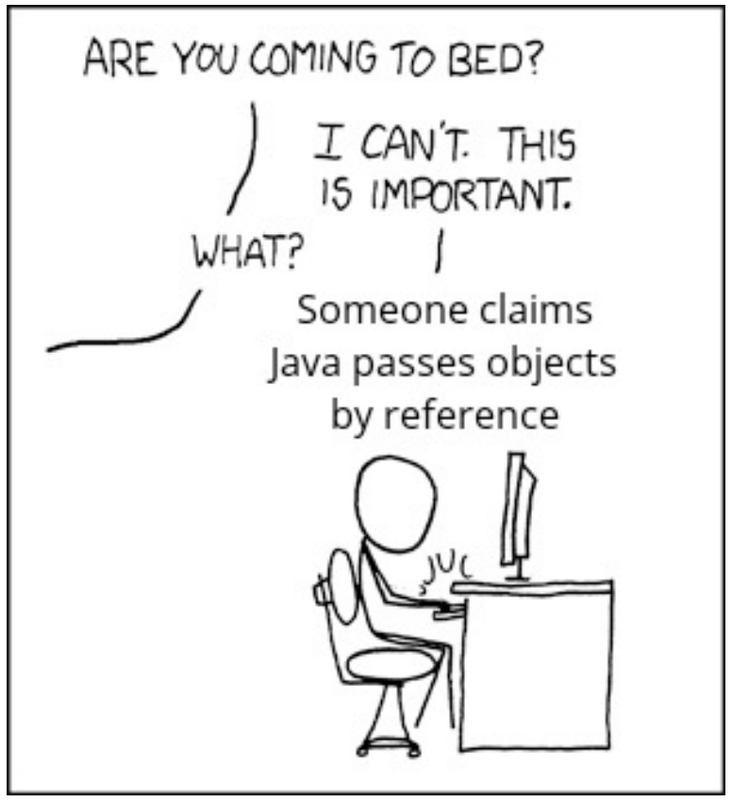
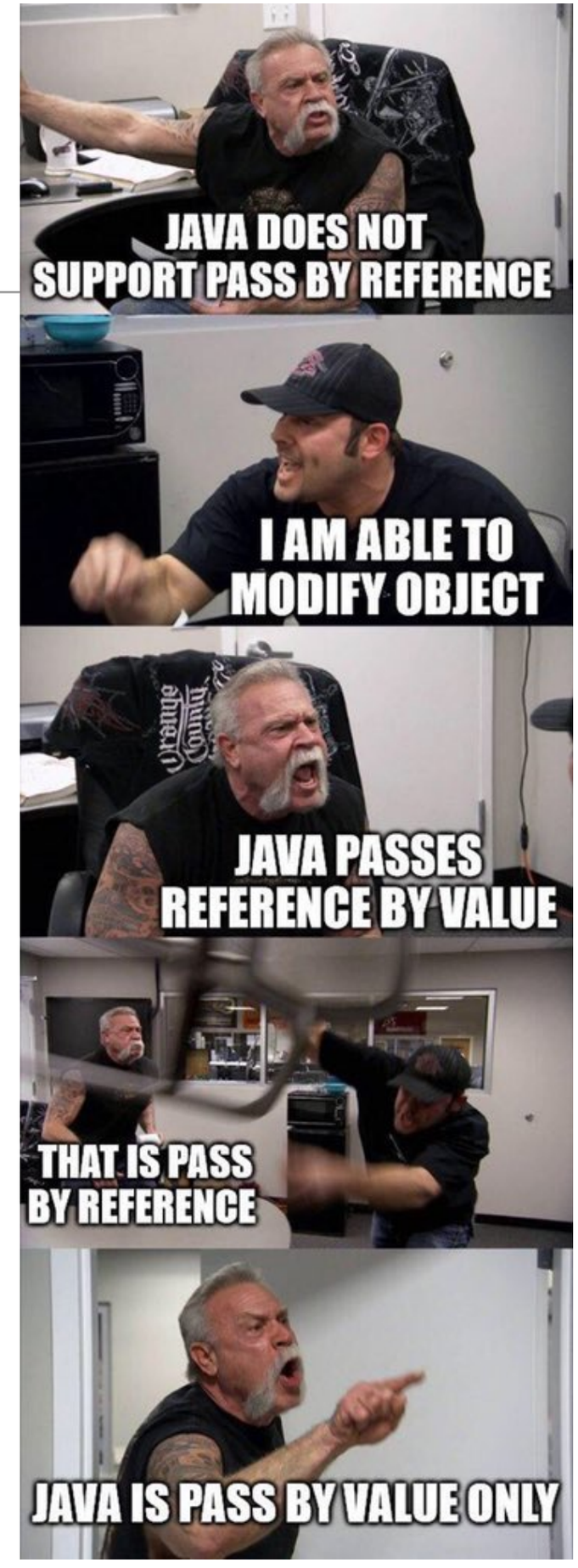
OOP in Java: Inheritance

CS 171: Intro to Computer Science II

But first, let's recap…
Java is pass-by-value language!





Java is Pass-By-Value: Strings

```
public static void changeString(String str){
    str = "hey";
}
```

```
String c = "hello";
System.out.println("c = " + c);
changeString(c);
System.out.println("c = " + c);
```

