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:

OS:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
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 including dynamic dispatch
  - 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 *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.
The Other Huge Point

- CSE351 has given you a “really different feeling” about what computers do and how programs execute

- We have occasionally contrasted to Java, but CSE143 and similar may still feel like, “a different world.”

- It’s not – it’s just a higher-level of abstraction
  - Connect these levels via how-one-could-implement-Java in 351 terms
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
  - Java’s portability-guarantee fixes the sizes of all types
    - Example: int is 4 bytes in Java regardless of implementation
  - No unsigned types to avoid conversion pitfalls
    - Added some useful methods in Java 8 (also use bigger signed types)

- null is typically represented as 0 but “you can’t tell”

- Much more interesting:
  - Arrays
  - Characters and strings
  - Objects
Data in Java: Arrays

- 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]; // C
int[] array = new int[5]; // Java
```

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

| 5  | 00 | 00 | 00 | 00 | 00 |

Java vs. C
Data in Java: Arrays

- 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

```java
int array[5]; // C
int[] array = new int[5]; // Java
```

Bounds-checking sounds slow, but:
1. Length field is likely in cache
2. Compiler may store length field in register for loops
3. Compiler may prove that some checks are redundant
Data in Java: Characters & Strings

- 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
  - All String objects read-only (vs. StringBuffer)

The string ‘CSE351’:

C: ASCII

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>4</th>
<th>7</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>53</td>
<td>45</td>
<td>33</td>
<td>35</td>
</tr>
</tbody>
</table>

Java: Unicode

| 6 | 00 | 43 | 00 | 53 | 00 | 45 | 00 | 33 | 00 | 35 | 00 | 31 |
Data structures (objects) in Java

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

**Example of array stored “inline”**

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

Java
```
class Rec {
    int i;
    int[] a = new int[3];
    Rec p;
    ...
}
```
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 \(r->a\)
- So, no Java field needs more than 8 bytes

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

```java
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(...);
```

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

Java vs. C
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 {
    size_t 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 Vehicle {
    int passengers;
}

class Car extends Vehicle {
    int wheels;
}

class Boat extends Vehicle {
    int propellers;
}

class Object {
    ...
}
```

// 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();

Vehicle v2 = v1;
Car c2 = new Boat();

Car c3 = new Vehicle();

Boat b2 = (Boat) v;
Car c4 = (Car) v2;
Car c5 = (Car) b1;
Type-safe casting in Java

- Can only cast compatible object references

```java
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 declared as type Vehicle
Car c2 = new Boat(); // Compiler error - Incompatible type - elements in Car that are not in Boat (classes are siblings)
Car c3 = new Vehicle(); // Compiler error - 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 refers to a Car at runtime
Car c5 = (Car) b1; // Compiler error - Incovertible types, b1 is declared as type Boat
```

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
- **Object header**: GC info, hashing info, lock info, etc. (no size – why?)
- When we call "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., `p.samePlace(q)`) is chosen *at run-time* by lookup in the vtable.

**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);
```
Method dispatch

p
Point object

q
Point object

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

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);
### 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”, use same vtable position
- 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");
    }
}

PtSubclass object

header | vtable | x | y | aNewField

vtable for PtSubClass (not Point)

constructor -> Pointer to old code for constructor

samePlace -> Pointer to new code for samePlace

sayHi -> Pointer to code for sayHi

aNewField tacked on at end
Dynamic dispatch

Point object

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

C pseudo-translation:
// works regardless of what p is
return p->vtable[1](p, q);

PtSubclass object

PtSubclass vtable

Point vtable
That’s the “magic”

- In CSE143, it may have seemed “magic” that an *inherited* method could call an *overridden* method
  - You were tested on this endlessly 😊

- The “trick” in the implementation is this part:
  \[ p->vtable[i](p, q) \]
  - In the body of the pointed-to code, any calls to (other) methods of this will use \( p->vtable \)
  - Dispatch determined by \( p \), not the class that defined a method
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 (something close to) the *source code* directly
- 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”

- 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
Compiling and Running Java

- The Java compiler converts Java into **Java bytecodes**
- **Java bytecodes** are stored in a .class 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, this interpreter is called the **Java Virtual Machine**
- To run the Java virtual machine:
  - `java Foo`
  - This loads the contents of **Foo.class** and interprets the bytecodes

Note: The Java virtual machine is different than the CSE VM running on VMWare
Virtual Machine Model

- High-Level Language Program (e.g. Java, C)
- Virtual Machine Language (e.g. Java bytecodes)
- Native Machine Language (e.g. x86, MIPS)

Compile time:
- Bytecode compiler (e.g. javac Foo.java)

Run time:
- Virtual machine (interpreter) (e.g. java Foo)
- JIT compiler

Ahead-of-time compiler
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, ...

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)

‘i’ load 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

Holds pointer ‘this’
Other arguments to method
Other local variables

machine:

bytecode:

bytecode:

variable table
operand stack
constant pool

0 1 2 3 4

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>n</th>
</tr>
</thead>
</table>

compiled to x86:

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

No registers or stack locations; all operations use operand stack.

‘i’ stands for integer, ‘a’ for reference, ‘b’ for byte, ‘c’ for char, ‘d’ for double, ...
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**: 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)
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.

Diagram:

- C# code
- VB.NET code
- J# code
  - Compilers
  - Common Language Infrastructure
    - Common Intermediate Language
    - Common Language Runtime

- .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.

Java vs. C
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
110000011111101000001111

Computer system:

OS:

Windows 8
Mac

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