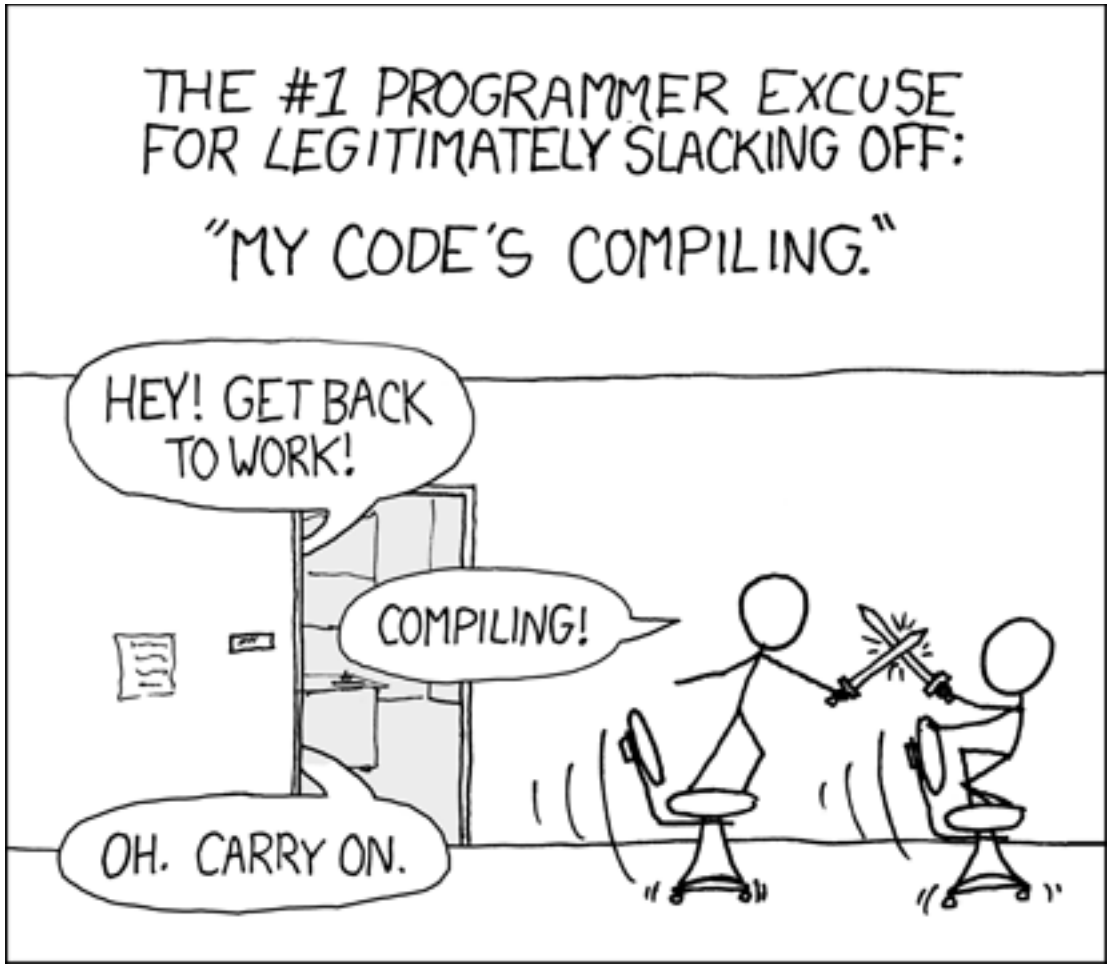
Java and C (part II)

CSE 351 Autumn 2024

Instructor:
Ruth Anderson

Teaching Assistants:
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub



https://xkcd.com/303

Relevant Course Information

- HW26 due Wednesday (12/04) @ 11:59 pm
- Final Exam Review in Section tomorrow! (12/05)
- Lab 5 (on Mem Alloc) due Thurs (12/05) @ 11:59pm
  - Closes Sunday 12/08 @11:59pm
- OPTIONAL HW on Java posted (for practice only)
- Final Exam, on Gradescope
  - Released Monday 12/09 at 12:01am
  - Due Wednesday 12/11 at 11:59pm
- Course evaluations now open - Please fill these out!
  - Separate ones for Lecture and Section

Polling Question

What would you expect to be the order of contents in an instance of the Car class?

