

# Java and C

CSE 351 Autumn 2019

## Instructor:

Justin Hsia

## Teaching Assistants:

Andrew Hu

Antonio Castelli

Cosmo Wang

Diya Joy

Ivy Yu

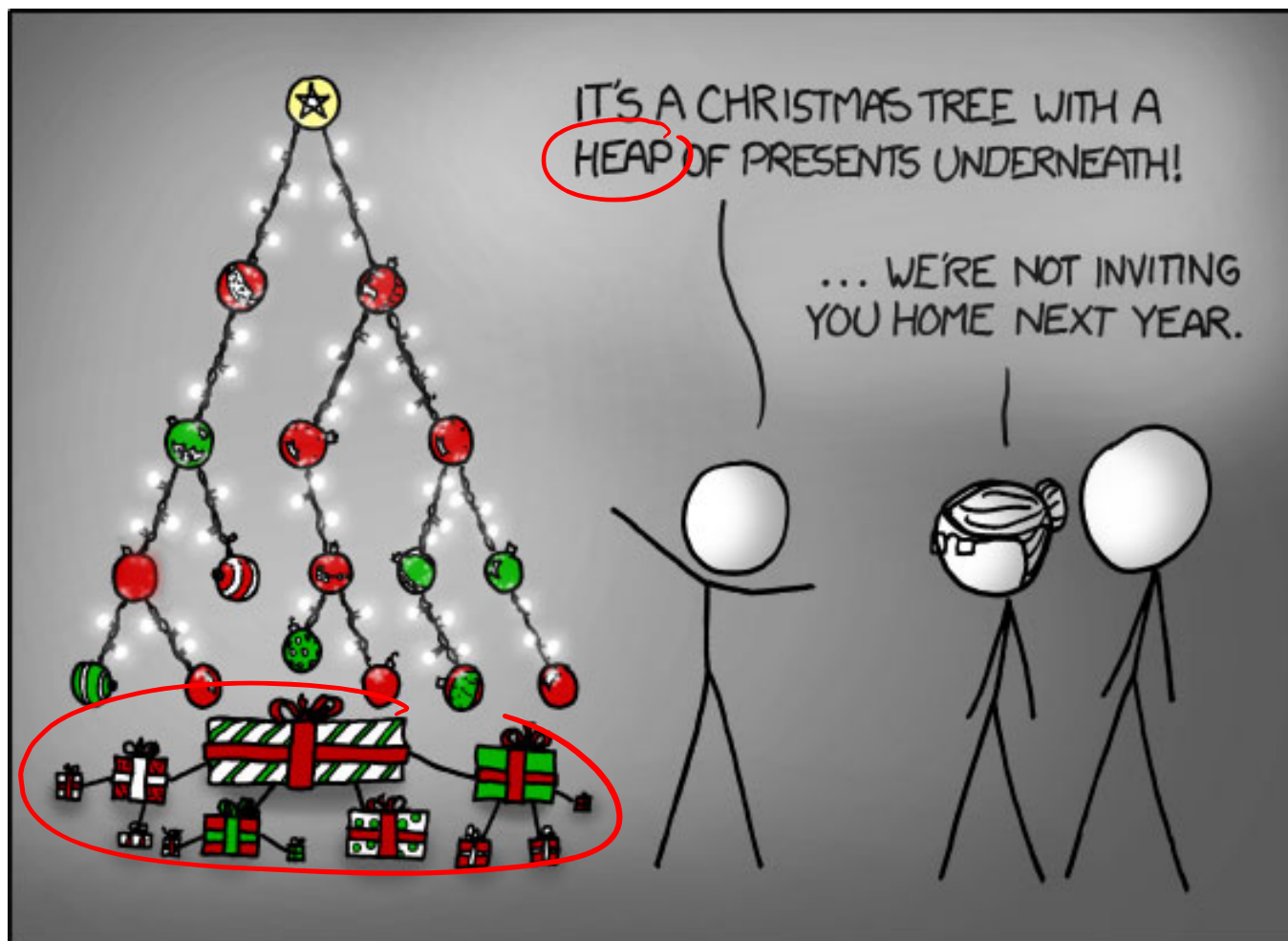
Kaelin Laundry

Maurice Montag

Melissa Birchfield

Millicent Li

Suraj Jagadeesh



<https://xkcd.com/835/>

# Administrivia

- ❖ Lab 5 due Friday (12/6)
  - **Hard deadline on Sunday (12/8) – 1 late day**
- ❖ Course evaluations now open
  - See Piazza post @597 for links (separate for Lec and Sec)
- ❖ **Final Exam:** Tue, 12/10, 12:30-2:20 pm in KNE 120
  - Review Session: Sun, 12/8, 3:30 – 6 pm in SAV 260
  - You get TWO double-sided handwritten 8.5×11” cheat sheets

