### 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 **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?*

```java
int array[5]:
C   ?? ?? ?? ?? ?
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.

```
int array[5]:
C   ?? ?? ?? ?? ?
Java 5 00 00 00 00 00
```

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’:

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

```
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

```java
class Rec {
    int i;
    int a[3];
    struct rec *p;
};
```

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

```java
Rec p = new Rec;
```
### 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

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

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

### Casting in C (example from Lab 5)

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

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

### 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

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

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

// 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

---
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 run-time by lookup in the vtable.

Java:

```java
Point p = new Point();
header vtable pointer x y
...
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);
```

Java objects

```
Point object

```

```java
Point p = new Point();
header vtable pointer x y
Point class vtable
```

- 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

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();
header vtable pointer x y
...
```
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”

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

Java vs. C

```
Java: Point p = ???;
// works regardless of what p is
return p->vtable[1](p, q);
```

```c
C pseudo-translation: ptSubClass vtable y x pNewField
```

Dynamic dispatch

- For Point object
  - Point vtable

- For PtSubClass object
  - PtSubClass vtable

- Java: Point p = ???;
- C pseudo-translation: `ptSubClass vtable y x pNewField`

Agenda

- Inside
- HW4 grades/feedback are up
- Lab 5 due tonight! Go, go, go!
  - If I’m not in the office today, I might be in the basement labs
- Tomorrow: Review Session
  - bring your own questions
- Final exam topics/materials
  - See past exams (website)
  - See topic manifest (website: last Friday’s slides)
- Today
  - Finish up Java
  - Brief tour of Parallel Processing
  - 351 Conclusions :(

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

- aNewField tacked on at end
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

Virtual Machine Model

High-Level Language Program

Bytecode

compile time

run time

Virtual Machine Language

Virtual machine (interpreter)

JIT

Ahead-of-time

compiler

Native Machine Language

Java bytecode

- like assembly code for JVM, but works on all JVMs: hardware-independent
- typed (unlike ASM)
- strong JVM protections
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:  
\[ \text{2A B4 00 05 B0} \]


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)
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