Vote in Ed Lessons

```
class Vehicle {
  int passengers;
  // methods not shown
}
class Car extends Vehicle {
  int wheels;
  // methods not shown
}
```

A. header, Vehicle vtable ptr, passengers, Car vtable ptr, wheels

B. Vehicle vtable ptr, passengers, wheels

C. header, Vehicle vtable ptr, Car vtable ptr, passengers, wheels

D. header, Car vtable ptr, passengers, wheels

E. We're lost…

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

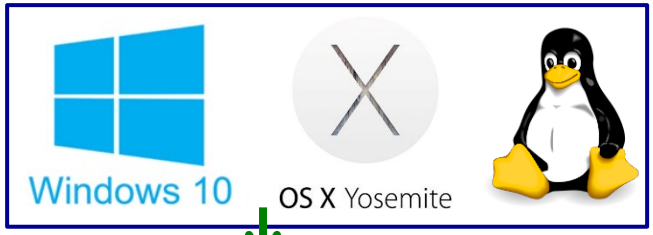
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

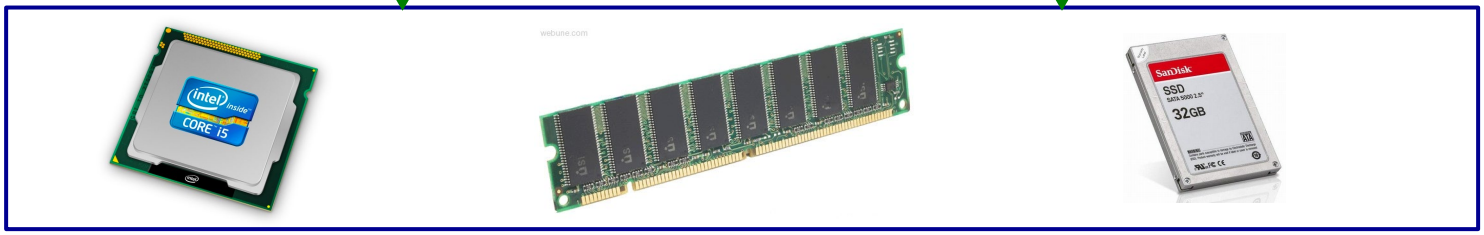
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

OS:



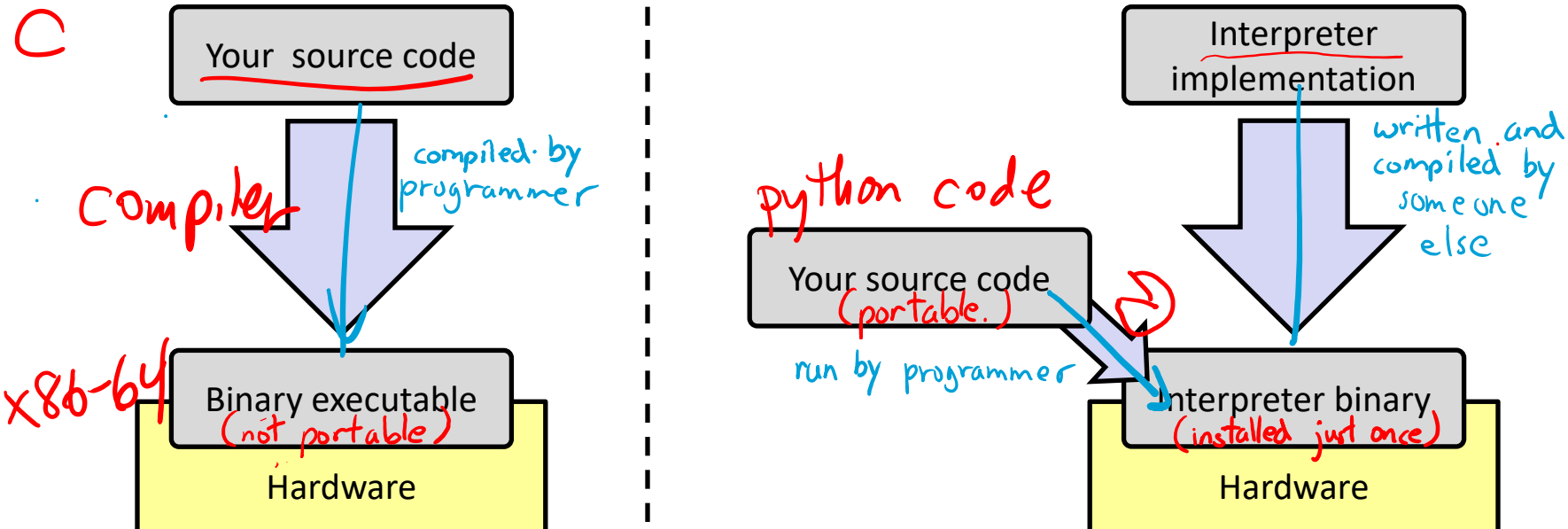
Computer system:



Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Memory allocation
Processes
Virtual memory
Java vs. C

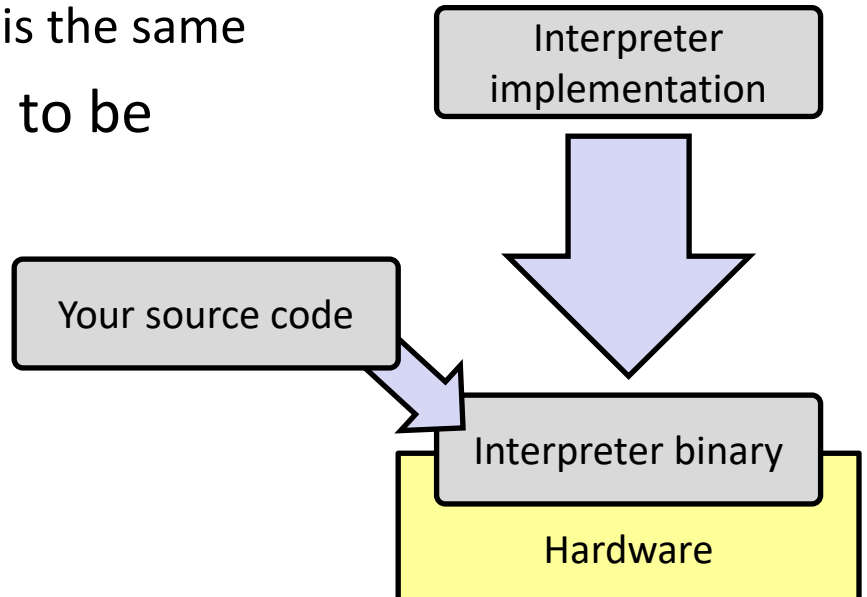
Implementing Programming Languages

- 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

Foo.java

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

2. To run the Java compiler:
   - `javac Foo.java` → produces file: Foo.class
     - 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

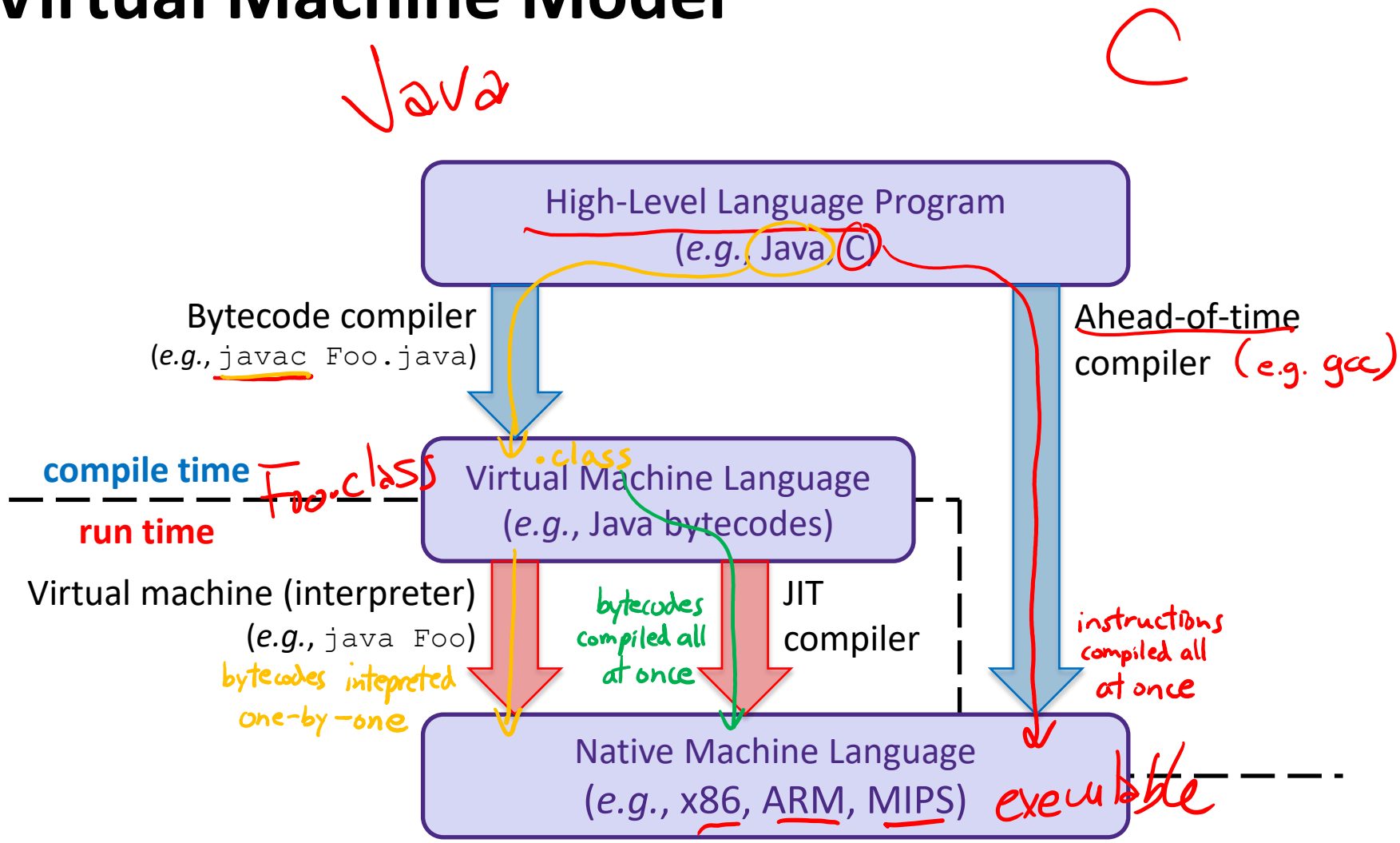
interpreter

"The JVM"

Note: The JVM is different than the CSE VM running on VMWare. Yet *another* use of the word "virtual"!

- 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



Polling Question – Answer in Ed Lessons

- You type **`javac`** and **`java`** at the command line. You provide an argument to both commands.

`javac`:

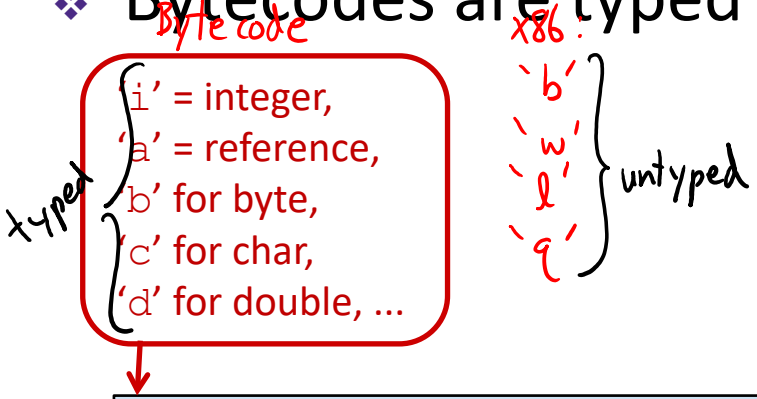
- **A) Is a: java source file/bytecode file/executable**
- **B) Its argument should refer to:** a Java source file
- **C) It does this:** Translates the source file into Java bytecodes put into a .class file

`java`:

- **A) Is a java source file/bytecode file/executable**
- **B) Its argument should refer to** the name of a class, where a .class file exists containing Java bytecodes
- **C) It does this:** Interprets the Java bytecodes

Java Bytecodes

- "Assembly code" for the Java Virtual Machine (JVM)
  - works on *all* JVMs
  - Hardware-independent! The JVM is just a program that has been compiled to run on this particular hardware
- Bytecodes are typed (unlike x86 assembly)



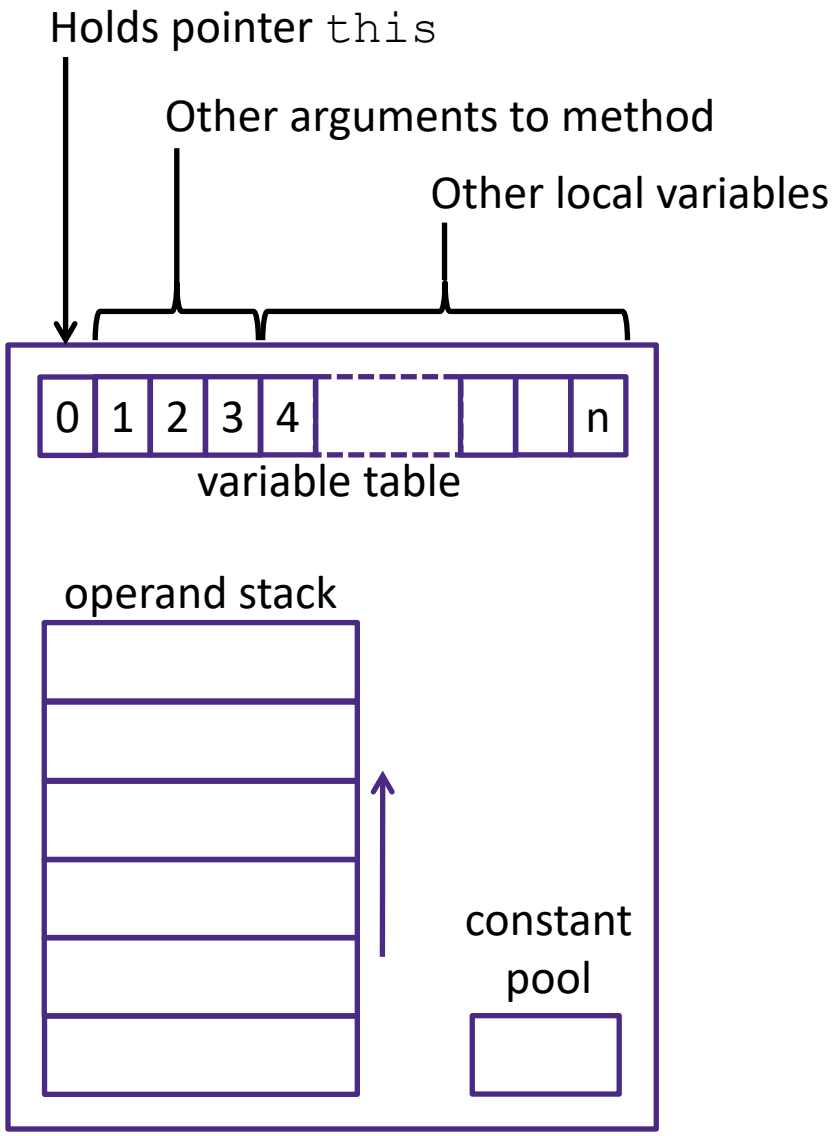

```
iload 1     // push 1st argument from variable table onto operand stack
iload 2     // push 2nd argument from variable table onto operand stack
iadd        // pop top 2 elements from operand stack, add together, and
            // push result back onto operand stack
istore 3    // pop result and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

The Java Virtual Machine (JVM)

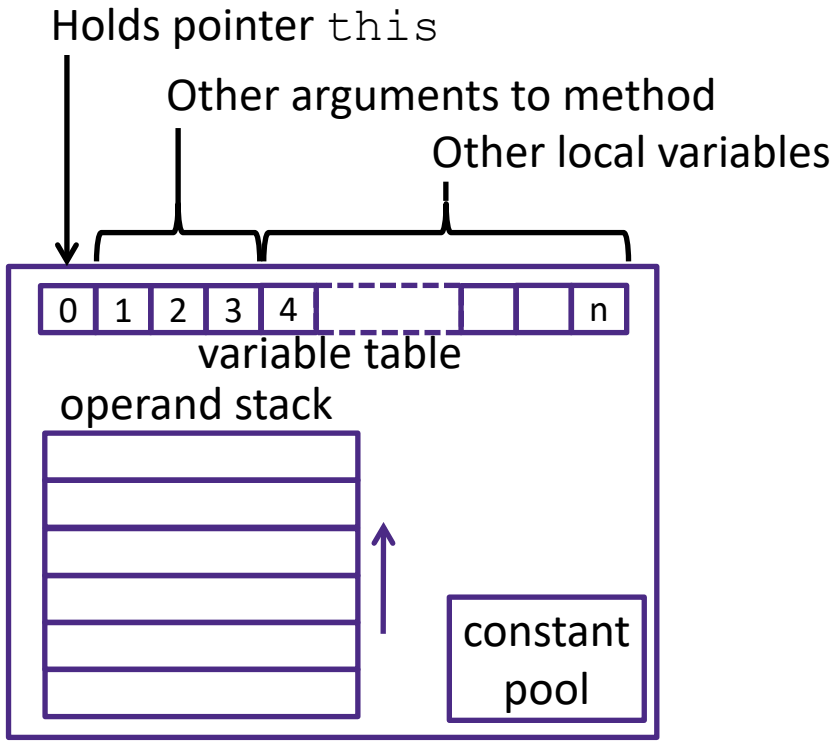
- Similar to how we described the state that x86 assembly instructions could modify: registers, memory, condition codes
- Java Bytecodes modify the state of the JVM: operand stack, variable table

the JVM model:
(not real hardware – virtual!)

- The state that x86 assembly modifies is **actual hardware**!
- The state that Java bytecodes modify is the **state of a program**!



Java Bytecode in Action



Java 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)
```

A Simple Java Method

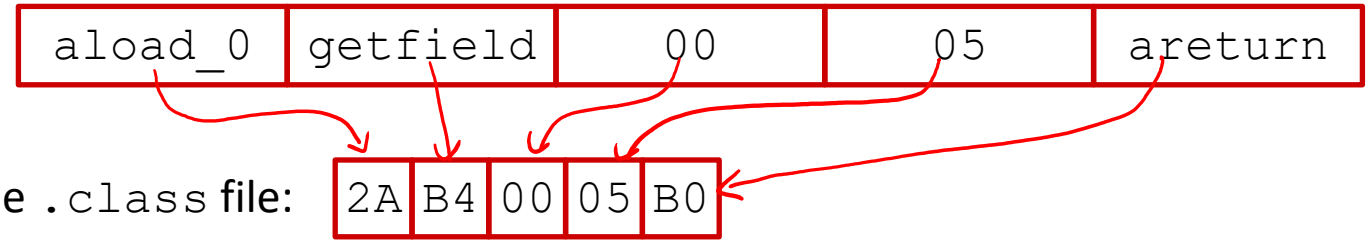
```
Method java.lang.String getEmployeeName()
0 aload_0          // "this" object is stored at 0 in the var table

1 getfield #5 <Field java.lang.String name>
                   // getfield instruction has a 3-byte encoding
                   // Pop an element from top of stack, retrieve its
                   //   specified instance field and push it onto stack
                   // "name" field is the fifth field of the object

4 areturn          // Returns object at top of stack
```

instruction "address"

two-byte argument

reference

Byte number: 0 1 4



As stored in the `.class` file:

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

Class File Format

- Every class in Java source code is compiled to its own class file
- 10 sections in the Java class file structure:
  - **Magic number**: 0xCAFEBABE (legible hex from James Gosling – Java's inventor)
  - **Version of class file format**: The minor and major versions of the class file
  - **Constant pool**: Set of constant values for the class
  - **Access flags**: For example whether the class is abstract, static, final, etc.
  - **This class**: The name of the current class
  - **Super class**: The name of the super class
  - **Interfaces**: Any interfaces in the class
  - **Fields**: Any fields in the class
  - **Methods**: Any methods in the class
  - **Attributes**: Any attributes of the class (for example, name of source file, etc.)
- A `.jar` file collects together all of the class files needed for the program, plus any additional resources (e.g. images)

Disassembled Java Bytecode

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

```
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)
…
```