# 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();
```

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

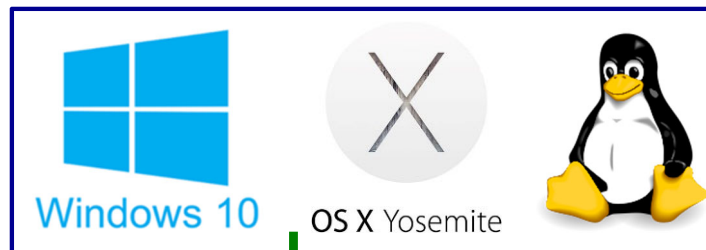
Assembly language:

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

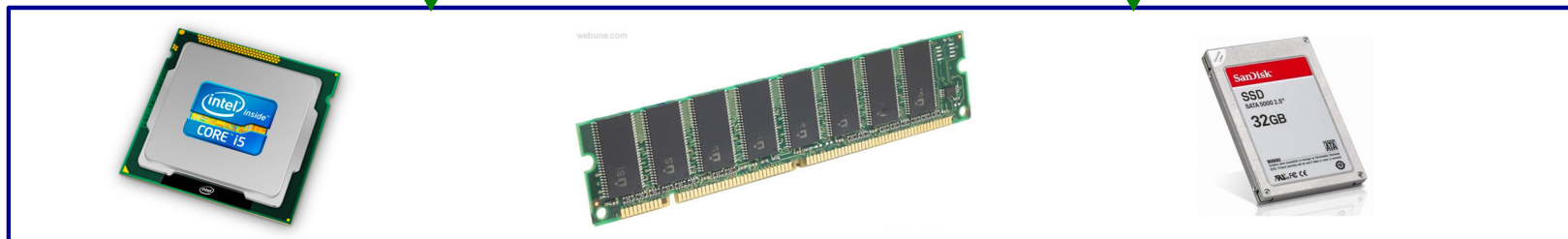
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



# 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

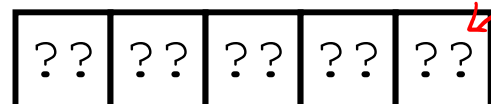
# Data in Java

- ❖ 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”
- ❖ 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];`

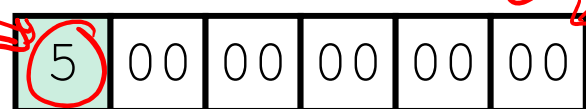


0 4 20

garbage!

**Java:**

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



0 4 20 24

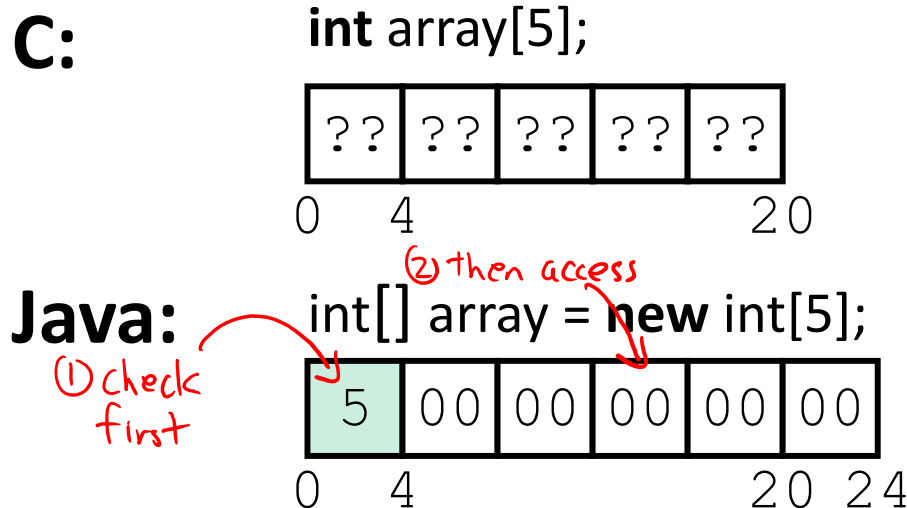
guaranteed zeros

size/length info only



# 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 bounds-check
  - Code is added to ensure the index is within bounds
  - Exception if out-of-bounds



## To speed up bounds-checking:

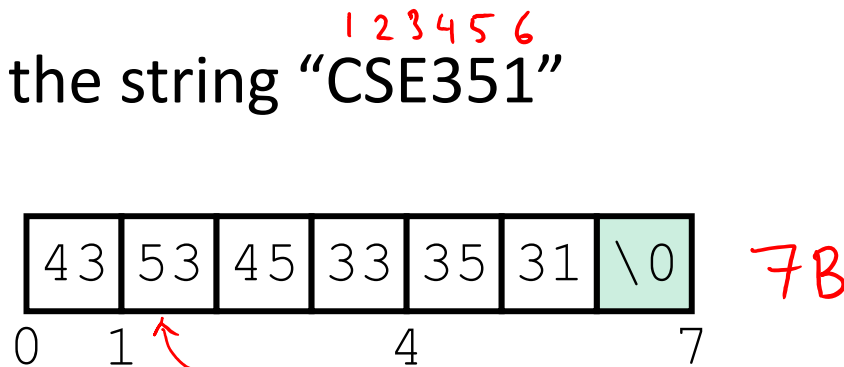
- Length field is likely in cache
- Compiler may store length field in register for loops
- Compiler may prove that some checks are redundant

# Data in Java: Characters & Strings

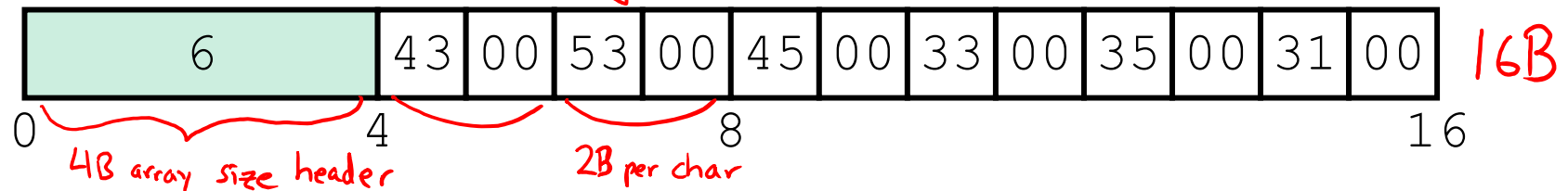
- ❖ Two-byte Unicode instead of ASCII
  - Represents most of the world's alphabets
- ❖ String not bounded by a ' \0 ' (null character)
  - Bounded by hidden length field at beginning of string
- ❖ All String objects read-only (vs. StringBuffer)

Example: the string "CSE351"

**C:**  
(ASCII)



**Java:**  
(Unicode)



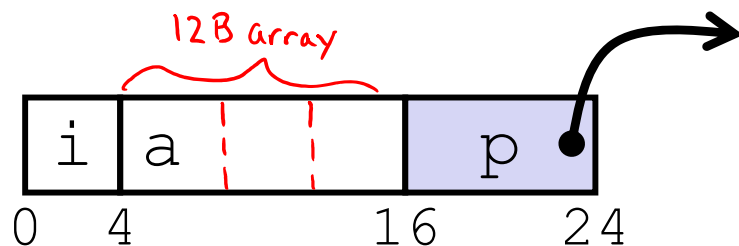
# Data in Java: Objects

- ❖ 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;
};
```

