Java and C (part I)

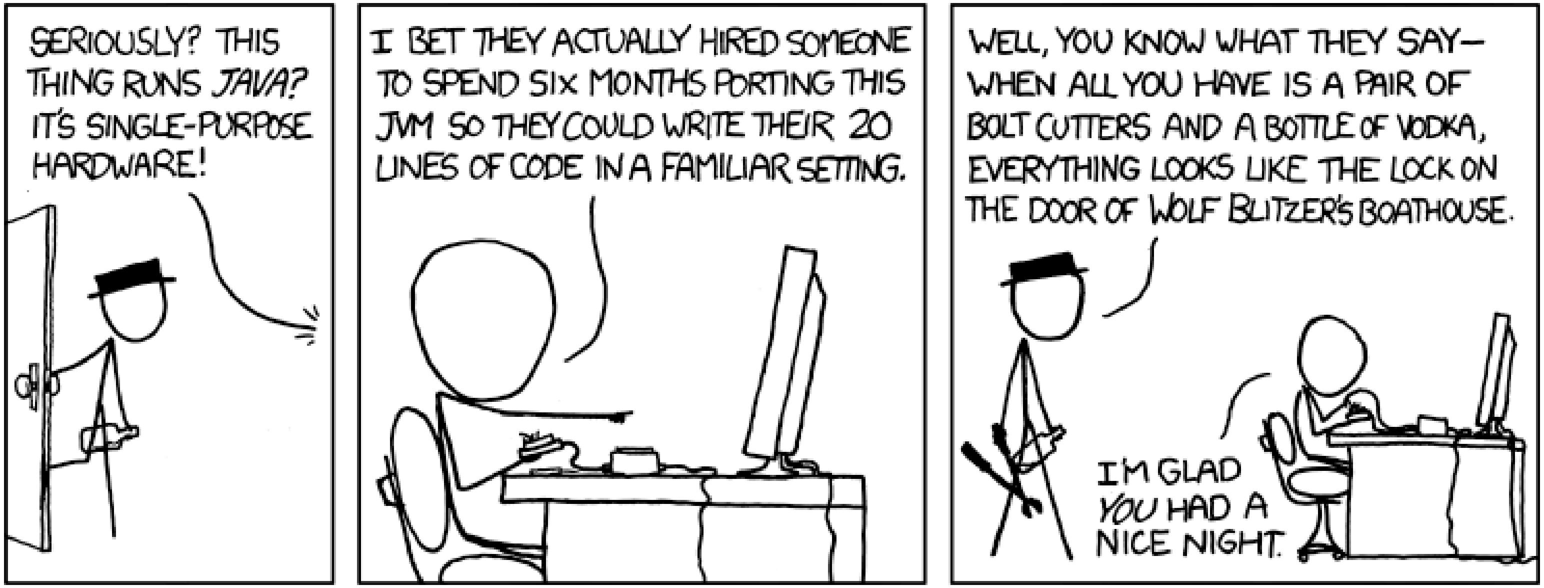
CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael, Connie Chen, Chloe Fong, Chendur Jayavelu, Joshua Tan, Nikolas McNamee, Nahush Shrivatsa, Naama Amiel, Neela Kausik, Renee Ruan, Rubee Zhao, Samantha Dreussi, Sean Siddens, Waleed Yagoub



https://xkcd.com/801/

Relevant Course Information

- HW25 due Monday (12/02) @ 11:59 pm
- HW26 due Wednesday (12/04) @ 11:59 pm
- Lab 5 (on Mem Alloc) due Thurs (12/05) @ 11:59pm
  - 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
  - Light style grading

Lab 5 Tips

- Struct pointers can be used to access field values, even if no struct instances have been created – just reinterpreting the data in memory
- Pay attention to boundary tag data
  - Size value + 2 tag bits – when do these need to be updated and do they have the correct values?
  - The `examine_heap` function follows the implicit free list searching algorithm – don't take its output as "truth"
- Learn to use and interpret the trace files for testing!!!
- A special heap block marks the end of the heap

