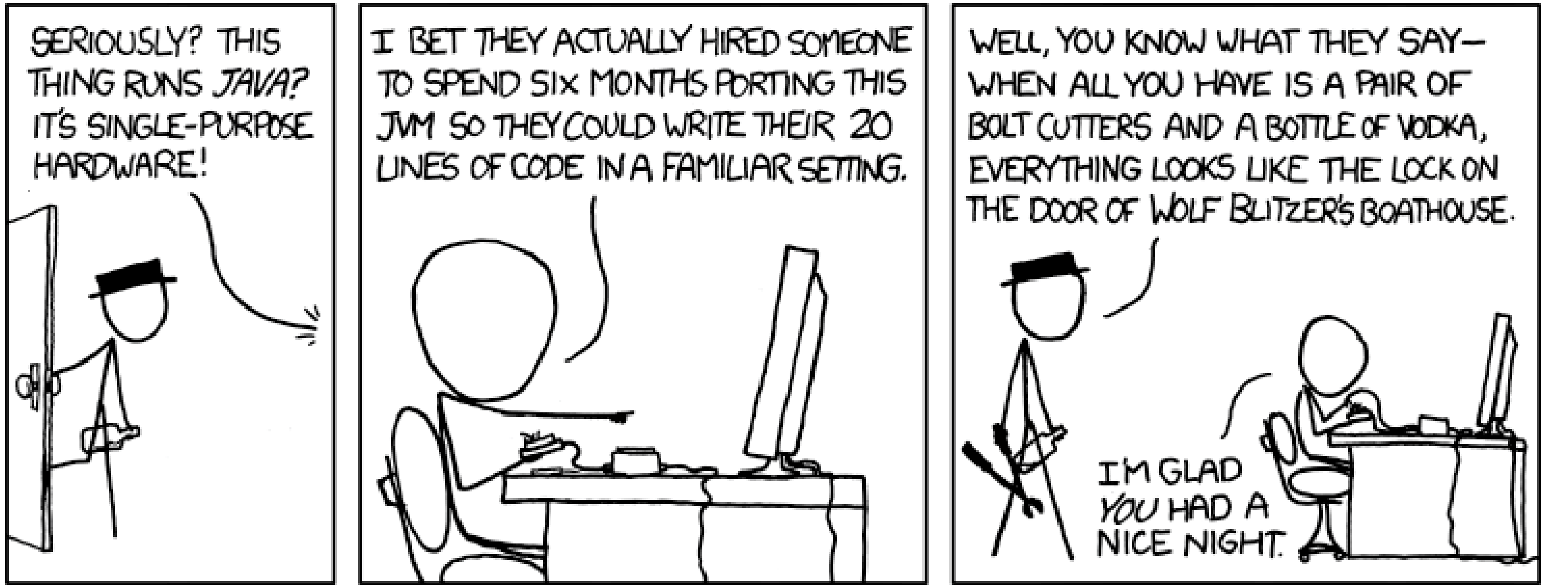
Java and C (condensed)

CSE 351 Autumn 2023

Instructor:
Justin Hsia

Teaching Assistants:
Afifah Kashif, Malak Zaki, Bhavik Soni, Naama Amiel, Cassandra Lam, Nayha Auradkar, Connie Chen, Nikolas McNamee, David Dai, Pedro Amarante, Dawit Hailu, Renee Ruan, Ellis Haker, Simran Bagaria, Eyoel Gebre, Will Robertson, Joshua Tan



http://xkcd.com/801/

Relevant Course Information

- HW25 due Wednesday (12/6)
- Lab 5 due Thursday (12/7)

- Course evaluations now open
  - See Ed Discussion post for links (separate for Lec and Sec)

- **Final Exam:** 12/11-13
  - Review Session: Friday 12/8 on Zoom, 2 hours TBD
  - Final review section on 12/7
  - Will be structured similarly to the Midterm

Potential Java Data Implementation

Java vs. C

- Reconnecting to Java (hello, CSE143!)
  - But now you know a lot more about what really happens when we execute programs
- We’ve learned about the following items in C; now we’ll see what they look like for Java:
  - Representation of data
  - Pointers / references
  - Casting
  - Function / method calls including dynamic dispatch

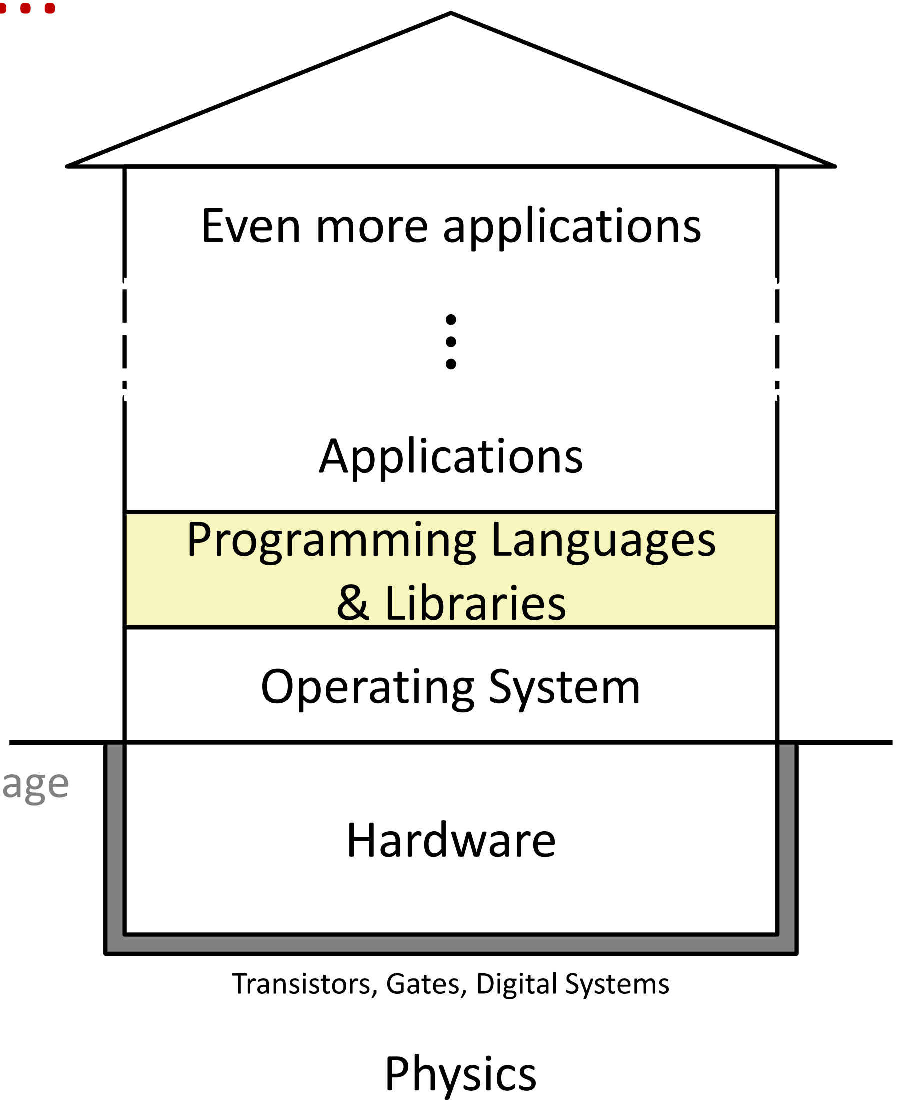
The Hardware/Software Interface

Everything applies more generally than just C!!!