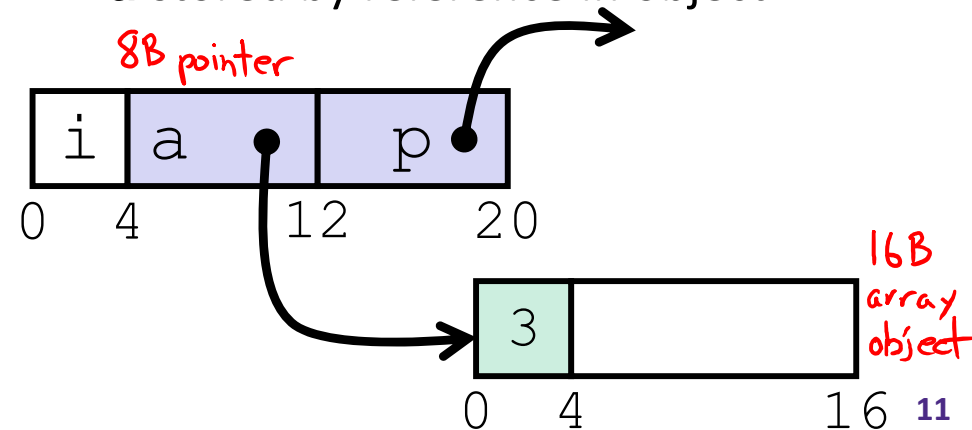
- a[] stored “inline” as part of struct



## 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  $r \rightarrow a$
- ❖ 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  $r \rightarrow a$  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;
```

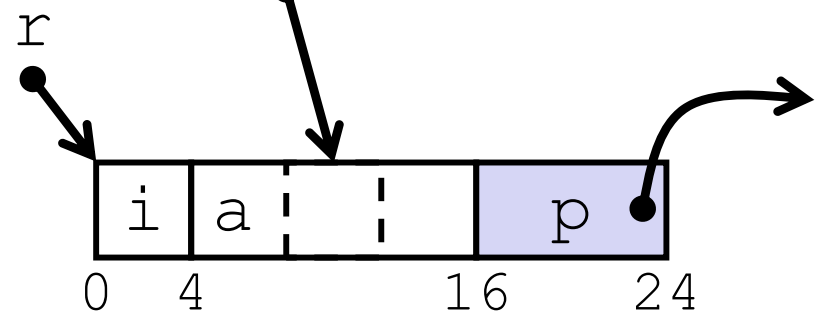
references

# 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

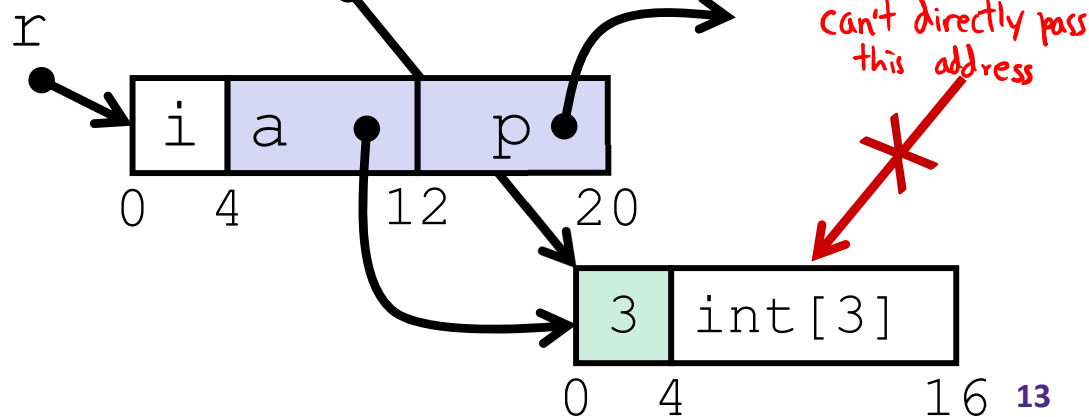
**C:**

```
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

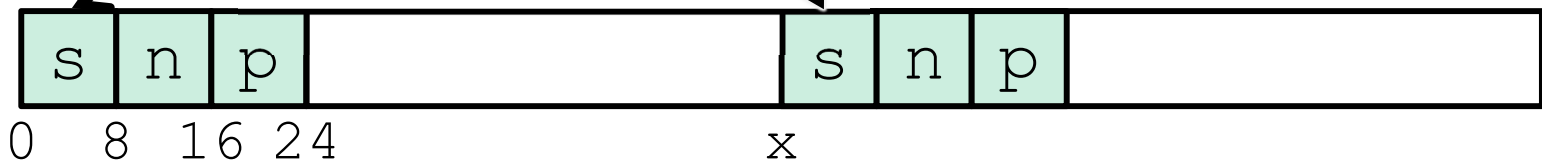
```

struct 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

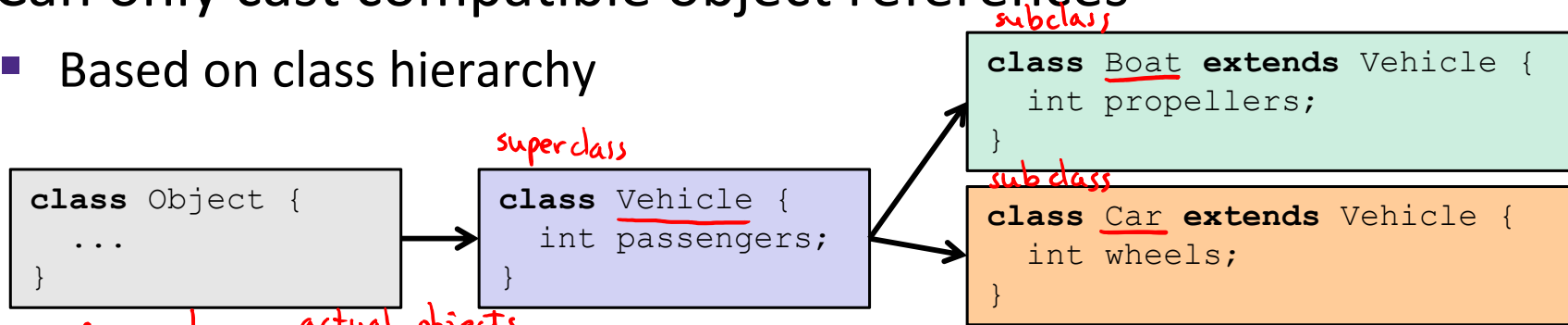
move by x bytes



# 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;
    
```

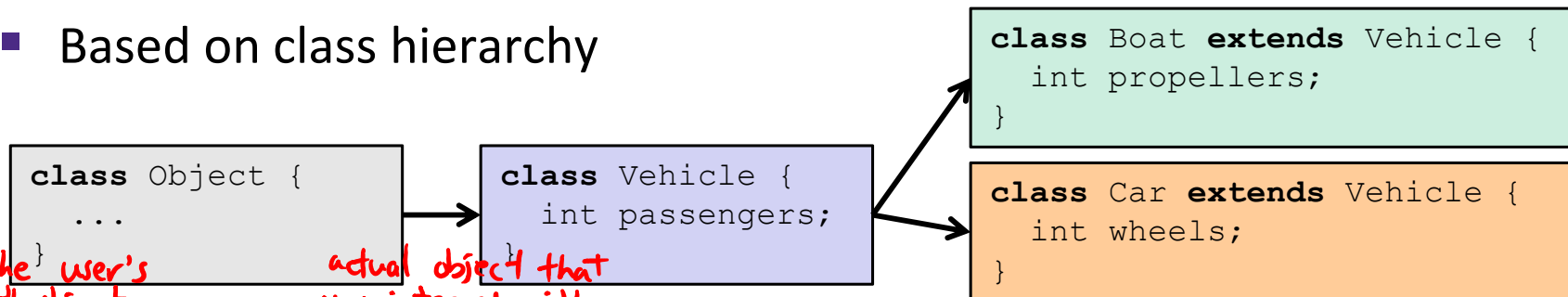
*Handwritten annotations:*

- references!* (circled in red) pointing to the variable names: v, b1, c1, v1, v2, c2, c3, b2, c4, c5.
- actual objects* (circled in red) pointing to the object creation expressions: new Vehicle(), new Boat(), new Car(), new Car(), v1, new Boat(), new Vehicle(), (Boat) v, (Car) v2, (Car) b1.

# Type-safe casting in Java

❖ Can only cast compatible object references

- Based on class hierarchy



*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(); // ✓ 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)
what if: Boat v = new Boat();
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 ();  
...
```

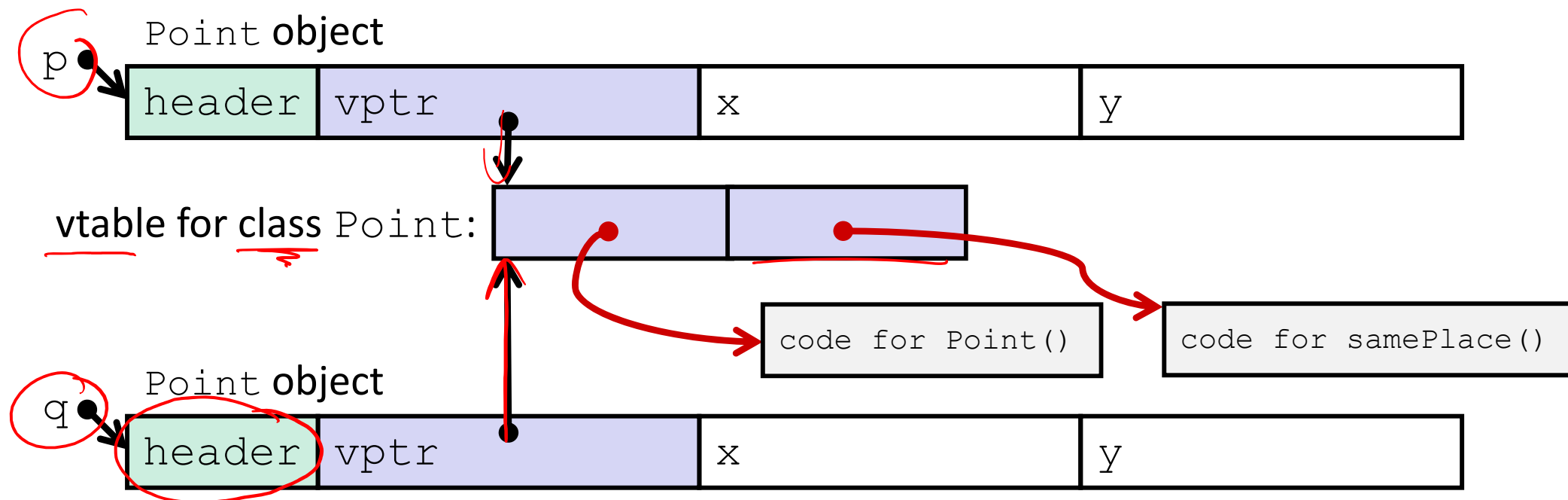
fields

