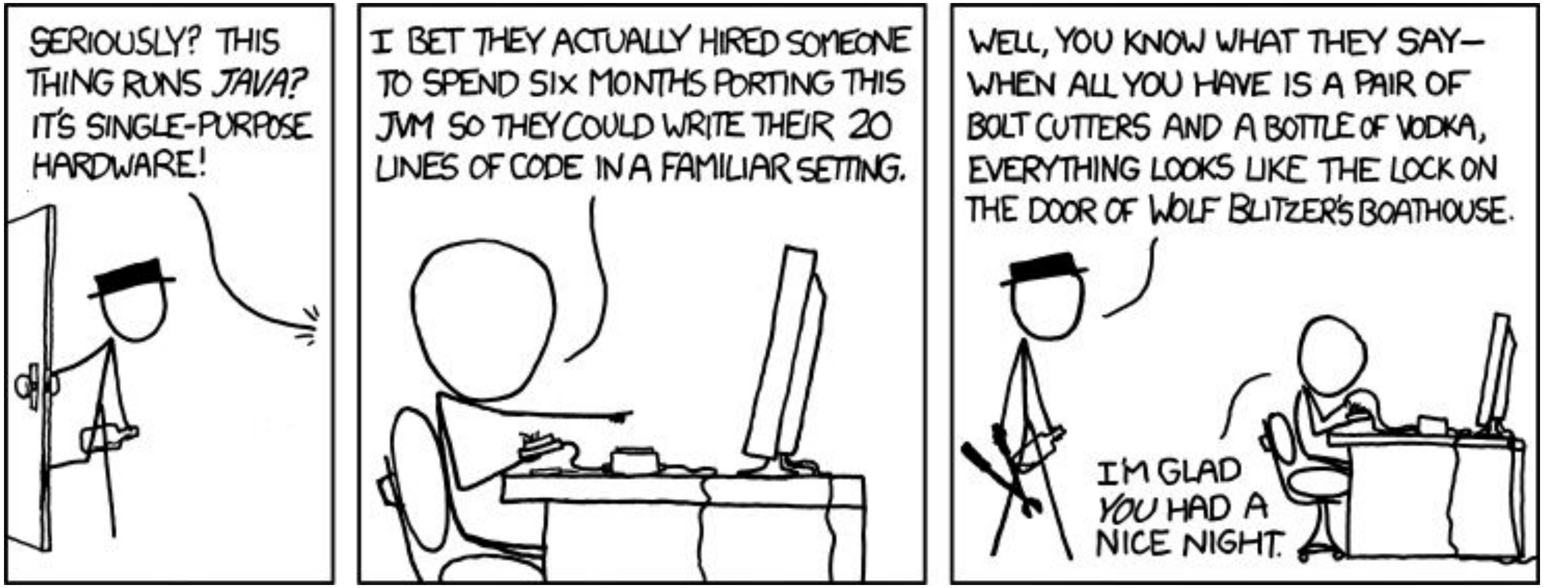
Java and C

CSE 351 Summer 2021

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobs
Tim Mandzyuk

"Home" by Andrew York


SERIOUSLY? THIS THING RUNS JAVA? IT'S SINGLE-PURPOSE HARDWARE!
I BET THEY ACTUALLY HIRED SOMEONE TO SPEND SIX MONTHS PORTING THIS JVM SO THEY COULD WRITE THEIR 20 LINES OF CODE IN A FAMILIAR SETTING.
WELL, YOU KNOW WHAT THEY SAY— WHEN ALL YOU HAVE IS A PAIR OF BOLT CUTTERS AND A BOTTLE OF VODKA, EVERYTHING LOOKS LIKE THE LOCK ON THE DOOR OF WOLF BLITZER'S BOATHOUSE.
I'M GLAD YOU HAD A NICE NIGHT.

https://xkcd.com/801/

Gentle, Loving Reminders

- Lab 5 due tonight!!!!
  - Reach out if you're using late days
- Unit Summary #3 due Friday!
  - No late days!

- Section tomorrow is TA's Choice & time for questions
  - See cool things! Ask your TAs questions!

Course Evals are out!

I’d really appreciate feedback!

Only 15% so far, *due friday*!

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"

Data in Java

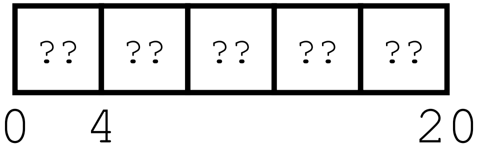
- Much more interesting:
  - **Arrays**
  - **Characters and strings**
  - **Objects**

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?*

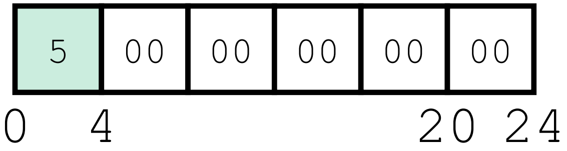
C:

int array[5];



Java:

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

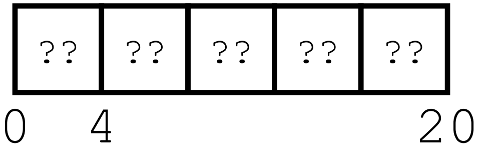


Data in Java: Arrays

- 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

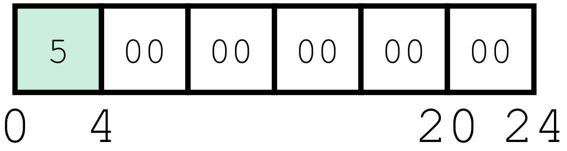
C:

int array[5];



Java:

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



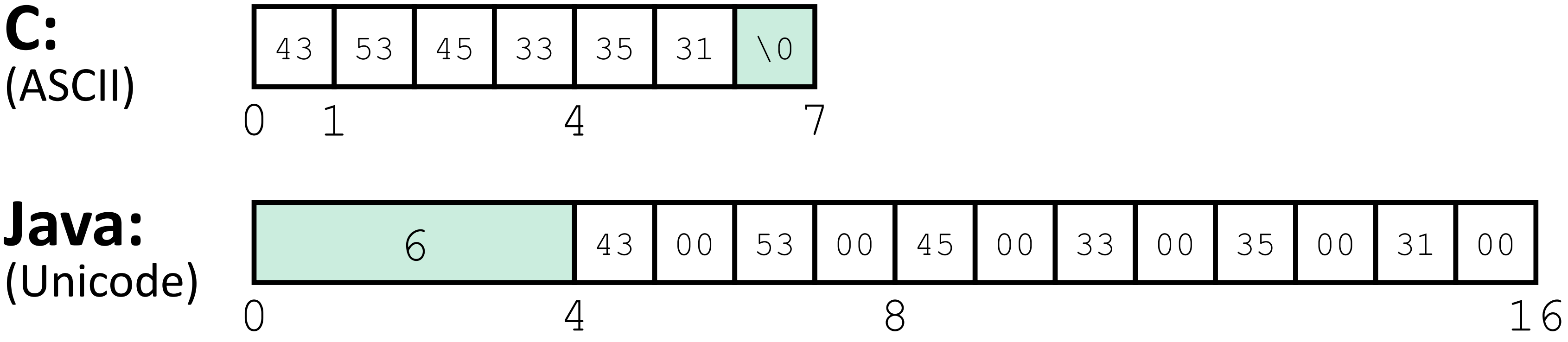
To speed up bounds-checking:

- 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”



C: (ASCII)

43	53	45	33	35	31	\0

Java: (Unicode)

6	43	00	53	00	45	00	33	00	35	00	31	00

Data in Java: Objects

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

C:

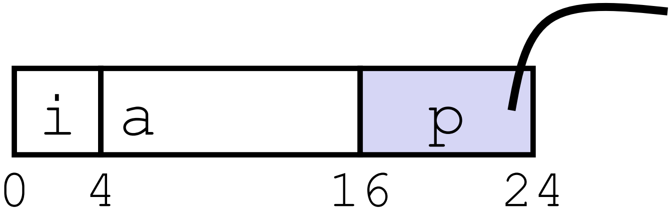
```
struct rec {
  int i;
  int a[3];
  struct rec *p;
};
```

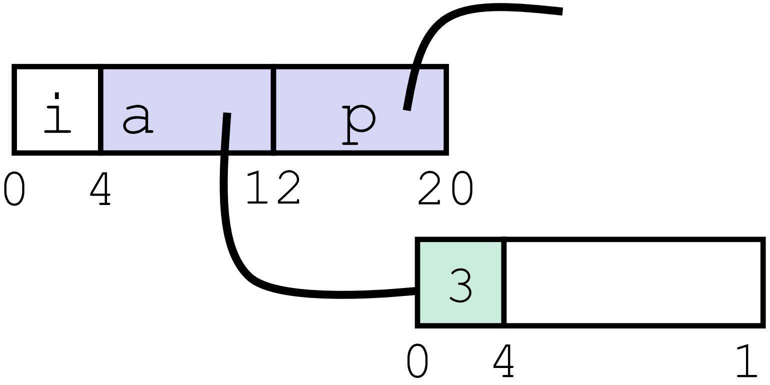
- `a[]` stored “inline” as part of struct

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;
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

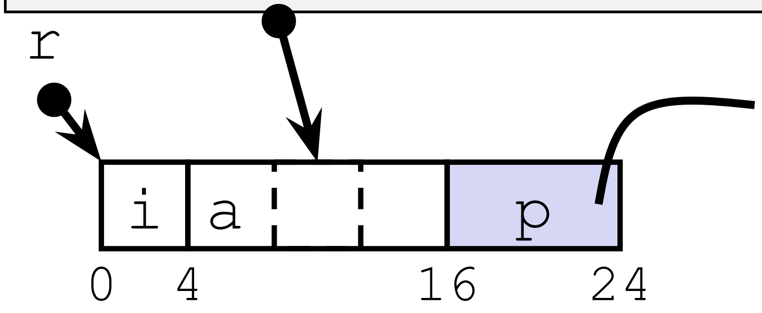
- *Pointers* in C can point to any memory address
- *References* in Java can only point to [the starts of] objects
  - Can only be dereferenced to access a field or element of that object

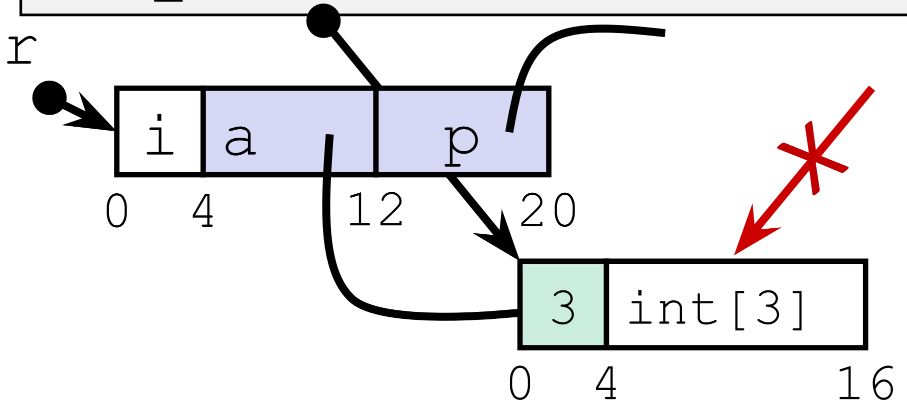
C:

```
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
```





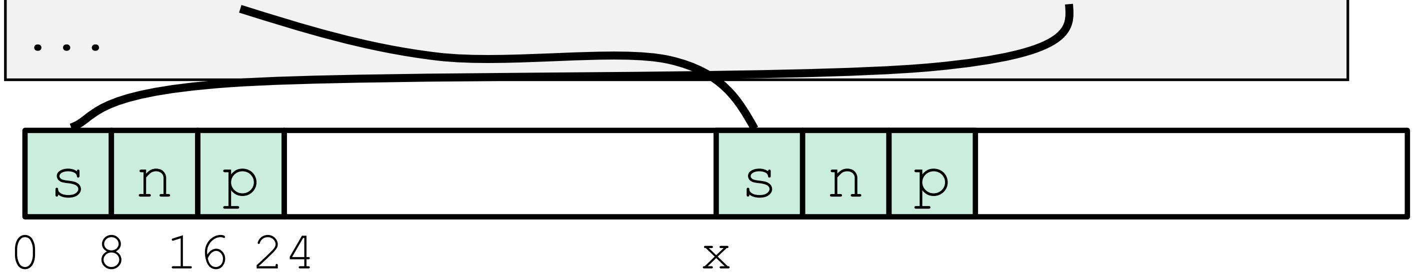
Casting in C (example from Lab 5)

- 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 );
...
```

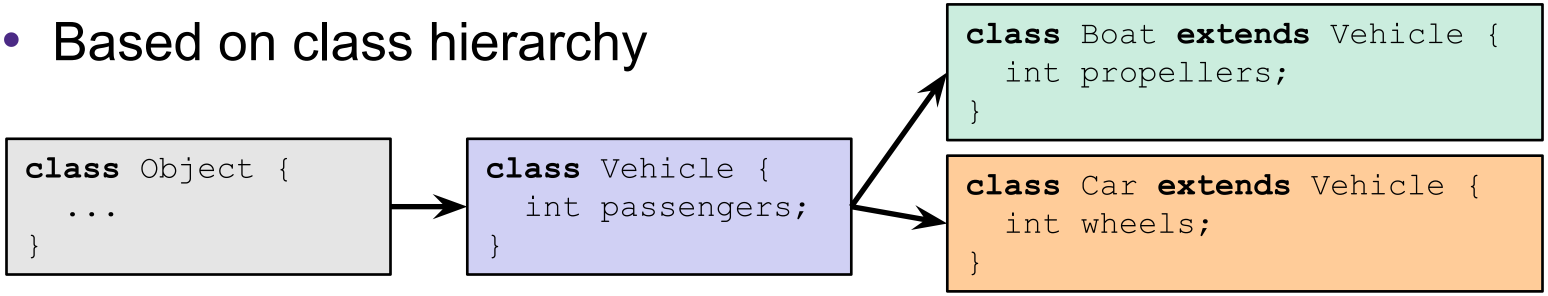
Cast b into char * to do unscaled addition

Cast back into BlockInfo * to use as BlockInfo struct



Type-safe casting in Java

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



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

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

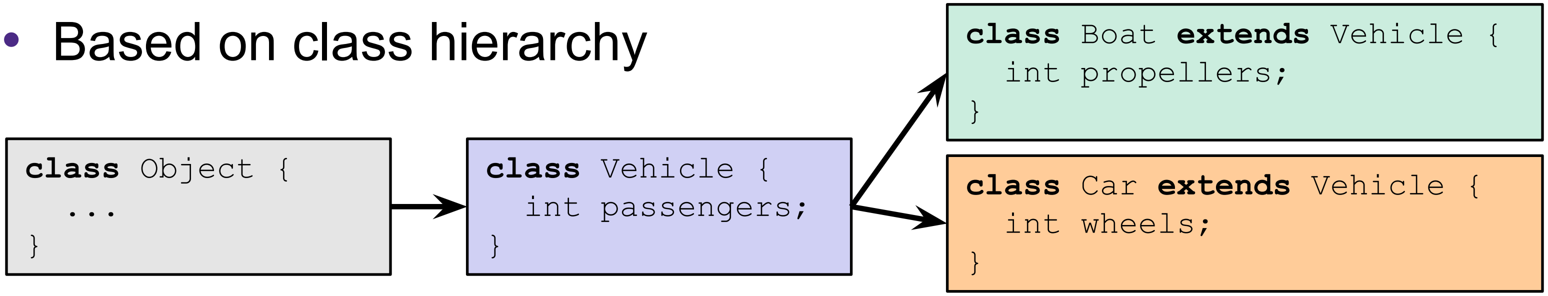
```
class Boat extends Vehicle {
  int propellers;
}
```

```
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;
```

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();     ←  ✓ 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();

Boat    b2 = (Boat) v;

Car     c4 = (Car) v2;
Car     c5 = (Car) b1;
```

Polling Question [Java I]

- Given:

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

- What happens with this line of code:

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

🐶 Compiles and Runs with no errors

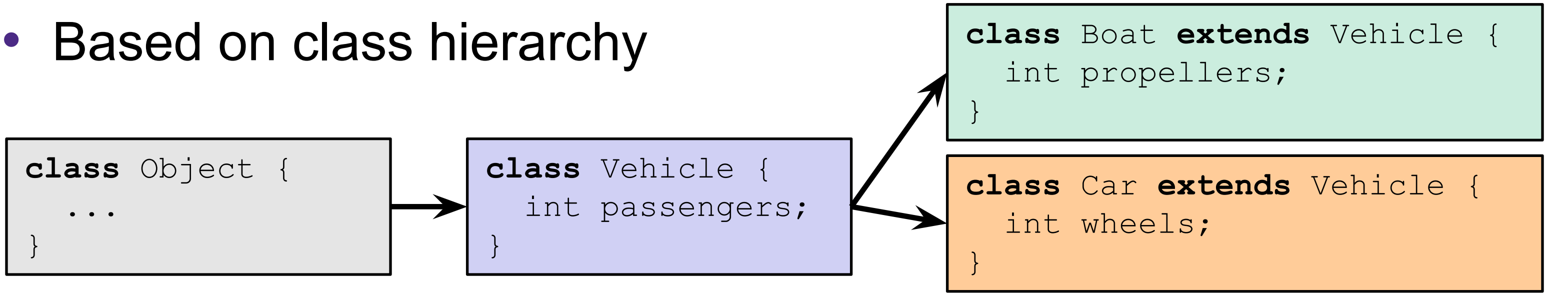
🐈 Compiler error

🐑 Compiles fine, then Run-time error

🥶 Help!

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();` ← ✓ 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)

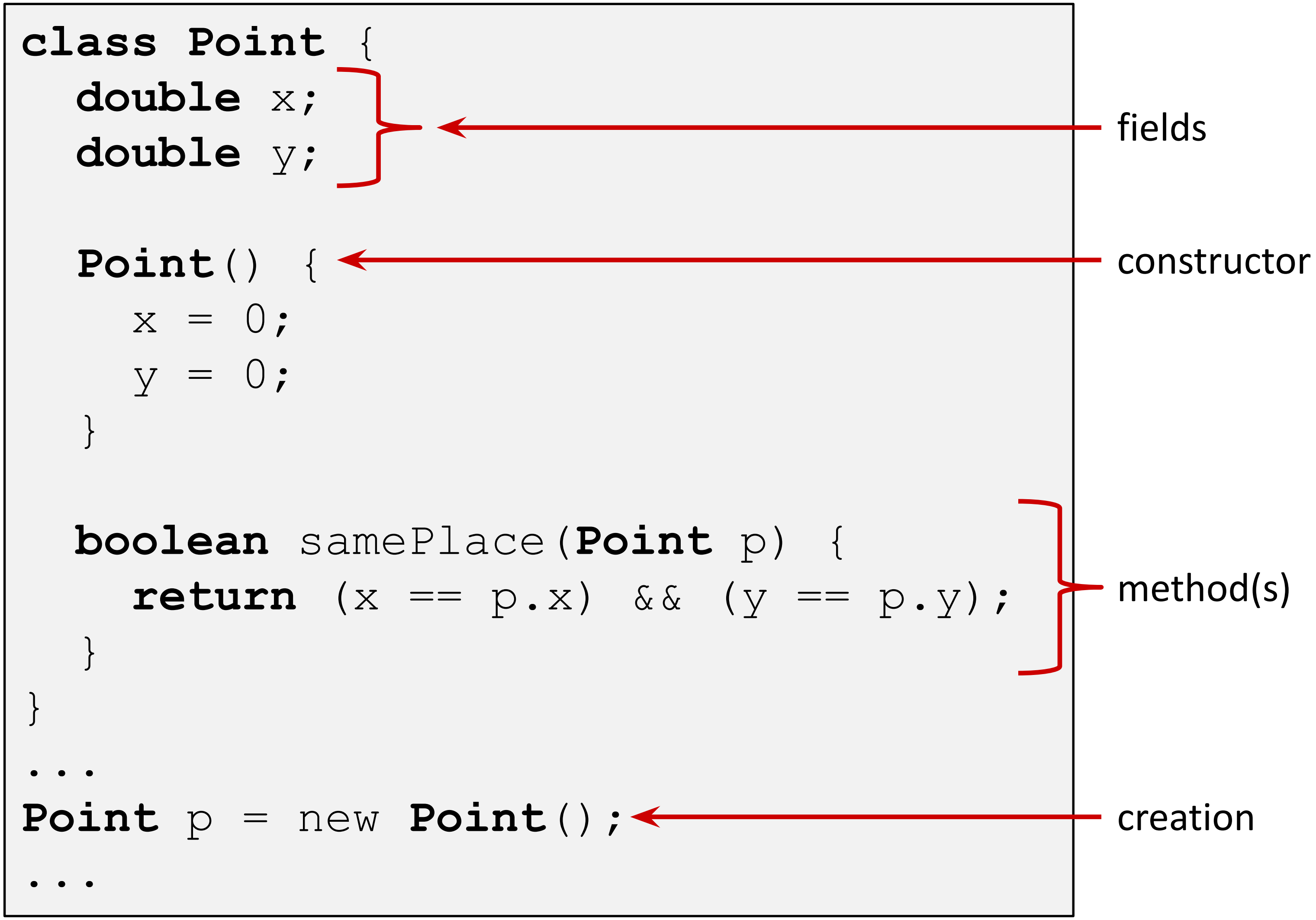
`Car c3 = new Vehicle();` ← ✗ Compiler error: Wrong direction – elements `Car` not in `Vehicle` (`wheels`)

`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`

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();
...
```

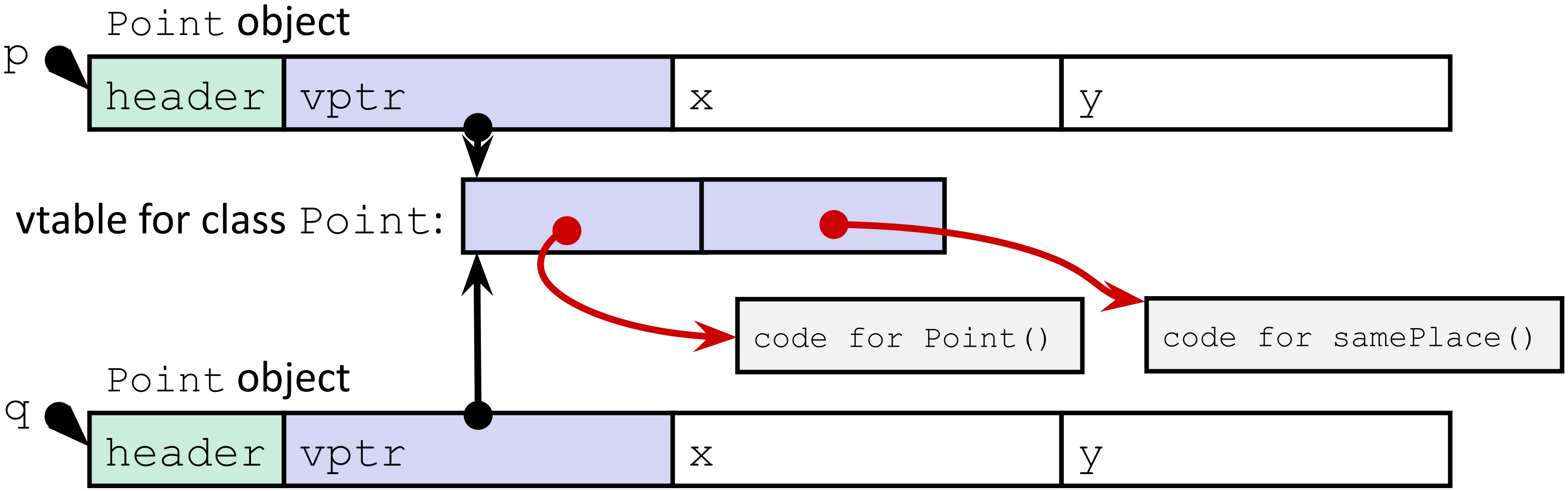
fields

constructor

method(s)

creation

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.

Java Constructors

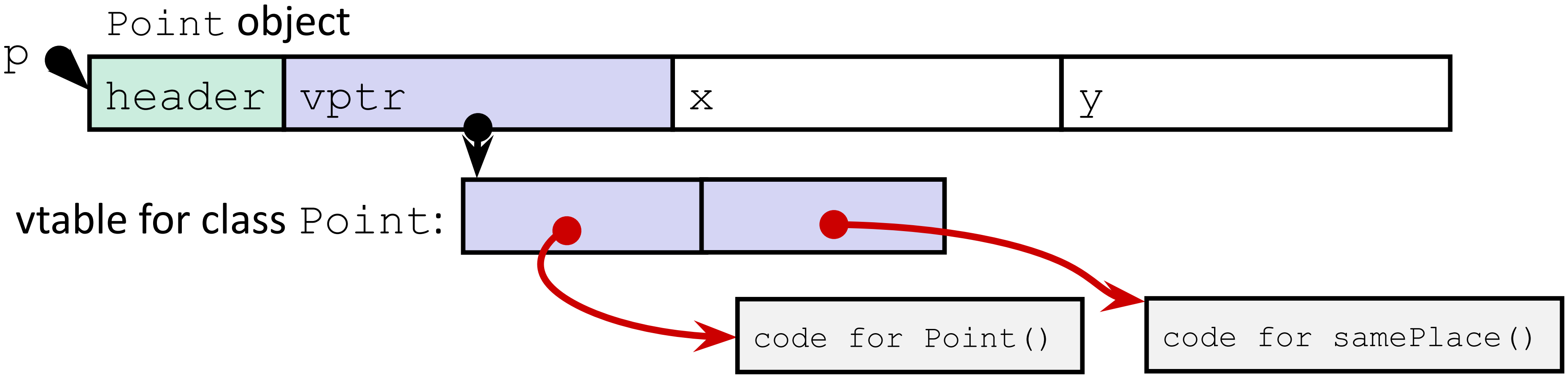
- **When we call `new`:** allocate space for object (data fields and references), initialize to zero, and run constructor

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

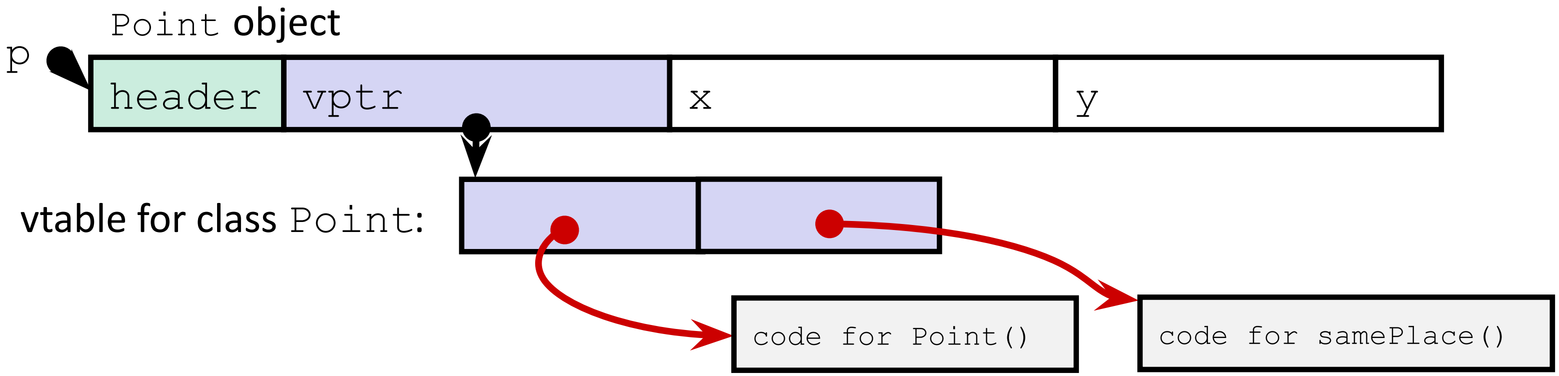
- Static methods are just like functions
- Instance methods:
  - Can refer to *this;*
  - Have an implicit first parameter for *this;* and
  - Can be overridden in subclasses
- The code to run when calling an instance method is chosen *at runtime* by lookup in the vtable

Java:

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

C pseudo-translation:

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



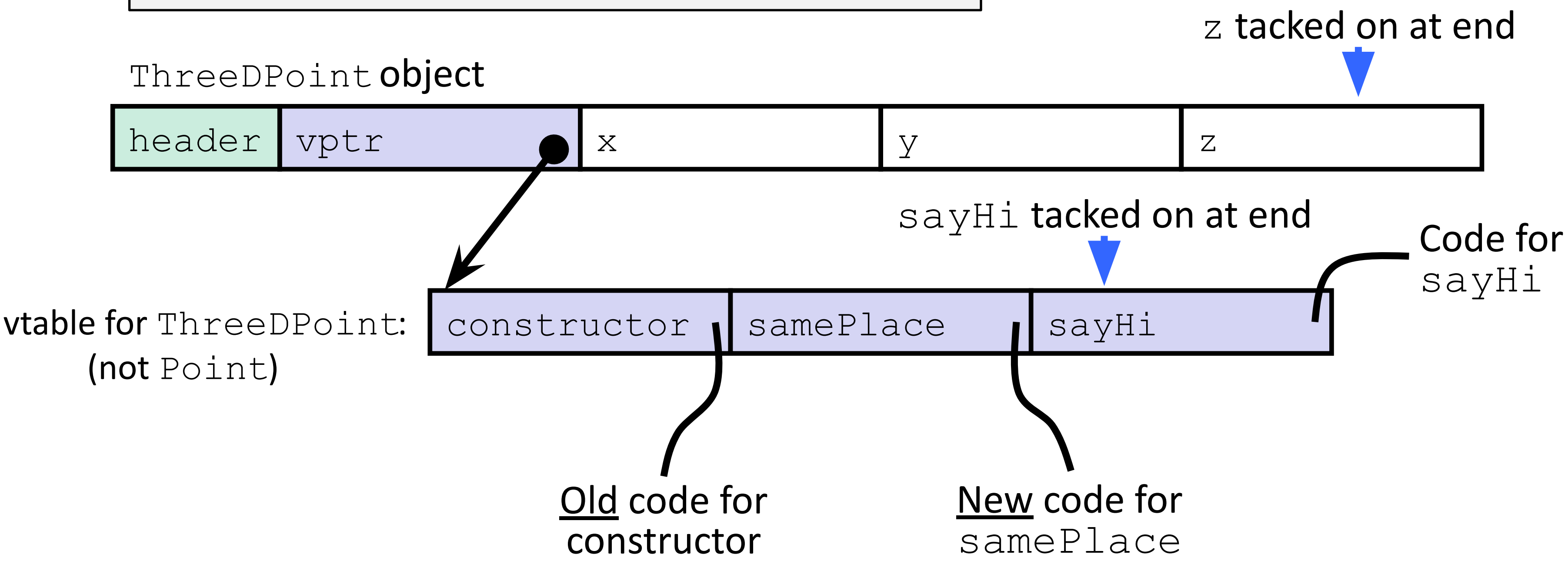
Subclassing

```
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");
    }
}
```



z tacked on at end

ThreeDPoint object

sayHi tacked on at end

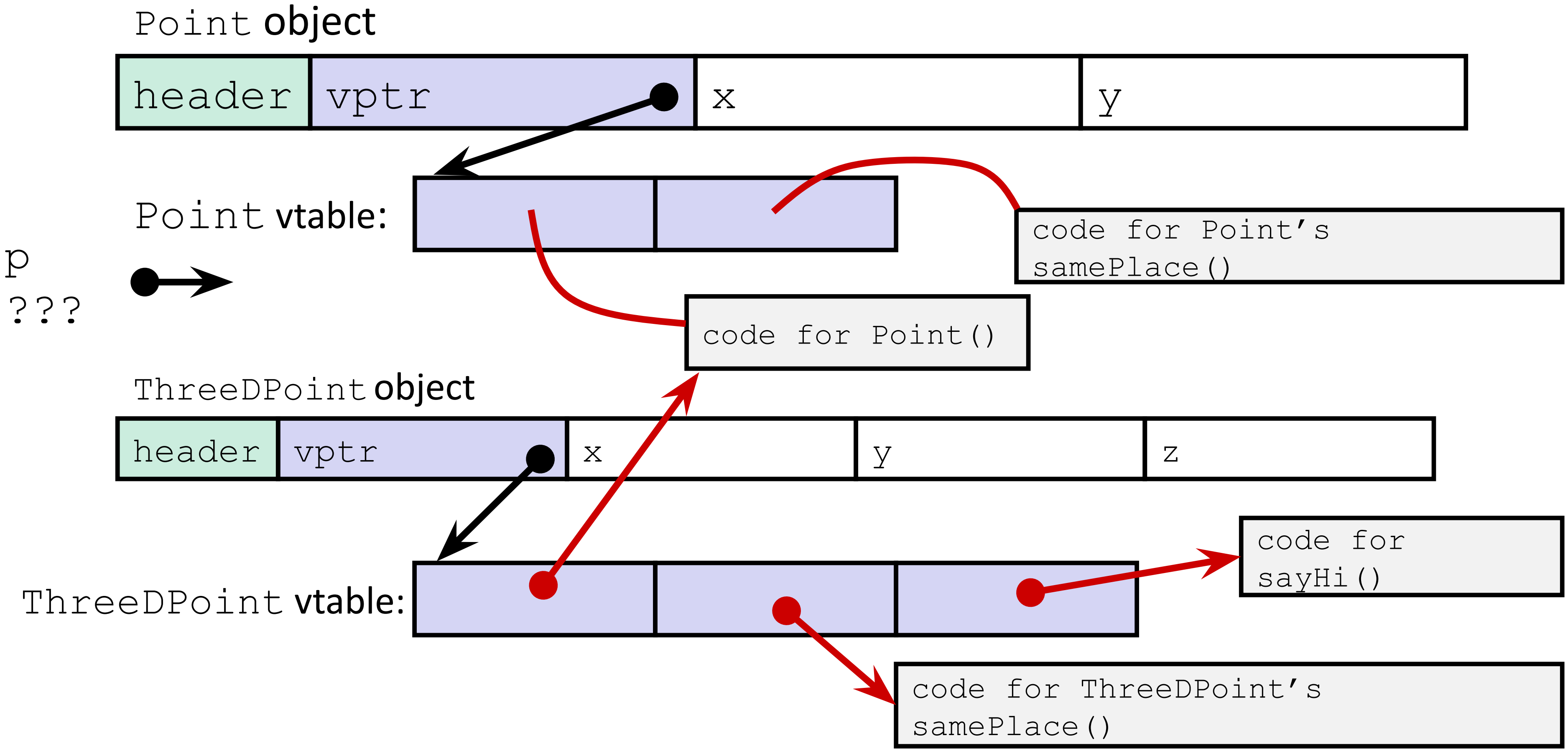
Code for sayHi

vtable for ThreeDPoint:
(not Point)

Old code for constructor

New code for samePlace

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

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
}
```

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

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

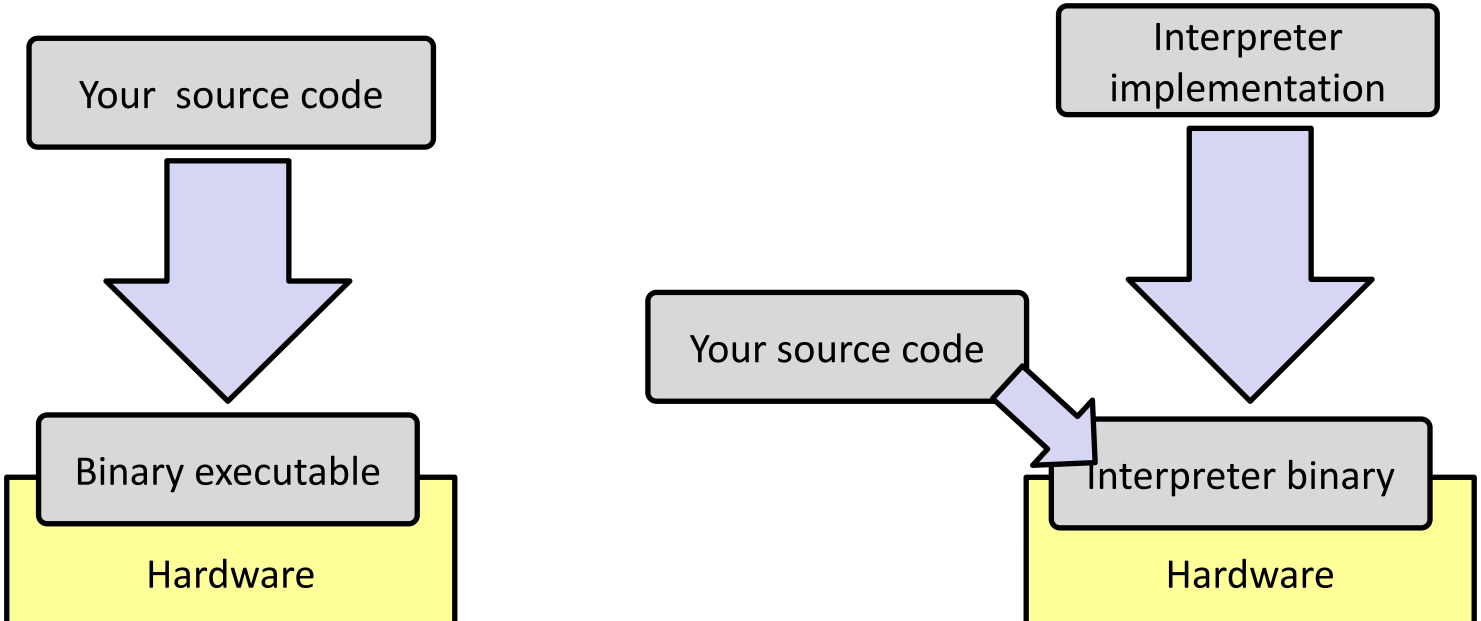
C. header, `Vehicle` vtable ptr, `Car` vtable ptr, `passengers`, `wheels`

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, …


Your source code

Binary executable

Hardware

Interpreter implementation

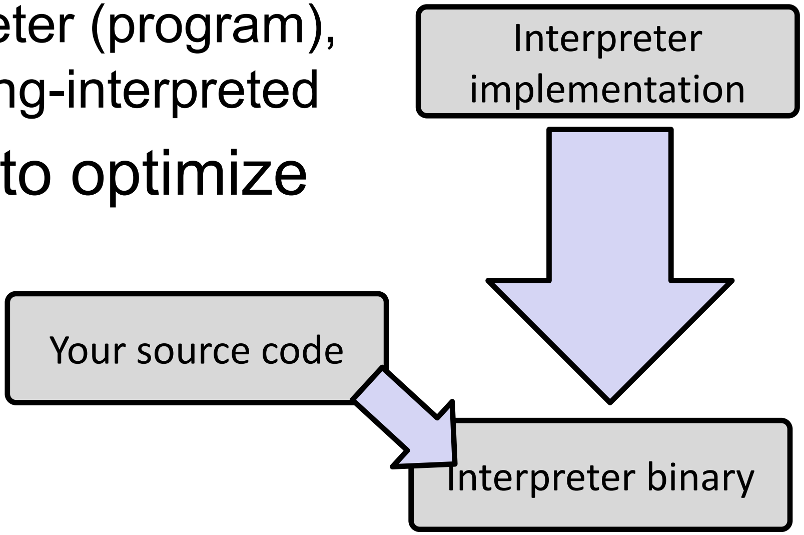
Your source code

Interpreter binary

Hardware

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"

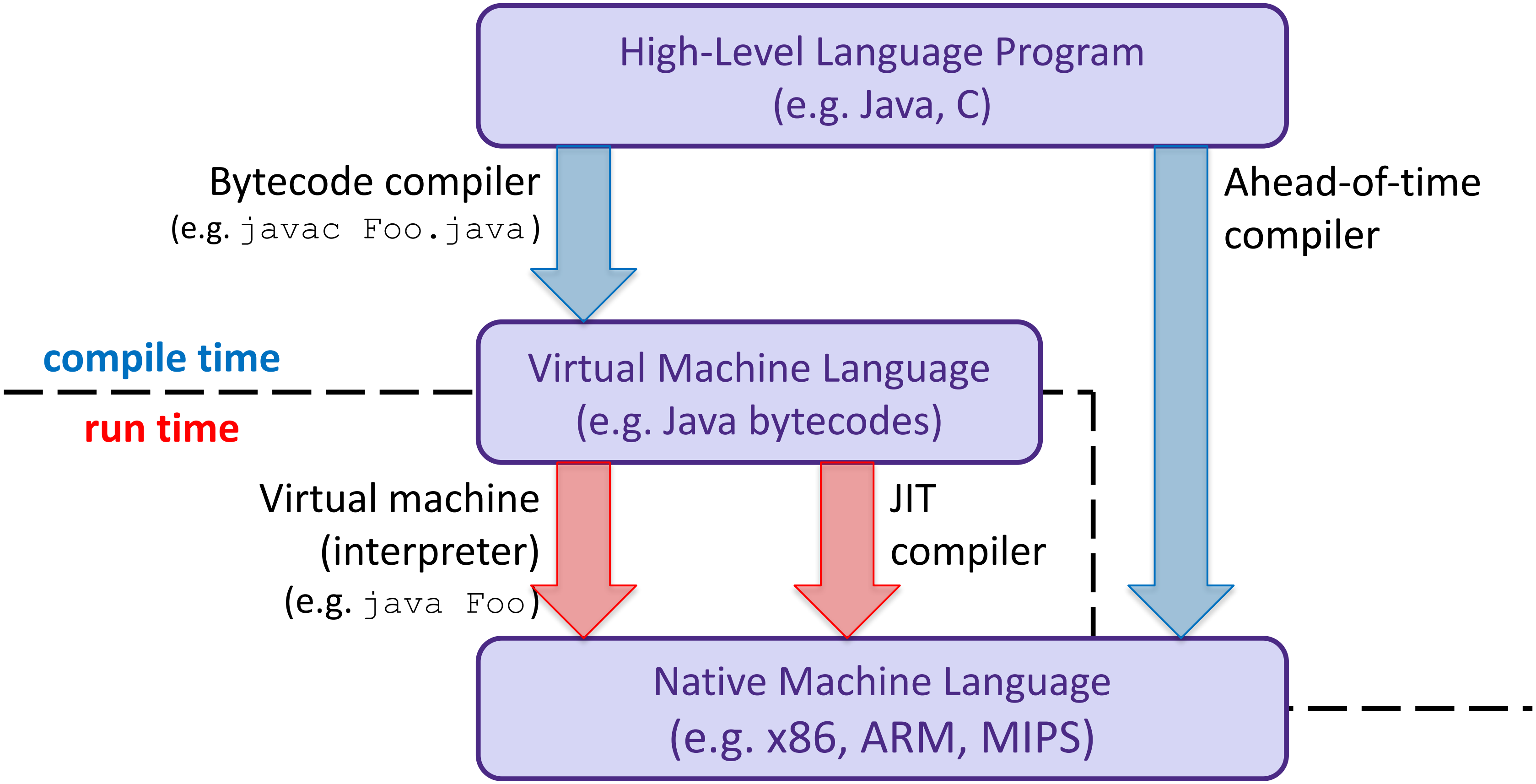
Note: The JVM is different than the CSE VM running on VMWare. Yet *another* use of the word "virtual"!

- 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

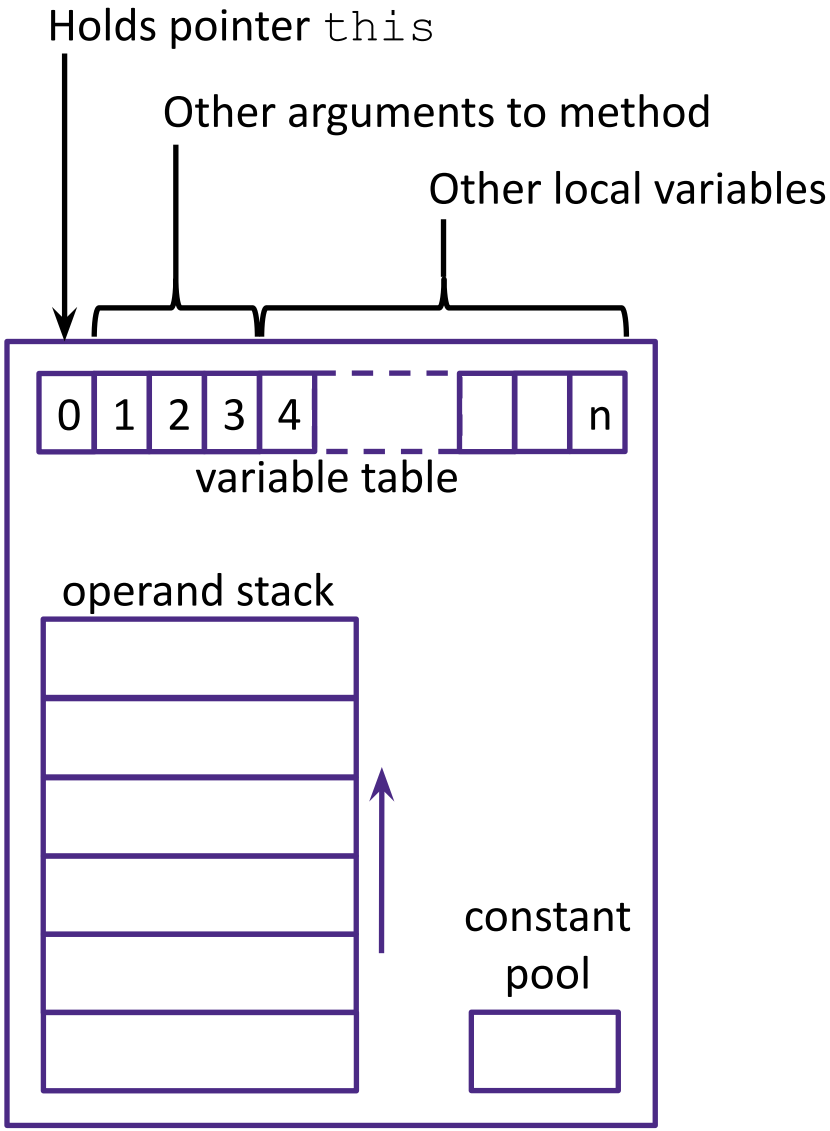
1. Save your Java code in a `.java` file
2. To run the Java compiler:
    - `javac Foo.java`
    - The Java compiler converts Java into *Java bytecodes*
        - Stored in a `.class` file
3. To execute the program stored in the bytecodes, Java bytecodes can be interpreted by a program (an interpreter)
    - 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



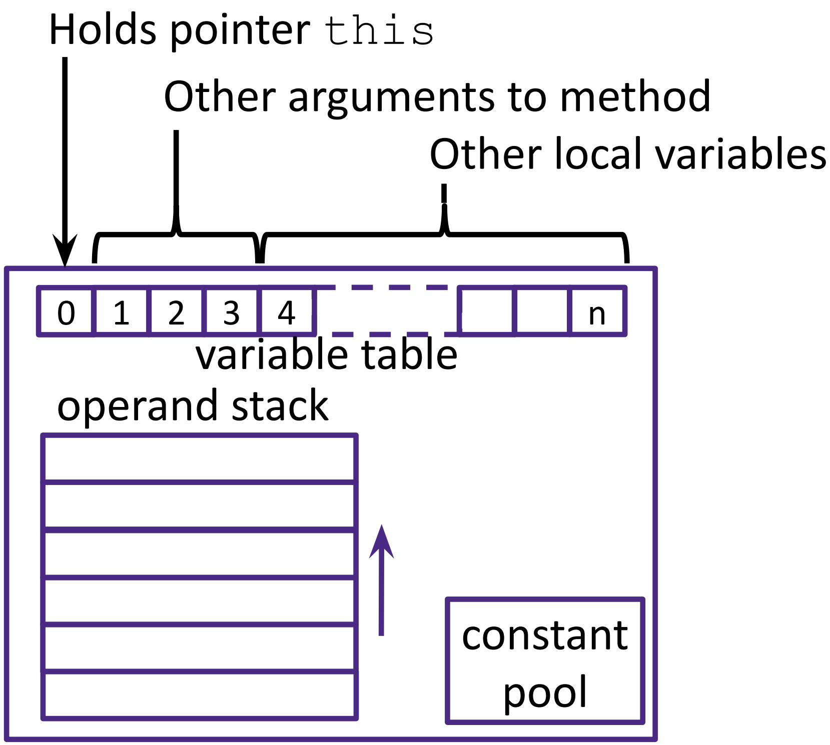
Java Bytecode

- Like assembly code for JVM, but works on *all* JVMs
  - Hardware-independent!
- Typed (unlike x86 assembly)
- Strong JVM protections



JVM Operand Stack

JVM:


Holds pointer `this`

Other arguments to method

Other local variables

‘i’ = integer,
‘a’ = 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 (IA32) 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>
                  // getfield instruction has a 3-byte encoding
                  // 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
```

Byte number:	0	1			4
	aload_0	getfield	00	05	areturn

As stored in the `.class` file:

2A	B4	00	05	B0

http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Class File Format

- Every class in Java code is compiled to its own class file
- 10 sections in the Java class file structure:
  - **Magic number**: 0xCAFEBABE (legible hex)
  - **Version of class file format**: minor & 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 (e.g. 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

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

```
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)
…
```

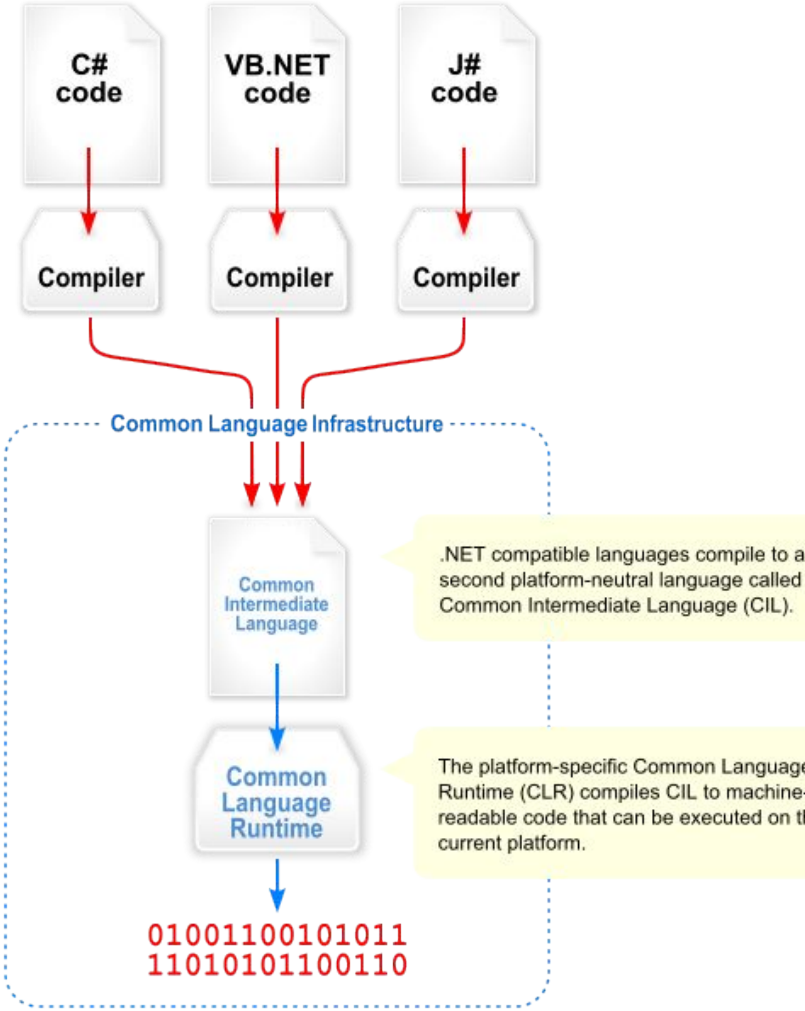
http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

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
  - **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!
- Originally, JVMs were designed and built for Java (still the major use) but JVMs are also viewed as a safe, GC’ed platform

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



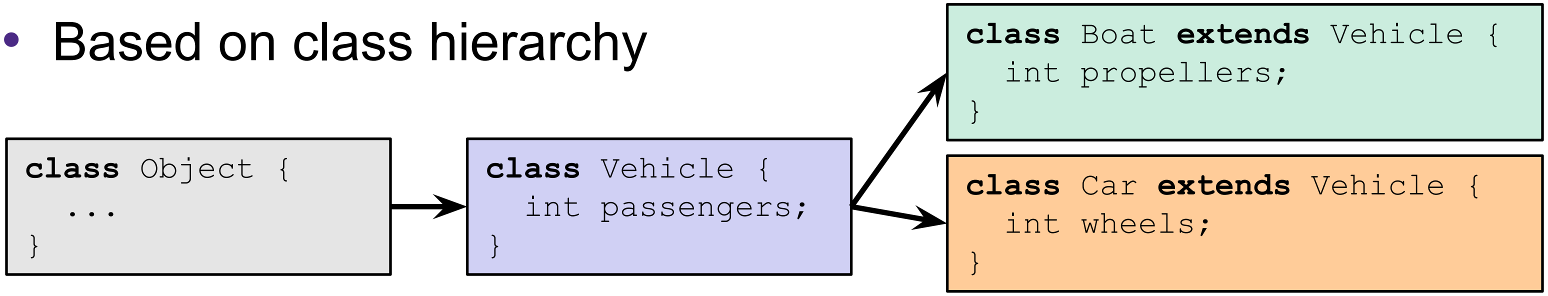
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).

The platform-specific Common Languag Runtime (CLR) compiles CIL to machine-readable code that can be executed on t current platform.

Questions?

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();` ⟵ ✓ 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`