Java and C II
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Lab 5 – Due TONIGHT! Fri 6/2

- Course evaluations now open
  - Please fill out evals for lecture AND separate eval for section!

- Final Exam: Wed, June 7, 2017 2:30-4:20pm in our regular lecture room
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, ...

![Diagram showing compilation vs. interpretation process]

- Your source code
  - Compile
  - Binary executable
  - Hardware
- Interpreter implementation
  - Interpreter binary
  - Hardware
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
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

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

- High-Level Language Program (e.g. Java, C)
  - Bytecode compiler (e.g. javac Foo.java)
    - Compile time
  - Virtual Machine Language (e.g. Java bytecodes)
    - Virtual machine (interpreter) (e.g. java Foo)
      - Run time
    - JIT compiler
  - Native Machine Language (e.g. x86, ARM, MIPS)
    - Ahead-of-time compiler
Java Bytecode

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

\[ c = a + b \]

\[ \text{‘i’ = integer, ‘a’ = reference, ‘b’ for byte, ‘c’ for char, ‘d’ for double, …} \]

**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

**Compiled to x86:**

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

**Holds pointer this**

**Other arguments to method**

**Other local variables**

**JVM:**

- Constant pool
- Variable table
- Operand stack

**No registers or stack locations! All operations use operand stack**
A Simple Java Method

Method `java.lang.String getEmployeeName()`

1. `aload 0 // "this" object is stored at 0 in the var table`
2. `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
3. `areturn // Returns object at top of stack`

Byte number: 0 1 4

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>aload_0</td>
<td>getfield</td>
<td>00 05</td>
</tr>
<tr>
<td>areturn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As stored in the .class file: 2A B4 00 05 B0

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**: 0xCAFEBAEBE (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

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! 😊 😄 😃

C:
```c
#define car  

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

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

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

Machine code:
```
0111010000011000 1000110100000100000000101000100111000010110000011111101000011111
```

OS:
- Windows
- macOS

Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks

Executables
- Arrays & structs
- Memory & caches

Processes
- Memory allocation
- Java vs. C