- Topic Group 1: **Data**
  - **Memory**, ~~**Data**~~ **Objects**, Integers, Floating Point, **Arrays**, **Structs**
- Topic Group 2: **Programs**
  - x86-64 Assembly, **Procedures**, **Stacks**, **Executables**

These apply to execution regardless of source language

- Topic Group 3: **Scale & Coherence**
  - Caches, Memory Allocation, Processes, Virtual Memory



Lecture Meta-Point

- CSE351 has given you a "really different feeling" about what computers do and how programs execute
  - Java is not a different world – it's just a higher-level of abstraction
  - Connect these levels via how-one-could-implement-Java in 351 terms
- The Java language specification provides an ***abstraction***
  - Tells us how code should behave for different language constructs, but we can't easily tell how things are really represented
  - But it is important to understand an *implementation* of the lower levels – useful in thinking about your program
    - None of the data representations we are going to talk about are *guaranteed* by Java

Data in Java

- Integers, floats, doubles, pointers – same as C
  - *References* in Java are much more constrained than C *pointers* in that they can only point to [the starts of] objects
  - Java's portability-guarantee fixes the sizes of all types
  - No unsigned types to avoid conversion pitfalls
    - Added some useful methods in Java 8 (also use bigger signed types)
- `null` is typically represented as `0` but "you can't tell"

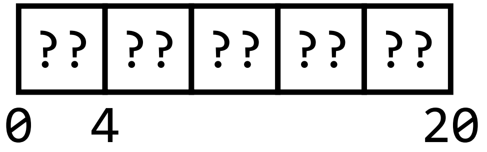
- Much more interesting:
  - **Arrays**
  - **Characters and strings**
  - **Objects**

Data in Java: Arrays

- Every element initialized to `0` or `null`
- Length specified in immutable field at start of array (`int`: 4B)
  - `array.length` returns value of this field
- *Since it has this info, what can it do?*

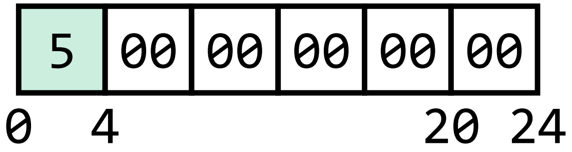
C:

```
int array[5];
```



Java:

```
int[] array = new int[5];
```

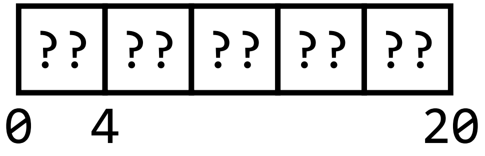


Data in Java: Arrays

- Every element initialized to `0` or `null`
- Length specified in immutable field at start of array (`int`: 4B)
  - `array.length` returns value of this field
- Every access triggers a <u>bounds-check</u>
  - Code is added to ensure the index is within bounds
  - Exception if out-of-bounds

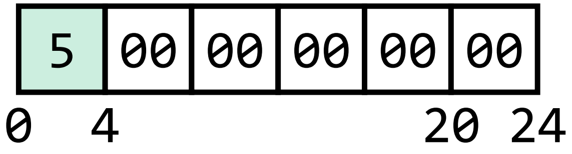
C:

```
int array[5];
```



Java:

```
int[] array = new int[5];
```



Discussion questions:

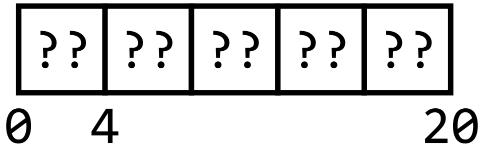
- What 351 concept does storing the array size here remind you of?
- What do you think the act of bounds-checking looks like at the assembly level?

Data in Java: Arrays

- Every element initialized to `0` or `null`
- Length specified in immutable field at start of array (`int`: 4B)
  - `array.length` returns value of this field
- Every access triggers a <u>bounds-check</u>
  - Code is added to ensure the index is within bounds
  - Exception if out-of-bounds

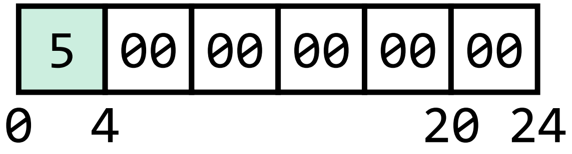
C:

```
int array[5];
```



Java:

```
int[] array = new int[5];
```



To speed up bounds-checking:

- Length field is likely in cache
- Compiler may store length field in register for loops
- Compiler may prove that some checks are redundant

Data in Java: Characters & Strings

- Two-byte Unicode instead of ASCII
- `String` not bounded by a `'\0'` (null character)
  - Bounded by hidden length field at beginning of string
  - All `String` objects read-only (vs. `StringBuffer`)

- Example: the string `"CSE351"`

C: (ASCII)

43	53	45	33	35	31	\0

Offsets: 0, 1, 4, 7

Java: (Unicode)

6	43	00	53	00	45	00	33	00	35	00	31	00

Offsets: 0, 4, 8, 16

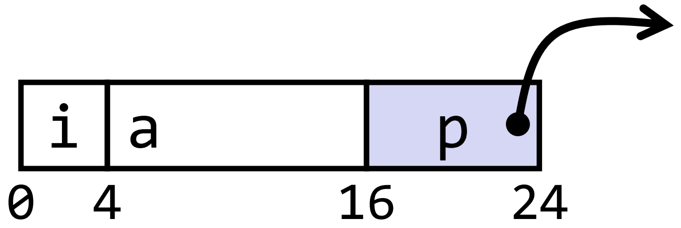
Data in Java: Objects

- Objects are always stored by reference, never stored "inline"
  - In Java, *all non-primitive variables are references to objects*
  - Access members using `r.a` notation (though just like `r->a` in C)

