Java and C (part II)

CSE 351 Spring 2021

Instructor:

Ruth Anderson

Teaching Assistants:

Allen Aby
Joy Dang
Alena Dickmann
Catherine Guevara
Corinne Herzog
Ian Hsiao
Diya Joy
Jim Limprasert
Armin Magness
Aman Mohammed
Monty Nitschke
Allie Pfleger
Mara Kirdani-Ryan
Alex Saveau
Sanjana Sridhar
Amy Xu

Administrivia

- Lab 5 (on Mem Alloc) due the last day of class (6/04)
  - Can be submitted at most ONE day late. (Sun 6/06)
- hw28 on Java and C – due Wed (6/09)
- Unit Summary #4 – due Wed (6/09)
  - No task #3 for Unit Summary #4
- Course evaluations now open
  - Please fill these out!
  - Separate ones for Lecture and Section
- **Questions Docs**: Use @uw google account to access!!
  - https://tinyurl.com/CSE351-21sp-Questions

Polling Question

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

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

Vote in Ed Lessons

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();
```

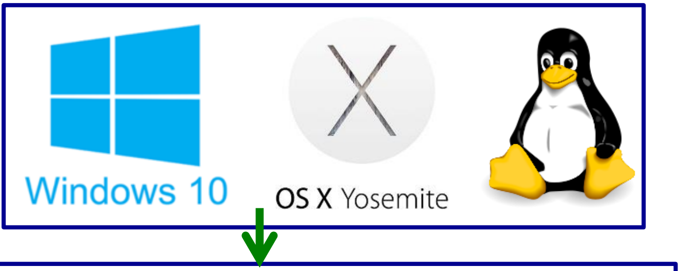
Assembly language:

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

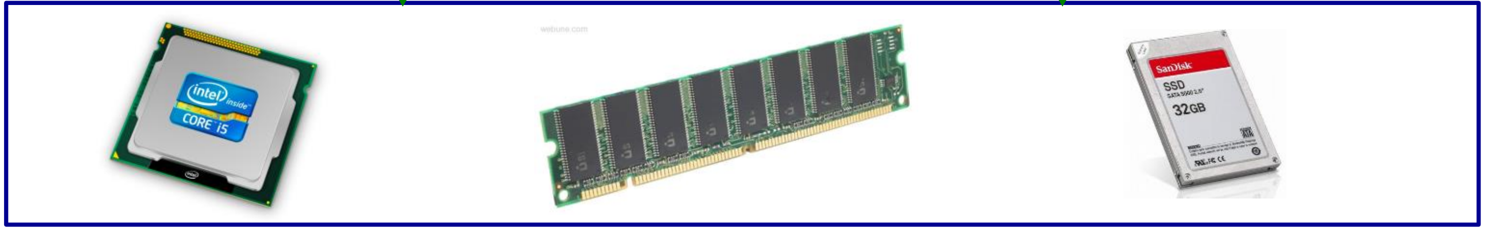