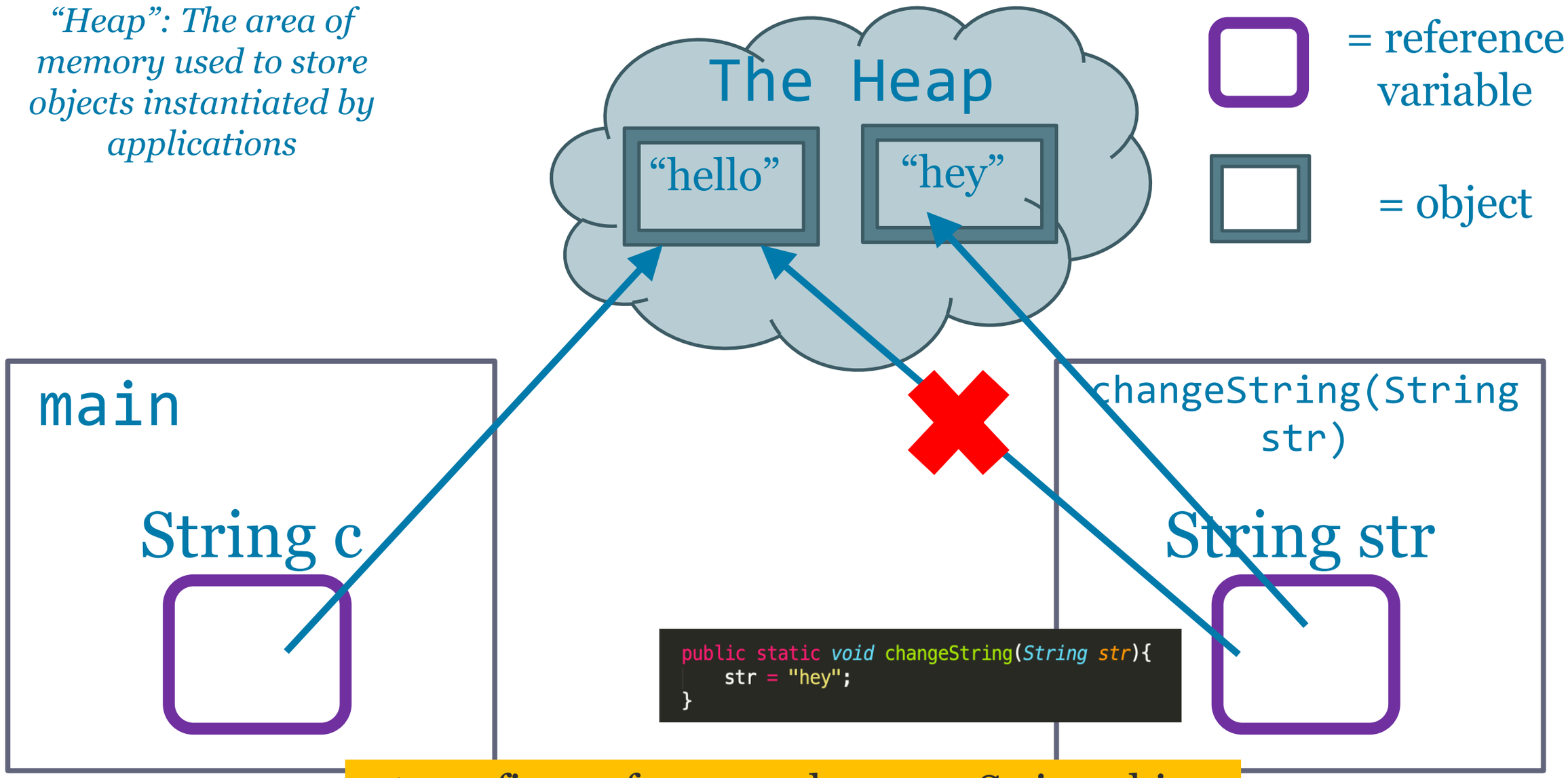
→

```
c = hello
c = hello
```

See code example on Canvas: lectures3-4-classes-passbyvalue => PassByValue.java

What's going on in memory?

"Heap": The area of memory used to store objects instantiated by applications



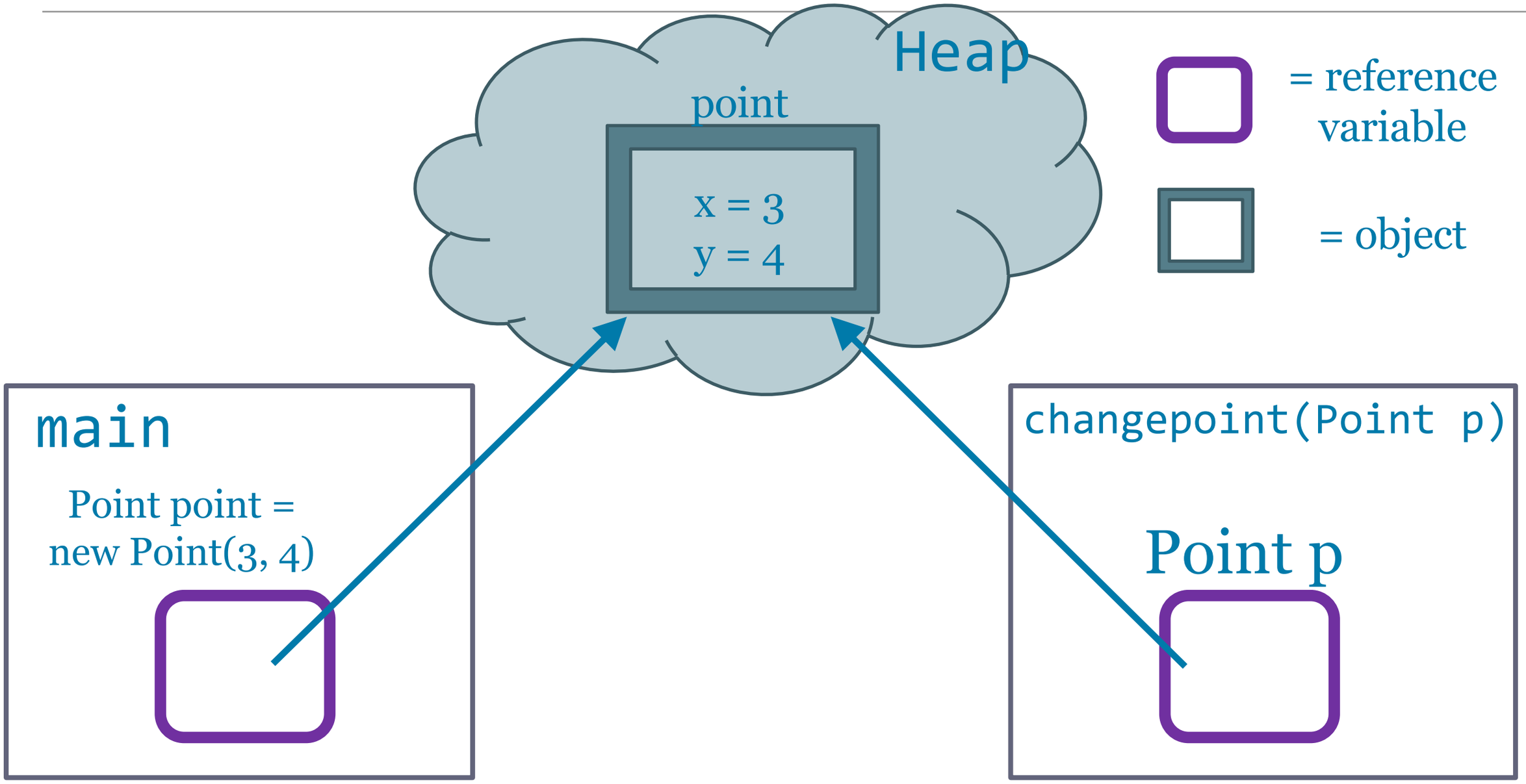
`str` at first references the same String object as `c`, but then references a new String object

Java is Pass-By-Value: Point

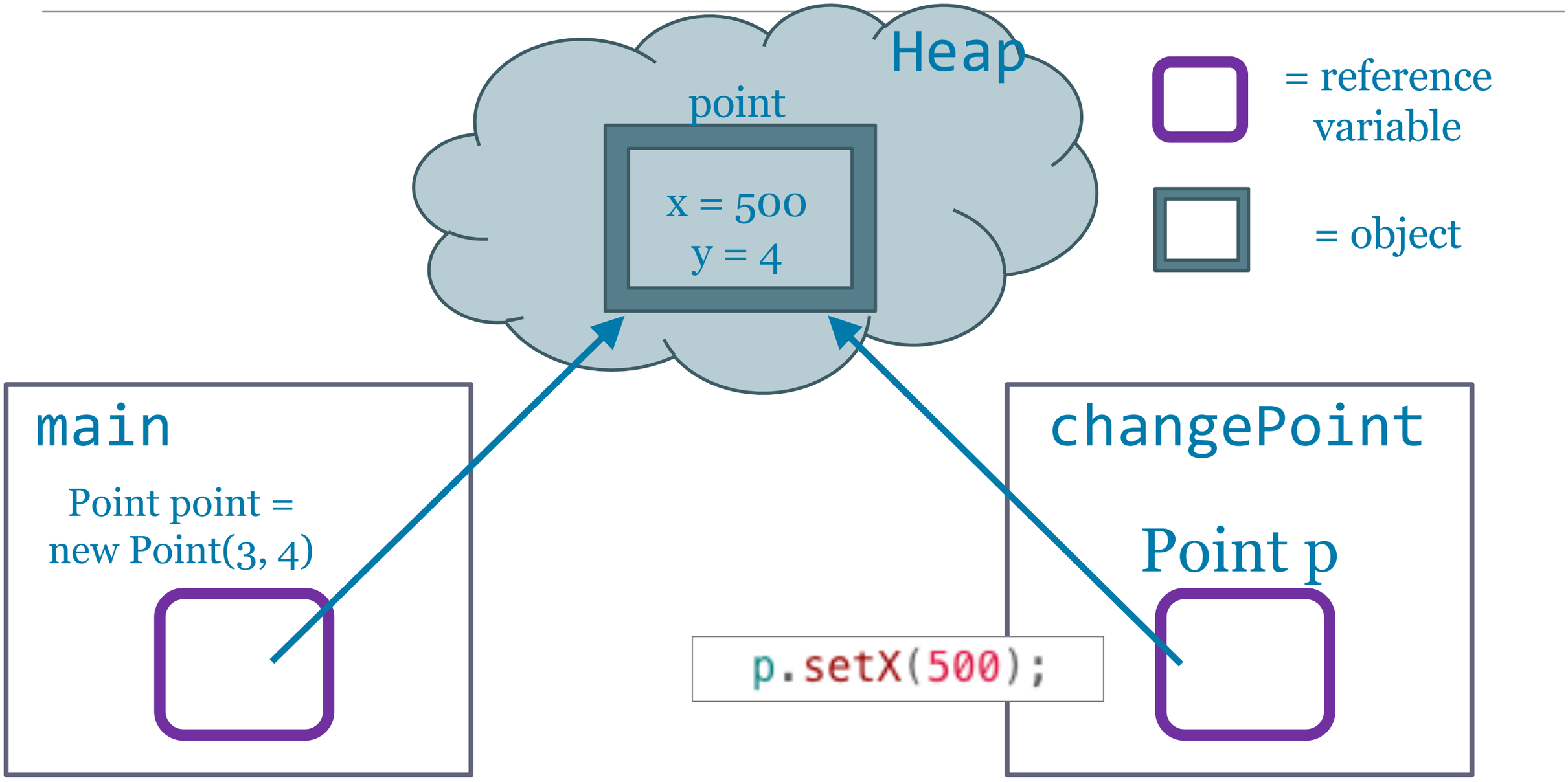
Recall the method changePoint in PointTester.java:

```
3     public static void changePoint(Point p){
4       // p = null;
5
6       // Test updating p's instance variables:
7       p.setX(500);
8       p.setY(600);
9
10      // What if....
11      // Point anotherPoint = new Point(); // default x,y
12      // p = anotherPoint;
13
14      // NOTE: Passing objects as parameters
15      //-------------------------------------
16      // (1) Re-assigning the object inside the method to something
17      // else (e.g. another object or null) does not affect
18      // the original object!
19      //
20      // (2) Updating the object's member variables
21      // (instance variables) does indeed get reflected
22      // as it is directly changing the object's contents!
23    }
```

Java is Pass-By-Value: Point



Java is Pass-By-Value: Point



`point` and `p` reference the same object, so changing the member variable `x` through `setX` method is reflected back in `point`

Java is Pass-By-Value: Arrays

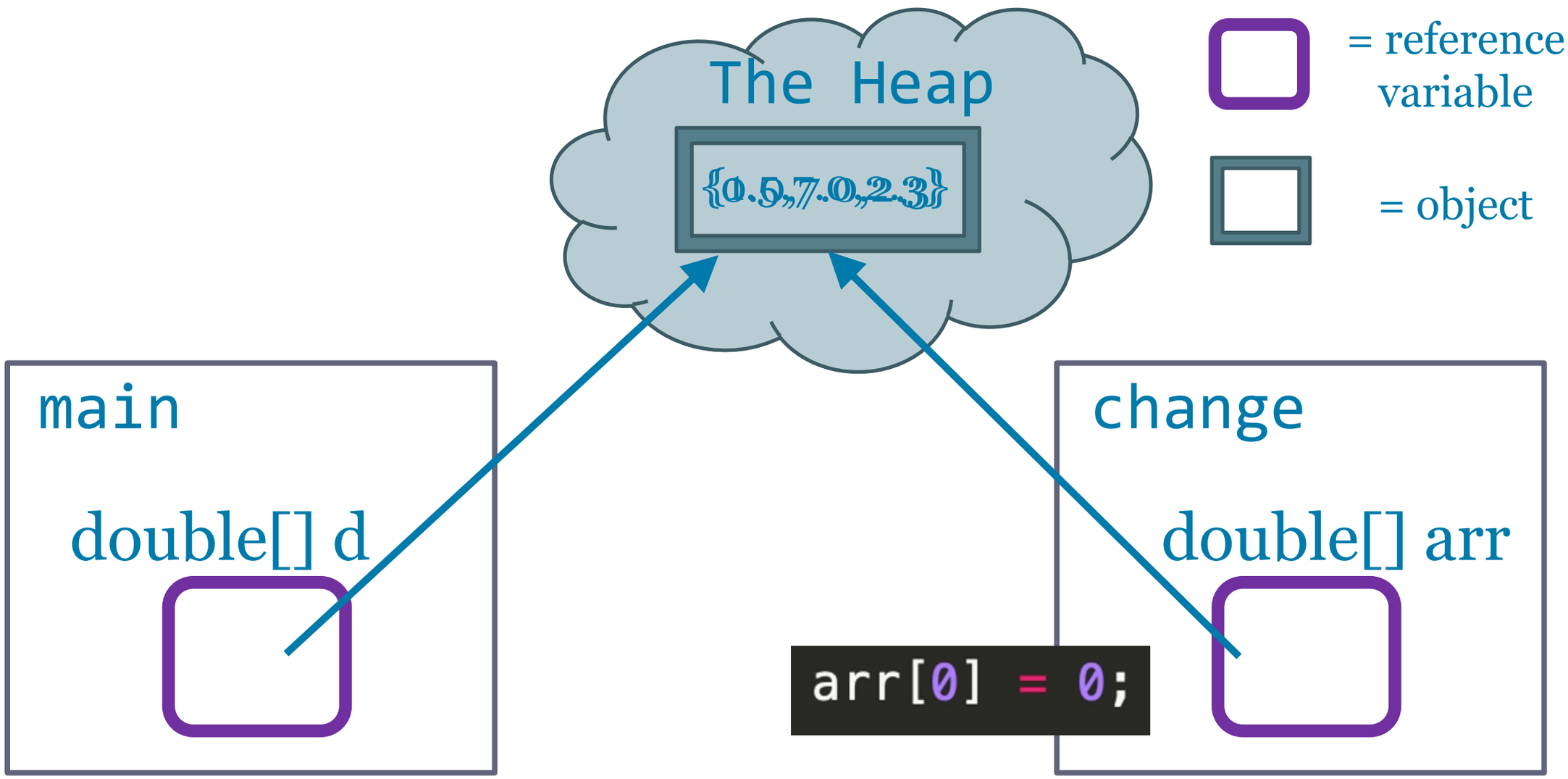
```
public static void changeArray(double[] arr){
    arr[0] = 0;
}
```

```
double[] d = {1.5, 7.0, 2.3};
System.out.println("d = {" + d[0] + "," + d[1] + "," + d[2] + "}");
changeArray(d);
System.out.println("d = {" + d[0] + "," + d[1] + "," + d[2] + "}");
```

↓

```
d = {1.5,7.0,2.3}
d = {0.0,7.0,2.3}
```

Java is Pass-By-Value: Arrays



`d` and `arr` reference the same array, so changes through `arr` are reflected in `d`

Pass-by-Value: Lessons Learned

Objects as Method Parameters:

1. Re-assigning the object inside the method to something else (e.g. another object or null) does not affect the original object! This is because the reference changes.

2. Updating the object's member variables (instance variables) does indeed get reflected as it is directly changing the object's contents

3. What if I want to re-assign the original object?
   - You can return and assign the new object in the caller!

Pass-by-Value: Lessons Learned

3. What if I want to re-assign the original object?
   - You can return and assign the new object in the caller!

See “PassByValue.java” examples:

```
public static String changeStrReturnNewString(String x){
  x = "xyz";
  return x; // We're now returning the object that references "xyz"
}
```

```
String x = "ab";
x = changeStr(x);
System.out.println("x contains: " + x); // now it's "xyz"
```

New Topic: Inheritance

Today we'll focus on understanding key concepts and Java examples, then on Wed we'll code our own inheritance applications.

For the rest of today's lecture, download this self-test and quiz yourself at the end of the lecture!

5-inheritance-definitions-selftest.docx

CS171 – OOP: Inheritance

By the end of today's lecture, quiz yourself!
What is the meaning of the following keywords, in Java, and when do we use them?

- `extends`
- `super`
- `static`
- `this`

- Give 3 examples showing the different ways you can use the keyword `super` inside a subclass, as a reference to its super (parent) class:
  1.
  2.
  3.

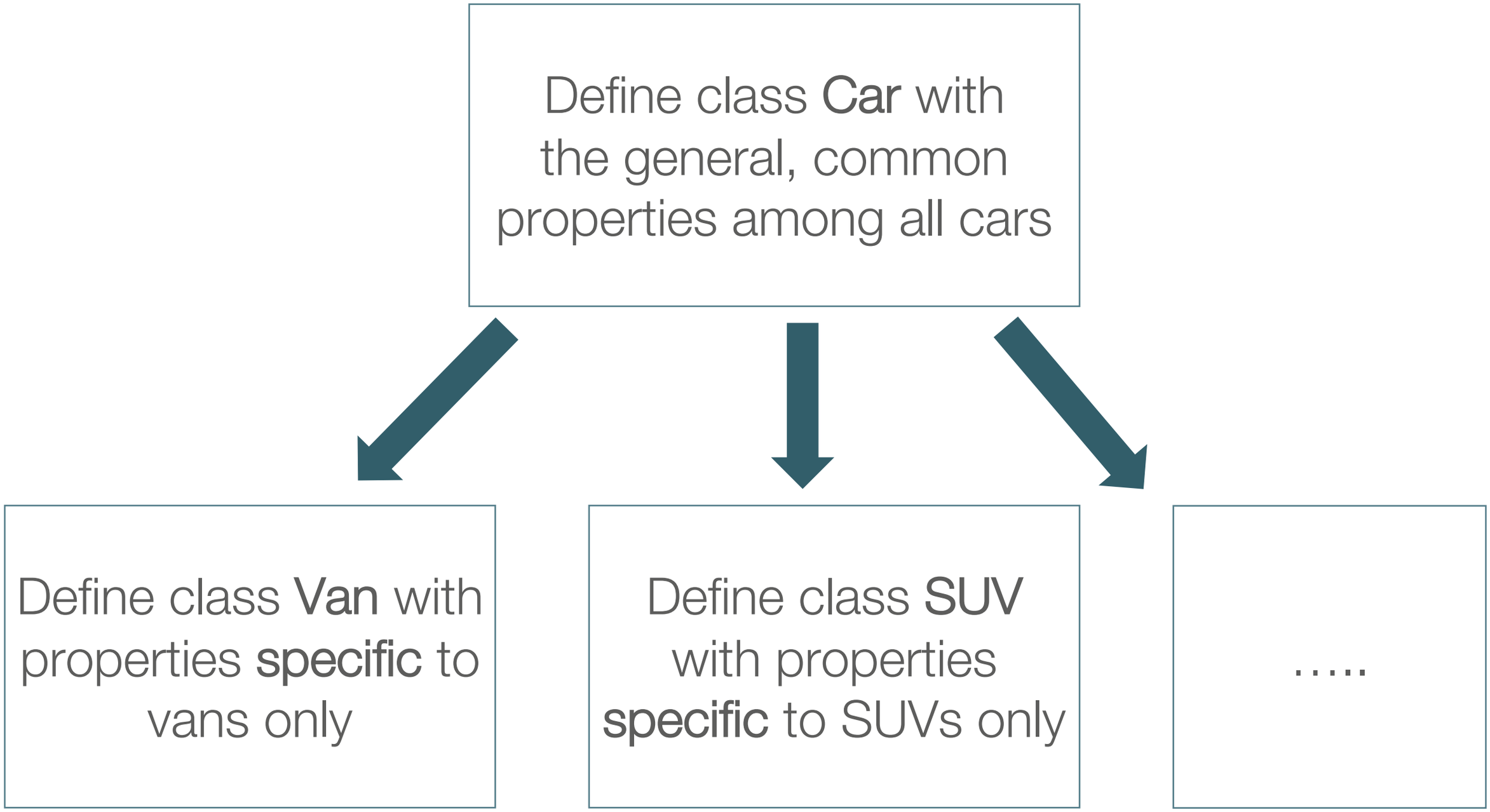
- What is method *overriding* in Java? Give an example?

What if we want to define more specific types of cars…

- Different types of cars
  - Sedan
  - SUV
  - Van

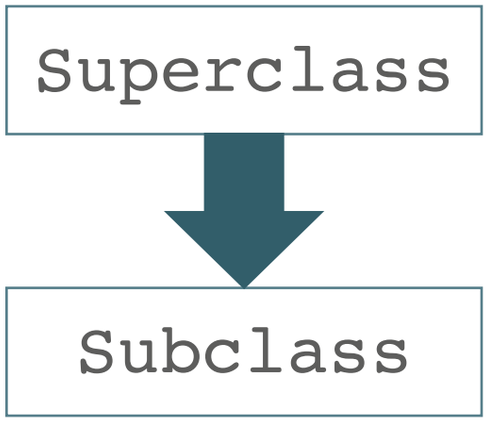
- What features are ***common*** for all cars?
- What features are ***specific***?

Instead of rewriting lots of code, here's an idea…



The Four Pillars

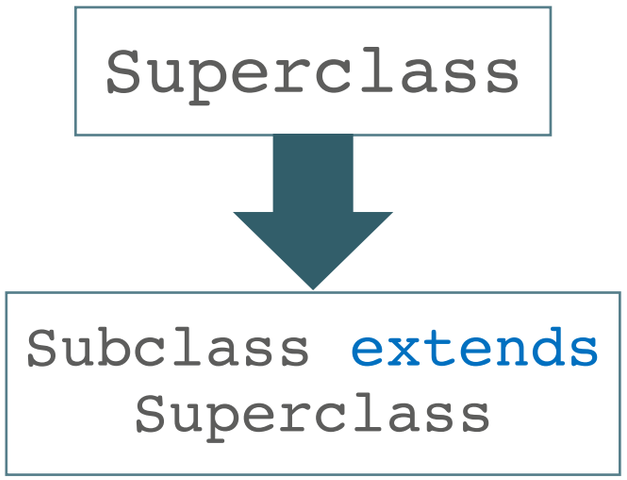
Inheritance to the rescue



- A subclass inherits all fields (data) and methods from the superclass
- Subclass can also
  - Add new fields
  - Add new methods
  - Override the methods of the superclass
- How about the superclass's constructor?
  - Superclass's constructor are not inherited — invoked explicitly or implicitly

Inheritance Keywords

- Q: How do I indicate that my class inherits from another superclass?
  - *extends*



- Q: Inside a subclass, can I access my superclass (parent)?
  - *super*

extends and super Keywords

- `extends` keyword indicates that one class (subclass) inherits from other class ➔
  `public class Child extends ParentClass`

- `super` refers to the superclass and can be used in a few ways:

  1. Call a superclass constructor ➔ `super(x,y);`

  2. Call a superclass method➔ `super.foo();`

  3. Access a superclass public/protected data field ➔ `super.name;`

Overriding Methods

- Subclass can modify the implementation of a method defined in the superclass — known as method overriding
  - Same exact signature (method name and parameter types) as a method in the superclass

It's like when I inherited my mother's blueberry muffins recipe but decided to make my own changes to it to make it "heathier" (more blueberries, brown sugar instead of white sugar)…..

.....it tasted worse.

Overriding Methods

- Subclass can modify the implementation of a method defined in the superclass — known as method overriding
  - Same exact signature (method name and parameter types) as a method in the superclass
  - Consider using @Override annotation (compiler checking)

```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

- A private method cannot be overridden because it is not accessible outside its own class
- Different from overloading

Overloading vs Overriding

- Overloading allows the same method name to be declared multiple times with different parameters
  - Usually done within the same class
  - Useful for processing different objects by similar logic
- Overriding
  - Only done by *subclass* 
  - Useful for incorporating additional information into the methods supported by the basic API of the superclass

Which is overloading/overriding?

```java
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

```java
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name,
Different Parameter

https://www.programcreek.com/2009/02/overriding-and-overloading-in-java-with-examples/

Which is overloading/overriding?

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

Overloading

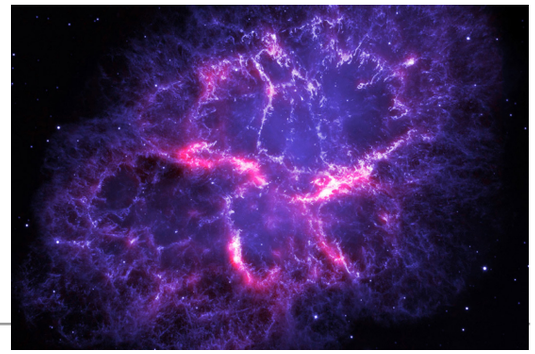
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

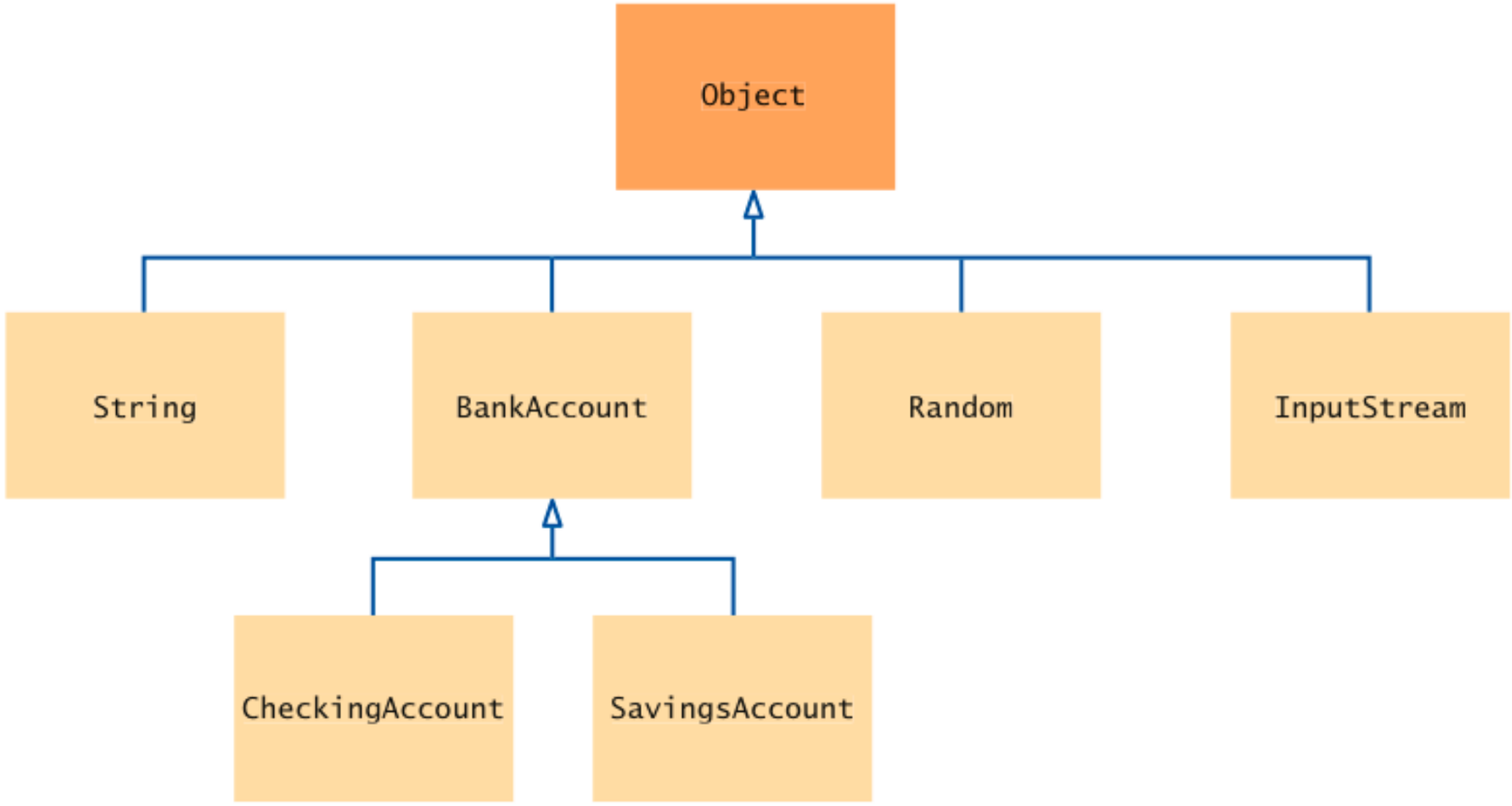
Same Method Name,
Different Parameter

https://www.programcreek.com/2009/02/overriding-and-overloading-in-java-with-examples/

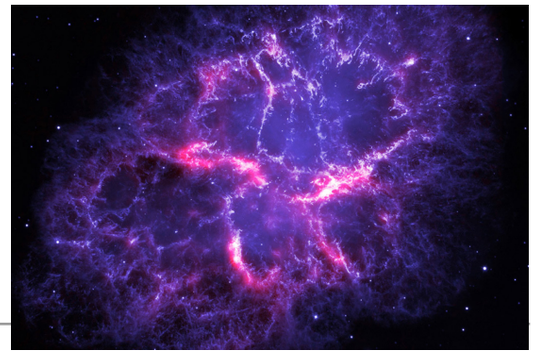
Object: The Cosmic Superclass



- Surprise! All classes defined without an explicit extends keyword automatically extend Object



Object:
The Cosmic Superclass



- Most useful methods in class Object:
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`

*** Good idea to override these methods ***

Overriding `toString()`

- Easy to read textual representation of an object
- Called when object reference is passed to `System.out.println()`
- Recommended to always override `toString()` to print something meaningful about the object (often instance variable values)

toString

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Returns:

a string representation of the object.

Overriding `toString()` in Point?

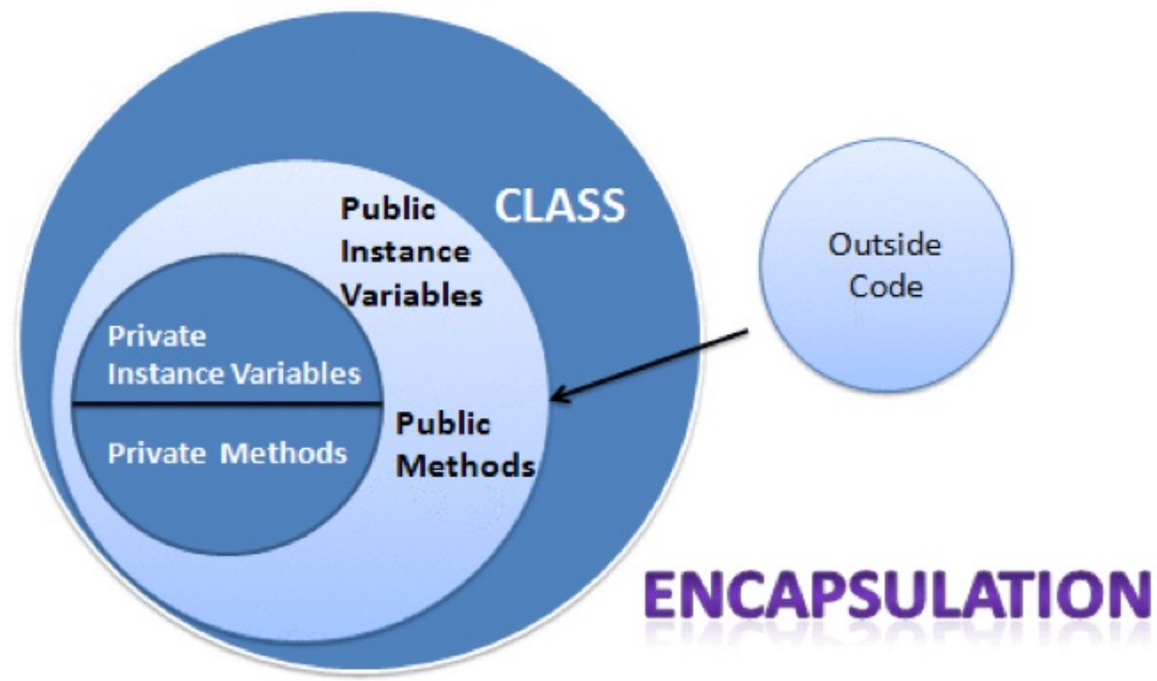

The Four Pillars
OOP
Abstraction
Inheritance
Polymorphism
Encapsulation

Encapsulation

- Protective barrier that prevents code and data being randomly controlled outside your class

- Make fields private, provide access via public methods

- Gives maintainability, flexibility, and extensibility to code



Accessibility

Access Levels

Modifier	Class	Package	Subclass	World
`public`				
`protected`				
no modifier				
`private`				

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Accessibility

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Inheritance Code Examples: Agenda

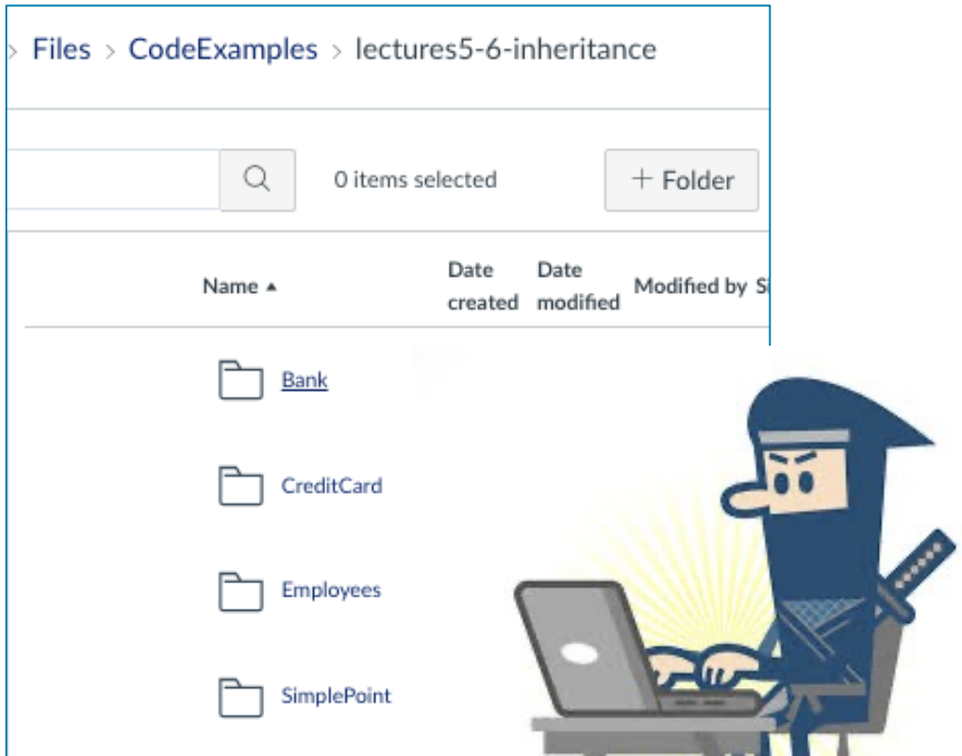
1. Start by simply extending Point to Point3D (a point that has 3 coordinates: x, y, z)

More exercises:

2. Credit card application with different type of cards

3. Company application with different types of Employees (salaried, hourly)

4. Bank application with different types of accounts (checking, savings)



Employee example

[Source code available on Canvas]

```java
/** a generic employee class */
public class Employee {
    private String name; // name of the employee
    public Employee(String n) { name = n; }
    public Employee() { name = "Unknown"; }
    public String getName() { return name; }
    public String toString() { return name; }
    public double earnings() { return 0; }
}
```

```java
/** An hourly employee that makes an earning based on hourly wage */
public class HourlyEmployee extends Employee {
    private double wage;
    private double hours;

    public HourlyEmployee(String n, double w, double h) {
        super(n); wage = w; hours = h;
    }
    public double earnings() {
        return wage * hours;
    }
}
```

```java
/** A salaried employee that makes a fixed salary */
public class SalariedEmployee extends Employee {
    private double weeklySalary;

    public SalariedEmployee(String n, double salary) {
        super(n); weeklySalary = salary;
    }
    public double earnings() {
        return weeklySalary;
    }
}
```

Employee example

[Source code available on Canvas]

```java
/** a generic employee class */
public class Employee {
    private String name; // name of the employee
    public Employee(String n) { name = n; }
    public Employee() { name = "Unknown"; }
    public String getName() { return name; }
    public String toString() { return name; }
    public double earnings() { return 0; }
}
```

```java
/** An hourly employee that makes an earning based on hourly wage */
public class HourlyEmployee extends Employee {
    private double wage;
    private double hours;

    public HourlyEmployee(String n, double w, double h) {
        super(n); wage = w; hours = h;
    }
    public double earnings() {
        return wage * hours;
    }
}
```

```java
/** A salaried employee that makes a fixed salary */
public class SalariedEmployee extends Employee {
    private double weeklySalary;

    public SalariedEmployee(String n, double salary) {
        super(n); weeklySalary = salary;
    }
    public double earnings() {
        return weeklySalary;
    }
}
```

Bank Example – Super Class

```java
public class BankAccount
{
   private double balance; //account balance

   public BankAccount() { balance = 0; }
   public BankAccount(double initialBalance) { balance = initialBalance; }

   public void deposit(double amount) {
      System.out.println("BankAccount.deposit("+amount+")");
      balance = balance + amount;
   }

   public void withdraw(double amount) {
      System.out.println("BankAccount.withdraw("+amount+")");
      balance = balance - amount;
   }

   public void transfer(double amount, BankAccount otherAccount) {
      System.out.println("BankAccount.transfer("+amount+")");
      // withdraw money from this account and deposit to the other
      withdraw(amount);
      otherAccount.deposit(amount);
   }

   public double getBalance() {
      System.out.println("BankAccount.getBalance()");
      return(balance);
   }
}
```

CheckingAccount

- Customer has a limited number of FREE transactions (i.e. deposits or withdraws) = 2
- After reaching that limit, customer pays an additional fee with every transaction

Bank Example - Subclass

```java
public class CheckingAccount extends BankAccount
{
   /* Static variables that are consistent across all checking accounts */
    private static final int FREE_TRANSACTIONS = 2;
    private static final double TRANSACTION_FEE_MULTIPLIER = 2.0;

    private int transactionCount;

    public CheckingAccount(double initialBalance) {
      super(initialBalance);
      transactionCount = 0;
   }

   @Override
   public void deposit(double amount) {
      //Override deposit to keep track of transactions
      transactionCount++;
      super.deposit(amount);
   }

   @Override
   public void withdraw(double amount) {
      //Override withdraw to keep track of transactions
       transactionCount++;
      super.withdraw(amount);
   }

   public void deductFees()
   {
     if (transactionCount > FREE_TRANSACTIONS) {
        double fees = TRANSACTION_FEE_MULTIPLIER * (transactionCount - FREE_TRANSACTIONS);
        withdraw(fees);
      }
      transactionCount = 0;
    }
}
```

Inherits “balance” but creates new data members

Subclass constructor calls superclass constructor, then initializes its own private var.

compiler annotation

Overriding deposit() so that we add an extra calculation before calling the original deposit

I can also call super.withdraw(fees) here

Checking Account Class

- Instance fields
  - `balance` — inherited from BankAccount but private
  - `transactionCount` —new to Checking Account
- Methods
  - `getBalance()` — inherited from BankAccount
  - `deposit(double amount), withdraw(double amount)` — override and update transaction count