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 *virtual 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/null, and run constructor method

**Java:**

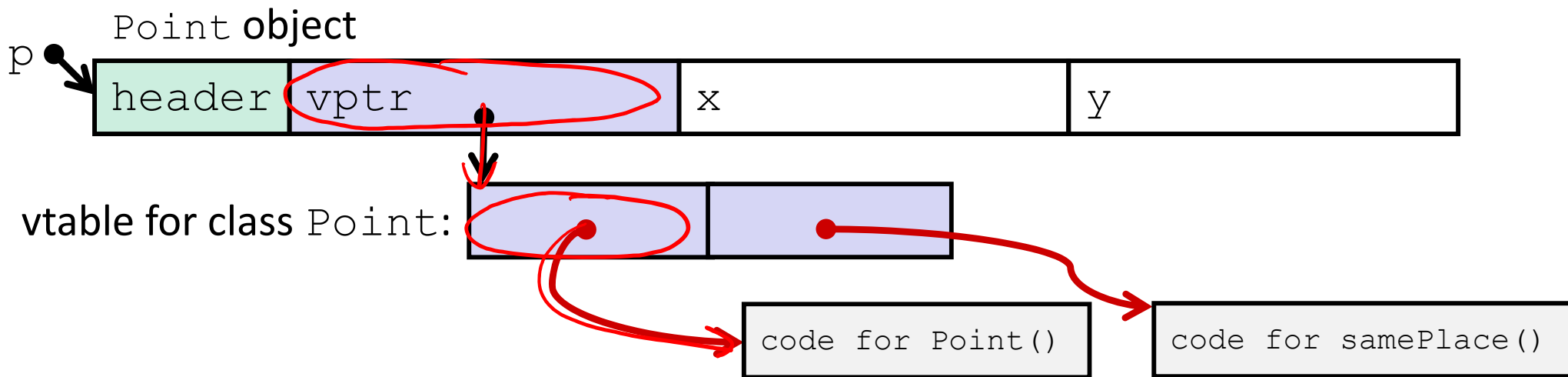
```
Point p = new Point();
```

**C pseudo-translation:**

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

*Zero out object data*

*run the constructor*



# Java Methods

- ❖ Static methods are just like functions
- ❖ Instance methods:
  - Can refer to *this*; reference to particular instance of class
  - 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 (i.e. dispatch)

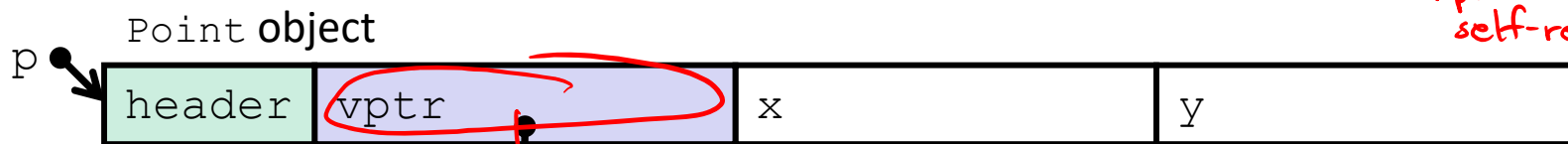
**Java:**

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

**C pseudo-translation:**

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

*implicit object self-reference*



vtable for class Point:



```
code for Point()
```

```
code for samePlace()
```

# Subclassing

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

← new field  
} override method  
} new method

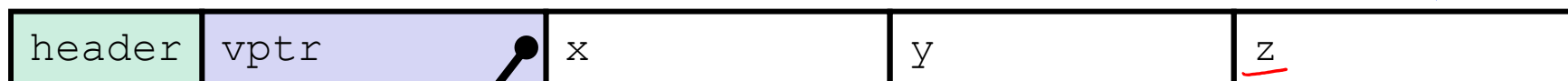
- ❖ Where does “z” go? At end of fields of `Point`
  - `Point` fields are always in the same place, so `Point` code can run on `ThreeDPoint` objects without modification
- ❖ Where does pointer to code for two new methods go?
  - No constructor, so use default `Point` constructor
  - To override “`samePlace`”,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

# Subclassing

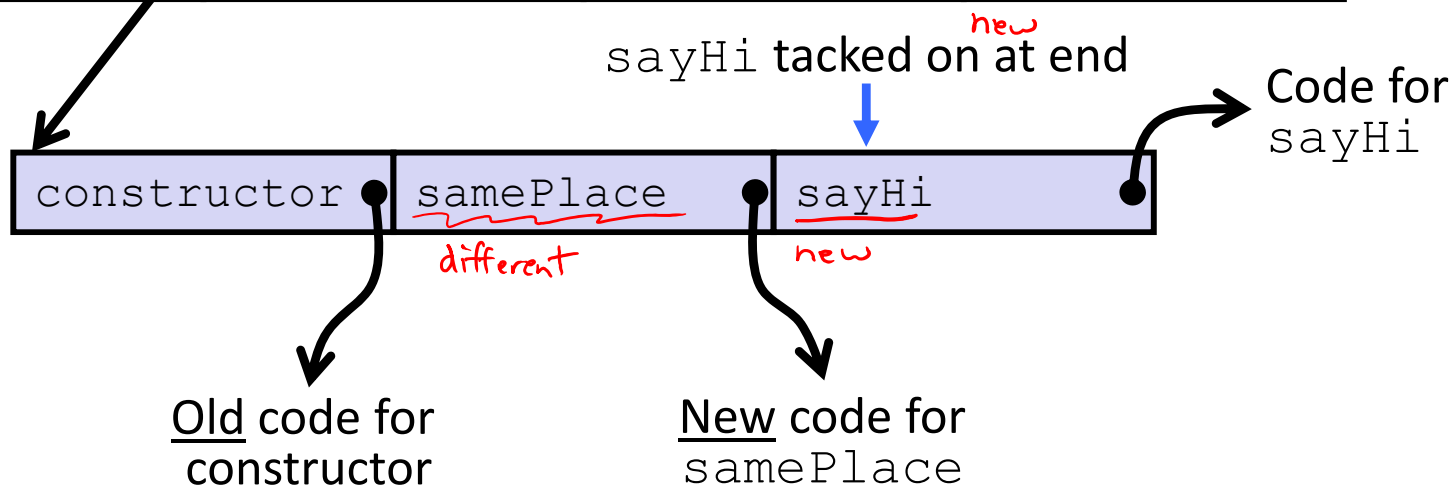
```

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

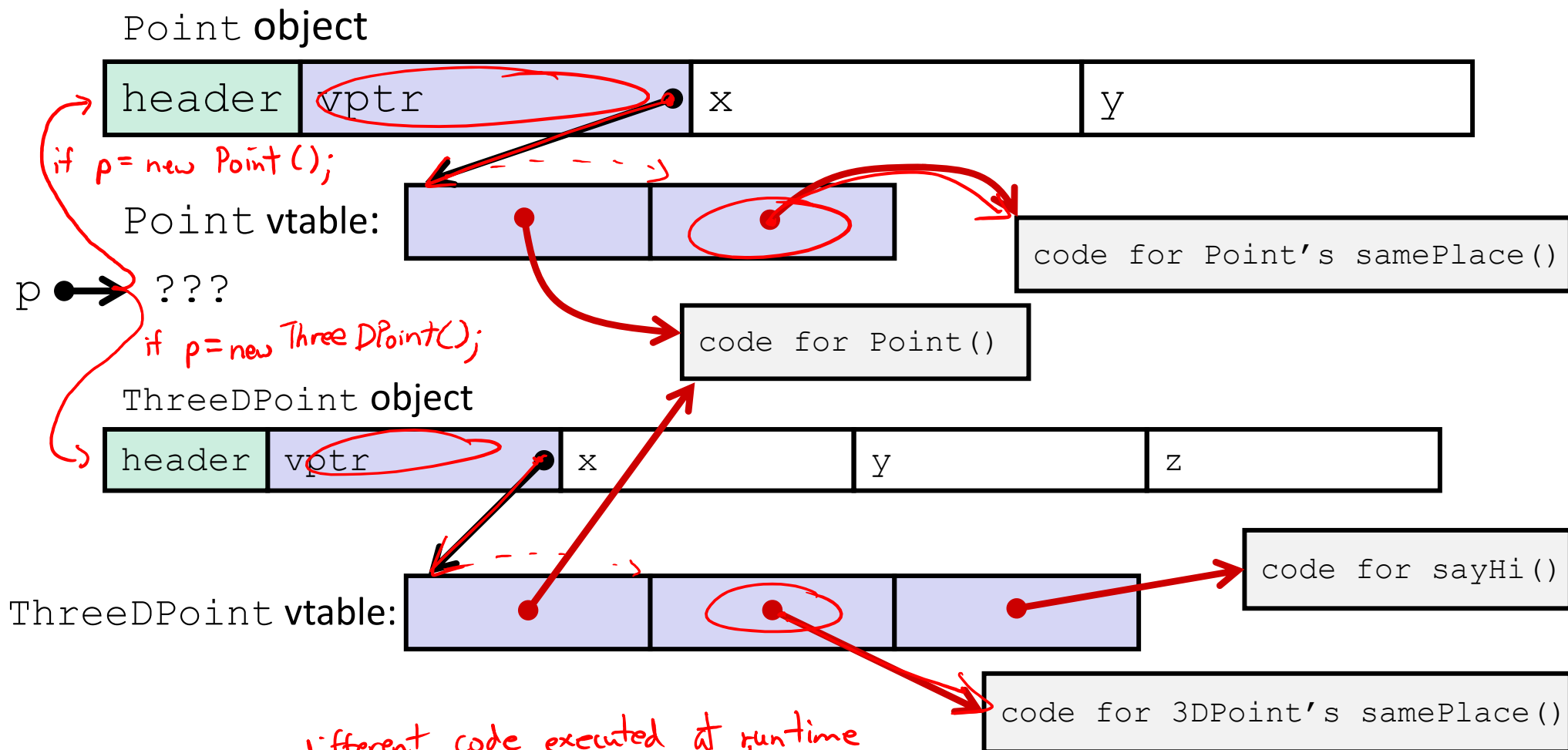
ThreeDPoint object



vtable for ThreeDPoint:  
(not Point)



