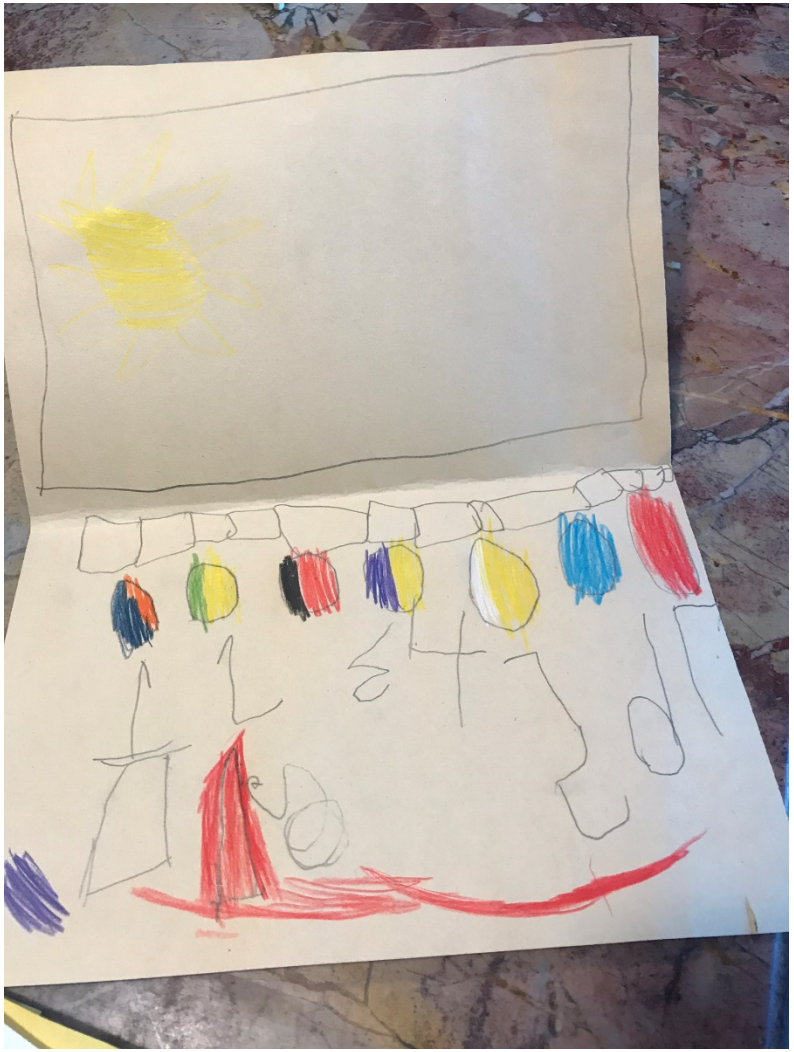
Java Virtual Machine

CSE 351 Spring 2018



Model of a Computer "Showing the Weather"
Pencil and Crayon on Paper
Matai Feldacker-Grossman, Age 4
May 22, 2018

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();
```

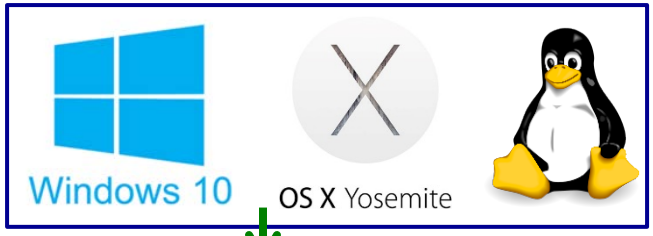
Assembly language:

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

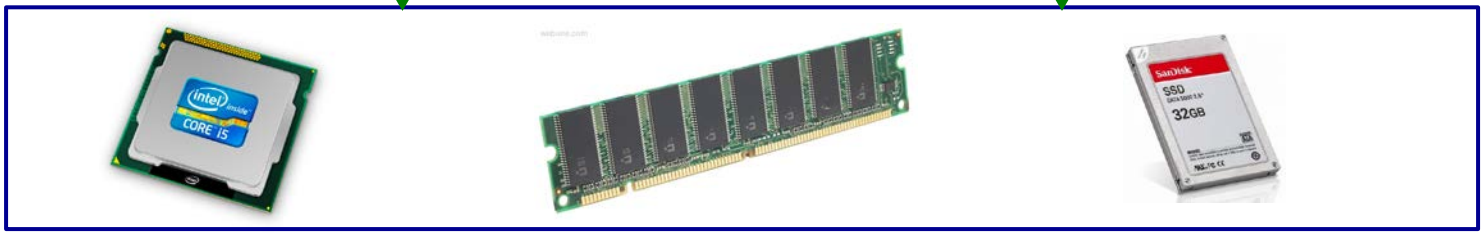
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

OS:



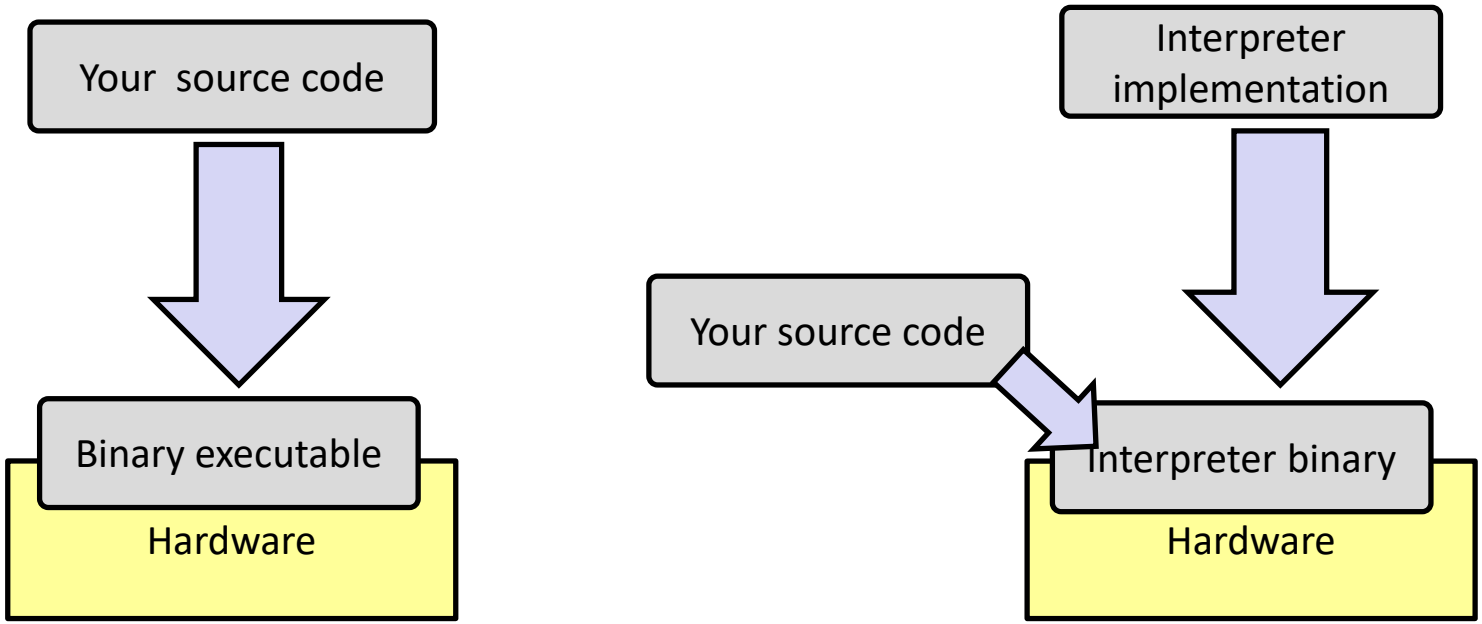
Computer system:



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

Implementing Programming Languages

