Roadmap

C:
```c
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
1000110100000100
1000100111000010
1100000111111010
```

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

OS:
```
Windows 8
```

Computer system:
```
Intel Core i7
```

Java vs. C

Spring 2014
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
Meta-point to this lecture

- None of the data representations we are going to talk about are \textit{guaranteed} by Java.
- In fact, the language simply provides an \textit{abstraction}.
- We can't easily tell how things are really represented.
- But it is important to understand \textit{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?*

```java
int array[5]:
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>5</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
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
  - Every access triggers a bounds-check
    - Code is added to ensure the index is within bounds
    - Exception if out-of-bounds

```
int array[5]:

   C  | ?? | ?? | ?? | ?? | ?? |
   0  | 4  |     |    |    |    |
   20 | 24 |

Java | 5  | 00 | 00 | 00 | 00 | 00
```

**Bounds-checking sounds slow, but:**
1. Length is likely in cache.
2. Compiler may store length in register for loops.
3. Compiler may prove that some checks are redundant.
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’:

```
Java: Unicode
6 00 43 00 53 00 45 00 33 00 35 00 31
```

```
C: ASCII
43 53 45 33 35 31 \0
```

Java vs. C
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
```c
struct rec {
    int i;
    int a[3];
    struct rec *p;
};
```

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

Spring 2014
Pointer/reference fields and variables

- In C, we have “->” and “.” for field selection depending on whether we have a pointer to a struct or a struct
  - (*r).a is so common it becomes r->a

- In Java, all non-primitive variables are references to objects
  - We always use r.a notation
  - But really follow reference to r with offset to a, just like C’s r->a

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

```
r = new Rec();
r2 = new Rec();
r.i = val;
r.a[2] = val;
r.p = r2;
```
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(...);
some_fn(&(r.a[1]))  //ptr
```

```java
class Rec {
    int i;
    int[] a = new int[3];
    Rec p;
}
Rec r = new Rec();
some_fn(r.a, 1)  // ref, index
```
Casting in C (example from Lab 5)

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

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

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 Car extends Vehicle {
    int wheels;
}

class Boat extends Vehicle {
    int propellers;
}

// 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 Car(); // ok, everything needed for Vehicle is also in Car
Vehicle v2 = v1; // ok, v1 is already a Vehicle
Car c2 = new Boat(); // incompatible type — Boat and Car are siblings
Car c3 = new Vehicle(); // wrong direction; elements in Car not in Vehicle (wheels)
Boat b2 = (Boat) v; // run-time error; Vehicle does not contain all elements in Boat (propellers)
Car c4 = (Car) v2; // 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

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 objects

- **vtable pointer**: points to *virtual method table*
  - like a jump table for instance ("virtual") methods plus other class info
  - one table per class

- **header**: GC info, hashing info, lock info, etc.
  - no size – why?

- **new**: allocate space for object; zero/null fields; run constructor
  - compiler actually resolves constructor like a static method
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., \texttt{p.samePlace(q)}) is chosen \textit{at run-time} by lookup in the vtable.

\textbf{Java:}

```java
Point p = new Point();
return p.samePlace(q);
```

\textbf{C pseudo-translation:}

```c
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:**

```java
Point p = new Point();

return p.samePlace(q);
```

**C pseudo-translation:**

```c
Point* p = calloc(1,sizeof(Point));
p->header = ...;
p->vtable = &Point_vtable;
p->vtable[0](p);

return p->vtable[1](p, q);
```
Subclassing

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

- **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

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

vtable for PtSubClass (not Point)
Pointer to old code for constructor
Pointer to new code for samePlace

vtable x y aNewField

aNewField tacked on at end
Dynamic dispatch

Point object

```
Point p = ???;
return p.samePlace(q);
```

// works regardless of what p is

```
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/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
  - 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

- 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

Compiled

C

Java

Java

Lisp

Interpreted
Virtual Machine Model

- High-Level Language Program
  - Bytecode compiler
  - Virtual Machine Language
    - Virtual machine (interpreter)
    - JIT compiler
  - Native Machine Language
    - Ahead-of-time compiler

compile time
run time

Spring 2014
Java vs. C
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

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