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



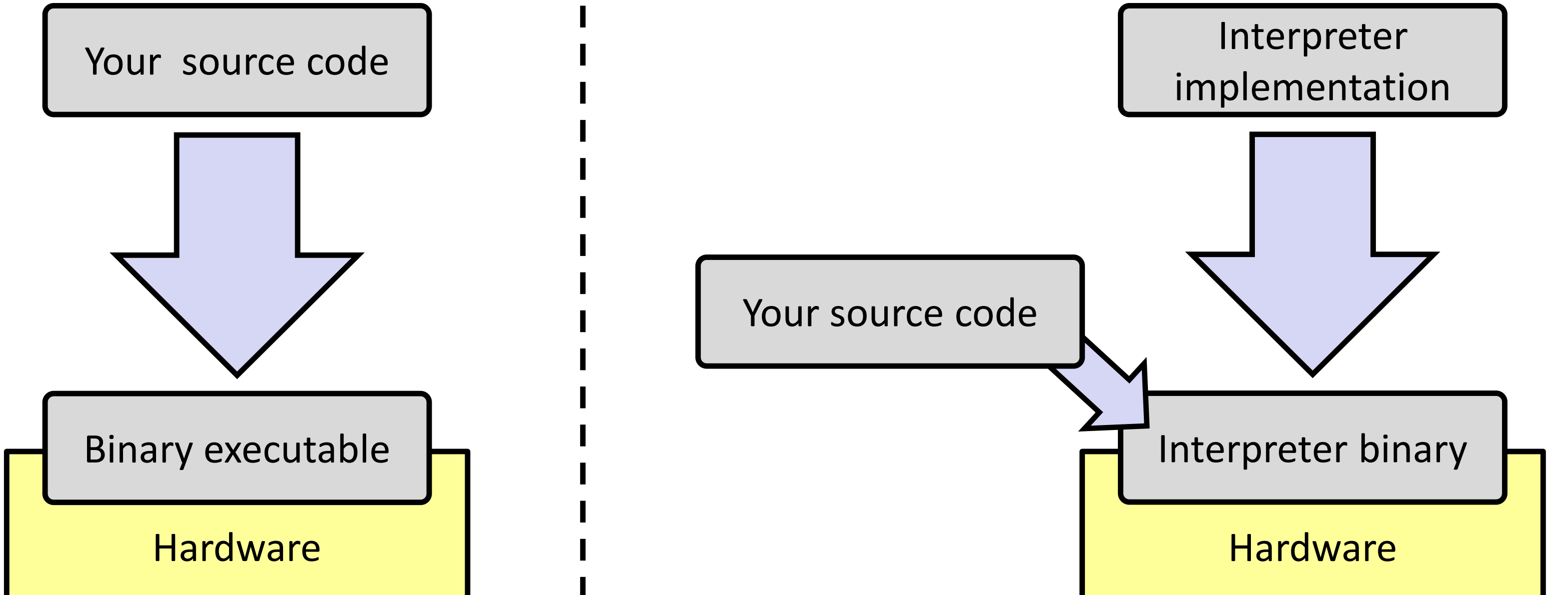
Computer system:



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

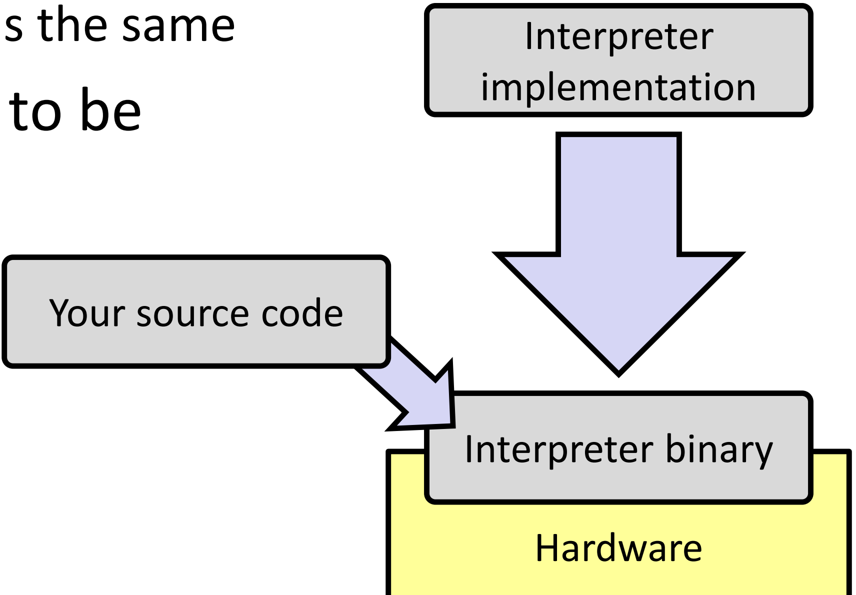
Implementing Programming Languages

- Many choices in 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

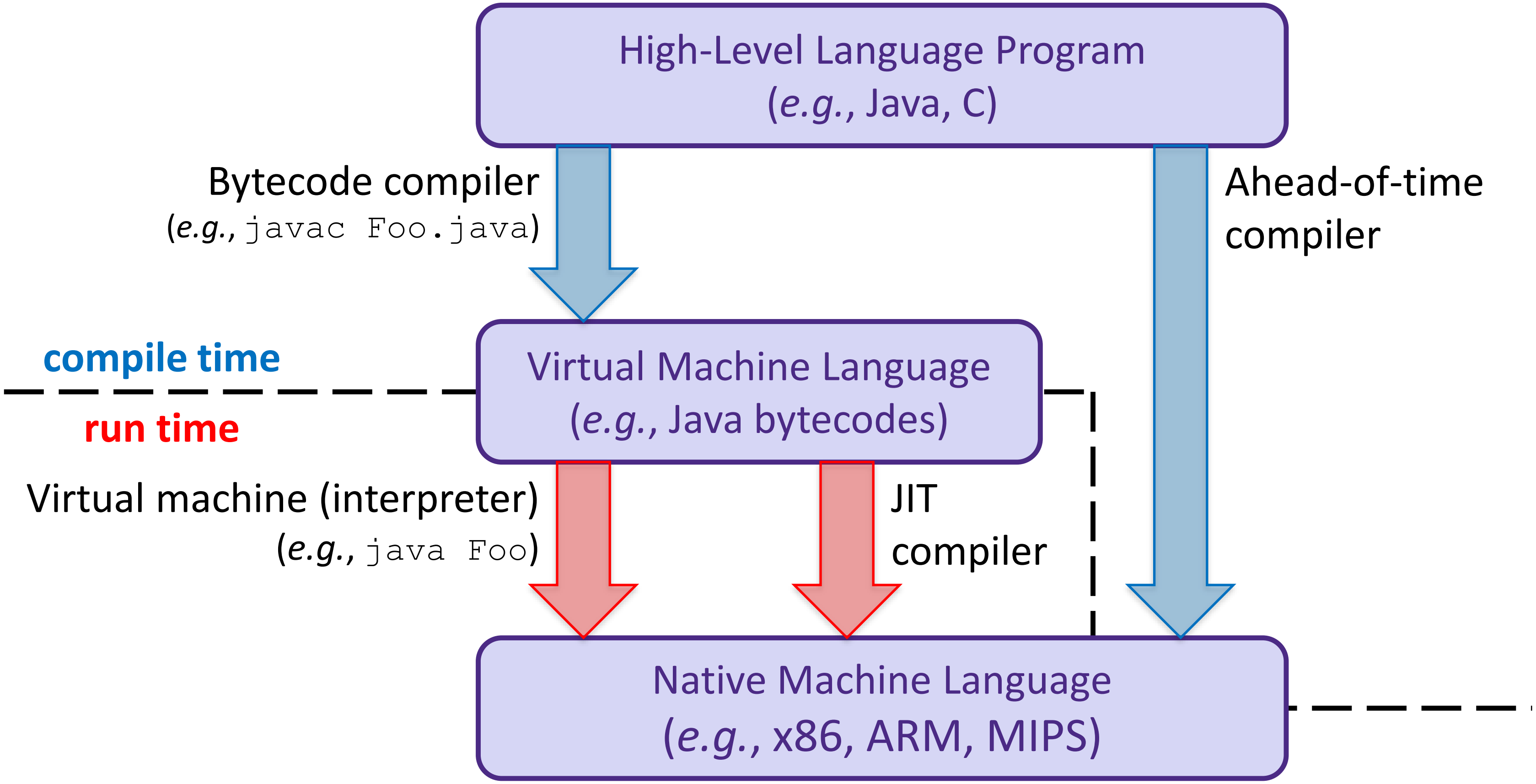
3. To execute the program stored in the bytecodes, these can be interpreted by the Java Virtual Machine (JVM)
   - Running the virtual machine: `java Foo`
   - Loads `Foo.class` and interprets the bytecodes