- Many choices in how to implement programming models
- We've talked about compilation, can also *interpret*
- Interpreting languages has a long history
  - Lisp, an early programming language, was interpreted
- Interpreters are still in common use:
  - Python, Javascript, Ruby, Matlab, PHP, Perl, …



An Interpreter is a Program

- Execute (something close to) the *source code* directly
- Simpler/no compiler – less translation
- More transparent to debug – less translation
- Easier to run on different architectures – runs in a simulated environment that exists only inside the *interpreter* process
  - Just port the interpreter (program), not the program-intepreted
- Slower and harder to optimize

Interpreter vs. Compiler

- An aspect of a language implementation
  - A language can have multiple implementations
  - Some might be compilers and other interpreters
- "Compiled languages" vs. "Interpreted languages" a misuse of terminology
  - But very common to hear this
  - And has *some* validation in real world (e.g., JavaScript vs. C)
- Also, as about to see, modern language implementations are often a mix of the two
  - Compiling to a bytecode language, then interpreting
  - *Just-in-time* compilation of parts to assembly for performance

"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***
  - Java is sometimes compiled ahead of time (AOT) like C

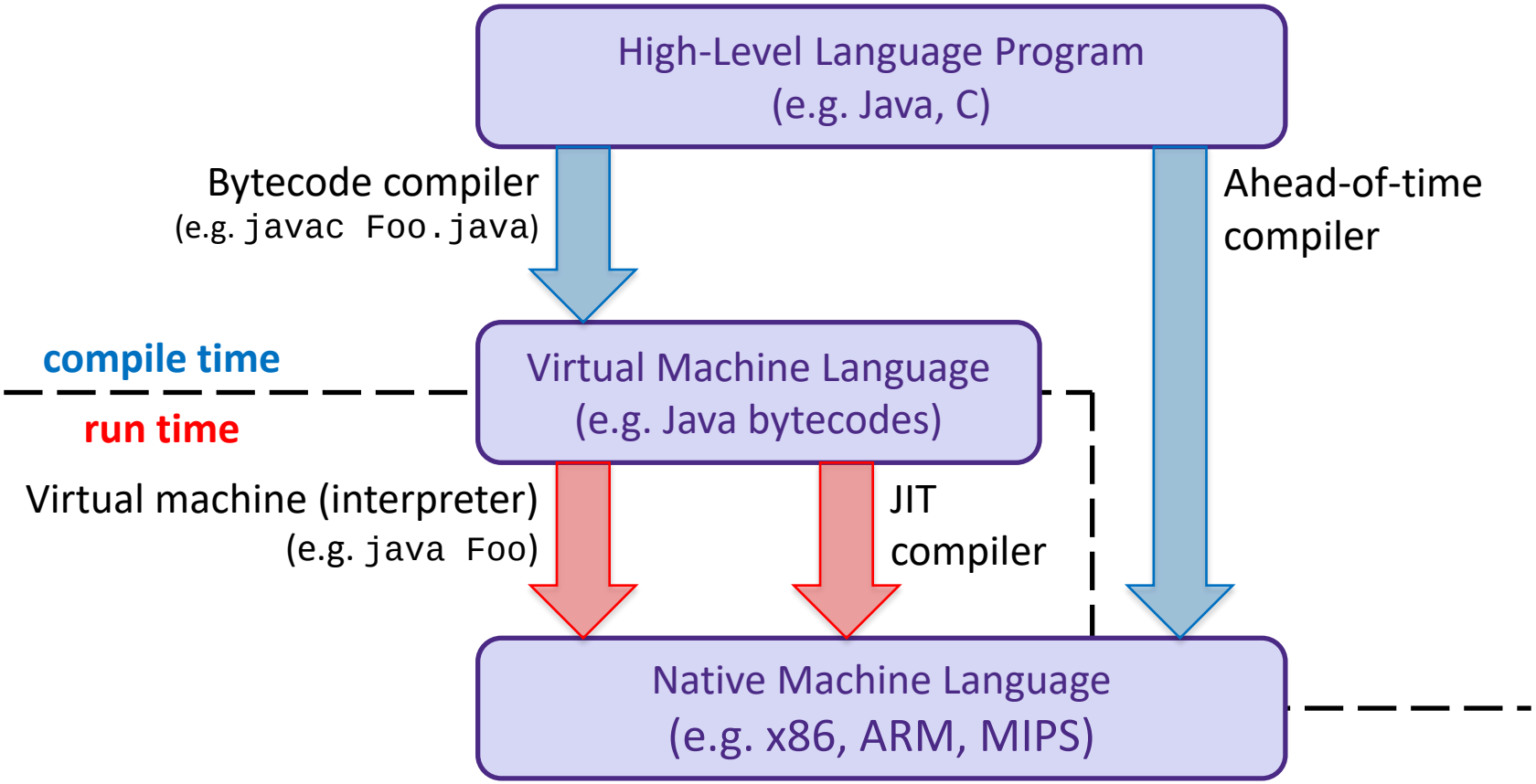
Compiling and Running Java

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

- To execute the program stored in the bytecodes, Java bytecodes can be interpreted by a program (an interpreter)
- For Java, this interpreter is called the Java Virtual Machine
- To run the Java virtual machine:
  - `java Foo`
  - **This loads the contents of `Foo.class` and interprets the bytecodes**

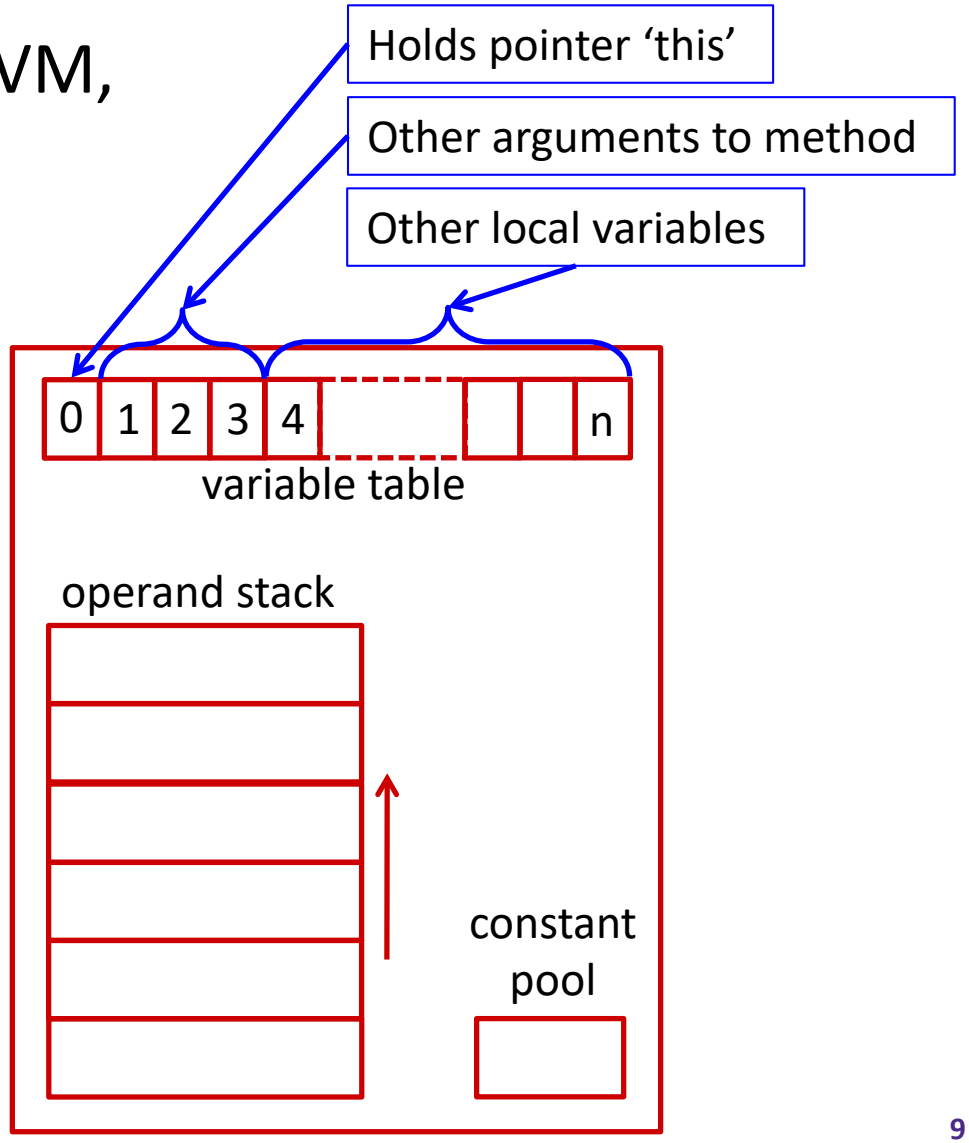
