The last week of 351

- Monday lecture (inside)
  - Java vs. C
  - Evaluations or start parallelism
- Wednesday lecture (outside?)
  - Parallelism
  - Conclusions
  - Evaluations (if not Monday)
- Wednesday: Lab 5 due!
- Thursday section (inside?)
  - Final exam review session (with Ben)
- Friday: final exam (inside)

Java vs. C

- Reconnecting to Java
  - Back to CSE143!
  - But now you know a lot more about what really happens when we execute programs

- We’ve learned about the following items in C; now we’ll see what they look like for Java:
  - Representation of data
  - Pointers / references
  - Casting
  - Function / method calls
  - Runtime environment
  - Translation from high-level code to machine code

Roadmap

<table>
<thead>
<tr>
<th>C:</th>
<th>Java:</th>
</tr>
</thead>
<tbody>
<tr>
<td>car *c = malloc(sizeof(car));</td>
<td>Car c = new Car();</td>
</tr>
<tr>
<td>c-&gt;miles = 100;</td>
<td>c.setMiles(100);</td>
</tr>
<tr>
<td>c-&gt;gals = 17;</td>
<td>c.setGals(17);</td>
</tr>
<tr>
<td>float mpg = get_mpg(c);</td>
<td>float mpg = c.getMPG();</td>
</tr>
<tr>
<td>free(c);</td>
<td></td>
</tr>
</tbody>
</table>

Assembly language:

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

Machine code:

```
0111011000001100
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

- Windows
- Mac

Meta-point to this lecture

- None of the data representations we are going to talk about are guaranteed by Java
- In fact, the language simply provides an abstraction
- We can’t easily tell how things are really represented
- But it is important to understand an implementation of the lower levels – useful in thinking about your program
  - just like caching, etc.
Data in Java

- **Integers, floats, doubles, pointers – same as C**
  - Yes, Java has pointers – they are called ‘references’ – however, Java references are much more constrained than C’s general pointers
- **Null is typically represented as 0**
- **Characters and strings**
- **Arrays**
- **Objects**

Data in Java

- **Arrays**
  - Every element initialized to 0 or null
  - Length specified in immutable field at start of array (int – 4 bytes)
    - `array.length` returns value of this field
    - *Since it has this info, what can it do?*

```
int array[5]:

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>4</th>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>5</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```

Data in Java

- **Characters and strings**
  - Two-byte Unicode instead of ASCII
    - Represents most of the world’s alphabets
  - String not bounded by a ‘\0’ (null character)
    - Bounded by hidden length field at beginning of string

```
the string ‘CSE351’:

<table>
<thead>
<tr>
<th>C: ASCII</th>
<th>43</th>
<th>53</th>
<th>45</th>
<th>33</th>
<th>35</th>
<th>31</th>
<th>\0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java: Unicode</td>
<td>6</td>
<td>00</td>
<td>43</td>
<td>00</td>
<td>53</td>
<td>00</td>
<td>45</td>
</tr>
</tbody>
</table>
```
Data structures (objects) in Java

- Objects are always stored by reference, never stored inline.
  - Include complex data types (arrays, other objects, etc.) using references

```c
struct rec {
    int i;
    int a[3];
    struct rec *p;
};
```

```java
class Rec {
    int i;
    int[] a = new int[3];
    Rec p;
}
```

Pointers/References

- Pointers in C can point to any memory address
- References in Java can only point to [the starts of] objects
  - And can only be dereferenced to access a field or element of that object

```c
struct rec {
    int i;
    int a[3];
    struct rec *p;
};
struct rec* r = malloc(...);
struct rec r2;
r->i = val;
r->a[2] = val;
r->p = &r2;
```

```java
Rec r = new Rec();
r2 = new Rec();
r.i = val;
r.a[2] = val;
r.p = r2;
```

Casting in C

- We can cast any pointer into any other pointer; just look at the same bits differently

```c
struct BlockInfo {
    int sizeAndTags;
    struct BlockInfo* next;
    struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;

newBlock = (BlockInfo*) ( (char*) b + x );
```

```java
Cast b into char pointer so that you can add byte offset without scaling
```

```java
Cast back into BlockInfo pointer so you can use it as BlockInfo struct
```
Type-safe casting in Java

- Can only cast compatible object references

```java
class Object {}  
class Vehicle {
    int passengers;
}
class Boat extends Vehicle {
    int propellers;
}
class Car extends Vehicle {
    int wheels;
}
```

// Vehicle is a super class of Boat and Car, which are siblings
Vehicle v = new Vehicle();
Car c1 = new Car();
Boat b1 = new Boat();
Vehicle v1 = new Vehicle(); // ok, everything needed for Vehicle
Car c2 = new Boat(); // ok, v1 is already a Vehicle
// is also in Car
Vehicle v2 = v1; // incompatible type - Boat and Car are siblings
Car c3 = new Vehicle();
Boat b2 = v; // not in Vehicle (wheels)
Car c4 = (Car) v; // ok, v2 started out as Car
Car c5 = (Car) b1; // incompatible types, b1 is Boat

Why are these problematic?

How is this implemented/enforced?

Java objects

```java
class Point {
    double x;
    double y;
    Point() {
        x = 0;
        y = 0;
    }
    boolean samePlace(Point p) {
        return (x == p.x) && (y == p.y);
    }
}
Point p = new Point();
```

Java Methods

- Static methods are just like functions.
- Instance methods
  - can refer to this;
  - have an implicit first parameter for this; and
  - can be overridden in subclasses.
- The code to run when calling an instance method (e.g., `p.samePlace(q)`) is chosen at runtime by lookup in the vtable.

Java:

```java
Point p = new Point();  // C pseudo-translation:
Point* p = calloc(1,sizeof(Point));
p->header = ...;
p->vtable = &Point_vtable;
p->vtable[0](p);
return p->vtable[1](p, q);
```
Method dispatch

Java:
Point p = new Point();
Point* p = calloc(1, sizeof(Point));
p->header = ...
p->vtable = &Point_vtable;
p->vtable[0](p);
return p->vtable[1](p, q);

C pseudo-translation:
header vtable pointer
x y

Point object
vtable pointer
x y

Subclassing

Java:
class PtSubClass extends Point{
    int aNewField;
    boolean samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}

C pseudo-translation:
class PtSubClass extends Point{
    int aNewField;
    bool samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}

Subclassing

Where does “aNewField” go? At end of fields of Point
- Point fields are always in the same place, so Point code can run on PtSubClass objects without modification.

Where does pointer to code for two new methods go?
- No constructor, so use default Point constructor
- To override “samePlace”, write over old pointer
- Add new pointer at end of table for new method “sayHi”

Subclassing

Java:
Point p = ???;
return p.samePlace(q);

C pseudo-translation:
return p->vtable[1](p, q);
Implementing Programming Languages

- Many choices in how to implement programming models
- We've talked about compilation, can also interpret
  - Execute line by line in original source code
  - Simpler compiler – less translation
  - Easier to debug – less translation
  - Easier to run on different architectures – runs in a simulated environment that exists only inside the interpreter process
  - Slower and harder to optimize
  - All errors at run time
- 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, ...

Interpreted vs. Compiled in practice

- Really a continuum, a choice to be made
  - More or less work done by interpreter/compiler
  - Compiled
    - C
    - Java
    - Lisp
- Interpreted
- Java programs are usually run by a virtual machine
  - 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

Virtual Machine Model

![Virtual Machine Model Diagram]

Java bytecode

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

![Diagram of JVM operand stack]

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: 0xCAFEBAEB (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, 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 the name of the source file, etc.)
- A .jar file collects together all of the class files needed for the program, plus any additional resources (e.g. images)

A Simple Java Method

```java
public java.lang.String getEmployeeName()
{
  0 load_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 field and push the value onto stack.
    // "name" field is the fifth field of the class
  4 areturn // Returns object at top of stack
}
```

Disassembled Java Bytecode

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

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

In the .class file:

```
0    aload_0
1    getfield    00 05 areturn
```

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