"The JVM"

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



High-Level Language Program
(*e.g.*, Java, C)

Bytecode compiler
(*e.g.*, `javac Foo.java`)

Ahead-of-time compiler

compile time

run time

Virtual Machine Language
(*e.g.*, Java bytecodes)

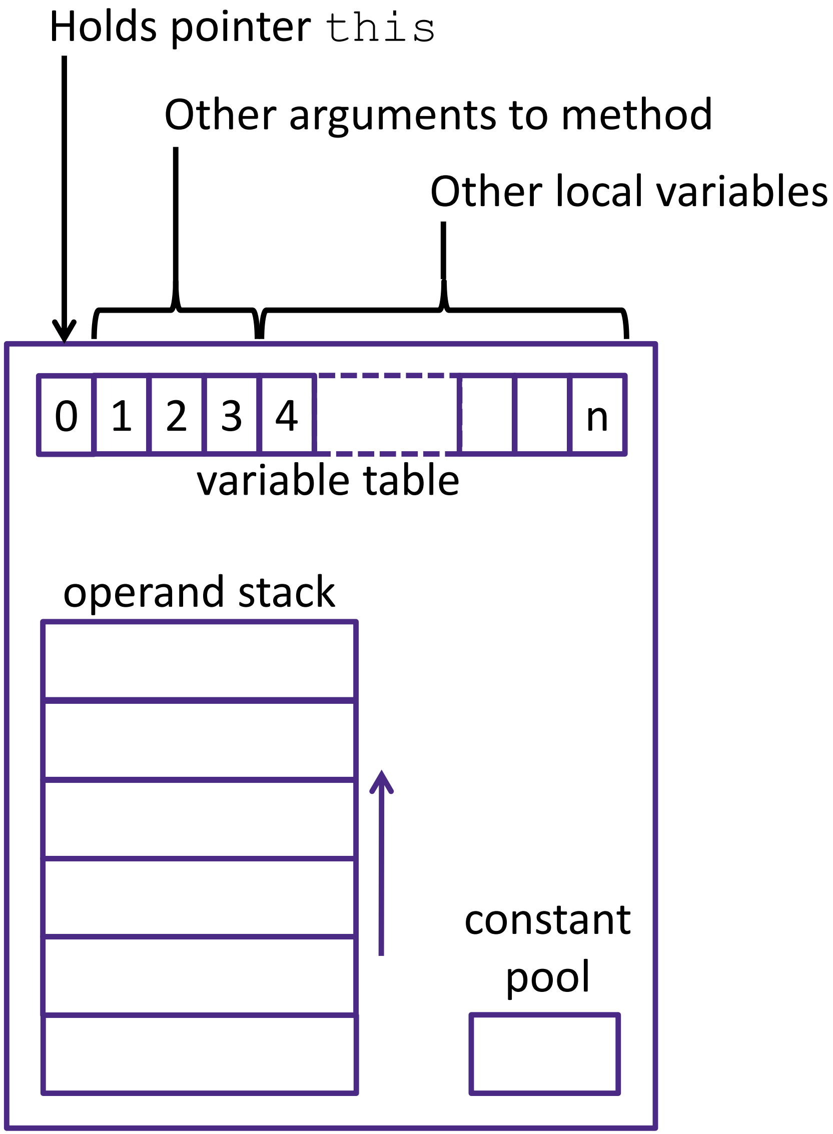
Virtual machine (interpreter)
(*e.g.*, `java Foo`)

JIT compiler

Native Machine Language
(*e.g.*, x86, ARM, MIPS)

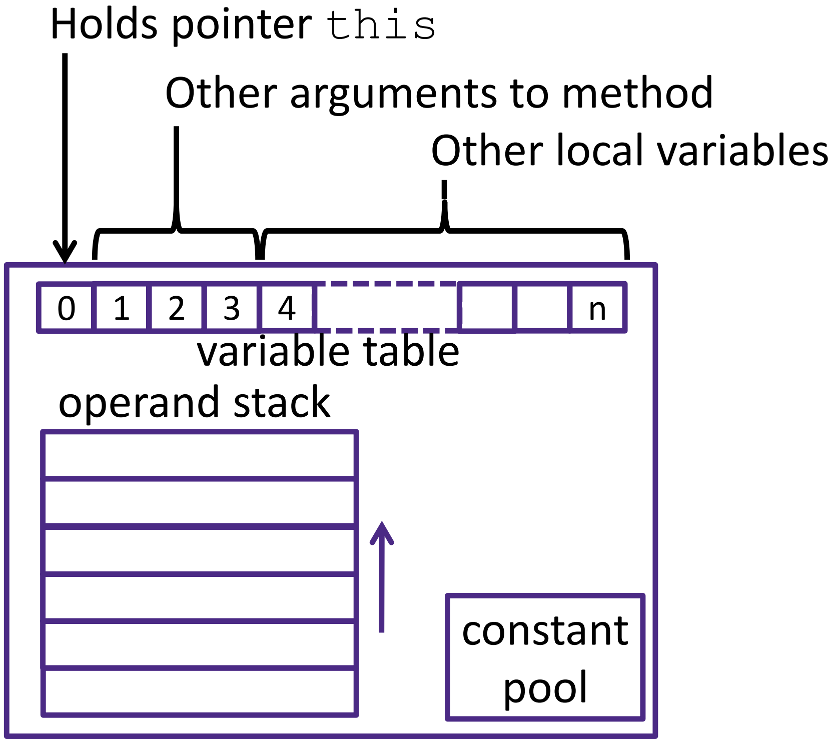
Java Bytecode

- 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)
```