C:

```
struct rec {
  int i;
  int a[3];
  struct rec* p;
};
```

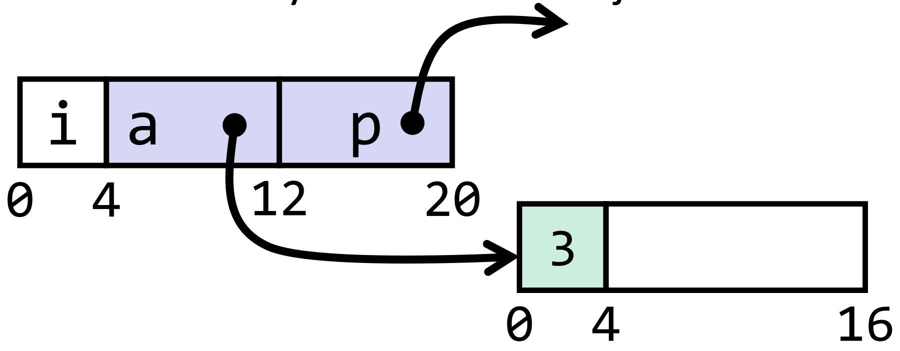
- a[] stored "inline" as part of struct



Java:

```
class Rec {
  int i;
  int[] a = new int[3];
  Rec p;
  ...
}
```

- a stored by reference in object



Struct vs. object discussion questions:

- What are the consequences for the memory layout?
- What are the consequences for the field access performance?

Casting in C (example from Lab 5)

- Can cast any pointer into any other pointer
  - Changes dereference and arithmetic behavior

```
struct block_info {
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
typedef struct block_info block_info;
...
int x;
block_info* b;
block_info* new_block;
...
new_block = (block_info*) ( (char*) b + x );
...
```

Cast b into char* to do unscaled addition

Cast back into block_info* to use as block_info struct

Type-safe casting in Java

- Can only cast compatible object references (class hierarchy)

```
class Object {
  ...
}
```

```
class Vehicle {
  int passengers;
}
```

```
class Boat extends Vehicle {
  int propellers;
}
```

```
class Car extends Vehicle {
  int wheels;
}
```

```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat     b1 = new Boat();   // |--> sibling
Car      c1 = new Car();    // |--> sibling

Vehicle v1 = new Car();
Vehicle v2 = v1;

Car      c2 = new Boat();
Car      c3 = new Vehicle();

Boat     b2 = (Boat) v;
Car      c4 = (Car) v2;
Car      c5 = (Car) b1;
```

Type-safe casting in Java

- Can only cast compatible object references (class hierarchy)

```
class Object {
  ...
}
```

→

```
class Vehicle {
  int passengers;
}
```

→

```
class Boat extends Vehicle {
  int propellers;
}
```

```
class Car extends Vehicle {
  int wheels;
}
```

```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat     b1 = new Boat();   // |--> sibling
Car      c1 = new Car();    // |--> sibling
```

```
Vehicle v1 = new Car();
Vehicle v2 = v1;
```

← ✓ Everything needed for Vehicle also in Car

← ✓ v1 is declared as type Vehicle

```
Car     c2 = new Boat();
Car     c3 = new Vehicle();
```

← ✗ Compiler error: Incompatible type – fields in Car that are not in Boat (siblings)

← ✗ Compiler error: Wrong direction – fields Car not in Vehicle (wheels)

```
Boat    b2 = (Boat) v;
Car     c4 = (Car) v2;
Car     c5 = (Car) b1;
```

← ✗ Runtime error: Vehicle does not contain all fields in Boat (propellers)

← ✓ v2 refers to a Car at *runtime*

← ✗ Compiler error: Unconvertable types – b1 is declared as type Boat

Java Object Definitions

```
class Point {
  double x;
  double y;

  Point() {
    x = 0;
    y = 0;
  }

  boolean samePlace(Point p) {
    return (x == p.x) && (y == p.y);
  }
}
...
Point p = new Point();
...
```

fields

constructor

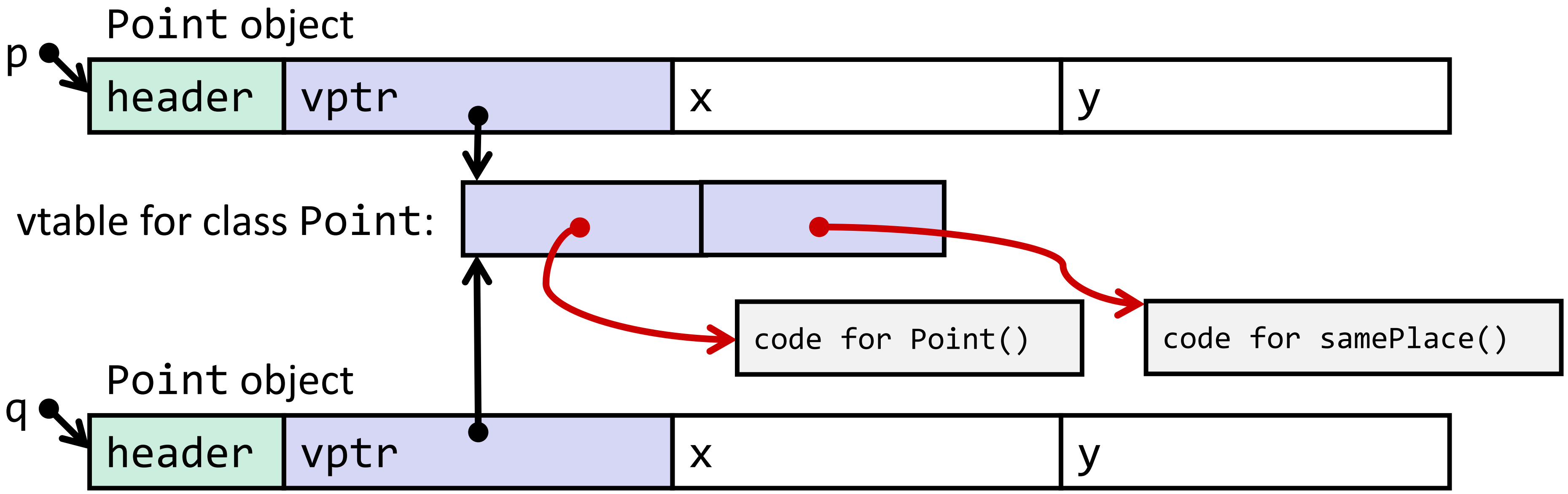
method(s)

creation

Discussion question:

- How might we represent Java objects in memory based on what we've learned in C?
  Hint: think about fields and methods separately.

Java Objects and Method Dispatch



- *Object header* : GC info, hashing info, lock info, etc.
- *Virtual method table (vtable)*
  - Like a jump table for instance ("virtual") methods plus other class info
  - One table per class
  - Each object instance contains a *vtable pointer (vptr)*

Java Constructors

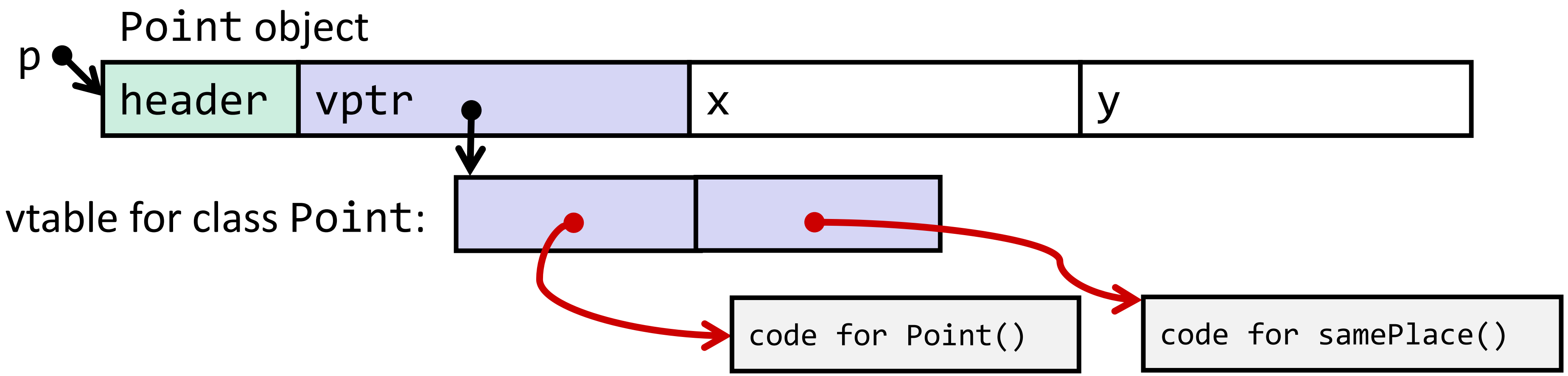
- **When we call new:** allocate space for object (data fields and references), initialize to zero/null, and run constructor method

Java:

```
Point p = new Point();
```

C pseudo-translation:

```
Point* p = calloc(1,sizeof(Point));
p->header = ...;
p->vptr = &Point_vtable;
p->vptr[0](p);
```



Java Methods

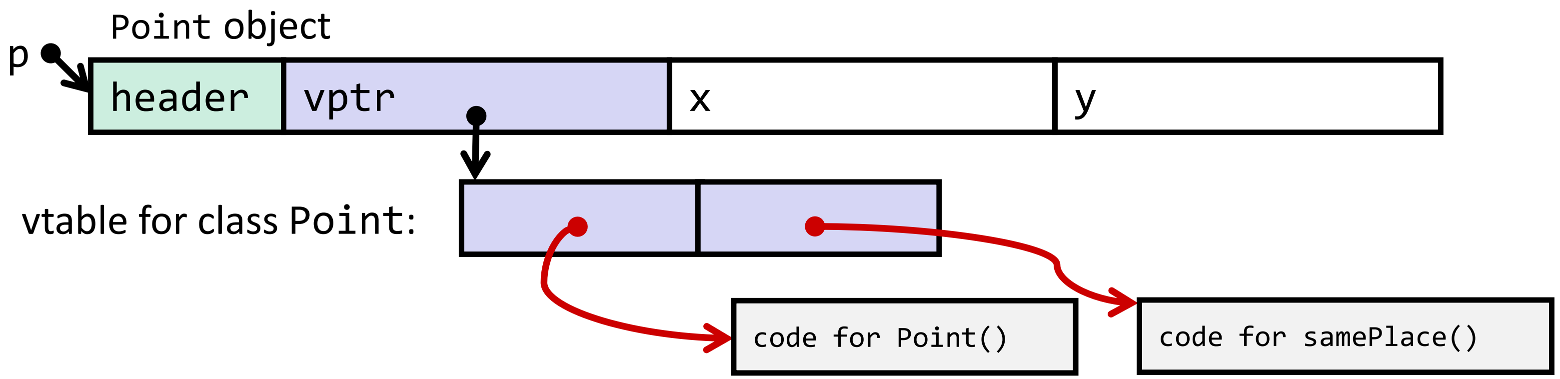
- Static methods are just like functions
- Instance methods:
  - Have an implicit first parameter for *this;* and
  - Can be overridden in subclasses
- The code to run when calling an instance method is chosen *at runtime* by lookup in the vtable

Java:

```
p.samePlace(q);
```

C pseudo-translation:

```
p->vptr[1](p, q);
```

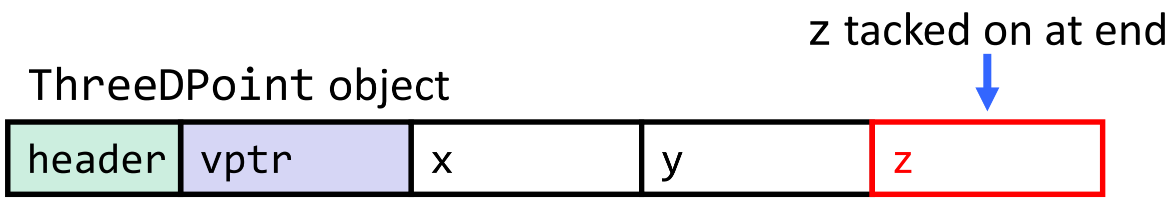


Subclassing

```
class ThreeDPoint extends Point {
    double z;
    boolean samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}
```

Subclassing

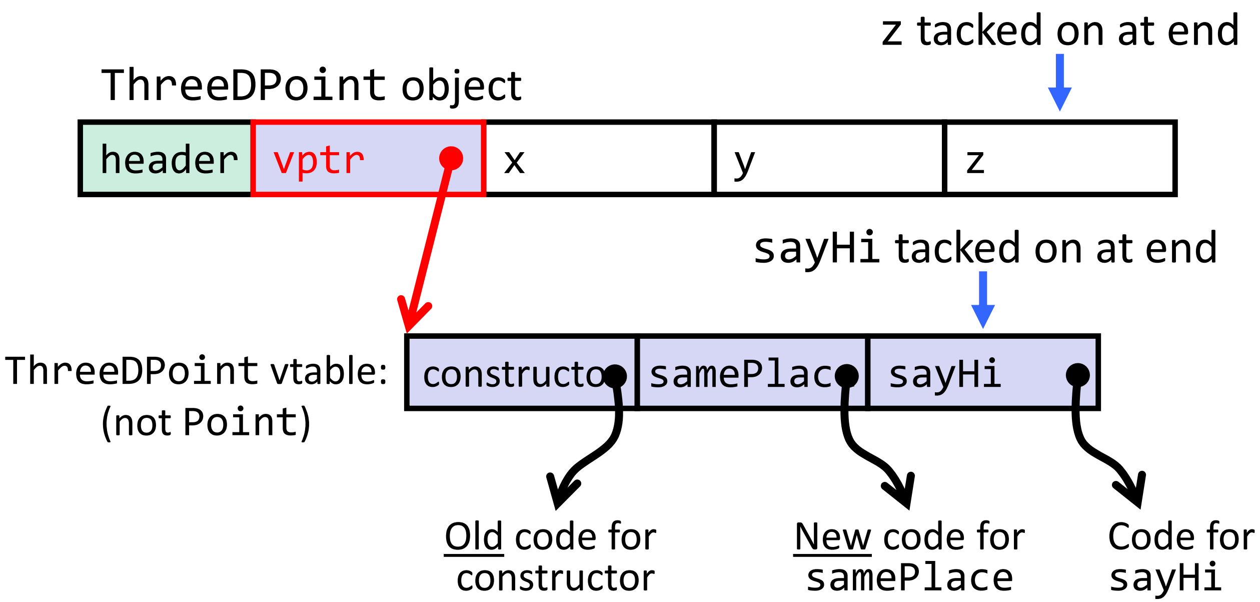
```java
class ThreeDPoint extends Point {
    double z;
    boolean samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}
```