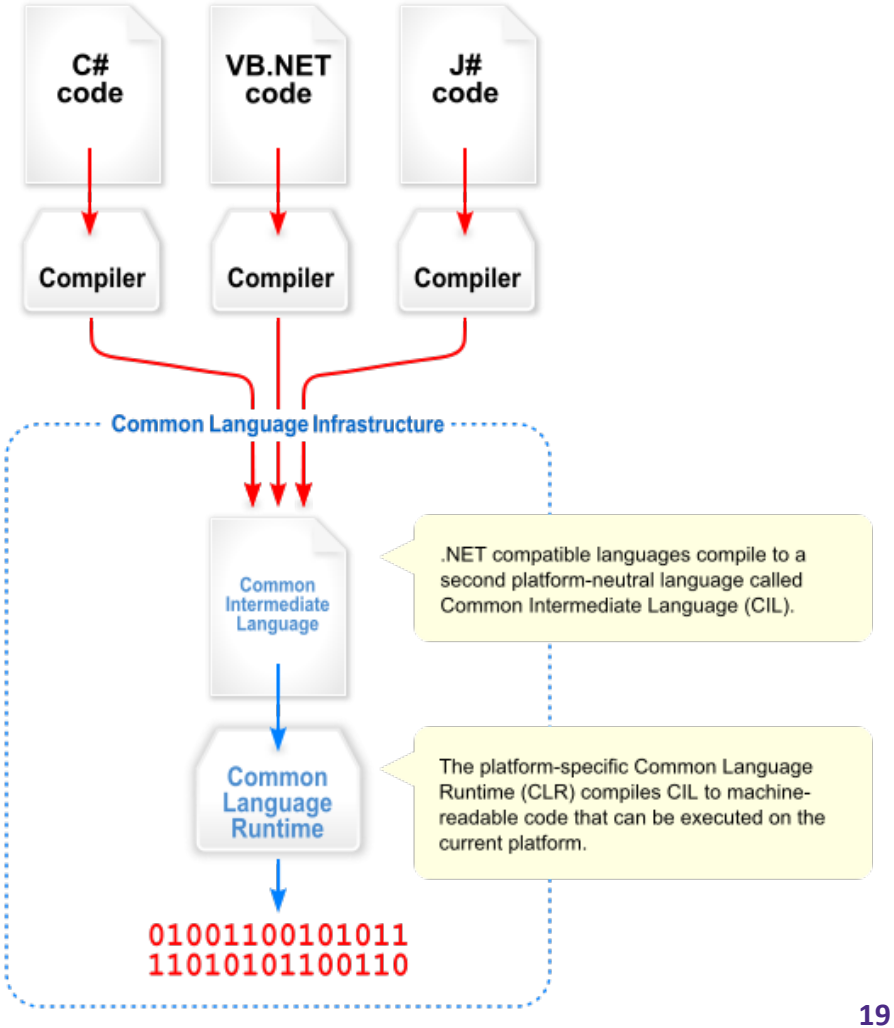
http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

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! 😊 😎 😂

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

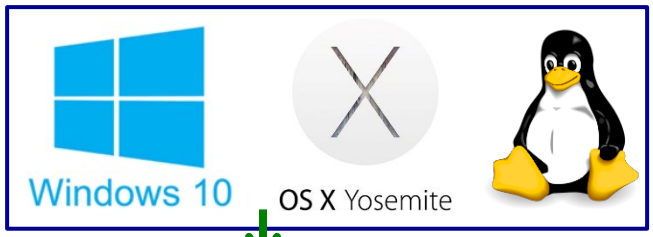
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

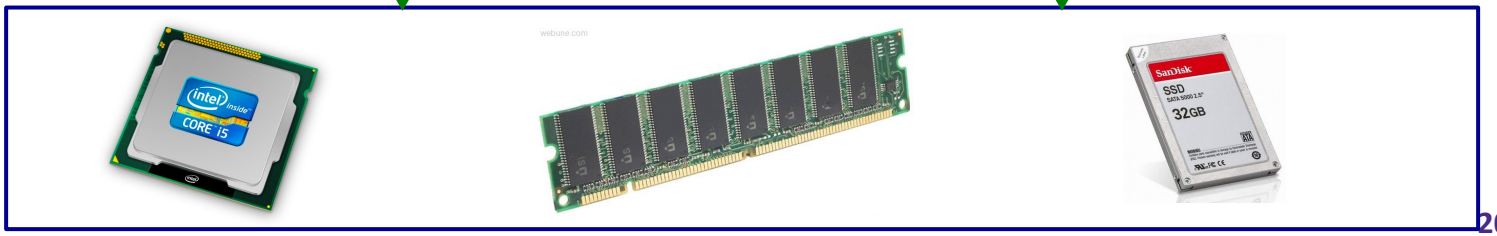
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111101000011111
```

OS:



Computer system:



Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Memory allocation
Processes
Virtual memory
Java vs. C