# Dynamic Dispatch



## Java:

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

## C pseudo-translation:

```
// works regardless of what p is
return p->vptr[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

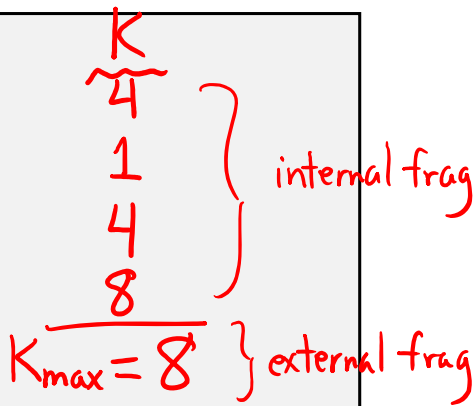


# Practice Question

- ❖ 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)

```

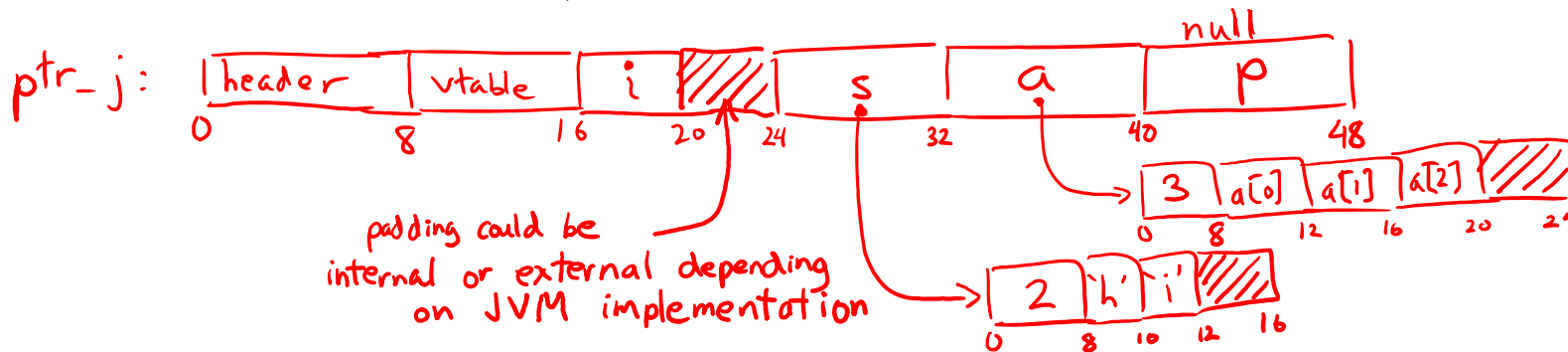
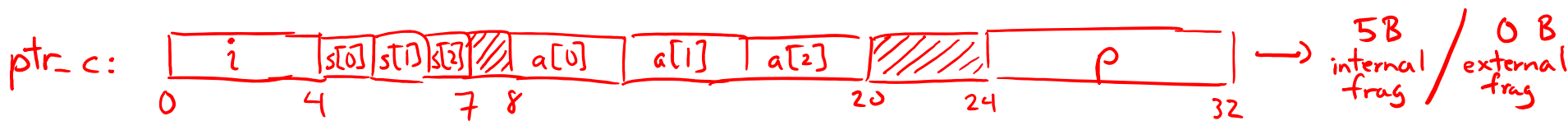
struct c {
    int i;
    char s[3];
    int a[3];
    struct c *p;
};
struct c* ptr_c;
    
```



```

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

no explicit methods, but still inherits constructor & methods from Object class



# 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();
```

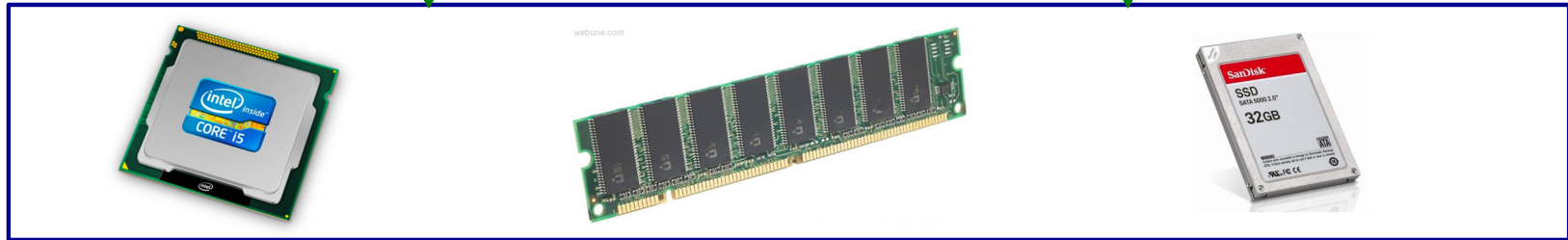
Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



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

OS:

