Java and C II

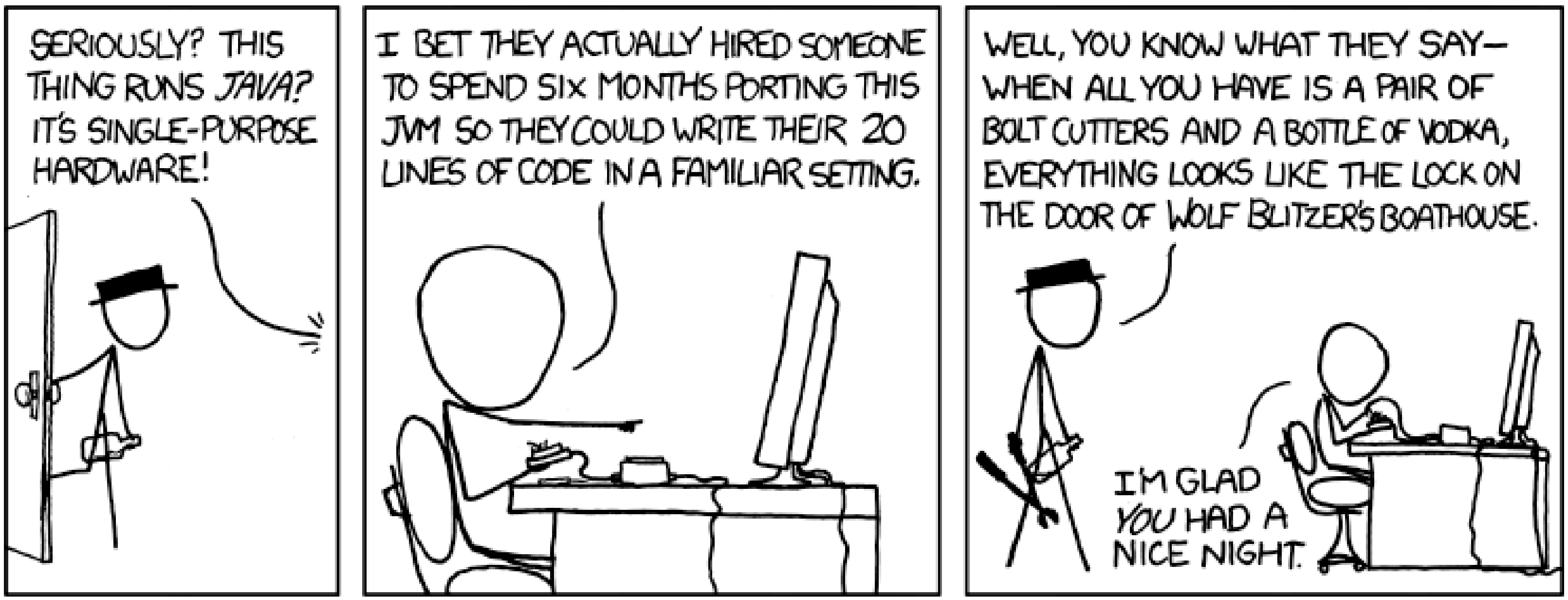
CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun



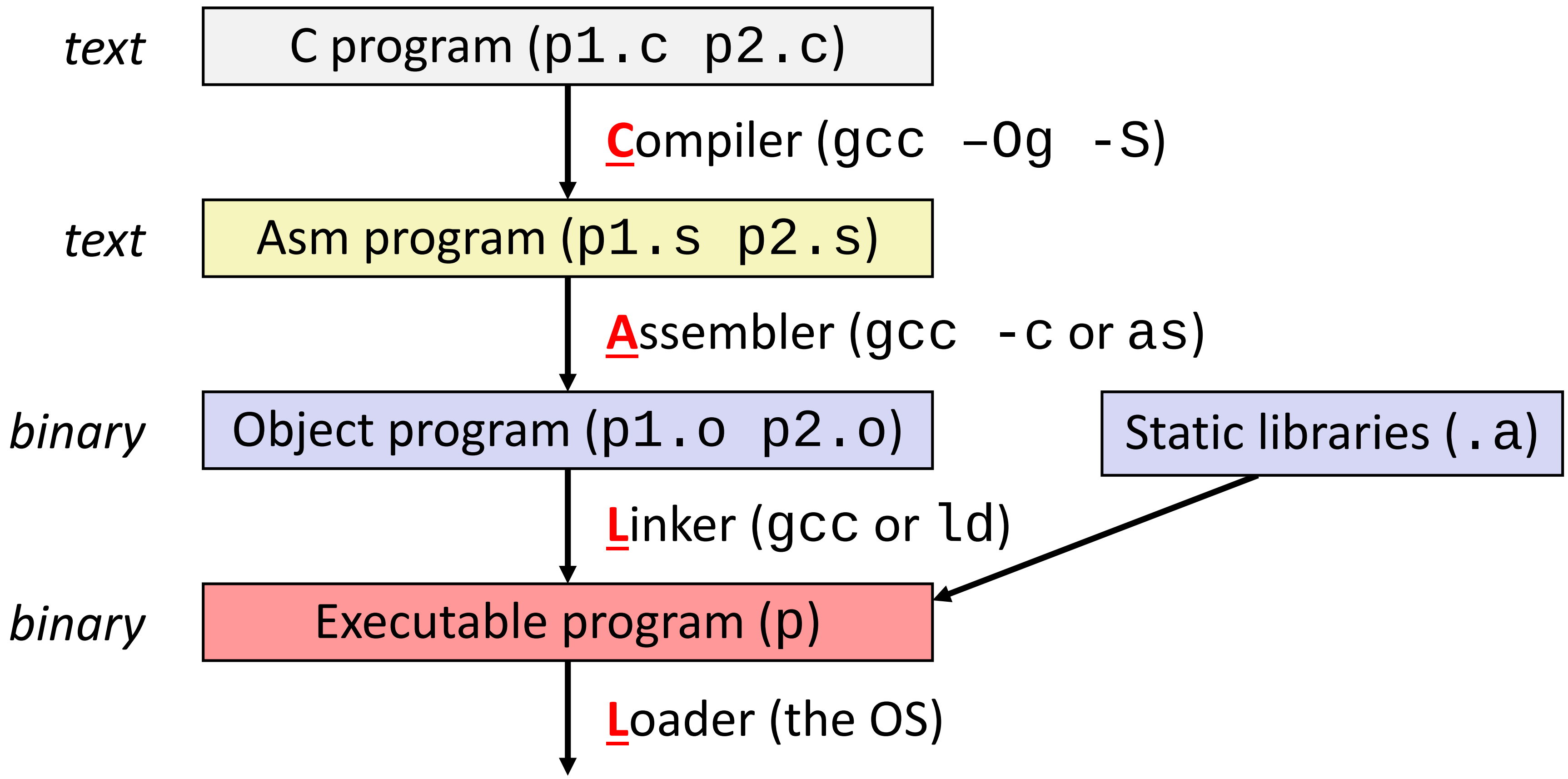
https://xkcd.com/801/

Administrivia

- Lab 5 due Friday @ 11:45pm
  - Hard deadline on Sunday @ 11:45pm
- Course evaluations now open
  - See Piazza post @465 for links (separate for Lec A/B)
- **Final Exam:** Tue, Dec. 13 @ 12:30pm in Kane 120
  - Review Session: Sun, Dec. 11 @ 1:30pm in EEB 105
  - Cumulative (midterm clobber policy applies)
  - TWO double-sided handwritten 8.5×11” cheat sheets
    - Recommended that you reuse or remake your midterm cheat sheet

Starting a C Program

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file p
- Run with command: `./p`



Compiler

- **Input:** Higher-level language code (e.g. C, Java)
  - `foo.c`
- **Output:** Assembly language code (e.g. x86, ARM, MIPS)
  - `foo.s`

- First there's a preprocessor step to handle #directives
  - Macro substitution, plus other specialty directives
  - If curious/interested: http://tigcc.ticalc.org/doc/cpp.html
- Super complex, take CSE401!
- Compiler optimizations
  - "Level" of optimization specified by capital 'O' flag (e.g. -Og, -O3)
  - Options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Assembler

- **Input:** Assembly language code (e.g. x86, ARM, MIPS)
  - `foo.s`
- **Output:** Object files (e.g. ELF, COFF)
  - `foo.o`
  - Contains *object code* and *information tables*

- Reads and uses *assembly directives*
  - e.g. `.text`, `.data`, `.quad`
  - x86: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
- Produces "machine language"
  - Does its best, but object file is *not* a completed binary

Producing Machine Language

This is extra (non-testable) material

- **Simple cases:** arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- What about the following?
  - Conditional jump
  - Accessing static data (e.g. global var or jump table)
  - `call`

- Addresses and labels are problematic because final executable hasn't been constructed yet!
  - So how do we deal with these in the meantime?