Note: The Java virtual machine is different than the CSE VM running on VMWare

Virtual Machine Model

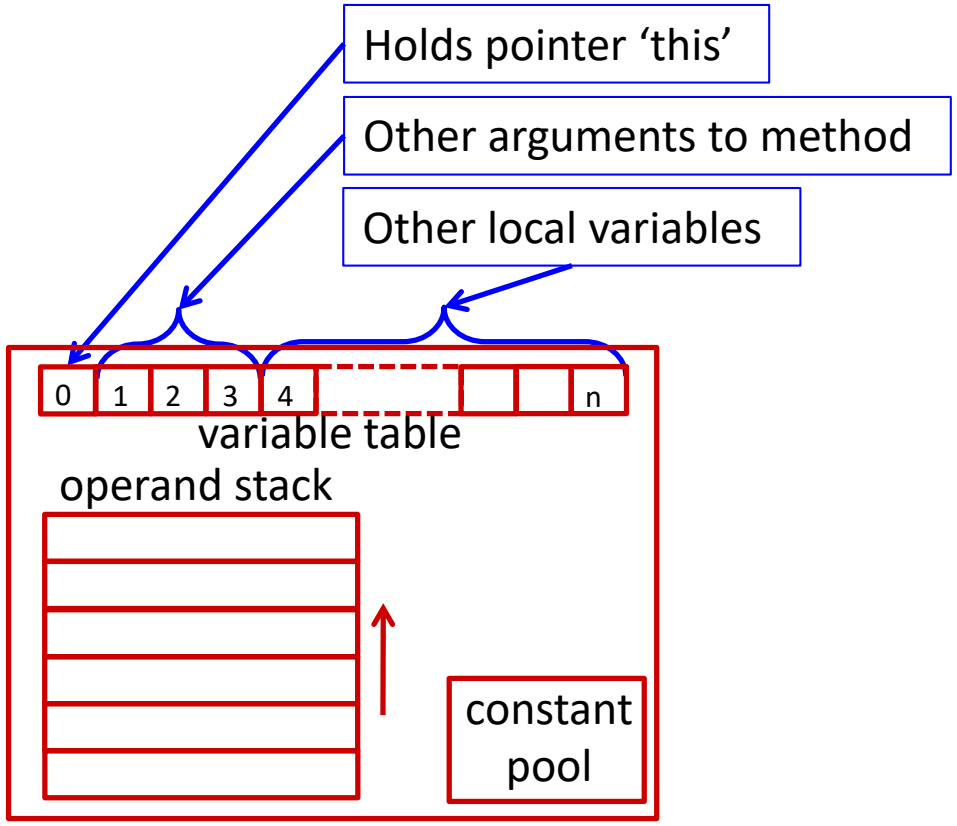


Java bytecode

- like assembly code for JVM, but works on *all* JVMs: hardware-independent
- typed (unlike ASM)
- strong JVM protections



JVM Operand Stack



'i' stands for integer,
'a' for 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 x86:

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

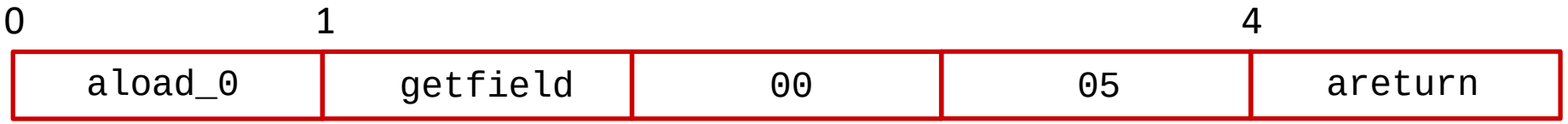
A Simple Java Method

```
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> // takes 3 bytes
                // 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