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
```

Byte number:	0	1			4
	aload_0	getfield	00	05	areturn

As stored 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
```

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

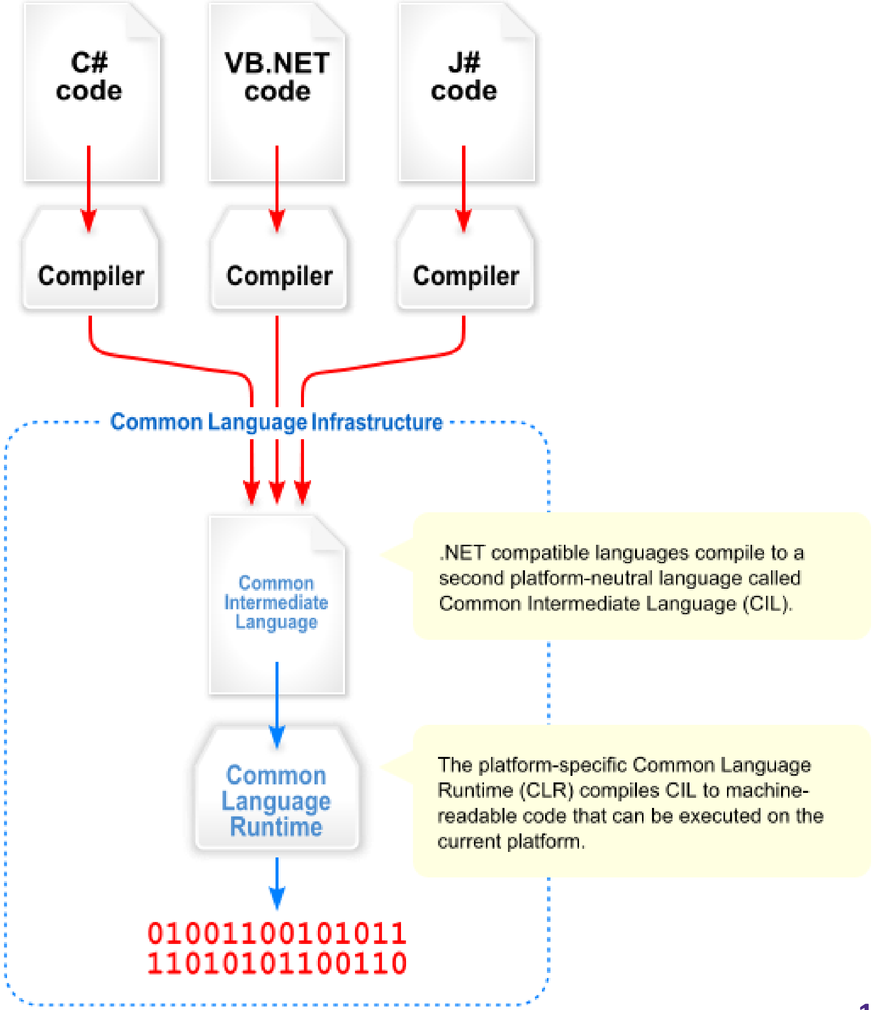
```
Compiled from Employee.java
class Employee extends java.lang.Object {
  public Employee(java.lang.String,int);
  public java.lang.String getEmployeeName();
  public int getEmployeeNumber();
}

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



.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).

The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.

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();
```

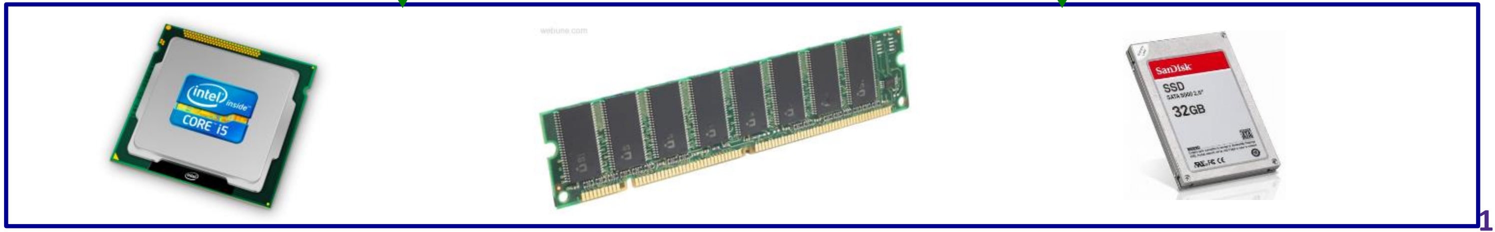
Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



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

OS: