside of arrays
  - It **knows** you can't change class headers
  - It **knows** you can't do invalid casts
  - …So it takes advantage of all of this.

Examples of how JITTest could be optimized.

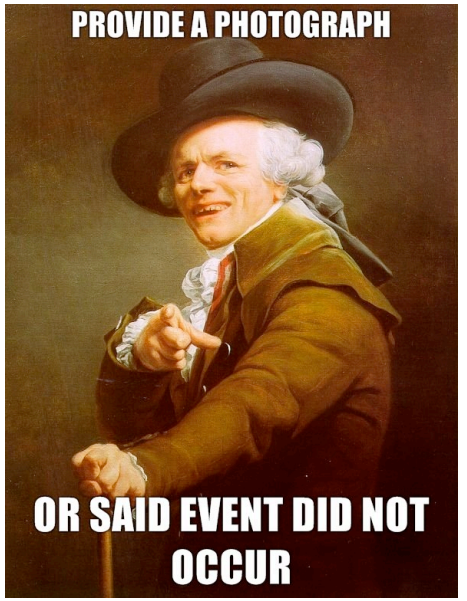
- Escape analysis – no need to heap alloc
- Constant prop.
- Virtual method call -> direct method call -> inlined
  - Okay, the virtual -> direct itself doesn't matter **so much**
    - Cache preloads
    - Branch prediction + speculative execution…
    - All make sure it's as fast as possible

Let's go deeper…

- Because I sorta lied earlier.
- Comparable.

What do we know?

- Comparable is an *interface*
  - From earlier, your reaction should be "EWWWW"
  - Let's walk through a few things
    - Namely PICs
      - "MICs" on the CLR



…So virtual dispatch one more time.

- Marking a method as final means that the method cannot be overridden.
  - Will this help performance?

NO. The JVM is smarter than that.

- (At the cost of your arm, leg, and RAM)

What.

- When optimizing methods, it only considers what's loaded
  - Is there only one concrete implementation of an abstract class?
    - Great! Do direct calls to that concrete implementation
    - Just add a note to recompile if another implementation is added
  - This expands to "it only considers what's overridden".
    - There's one subclass of ArrayList, but it doesn't override add()?
      - Cool. No reason to virtual dispatch on add().

Stop. Breathe. Stretch? Questions?

Final stretch (ha): Garbage Collectors.

- There are multiple kinds, some overlapping:
  - Generational
  - Non-generational
  - Inaccurate
  - Accurate
  - Parallel
  - Serial
  - Refcounting*
  - Mark+sweep

The JVM

- It has 4+ standard GCs, and a few beta come in every now and then
- Generally, choose the best for your application.
- I'll not go over *all* of them, because some are still magic to me
  - Will still go over big points though

JVM GCs (Pictured)

1965 No. 2 900 Mighty Dump
Mark A. Vaught collection, Lee Klancher

The JVM's "standard" GC

- Has been G1 since JDK 7u4
- Before was concurrent mark+sweep (CMS)

CMS is easier. Let's go with that.

- …Because the basis for both is similar anyway.

What's CMS

- Concurrent
- Mark+Sweep
- Generational

Let's ignore concurrency, because YOLO.

- (It's actually really interesting, but it's added complexity we don't need. Oracle has lots of neat articles on it; I recommend you check it out!)

- Aside: MRI uses a generational M&S GC

Mark+Sweep?

```
for allocation in all_allocations:
        if has_references(allocation):
            mark(allocation)

sweep_away(a for a in all_allocations if not marked(a))
```

Generational?

- The heap is "sectioned off" into multiple parts. Parts we care about:
  - Permgen
  - Eden ("New gen")
  - Survivor
  - Tenured ("Old gen")

Permgen

- .class file contents, along with a few other things. Rarely GCed, if at all. Things that are expected to live forever.

New gen

- New allocations

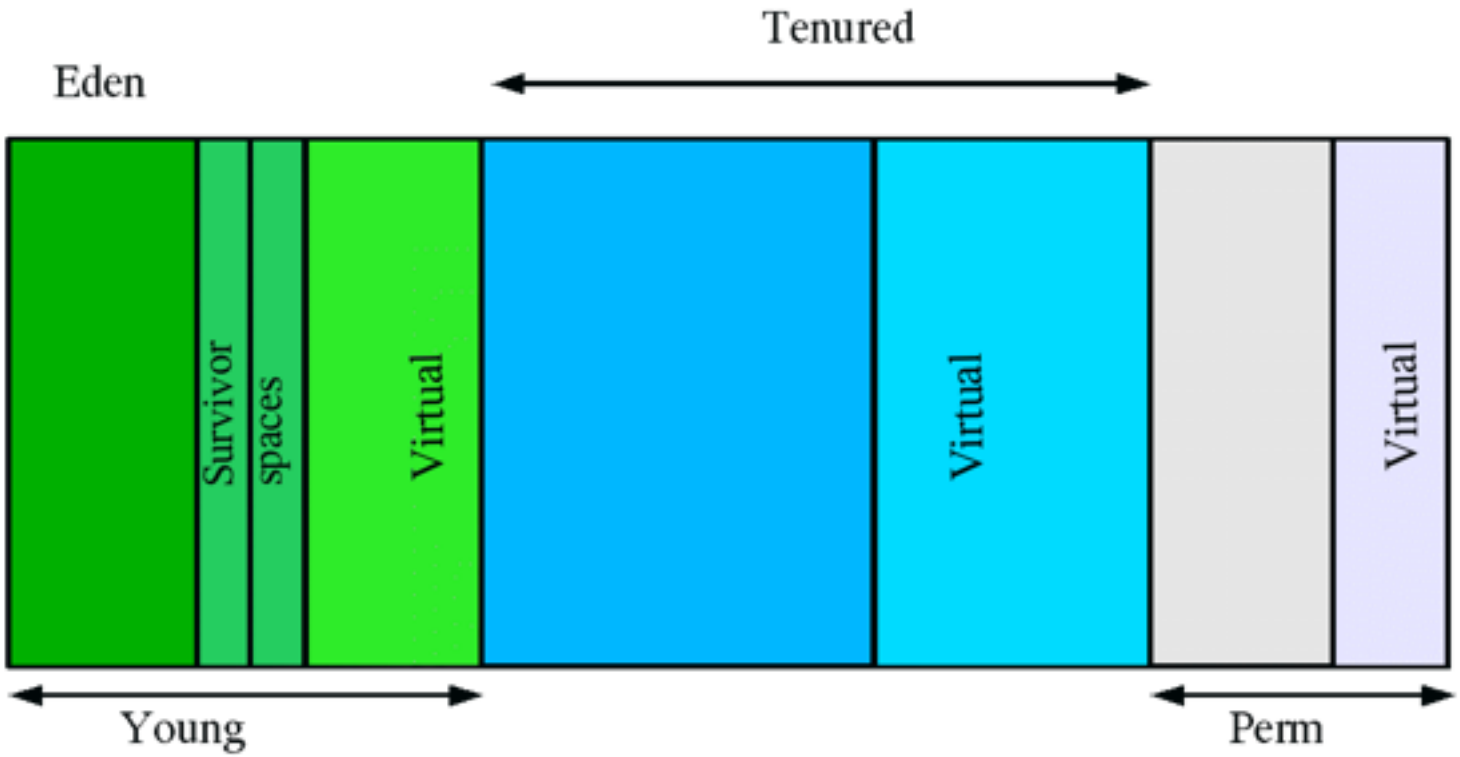
Old Gen

- OLD allocations
  - Shock and awe

Survivors

- …Teenager allocations?

How it looks



Courtesy of http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#generation_sizing

…Cool. Why?

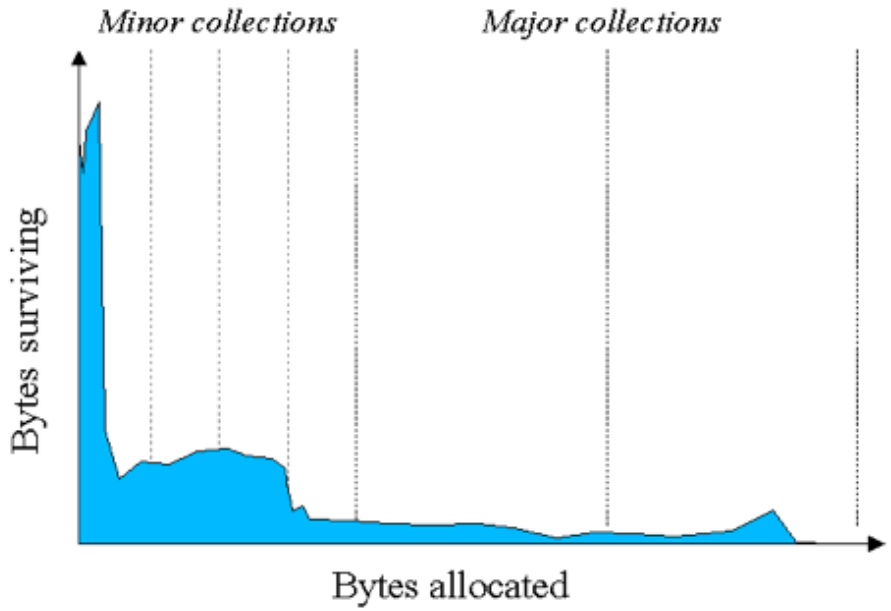
- Because new != C/C++-style malloc. New looks like:
  - void* result = eden_pointer;
  - eden_pointer += align16(sizeof(NewObject) + 4);
  - return result;
    - 16-byte alloc alignment
    - Just a pointer bump.
    - Gets more interesting when parallel happens

WHAT?!

- Yes. New is incredibly cheap.
  - This is because Eden GCs just walk Eden, finding all allocations that are still alive, and pop them in "survivor" territory.
  - After a survivor lives in survivor territory a few times, promoted to tenured/old gen
  - Old gen rarely collected

Isn't this arbitrary?

- Nope. See this graph of knowledge



Courtesy of http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#generation_sizing

What does it all mean?

- Most (> 90%) of allocations don't survive more than once.
- Those that do are ~50% likely to not survive to old gen
- Those that do are likely to stay there for a long time

** There are exceptions for large allocs/other things, but let's ignore those **

Takeaways

- GCs are expensive
  - But that makes `new` constant-time in the best+average cases
- If you have enough memory, you theoretically would never need to GC
- …Theoretically.

GCs done!

- WE MADE IT GUISE

Other things I can ramble about if there's interest

- Array Bounds Checks – Condition "lifting"
- JIT – Arch-specific stuff
- JIT Styles
  - JVM
    - Advantages/Disadvantages
  - CLR
    - Advantages/Disadvantages

Random commentary/AMA

- …About runtimes
- Any questions at all
- About runtimes.

Summary

- This is not going to get summarized in one slide.

Summary [2]

- Runtimes can be really cool
- Runtimes can do a lot for you
- Runtimes can take a lot of memory

Summary [3]

- The JVM is essentially the result of one brilliant idea:
  - "LET'S THROW EVERY THINKABLE OPTIMIZATION INTO ONE PIECE OF SOFTWARE AND SEE HOW IT PERFORMS"
- Has lots of GC algos
  - Most are generational
  - Lots are concurrent

Summary [4]

- Profile, then optimize. Don't optimize, then profile
  - Remember the DistanceTo thing?
  - Optimized for you without dirtying up your AbstractFactoryAdapterFacadeVisitor pattern goodness!

Summary [5]

- Thanks for your time! ☺