Roadmap

1970's 1990's

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();
```

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

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

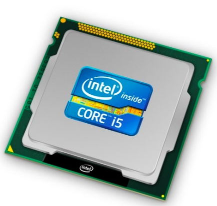
Machine code:

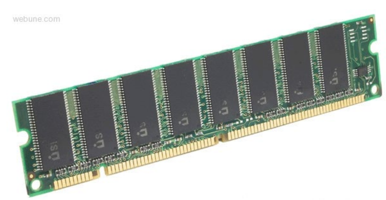
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10 OS X Yosemite

Computer system:





Java (1995) vs. C (1972)

- Reconnecting to Java (hello CSE12x/CSE14x!)
  - 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 CSE12x/CSE14x 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

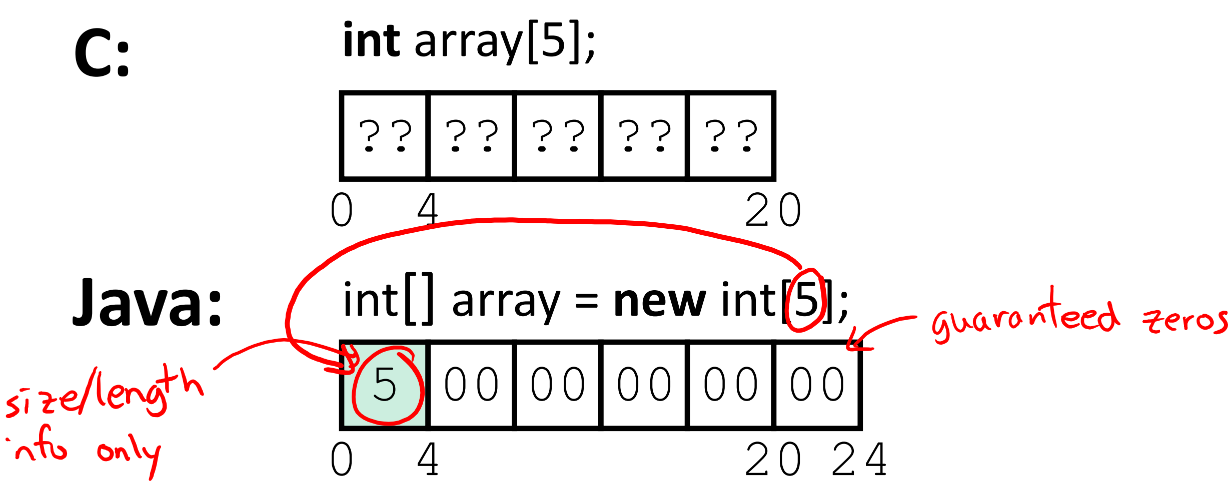
- In fact, the language simply provides an ***abstraction*** (Java language specification)
  - Tells us how code should behave for different language constructs, but we can't easily tell how things are really represented
  - But it is important to understand an ***implementation*** of the lower levels – 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"
- Much more interesting:
  - **Arrays**
  - **Characters and strings**
  - **Objects**

Data in Java: Arrays

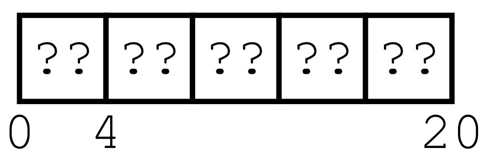
- Every element initialized to `0` or `null`
- Length specified in immutable field at start of array (`int`: 4B)
  - `array.length` returns value of this field
- *Since it has this info, what can it do?*



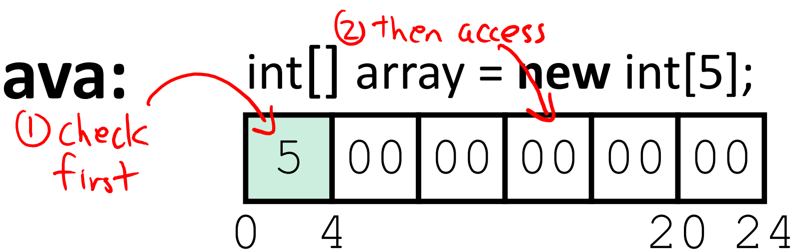
Data in Java: Arrays

- Every element initialized to `0` or `null`
- Length specified in immutable field at start of array (`int`: 4B)
  - `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];



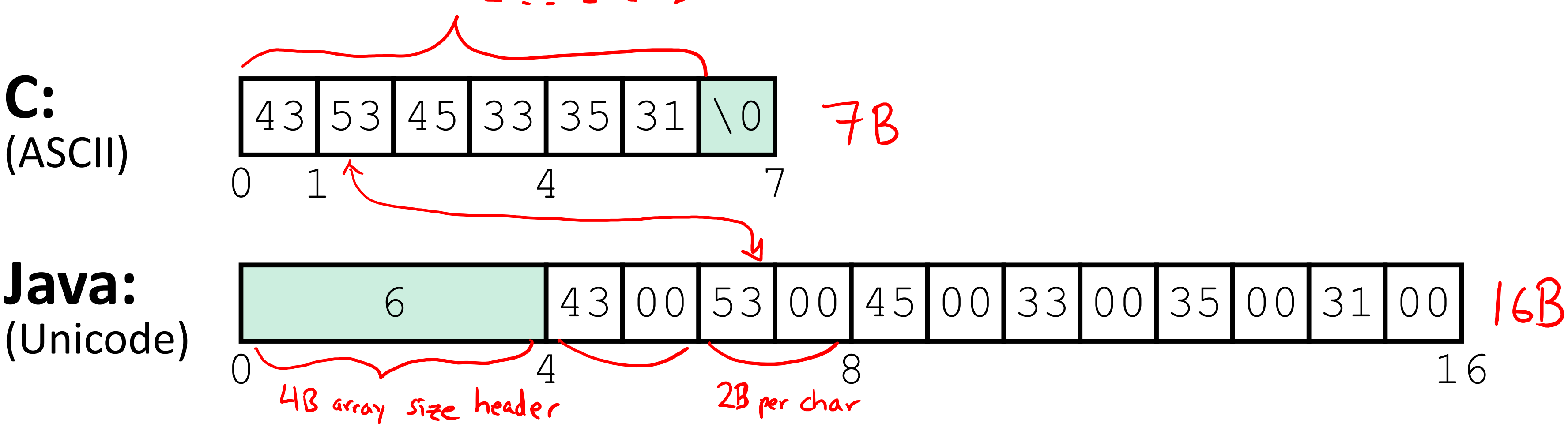
To speed up bounds-checking:

- 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"



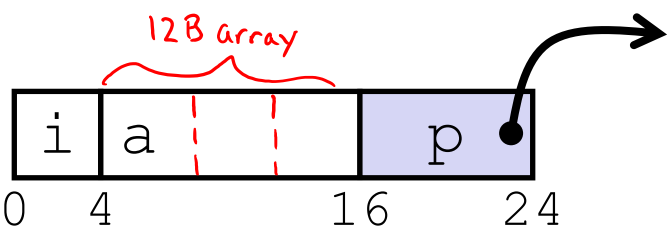
Data in Java: Objects

- 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

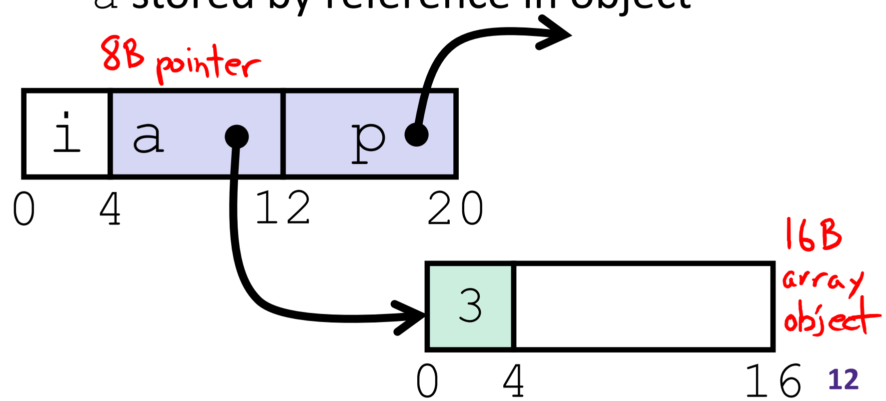


Java:

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

methods

- `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;
```

r2.i

// (*r).i

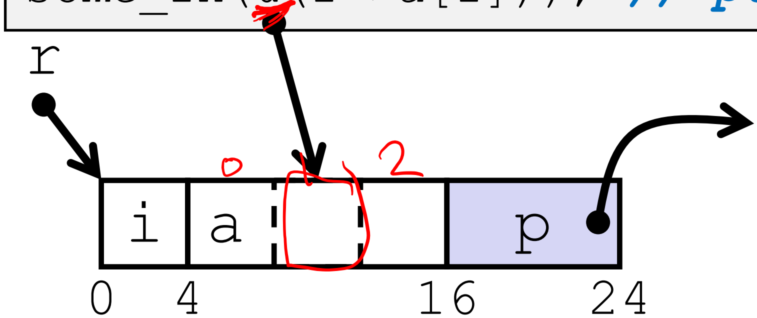
references

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

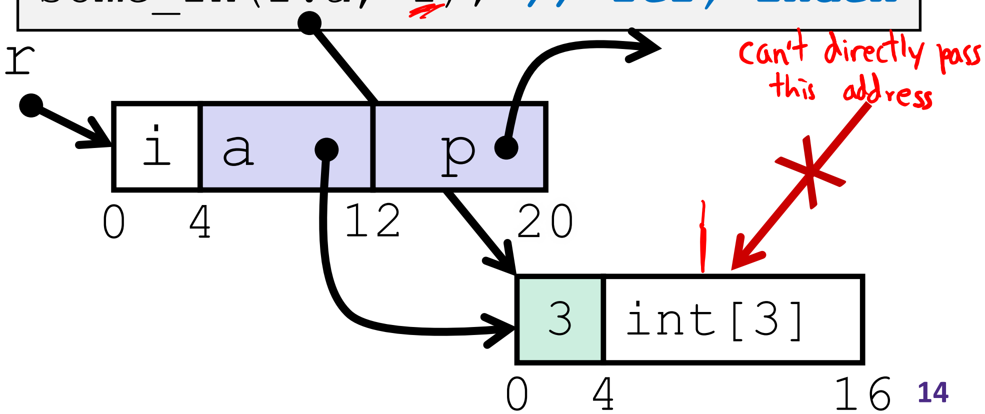
C:

```
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)

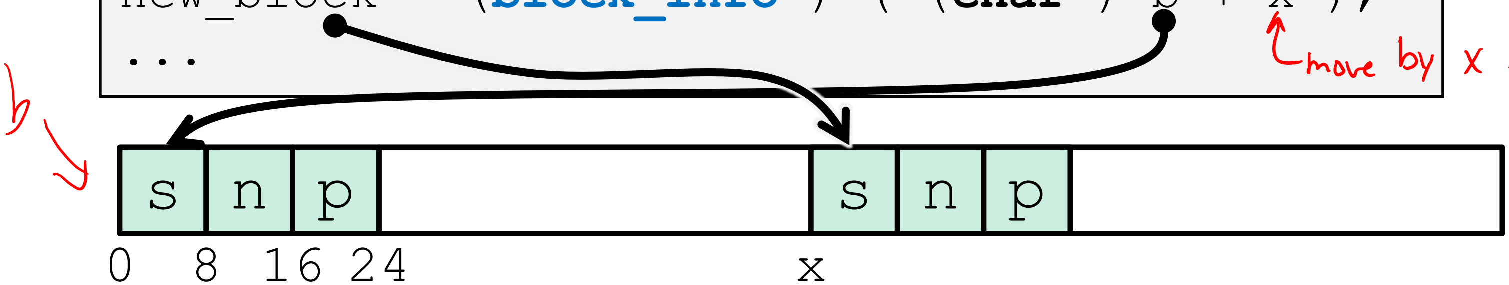
- Can cast any pointer into any other pointer
  - Changes dereference and arithmetic behavior

```
struct block_info {
        size_t size_and_tags;
        struct block_info* next;
        struct block_info* prev;
};
typedef struct block_info block_info;
...
int x;
block_info* b;
block_info* new_block;
...
new_block = (block_info*) ( (char*) b + x );
...
```

Cast b into char* to do unscaled addition

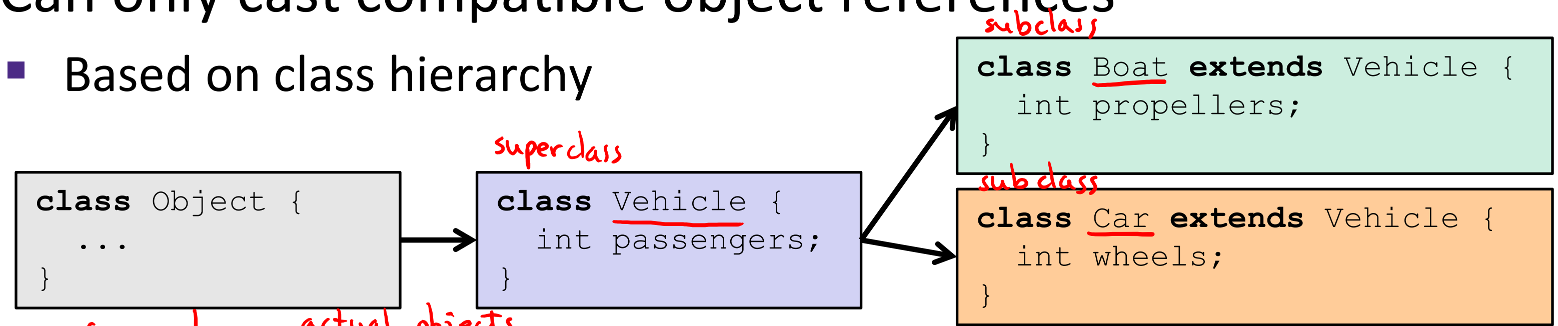
Cast back into block_info* to use as block_info struct

move by x bytes



Type-safe casting in Java

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

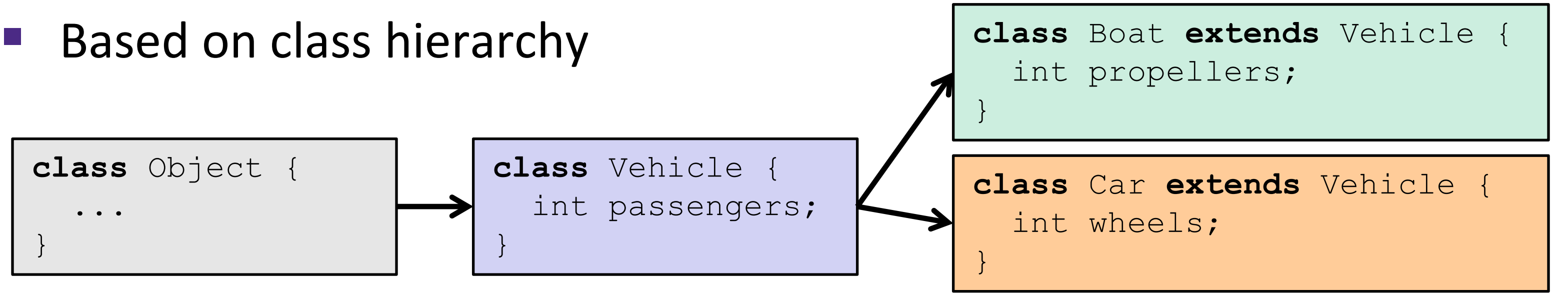


references! actual objects

```
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




```
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;
```

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;
}
```

defines the user's interface with object — actual object that you interact with

```
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;
```

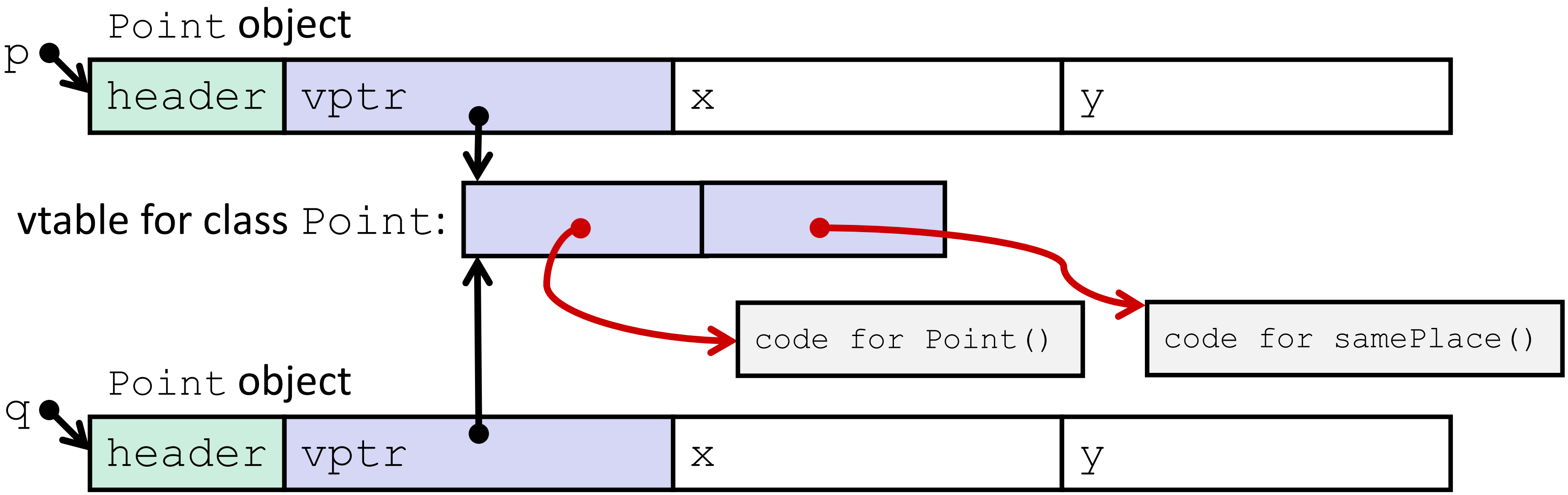
- `Vehicle v1 = new Car();` ← ✓ Everything needed for `Vehicle` also in `Car`
- `Vehicle v2 = v1;` ← ✓ `v1` is declared as type `Vehicle`
- `Car c2 = new Boat();` ← ✗ Compiler error: Incompatible type – elements in `Car` that are not in `Boat` (siblings)
- `Car c3 = new Vehicle();` ← ✗ Compiler error: 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`

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



- *Object header* : GC info, hashing info, lock info, etc.
- *Virtual method table (vtable)*
  - Like a **jump table** for instance ("virtual") methods plus other class info
  - Only one table per class
  - Each object instance contains a *vtable pointer (vptr)*

Java Constructors

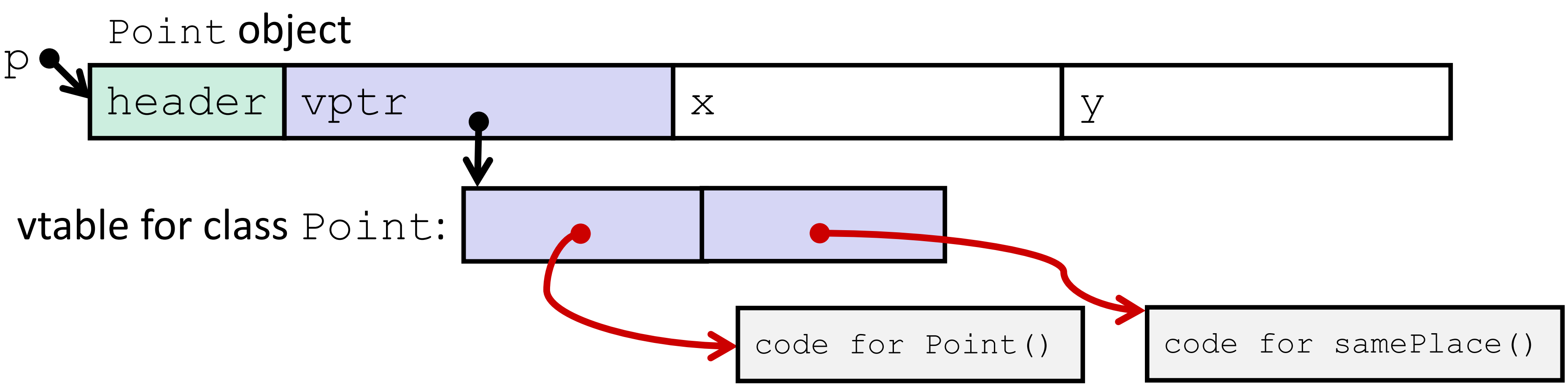
- **When we call `new`:** allocate space for object (data fields and references), initialize to zero/null, and run constructor method

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

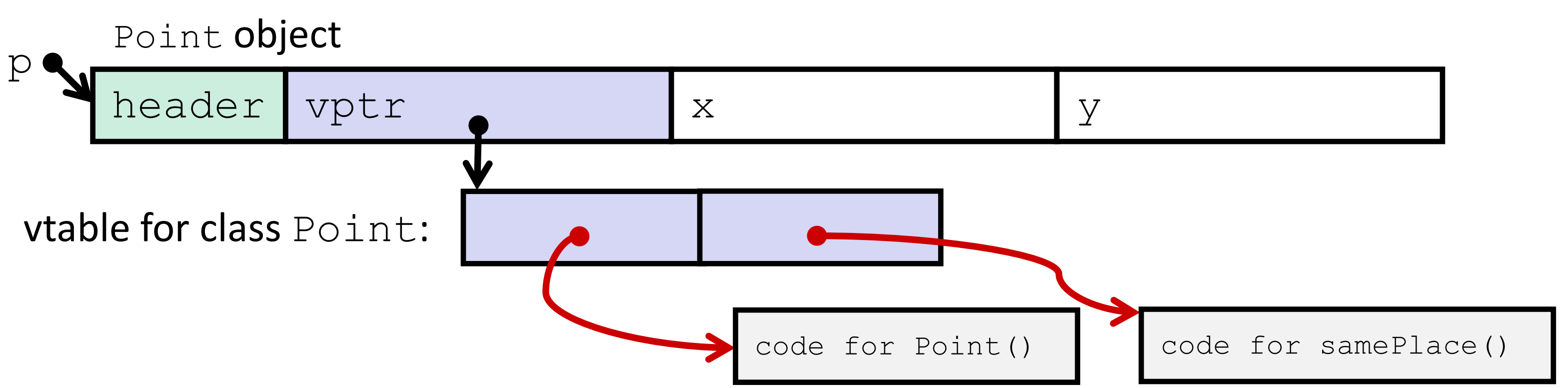
- Static methods are just like functions
- Instance methods:
  - Can refer to *this;*
  - Have an implicit first parameter for *this;* and
  - Can be overridden in subclasses
- The code to run when calling an instance method is chosen *at runtime* by lookup in the vtable

Java:

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

C pseudo-translation:

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



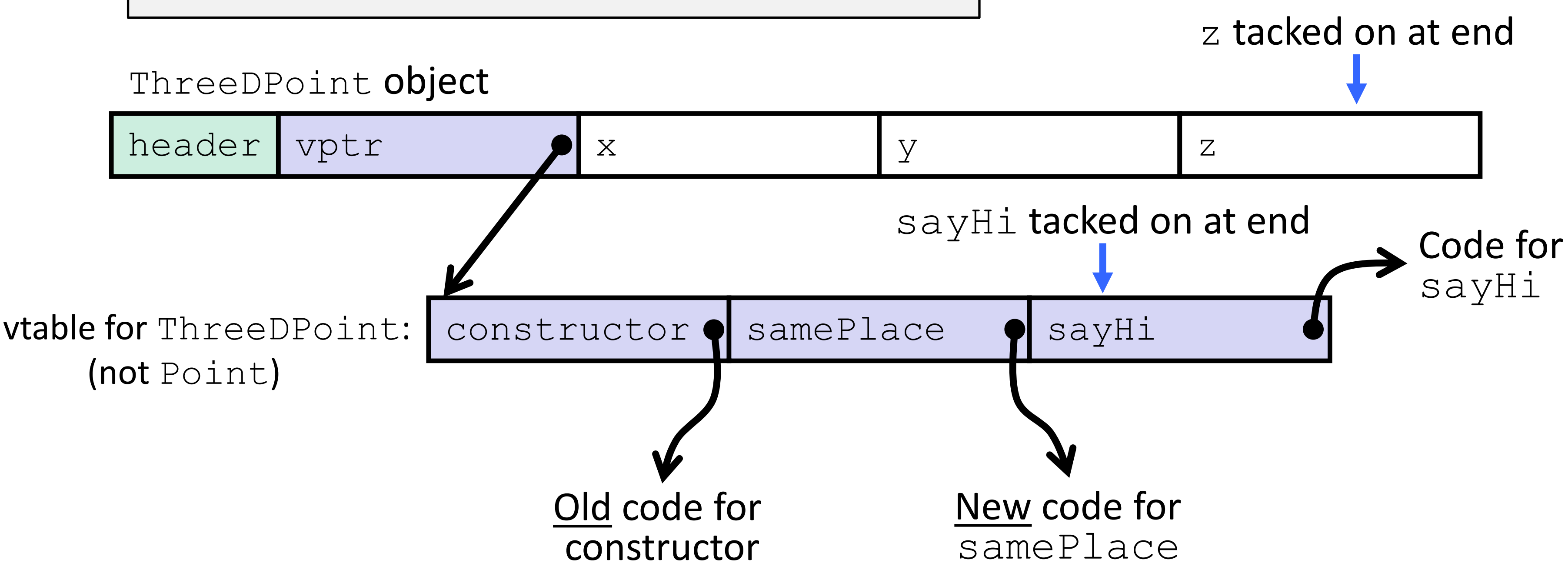
Subclassing

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

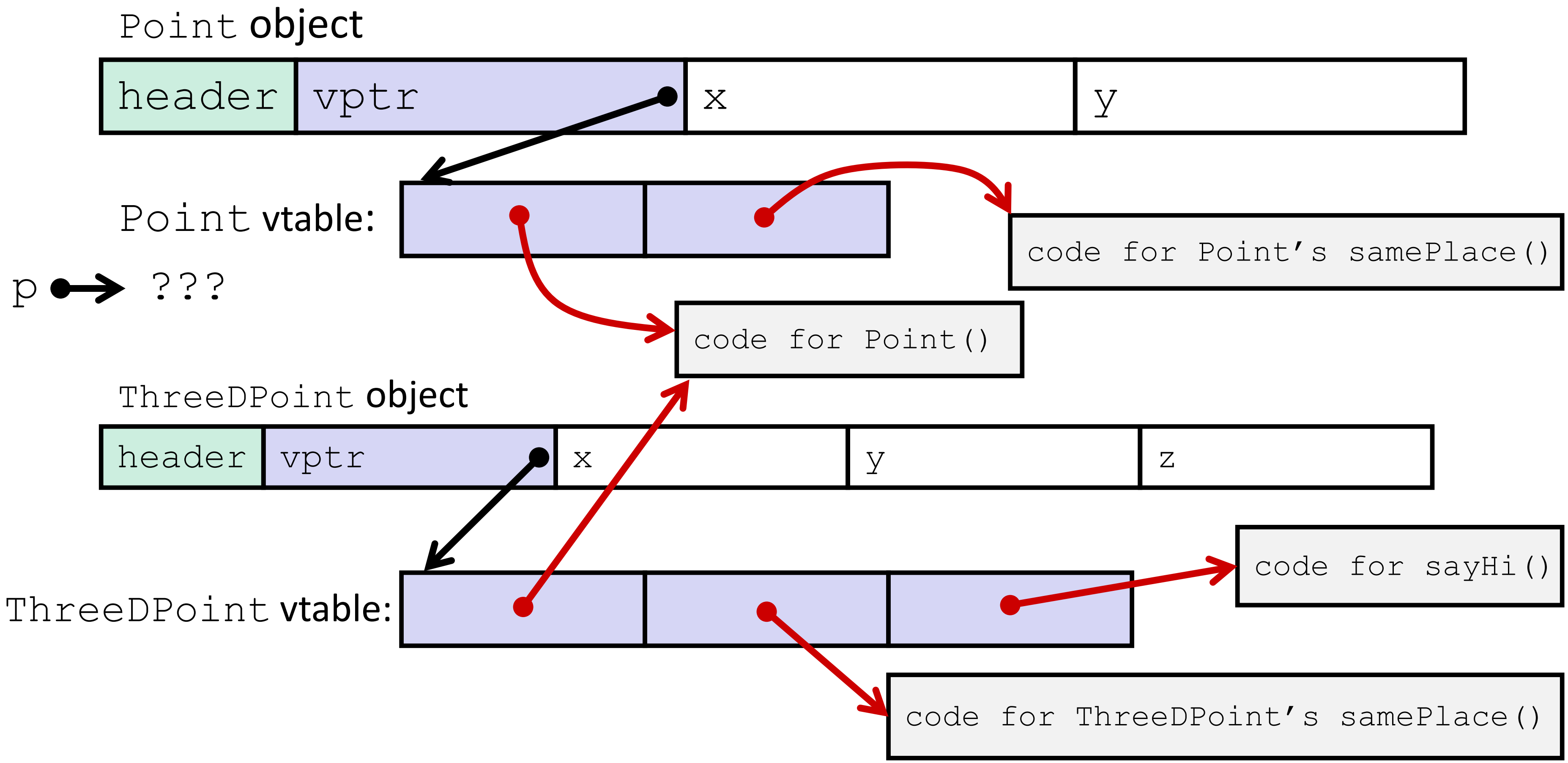
- Where does "`z`" go? At end of fields of `Point`
  - `Point` fields are always in the same place, so `Point` code can run on `ThreeDPoint` objects without modification
- Where does pointer to code for two new methods go?
  - No constructor, so use default `Point` constructor
  - To override "`samePlace`",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

Subclassing: Object Layout

```
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);
```

Inheritance and Overriding Methods

- In CSE12x/CSE14x, it may have seemed "magic" that an *inherited* method could call an *overridden* method

- 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