Object File Information Tables

This is extra (non-testable) material

- **Symbol Table** holds list of “items” that may be used by other files
  - *Non-local labels* – function names for `call`
  - *Static Data* – variables & literals that might be accessed across files

- **Relocation Table** holds list of “items” that this file needs the address of later (currently undetermined)
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external

- All files have their own symbol and relocation tables

Object File Format

This is extra (non-testable) material

1) object file header: size and position of the other pieces of the object file
2) text segment: the machine code
3) data segment: data in the source file (binary)
4) relocation table: identifies lines of code that need to be "handled"
5) symbol table: list of this file's labels and data that can be referenced
6) debugging information

- More info: ELF format
  - http://www.skyfree.org/linux/references/ELF_Format.pdf

Linker

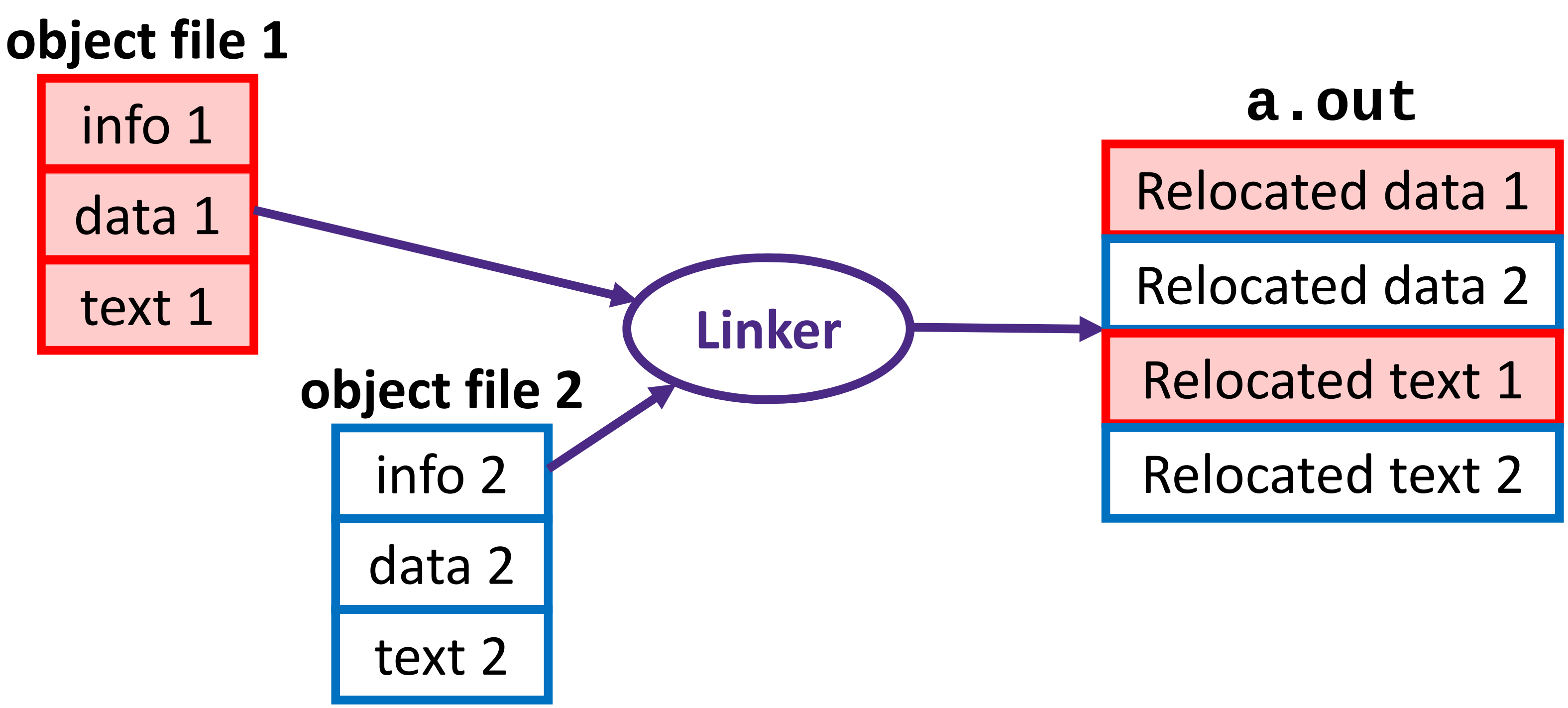
- **Input:** Object files (e.g. ELF, COFF)
  - `foo.o`
- **Output:** executable binary program
  - `a.out`

