Java and C (part I)

CSE 351 Summer 2020

Instructor:

Porter Jones

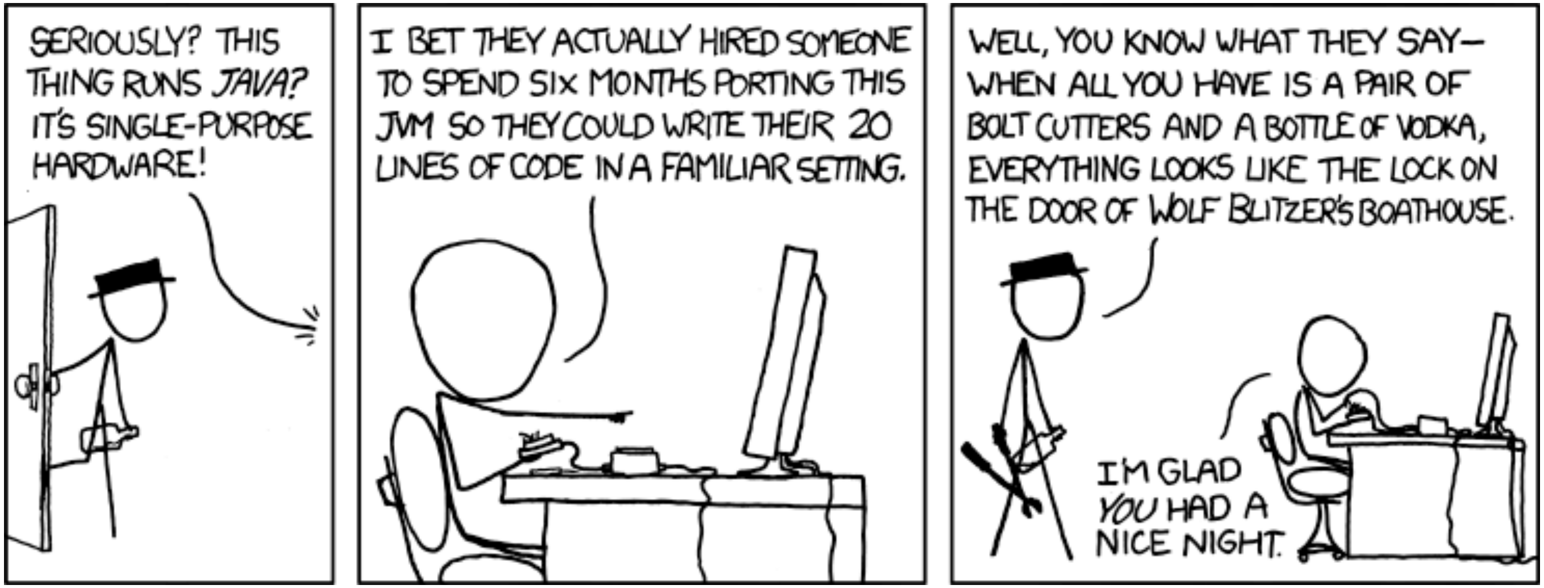
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



https://xkcd.com/801/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-8-19

- Can still do hw19 (it's optional/not for credit)
- hw23 due Monday (8/24) – 10:30am
  - Cover most of the material today, a few more things Friday

- Section tomorrow is TA's Choice & time for questions
  - See cool applications of 351 material and ask your TAs questions!

Administrivia

- Lab 5 due last day of quarter (Friday 8/21)
  - ***Cutoff is Saturday 8/22 @11:59pm (only one late day can be used!)***
  - The most significant amount of C programming you will do in this class – combines lots of topics from this class: pointers, bit manipulation, structs, examining memory
  - Understanding the concepts *first* and efficient *debugging* will save you lots of time
  - Can be difficult to debug so please start early and use OH
  - Light style grading
  - hw22 will help get you started!

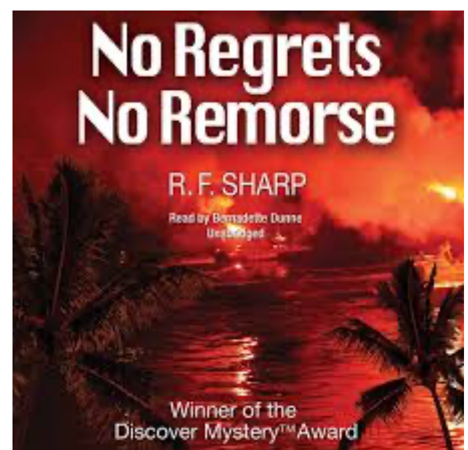
Java vs. C is included

- Unit Summary 3 due last day of quarter (Friday 8/21)
  - ***Cutoff is Saturday 8/22 @11:59pm (only one late day can be used!)***

Course Evaluation Reminder Meme

Due Friday @ 11:59pm

- Reminder to please fill out your course evaluations!! (you should have received an email with a link to the eval)
- The "Spamming Your Students About Course Evals" Starter Pack:



Roadmap

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
```

OS:

Windows 10 OS X Yosemite

Computer system:

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

Java vs. C

- Reconnecting to Java (hello 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

Worlds Colliding

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

- We have occasionally contrasted to Java, but CSE143 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

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*** (Java language specification)
  - Tells us how code should behave for different language constructs, but 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

Data in Java

- Integers, floats, doubles, pointers – same as C
  - "Pointers" are called "references" in Java, but 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 machine
  - 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"

~~Object o = 0;~~
Object o = null;

- Much more interesting:
  - **Arrays**
  - **Characters and strings**
  - **Objects**

Data in Java: Arrays

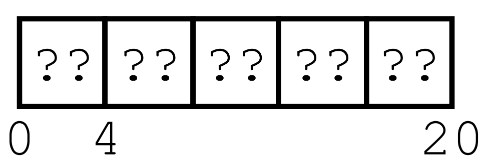
in C no initialization (except if you use calloc)

- 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?*

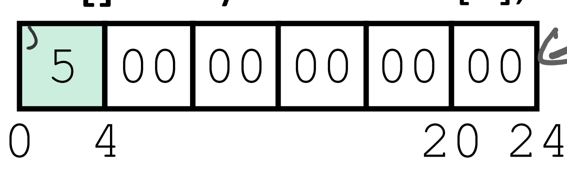
C: sum(int *ar, int size)

Java: sum(int[] ar) { sum ar.length }

C: **int** array[5];



Java: int[] array = **new** int[5];



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 <u>bounds-check</u>
  - Code is added to ensure the index is within bounds
  - Exception if out-of-bounds

C:

int array[5];

??	??	??	??	??

0 4 20

Java:

int[] array = **new** int[5];

5	00	00	00	00	00

0 4 20 24

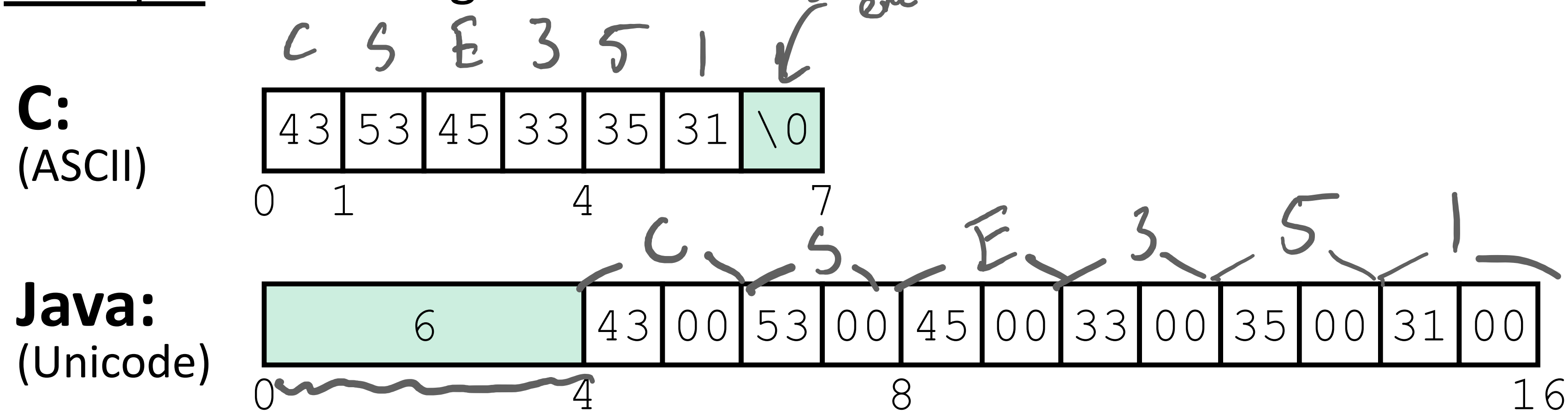
To speed up bounds-checking:

- Length field is likely in cache
- Compiler may store length field in register for loops
- Compiler may prove that some checks are redundant

Data in Java: Characters & 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)

Example: the string "CSE351"



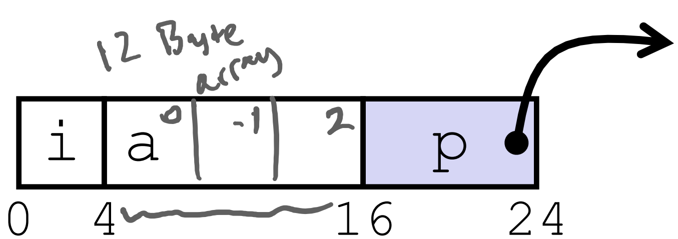
Data in Java: Objects

- Data structures (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;
};
```

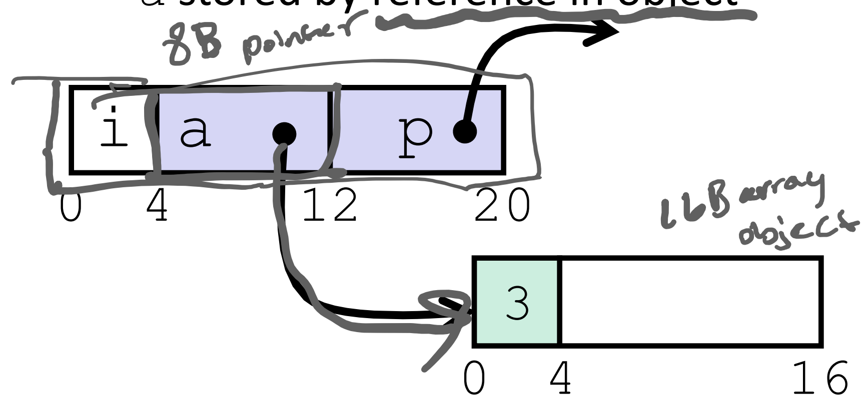
- `a[]` stored "inline" as part of struct



Java:

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

- `a` stored by reference in object



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 `r->a` in C
  - So no Java field needs more than 8 bytes

C:

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

references

Java:

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

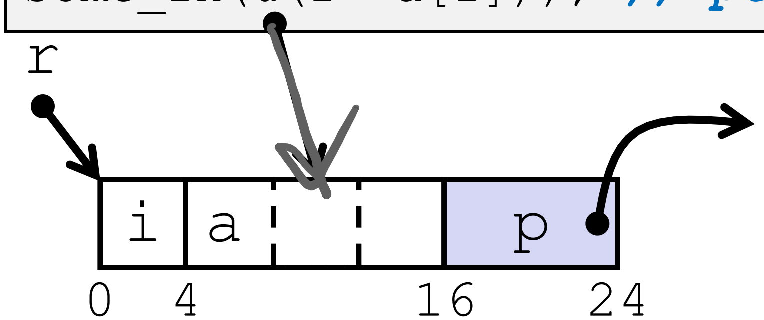
Pointers/References

both store addresses

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

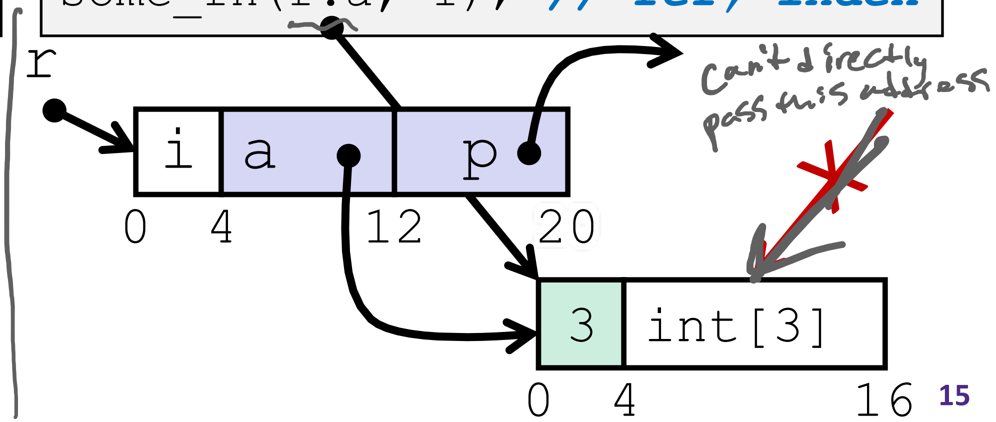
C:

```
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
```



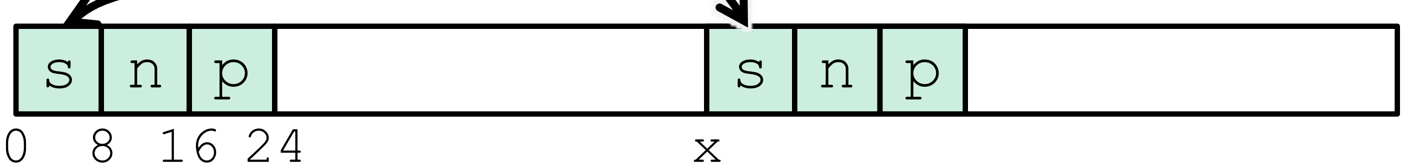
Casting in C (example from Lab 5)

- Can cast any pointer into any other pointer
  - Changes dereference and arithmetic behavior

```
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 * to do unscaled addition

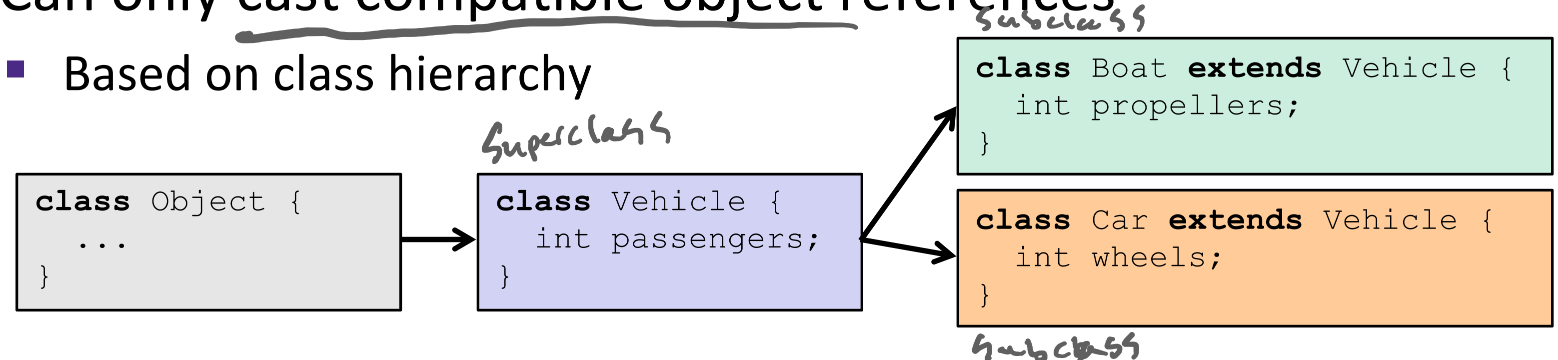
Cast back into BlockInfo * to use as BlockInfo struct



143 flash back

Type-safe casting in Java

- Can only cast compatible object references
  - Based on class hierarchy




```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat     b1 = new Boat();    // |--> sibling
Car      c1 = new Car();     // |--> sibling

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
  - Based on class hierarchy

```
class Object {
  ...
}
```

```
class Vehicle {
  int passengers;
}
```

```
class Boat extends Vehicle {
  int propellers;
}
```

```
class Car extends Vehicle {
  int wheels;
}
```

```
Vehicle  v  = new Vehicle(); // super class of Boat and Car
Boat     b1 = new Boat();    // |--> sibling
Car      c1 = new Car();     // |--> sibling

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;
```

- `Vehicle v1 = new Car();` ← ✓ Everything needed for `Vehicle` also in `Car`
- `Vehicle v2 = v1;` ← ✓ `v1` is declared as type `Vehicle`
- `Car c2 = new Boat();` ← ✗ Compiler error: Incompatible type – elements in `Car` that are not in `Boat` (siblings)

Polling Question [Java I]

Vehicle v2 = new Boat();

- Given:

```
Vehicle  v = new Vehicle();
```

- What happens with this line of code:

checked at runtime

```
Boat b2 = (Boat) v;
```

  - Vote at http://pollev.com/pbjones

A. Compiles and Runs with no errors

B. Compiler error

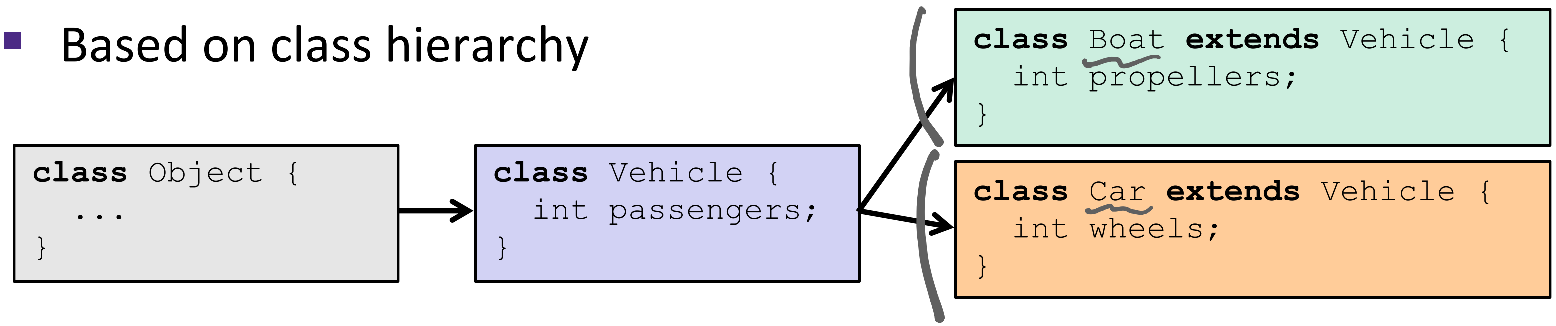
C. Compiles fine, then Run-time error

D. We're lost…

Vehicle v = new Boat();

Type-safe casting in Java

- Can only cast compatible object references
  - Based on class hierarchy




```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat     b1 = new Boat();   // |--> sibling
Car      c1 = new Car();    // |--> sibling

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;
```

- `Vehicle v1 = new Car();` ← ✓ Everything needed for `Vehicle` also in `Car`
- `Vehicle v2 = v1;` ← ✓ `v1` is declared as type `Vehicle`
- `Car c2 = new Boat();` ← ✗ Compiler error: Incompatible type – elements in `Car` that are not in `Boat` (siblings)
- `Car c3 = new Vehicle();` ← ✗ Compiler error: Wrong direction – elements `Car` not in `Vehicle` (`wheels`)
- ① `Boat b2 = (Boat) v;` ← ✗ Runtime error: `Vehicle` does not contain all elements in `Boat` (`propellers`)
- `Car c4 = (Car) v2;` ← ✓ `v2` refers to a `Car` at *runtime*
- ② `Car c5 = (Car) b1;` ← ✗ Compiler error: Unconvertable types – `b1` is declared as type `Boat`

Java Object Definitions

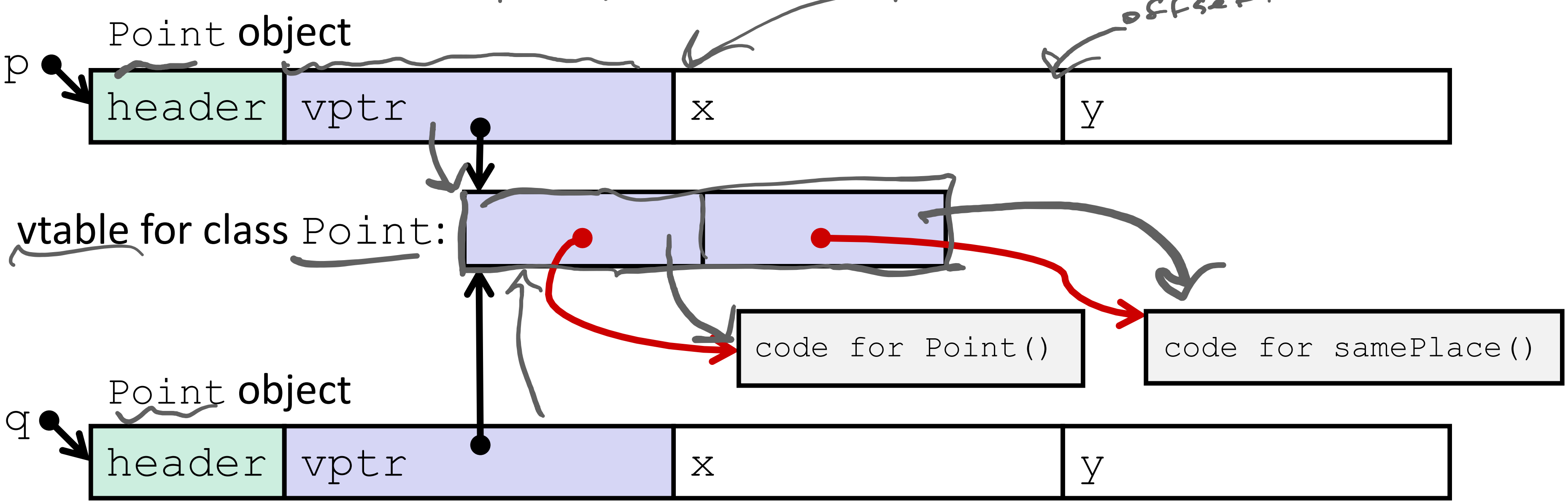
```
class Point {
  double x;
  double y;

  Point() {
    x = 0;
    y = 0;
  }

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

- fields: `double x; double y;`
- constructor: `Point() {`
- method(s): `boolean samePlace(Point p) { ... }`
- creation: `Point p = new Point();`

Java Objects and Method Dispatch



- ❖ *Virtual method table (vtable)*
  - Like a jump table for instance ("virtual") methods plus other class info
  - One table per class
  - Each object instance contains a *vtable pointer (vptr)*
- ❖ *Object header* : GC info, hashing info, lock info, etc.
  - Why no size?

Java Constructors

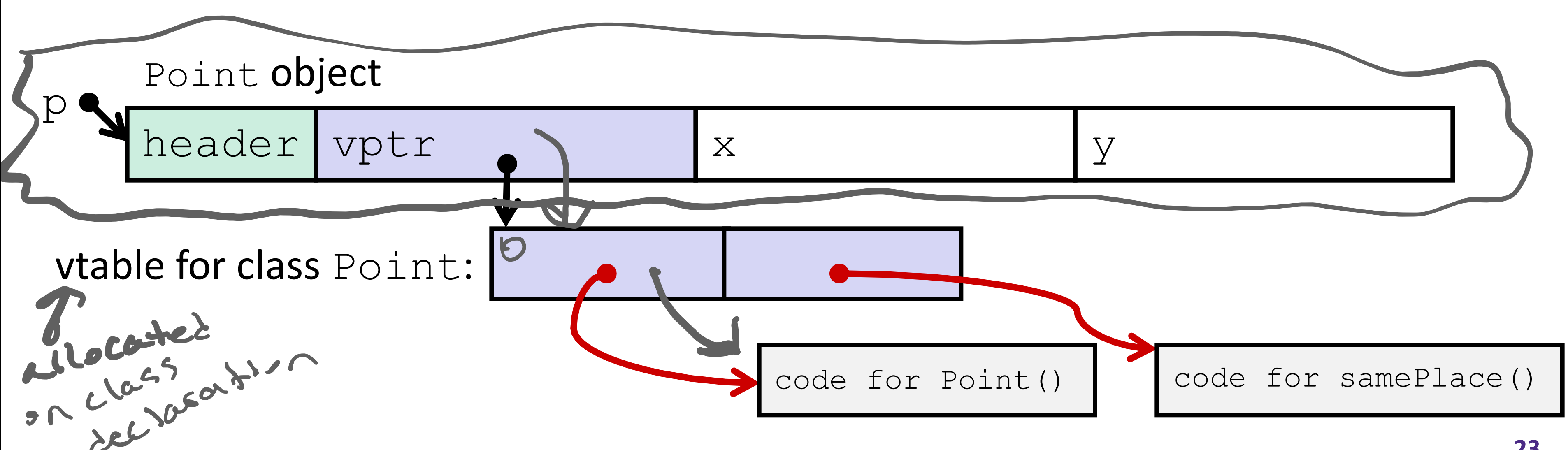
- **When we call `new`:** allocate space for object (data fields and references), initialize to zero/null, and run constructor method

Java:

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

C pseudo-translation:

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



Java Methods

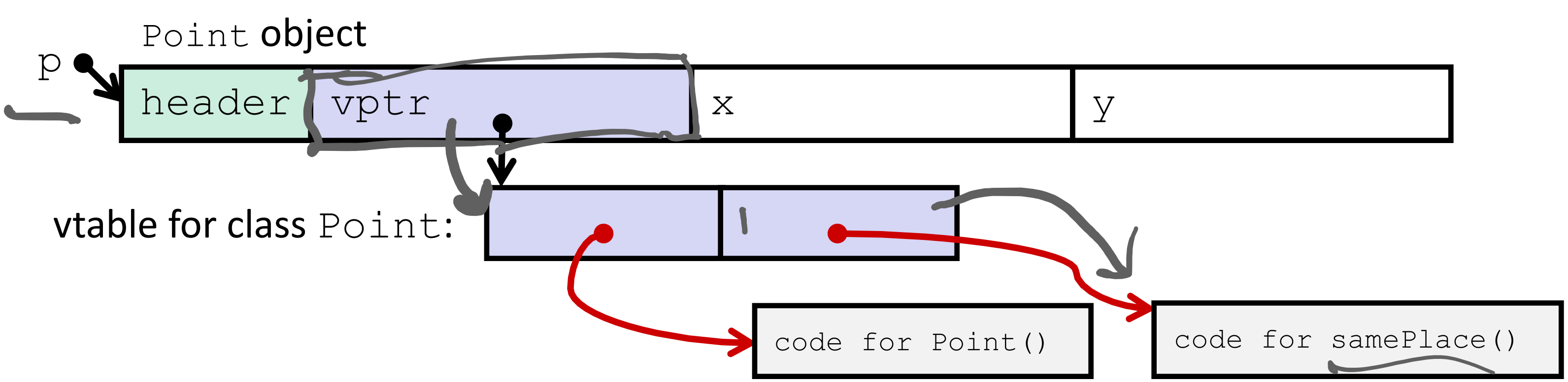
- Static methods are just like functions
- Instance methods:
  - Can refer to *this*;
  - Have an implicit first parameter for *this;* and
  - Can be overridden in subclasses
- The code to run when calling an instance method is chosen *at runtime* by lookup in the vtable

Java:

```
p.samePlace(q);
```

C pseudo-translation:

```
p->vptr[1](p, q);
```



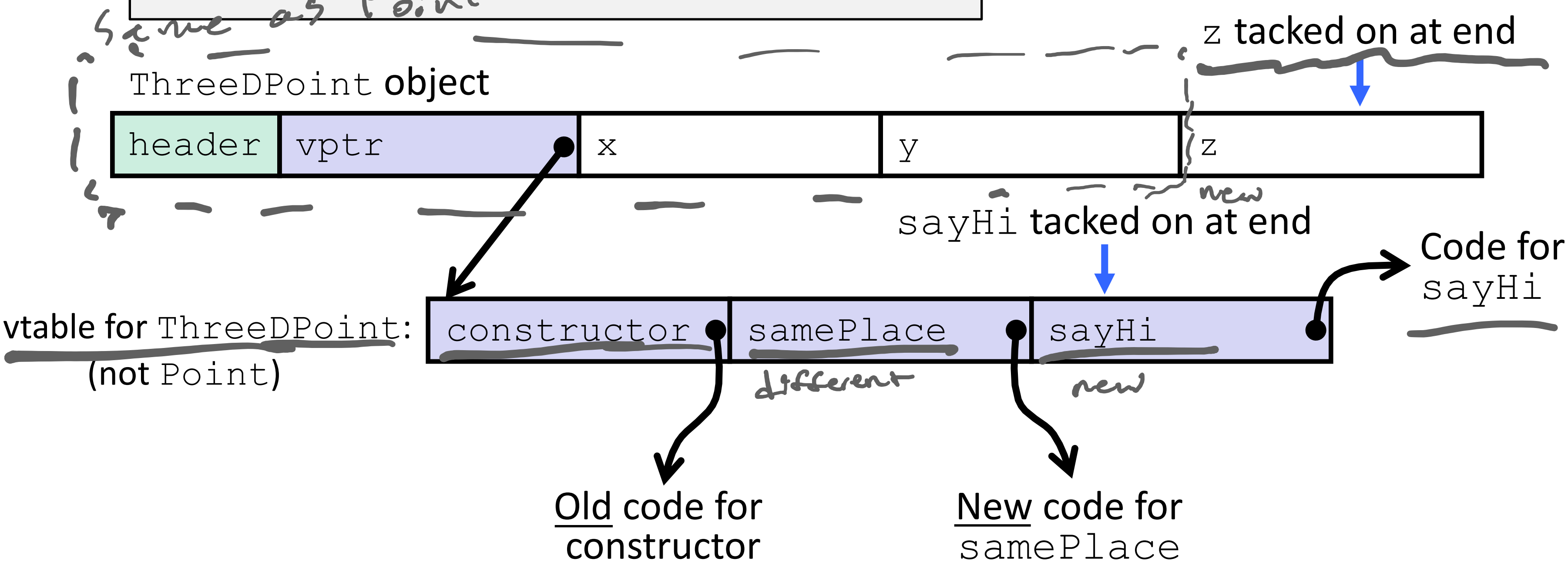
Subclassing

```
class ThreeDPoint extends Point {
    double z;
    boolean samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}
```

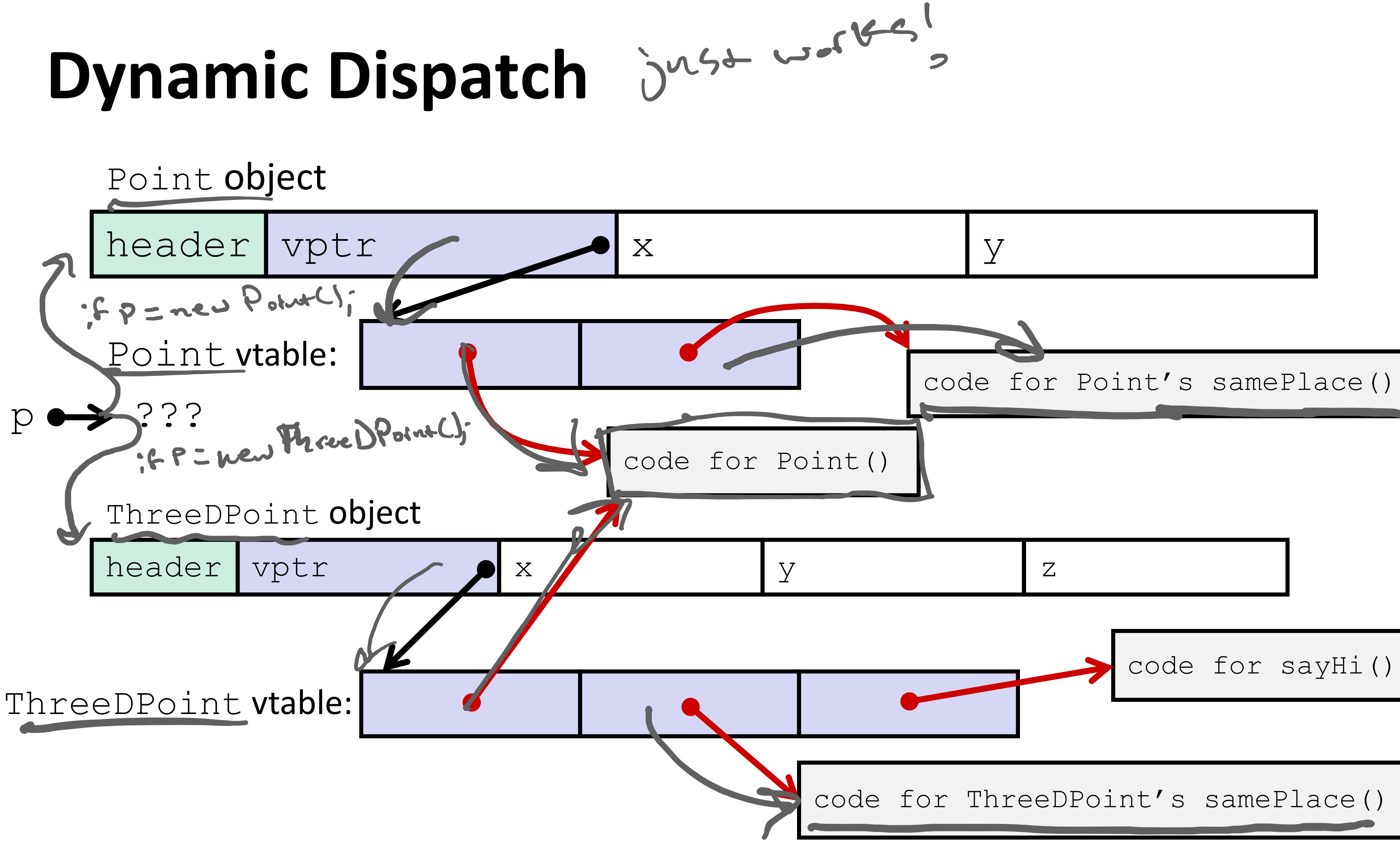
- Where does "`z`" go? At end of fields of `Point`
  - `Point` fields are always in the same place, so `Point` code can run on `ThreeDPoint` objects without modification
- Where does pointer to code for two new methods go?
  - No constructor, so use default `Point` constructor
  - To override "`samePlace`", use same vtable position
  - Add new pointer at end of vtable for new method "`sayHi`"

Subclassing

```
class ThreeDPoint extends Point {
    double z;
    boolean samePlace(Point p2) {
        return false;
    }
    void sayHi() {
        System.out.println("hello");
    }
}
```



Dynamic Dispatch



Java:

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

C pseudo-translation:

```
// works regardless of what p is
return p->vtr[1](p, q);
```

Ta-da!

- 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->vptr[i](p,q)
```

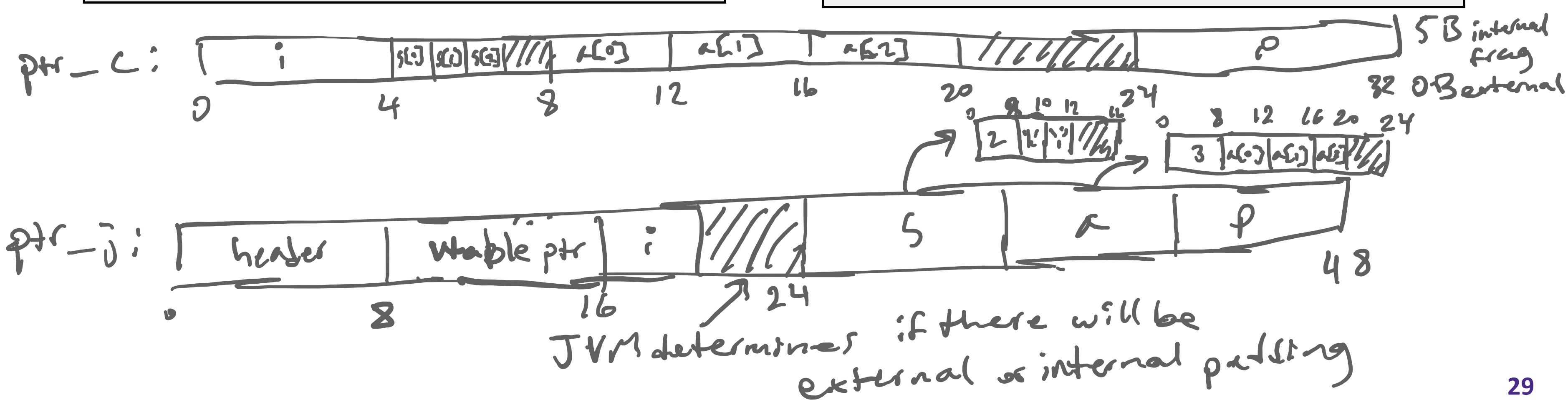
  - In the body of the pointed-to code, any calls to (other) methods of `this` will use `p->vptr`
  - Dispatch determined by `p`, not the class that defined a method

Practice Question

- Assume: 64-bit pointers, Java objects aligned to 8 B with 8-B header
- What are the sizes of the things being pointed at by `ptr_c` 32 B and `ptr_j`? 48 B

```
struct c {
  int i;
  char s[3];
  int a[3];
  struct c *p;
};
struct c* ptr_c;
```

```
class jobj {
  int i;
  String s = "hi";
  int[] a = new int[3];
  jobj p;
}
jobj ptr_j = new jobj();
```



Practice Question

What would you expect to be the order of contents in an instance of the Car class?

```
class Vehicle {
  int passengers;
  // methods not shown
}
class Car extends Vehicle {
  int wheels;
  // methods not shown
}
```

comes before Car fields

A. header, `Vehicle` vtable ptr, `passengers`, `Car` vtable ptr, `wheels`

B. `Vehicle` vtable ptr, `passengers`, `wheels`

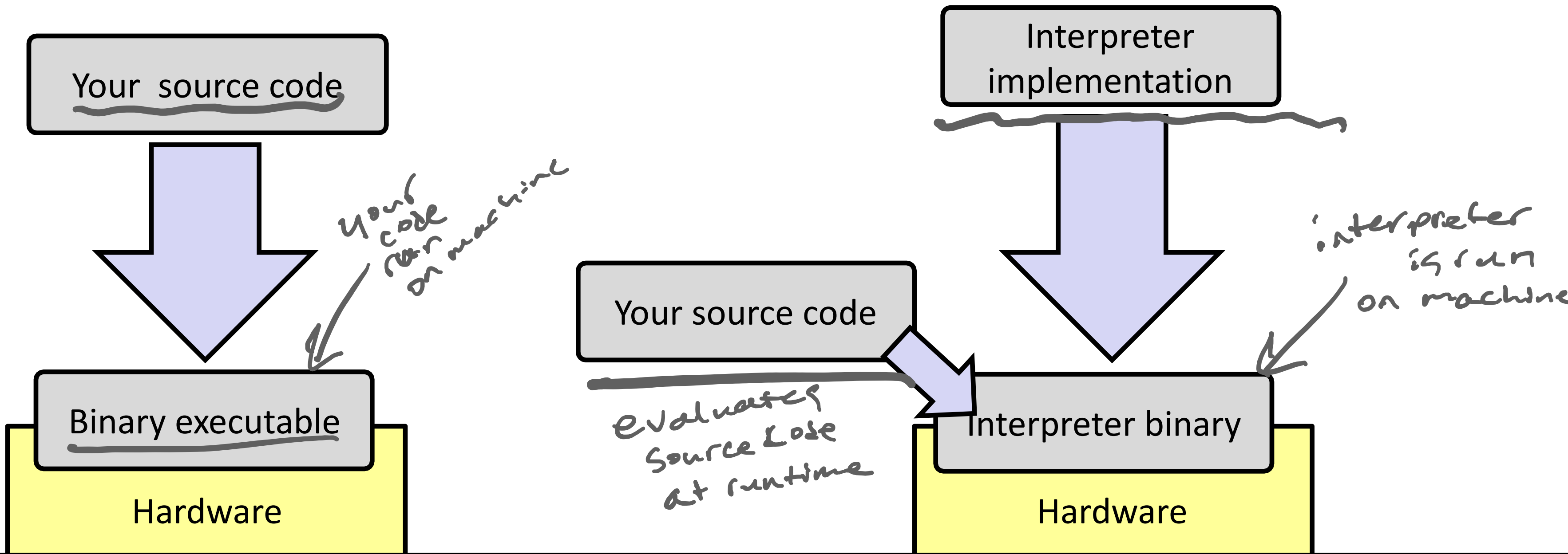
C. header, `Vehicle` vtable ptr, `Car` vtable ptr, `passengers`, `wheels`

D. header, `Car` vtable ptr, `passengers`, `wheels`

E. We're lost…

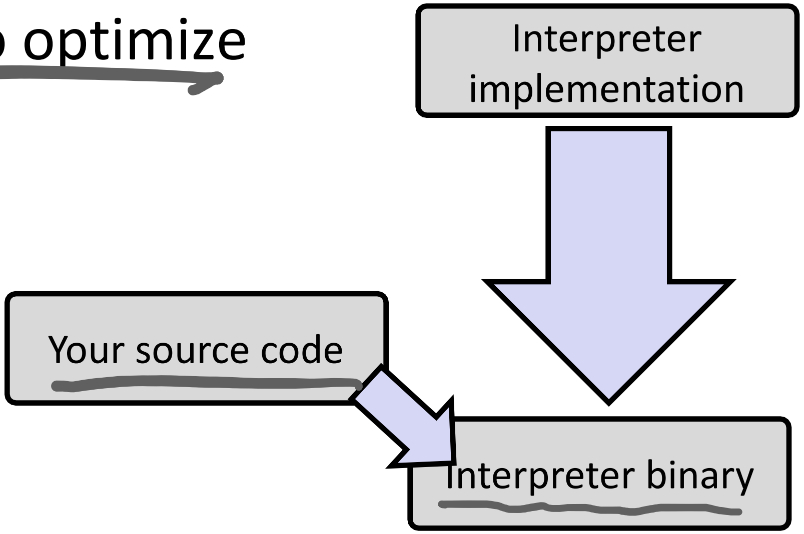
Implementing Programming Languages

- 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 (program), not the program-being-interpreted
- 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. E.g. :
  - Compiling to a bytecode language, then interpreting
  - Doing just-in-time compilation of parts to assembly for performance

"The JVM"

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

- 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**
    - http://en.wikipedia.org/wiki/Just-in-time_compilation
  - Java is sometimes compiled ahead of time (AOT) like C

Compiling and Running Java

143 flashback

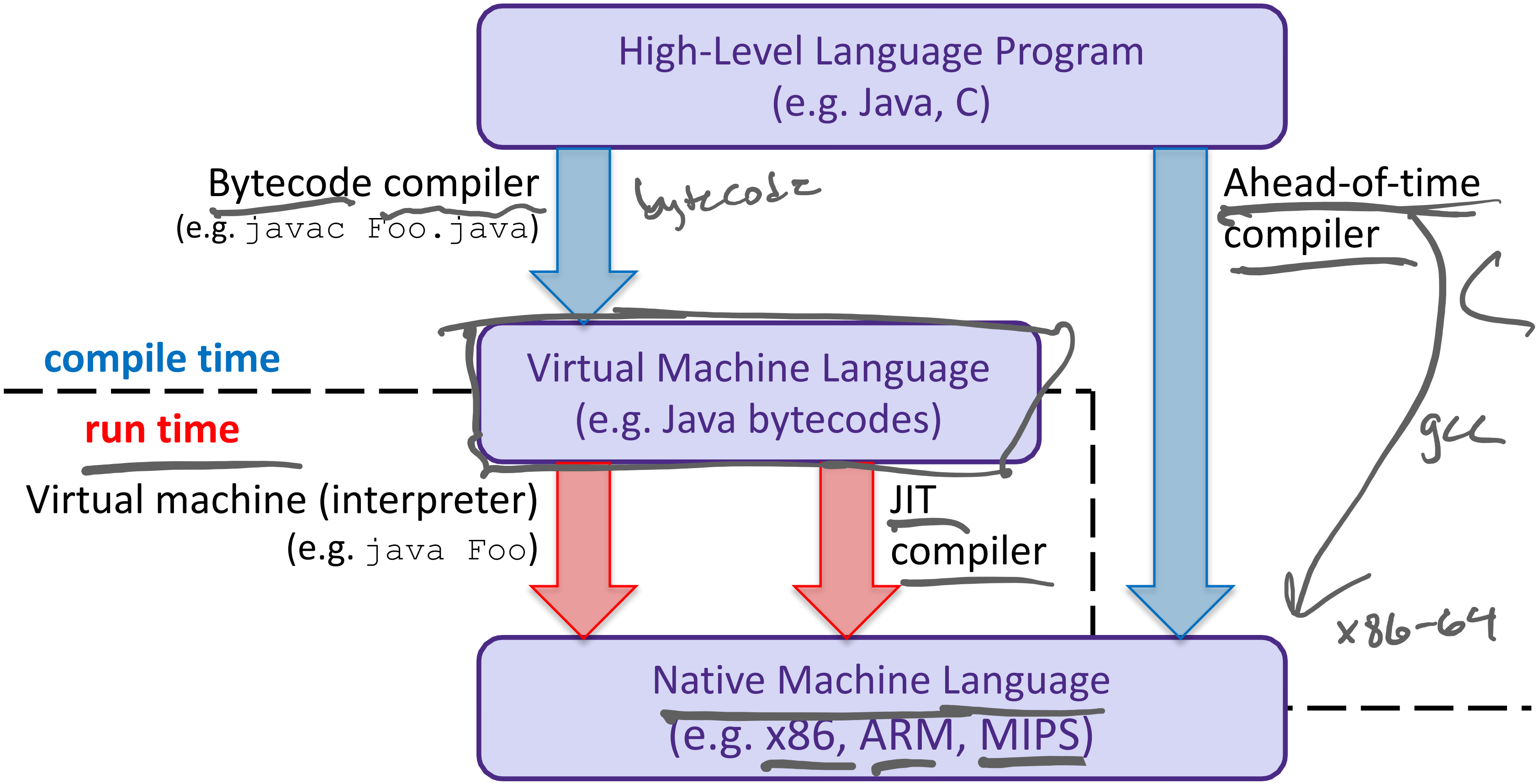
1. Save your Java code in a `.java` file

2. To run the Java compiler:
   - `javac Foo.java`
   - The Java compiler converts Java into *Java bytecodes*
     - Stored in a `.class` file

   not machine code

3. 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 (the JVM)
   - To run the virtual machine:
   - `java Foo`
   - This Loads the contents of `Foo.class` and interprets the bytecodes

Virtual Machine Model



Type-safe casting in Java

- Can only cast compatible object references
  - Based on class hierarchy

```
class Object {
  ...
}
```

```
class Vehicle {
  int passengers;
}
```

```
class Boat extends Vehicle {
  int propellers;
}
```

```
class Car extends Vehicle {
  int wheels;
}
```

```
Vehicle  v = new Vehicle(); // super class of Boat and Car
Boat     b1 = new Boat();   // |--> sibling
Car      c1 = new Car();    // |--> sibling
```

```
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;
```

- `Vehicle v1 = new Car();` ← ✓ Everything needed for `Vehicle` also in `Car`
- `Vehicle v2 = v1;` ← ✓ `v1` is declared as type `Vehicle`
- `Car c2 = new Boat();` ← ✗ Compiler error: Incompatible type – elements in `Car` that are not in `Boat` (siblings)
- `Car c3 = new Vehicle();` ← ✗ Compiler error: Wrong direction – elements `Car` not in `Vehicle` (`wheels`)
- `Boat b2 = (Boat) v;` ← ✗ Runtime error: `Vehicle` does not contain all elements in `Boat` (`propellers`)
- `Car c4 = (Car) v2;` ← ✓ `v2` refers to a `Car` at *runtime*
- `Car c5 = (Car) b1;` ← ✗ Compiler error: Unconvertable types – `b1` is declared as type `Boat`