- New fields (z) added to end of fields of subclass (x, y)
  - Point fields remain in the same place, so Point code can run on ThreeDPoint objects without modification!

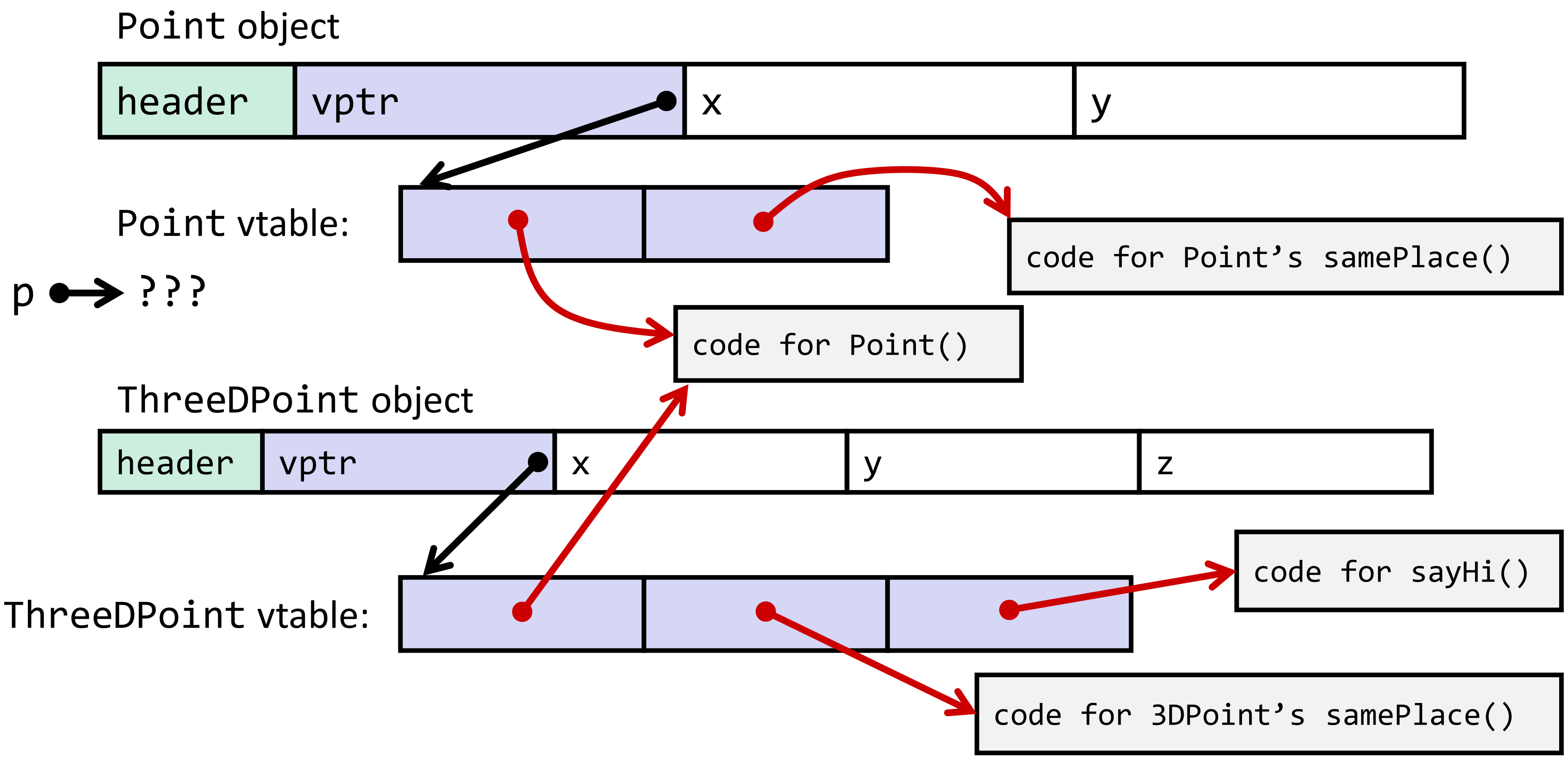
Subclassing

```
class ThreeDPoint extends Point {
    double z;
    boolean samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}
```



- Method modifications:
  - Add new pointer at end of vtable for new method "sayHi"
  - No constructor definition, so use default Point constructor
  - To override "samePlace", use same vtable position

Dynamic Dispatch



Java:

```
Point p = ???;
return p.samePlace(q);
```

C pseudo-translation:

```
// works regardless of what p is
return p->vptr[1](p, q);
```

Ta-da!

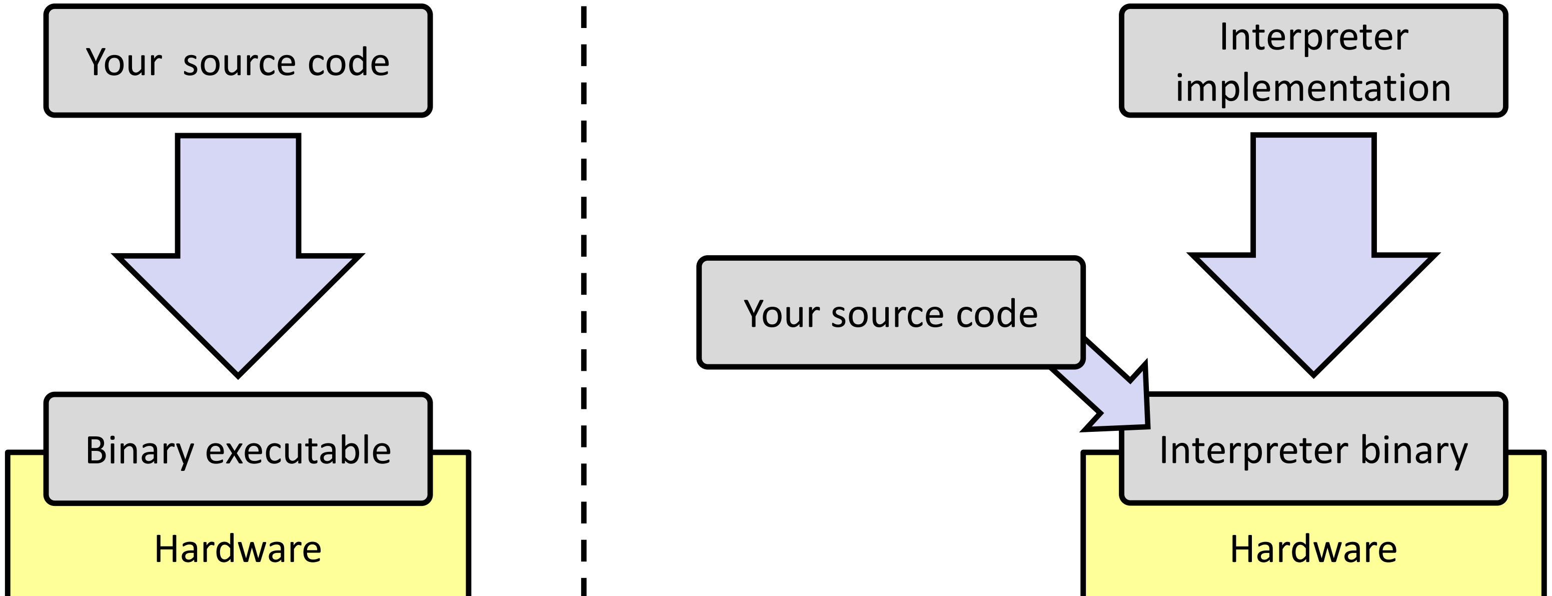
- In CSE123 or CSE143, it may have seemed "magic" that an *inherited* method could call an *overridden* method
  - You were tested on this endlessly

- The "trick" in the implementation is this part: **`p->vptr[i](p,q)`**
  - In the body of the pointed-to code, any calls to (other) methods of `this` will use `p->vptr`
  - Dispatch determined by `p`, not the class that defined a method

The Java Virtual Machine (JVM)

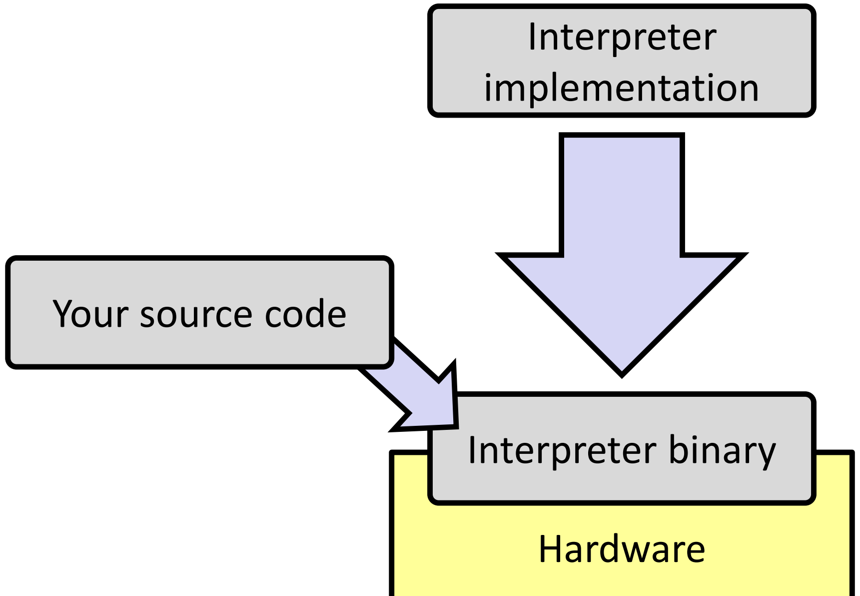
Implementing Programming Languages

- Many choices in programming model implementation
  - We've previously discussed compilation
  - One can also *interpret*
- Interpreters have a long history and are still in use
  - *e.g.*, Lisp, an early programming language, was interpreted
  - *e.g.*, Python, Javascript, Ruby, Matlab, PHP, Perl, …



Interpreters

- Execute (something close to) the *source code* directly, meaning there is less translation required
  - This makes it a simpler program than a compiler and often provides more transparent error messages
- Easier to run on different architectures – runs in a simulated environment that exists only inside the *interpreter* process
  - Just port the interpreter (program), and then interpreting the source code is the same
- Interpreted programs tend to be slower to execute and harder to optimize



Interpreters vs. Compilers

- Programs that are designed for use with particular language implementations
  - You can choose to execute code written in a particular language via either a compiler or an interpreter, if they exist
- “Compiled languages” vs. “interpreted languages” a misuse of terminology
  - But very common to hear this
  - And has *some* validation in the real world (*e.g.*, JavaScript vs. C)
- Some modern language implementations are a mix
  - *e.g.*, Java compiles to bytecode that is then interpreted
  - Doing just-in-time (JIT) compilation of parts to assembly for performance

Compiling and Running Java

1. Save your Java code in a `.java` file

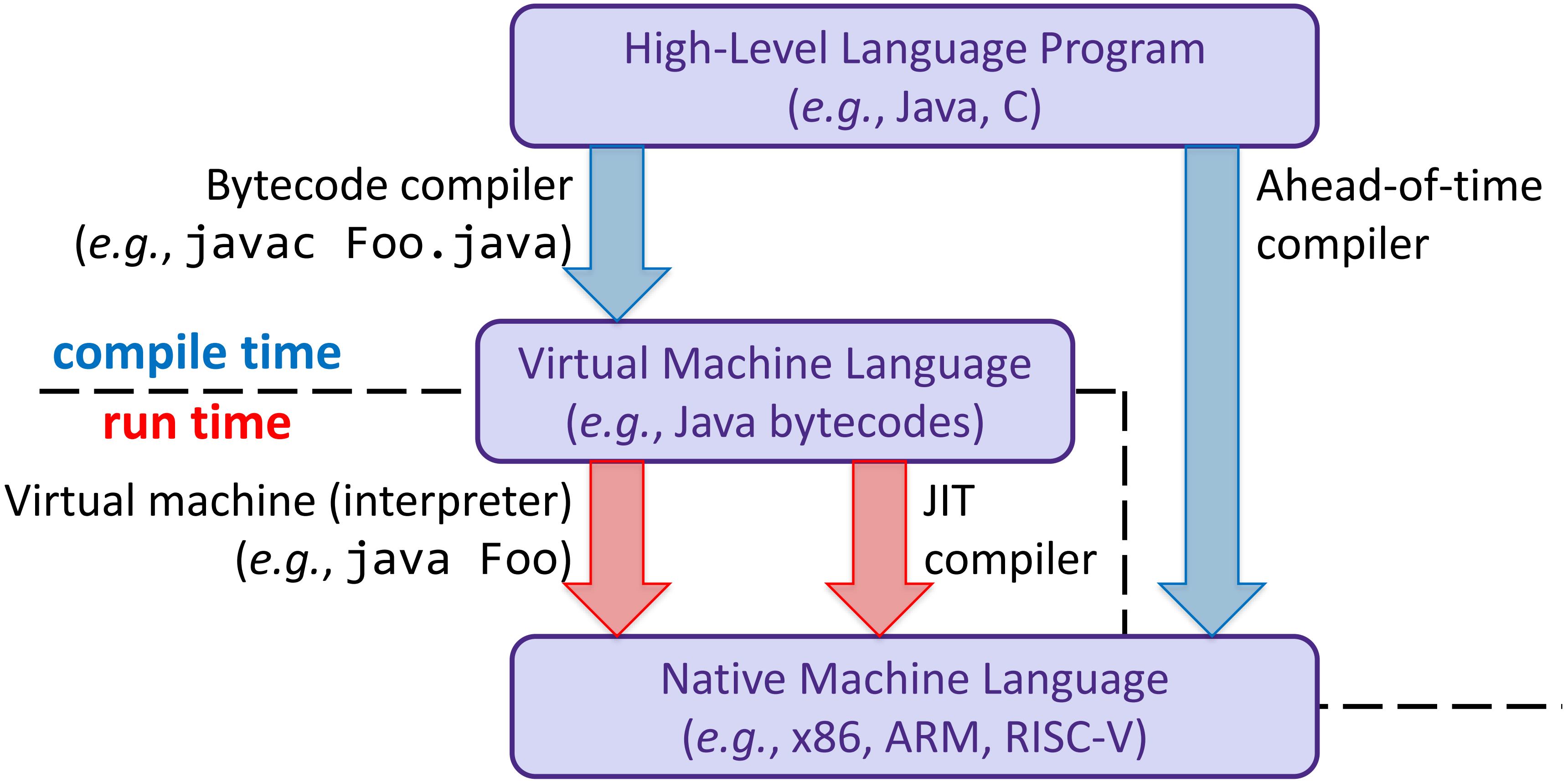
2. To run the Java compiler:
   - `javac Foo.java`
   - The Java compiler converts Java into *Java bytecodes*
     - Stored in a `.class` file

3. To execute the program stored in the bytecodes, these can be interpreted by the Java Virtual Machine (JVM)
   - Running the virtual machine: `java Foo`
   - Loads `Foo.class` and interprets the bytecodes

"The JVM"

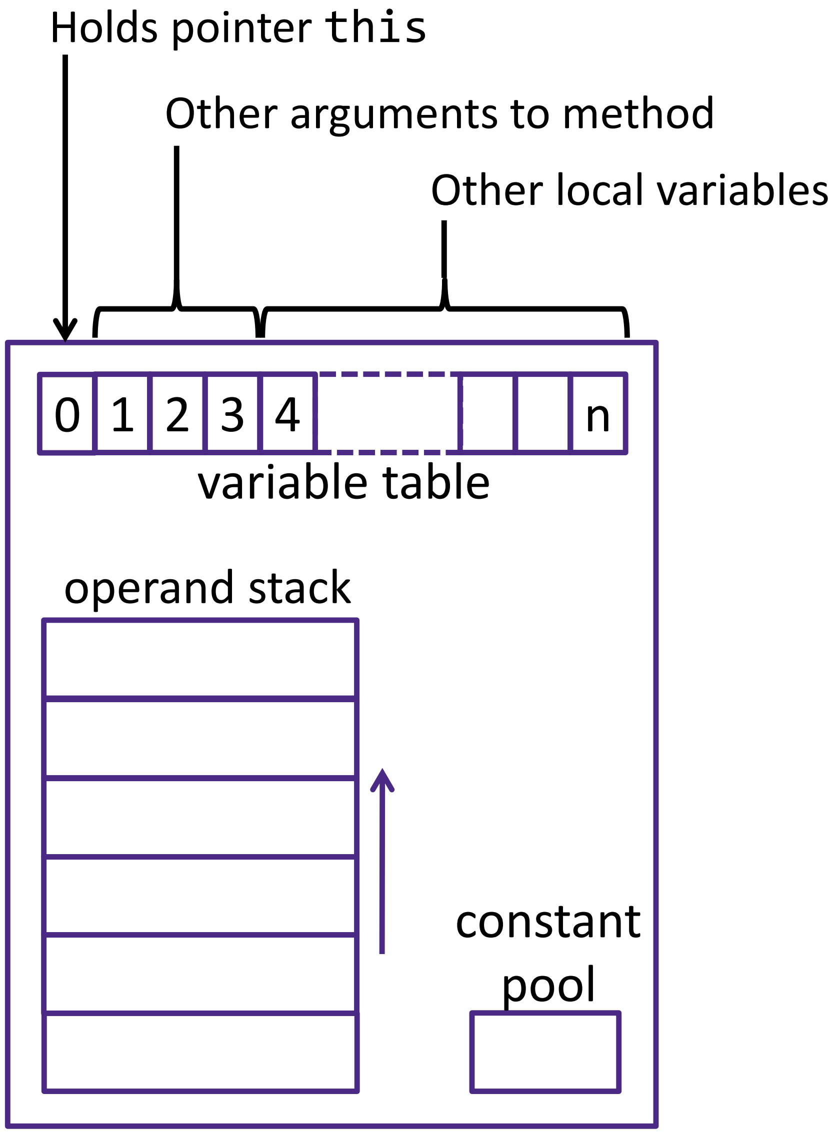
- Java programs are usually run by a Java *virtual machine* *(JVM)*
  - JVMs interpret an intermediate language called *Java bytecode*
  - Many JVMs compile bytecode to native machine code
    - **Just-in-time (JIT) compilation**
    - http://en.wikipedia.org/wiki/Just-in-time_compilation
  - Java is sometimes compiled ahead of time (AOT) like C

Virtual Machine Model



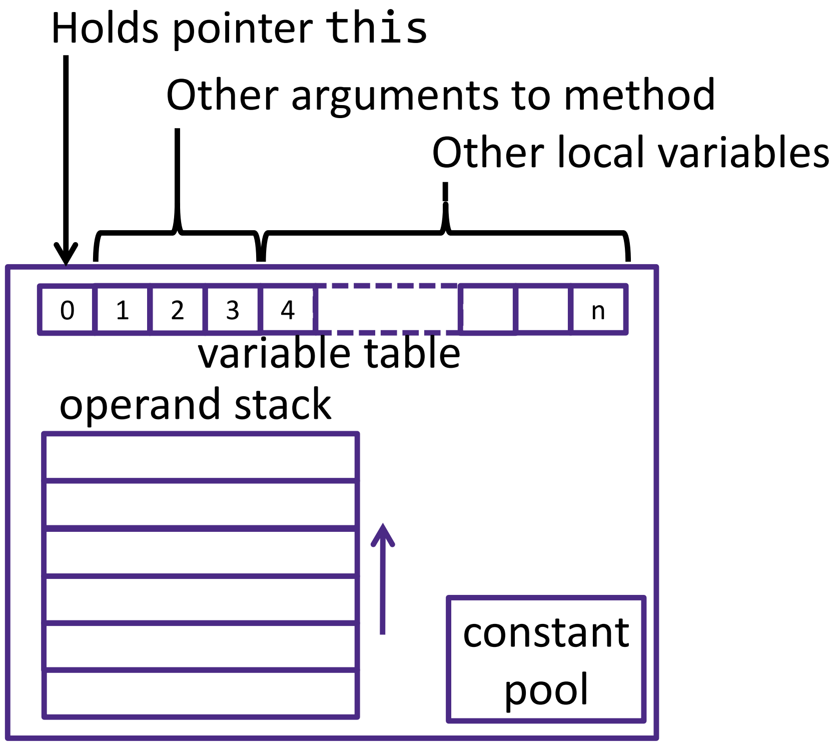
Java Bytecode

- Like assembly code for JVM, but works on *all* JVMs
  - Hardware-independent!
- Typed (unlike x86 assembly)
- Strong JVM protections



JVM Operand Stack

JVM:



‘i’ = integer,
‘a’ = reference,
‘b’ for byte,
‘c’ for char,
‘d’ for double, ...

Bytecode:

```
iload 1     // push 1st argument from table onto stack
iload 2     // push 2nd argument from table onto stack
iadd        // pop top 2 elements from stack, add together, and
            // push result back onto stack
istore 3    // pop result and put it into third slot in table
```

No registers or stack locations!
All operations use operand stack

Compiled to (IA32) x86:

```
mov 8(%ebp),  %eax
mov 12(%ebp), %edx
add %edx, %eax
mov %eax, -8(%ebp)
```

Disassembled Java Bytecode

```
> javac Employee.java
> javap -c Employee
```

http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

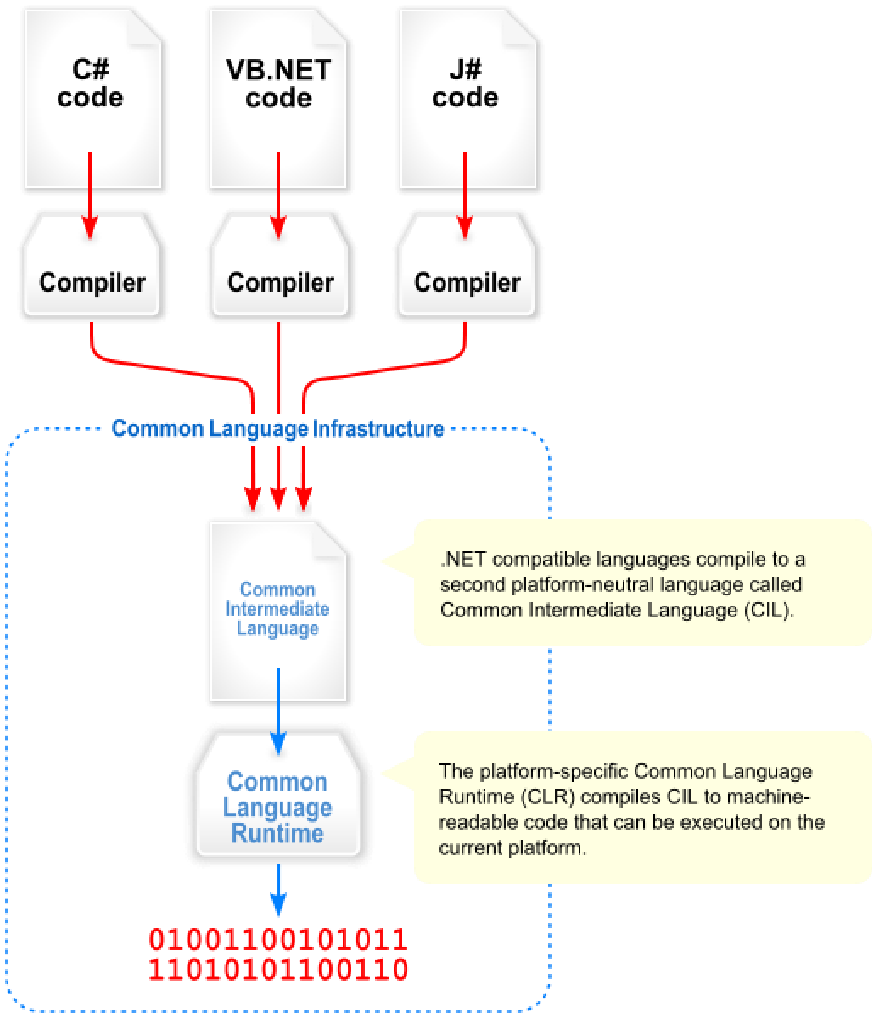
```
Compiled from Employee.java
class Employee extends java.lang.Object {
  public Employee(java.lang.String,int);
  public java.lang.String getEmployeeName();
  public int getEmployeeNumber();
}

Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object()>
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void
                      storeData(java.lang.String, int)>
20 return

Method java.lang.String getEmployeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn

Method int getEmployeeNumber()
0 aload_0
1 getfield #4 <Field int idNumber>
4 ireturn

Method void storeData(java.lang.String, int)
…
```

Other languages for JVMs

- JVMs run on so many computers that compilers have been built to translate many other languages to Java bytecode:
  - **AspectJ**, an aspect-oriented extension of Java
  - **ColdFusion**, a scripting language compiled to Java
  - **Clojure**, a functional Lisp dialect
  - **Groovy**, a scripting language
  - **JavaFX** Script, a scripting language for web apps
  - **JRuby**, an implementation of Ruby
  - **Jython**, an implementation of Python
  - **Rhino**, an implementation of JavaScript
  - **Scala**, an object-oriented and functional programming language
  - And many others, even including C!
- Originally, JVMs were designed and built for Java (still the major use) but JVMs are also viewed as a safe, GC'ed platform

Microsoft's C# and .NET Framework

- C# has similar motivations as Java
  - Virtual machine is called the *Common Language Runtime*
  - *Common Intermediate Language* is the bytecode for C# and other languages in the .NET framework



We made it! 😚 😎 😂

- Topic Group 1: **Data**
  - Memory, Data, Integers, Floating Point, Arrays, Structs
- Topic Group 2: **Programs**
  - x86-64 Assembly, Procedures, Stacks, Executables
- Topic Group 3: **Scale & Coherence**
  - Caches, Memory Allocation, Processes, Virtual Memory