```



In the .class file:

2A	B4	00	05	B0

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)
…
```

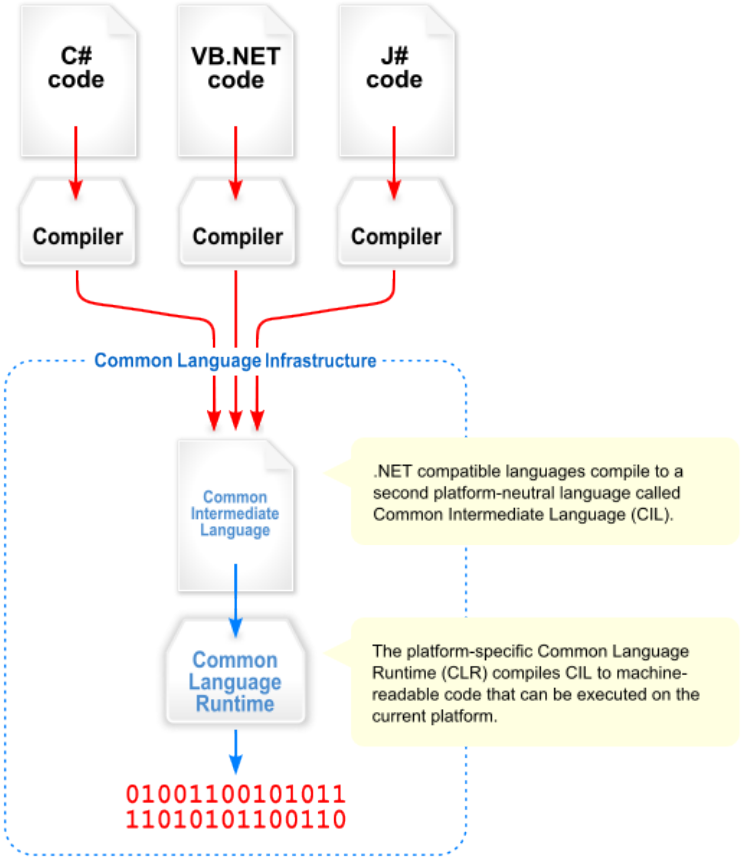
Other languages for JVMs

- 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!

- Traditionally, JVM definition and implementation was engineered for Java and still true first-and-foremost, but has evolved 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:

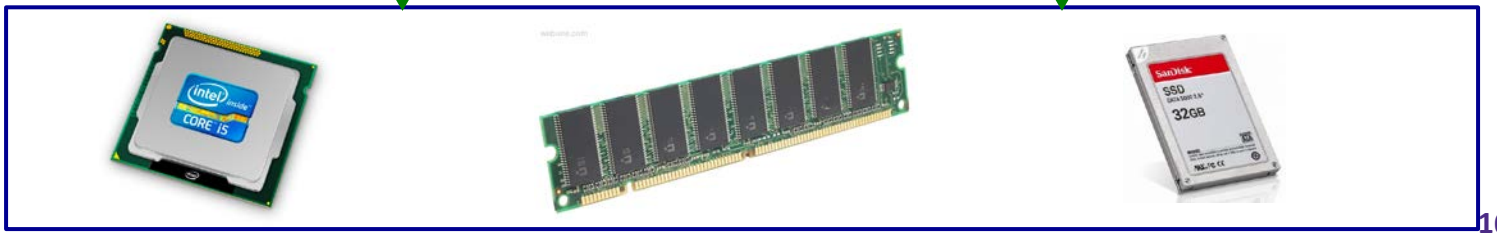
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100001111
```

OS:

Computer system:



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