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program

Linking

This is extra (non-testable) material

1) Take text segment from each `.o` file and put them together
2) Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry



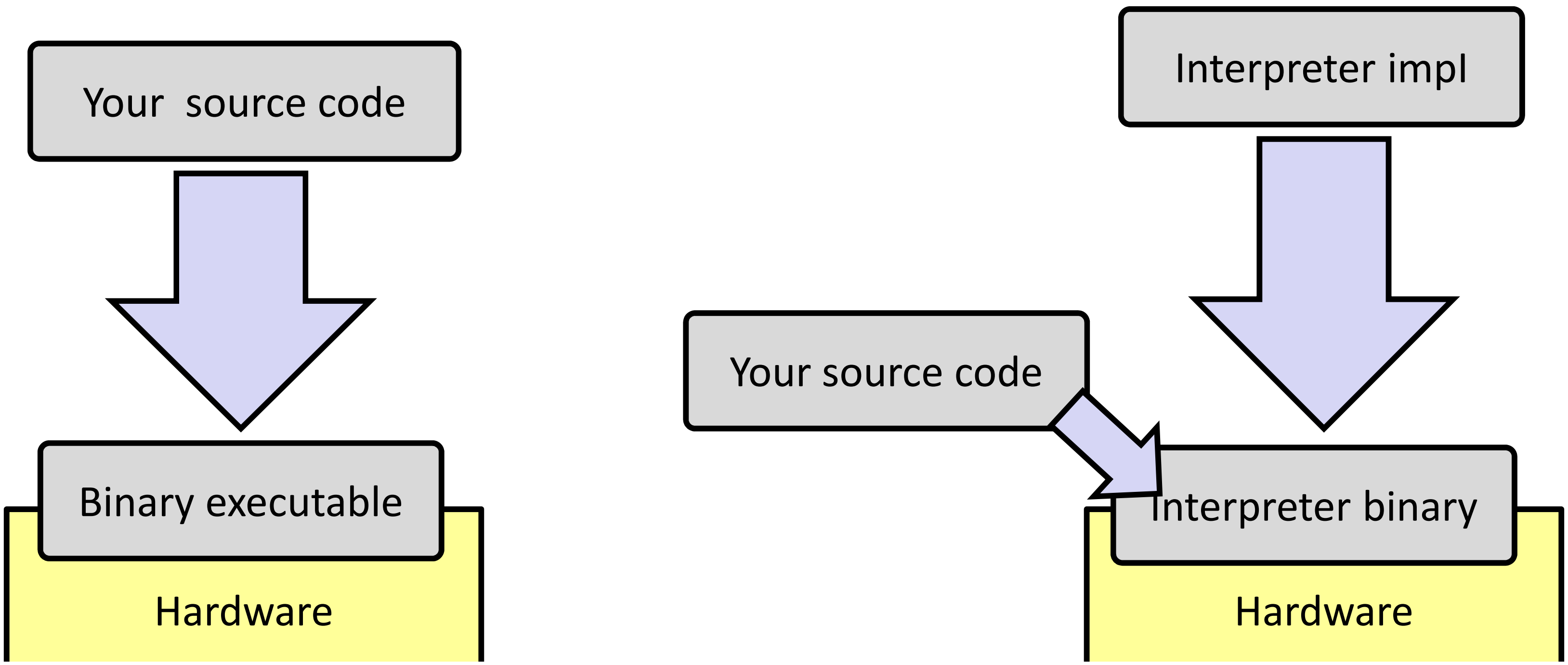
Loader

- **Input:** executable binary program, command-line arguments
  - `./a.out arg1 arg2`
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
- If run from terminal, shell calls `fork` and `execv`
- `execv` will read executable's header to initialize *virtual address space* with correctly-sized text, data, and stack segments
  - Initializes Instructions and Static Data from executable file
  - Initializes Stack with environment and argument strings
  - Jumps to start-up routine to initialize registers
    - To begin `int main(int argc, char *argv[])`, sets up `%rdi`, `%rsi`, `%rip`

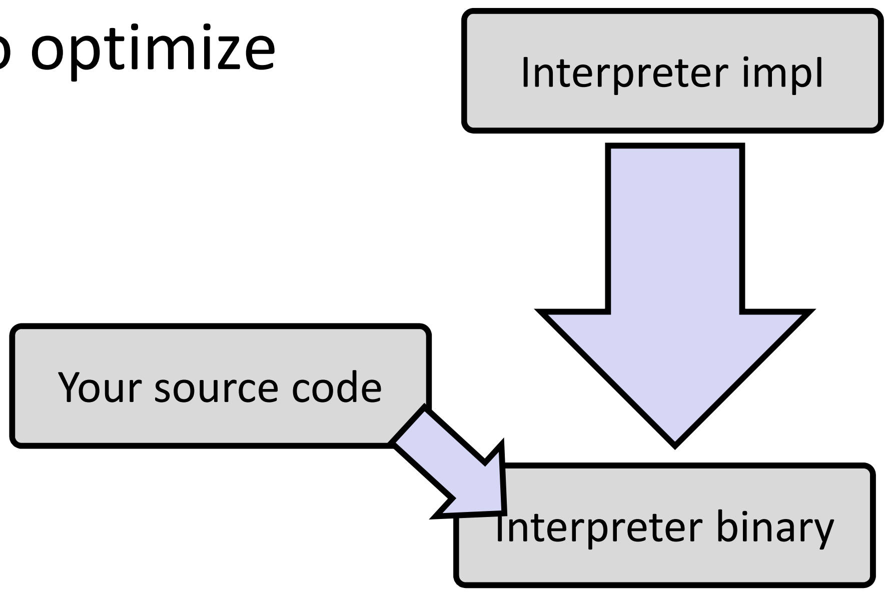
Implementing Programming Languages

- 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 the *source code* directly (or something close)
- 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
- 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 the 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
  - Doing just-in-time compilation of parts to assembly for performance

"The JVM"

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

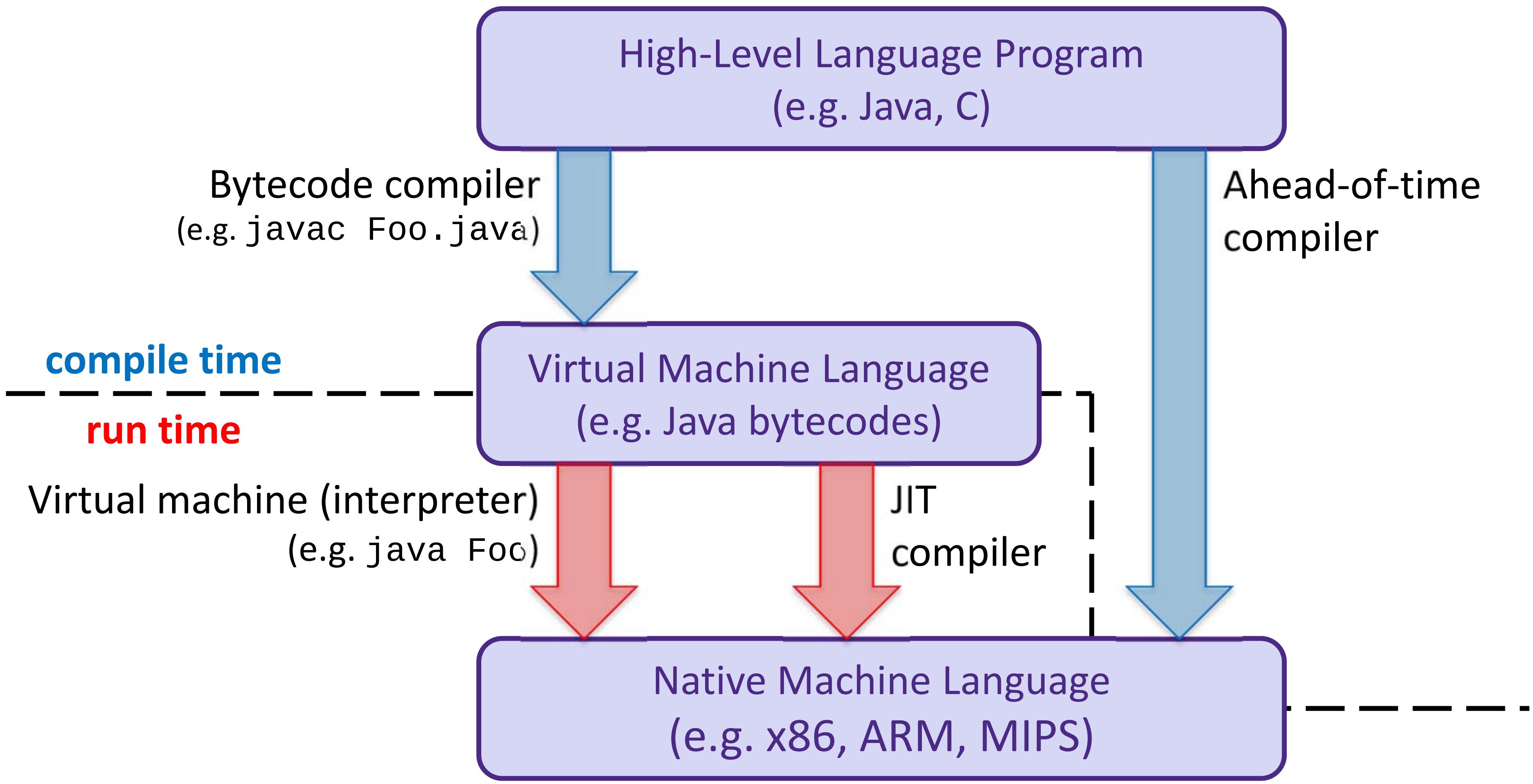
- Java programs are usually run by a Java *virtual machine* (*JVM*)
  - JVMs interpret an intermediate language called *Java bytecode*
  - Many JVMs compile bytecode to native machine code
    - **Just-in-time (JIT) compilation**
    - http://en.wikipedia.org/wiki/Just-in-time_compilation
  - Java is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

Compiling and Running Java

- The Java compiler converts Java into *Java bytecodes*
  - Stored in a `.class` file

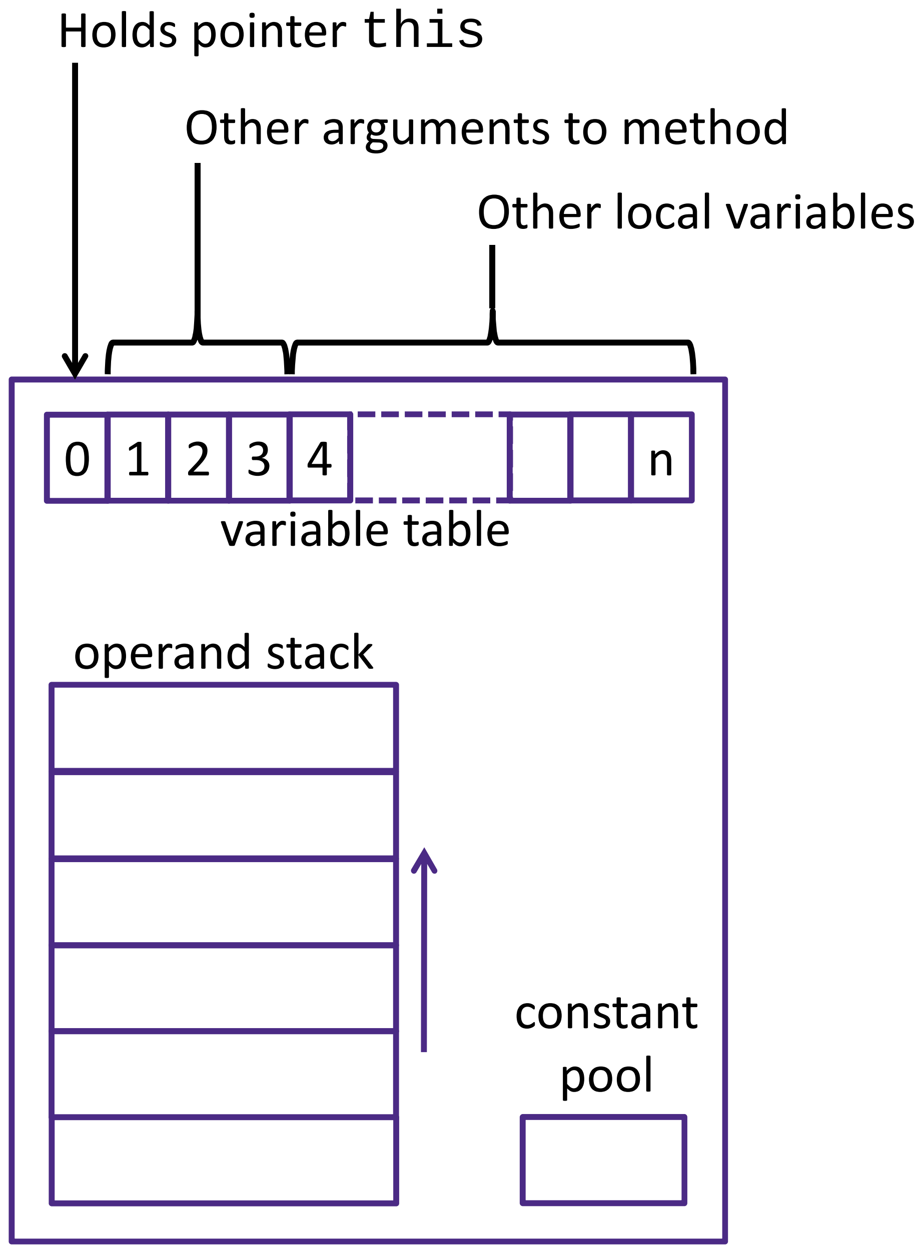
- Save your Java code in a `.java` file
- To run the Java compiler:
  - `javac Foo.java`
- To execute the program stored in the bytecodes, Java bytecodes can be interpreted by a program (an interpreter)
  - For Java, the JVM is the interpreter
  - `java Foo` runs the Java virtual machine
    - Loads the contents of `Foo.class` and interprets the bytecodes

Virtual Machine Model



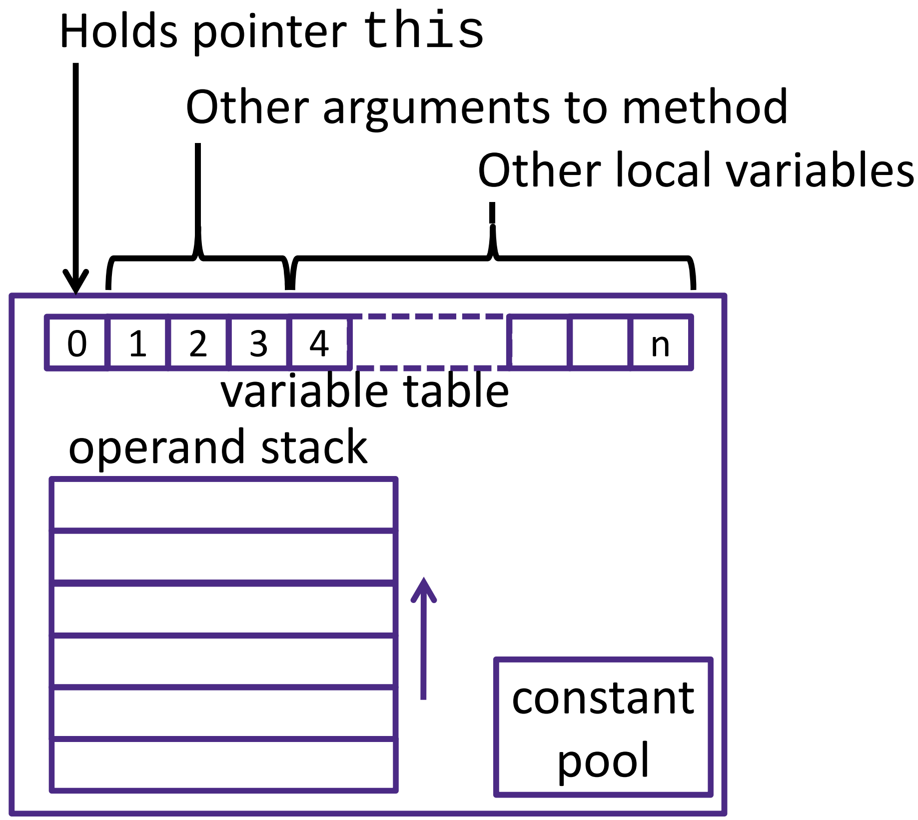
Java Bytecode

- 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 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

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

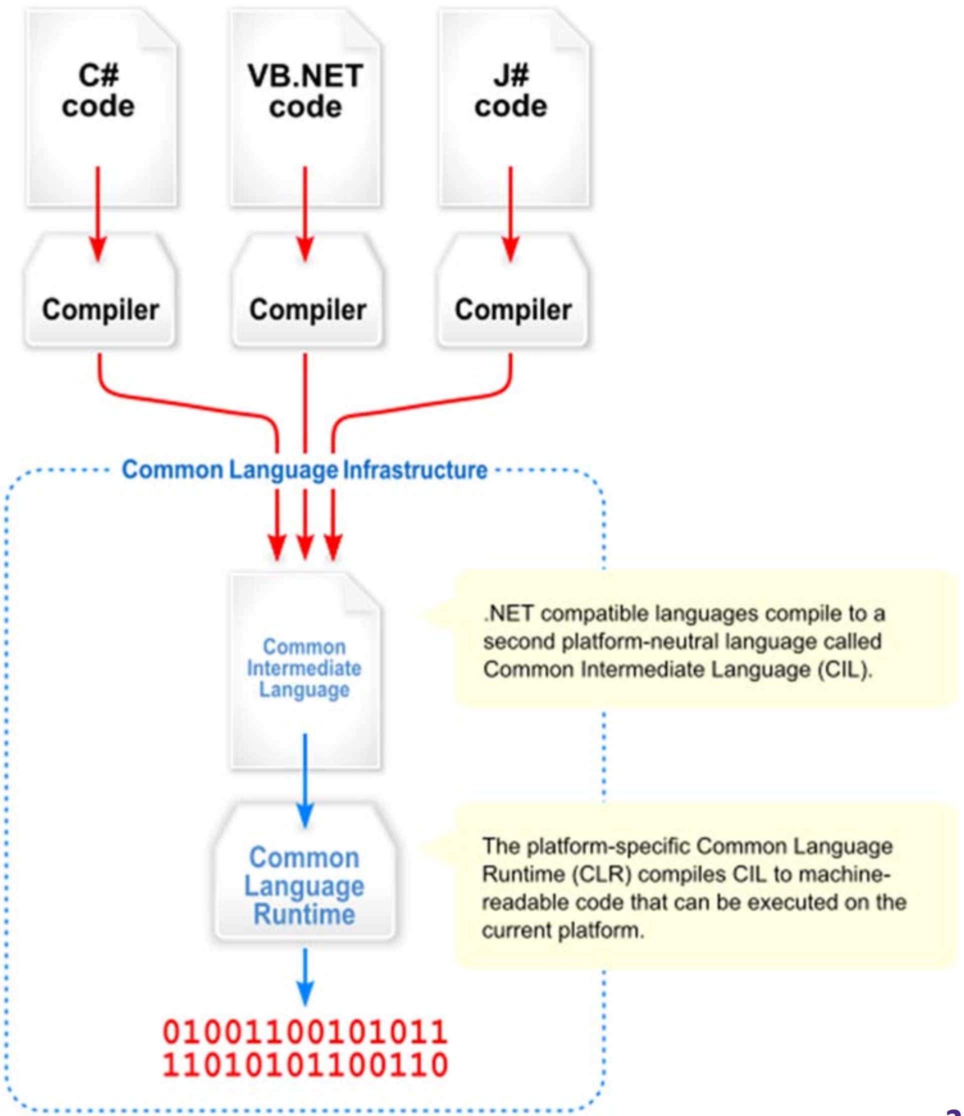
```
Compiled from Employee.java
class Employee extends java.lang.Object {
  public Employee(java.lang.String,int);
  public java.lang.String getEmployeeName();
  public int getEmployeeNumber();
}

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!

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! ☺ 😎 😂

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

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:

Windows 8 Mac

Computer system: