Java Classes – Contd.

CS 171: Intro to Computer Science II

Reminders & Agenda

- Quiz#1 this Friday 1/27 (see Canvas for details)

Today:

- Pass by value
  - What does it mean, and what are its implications?
- Code Exercise: Applications on Classes & Objects and pass-by-value
  - Point.java, PointTester.java
  - PassByValue.java
- OOP Pillars
  - Inheritance

Code Example: Creating a new type for "Point" coordinates

We coded Point.java, PointTester.java,
uploaded on Canvas under:
Files > CodeExamples > lectures3-4-classes-passbyvalue >

Invoking a Method: Parameters

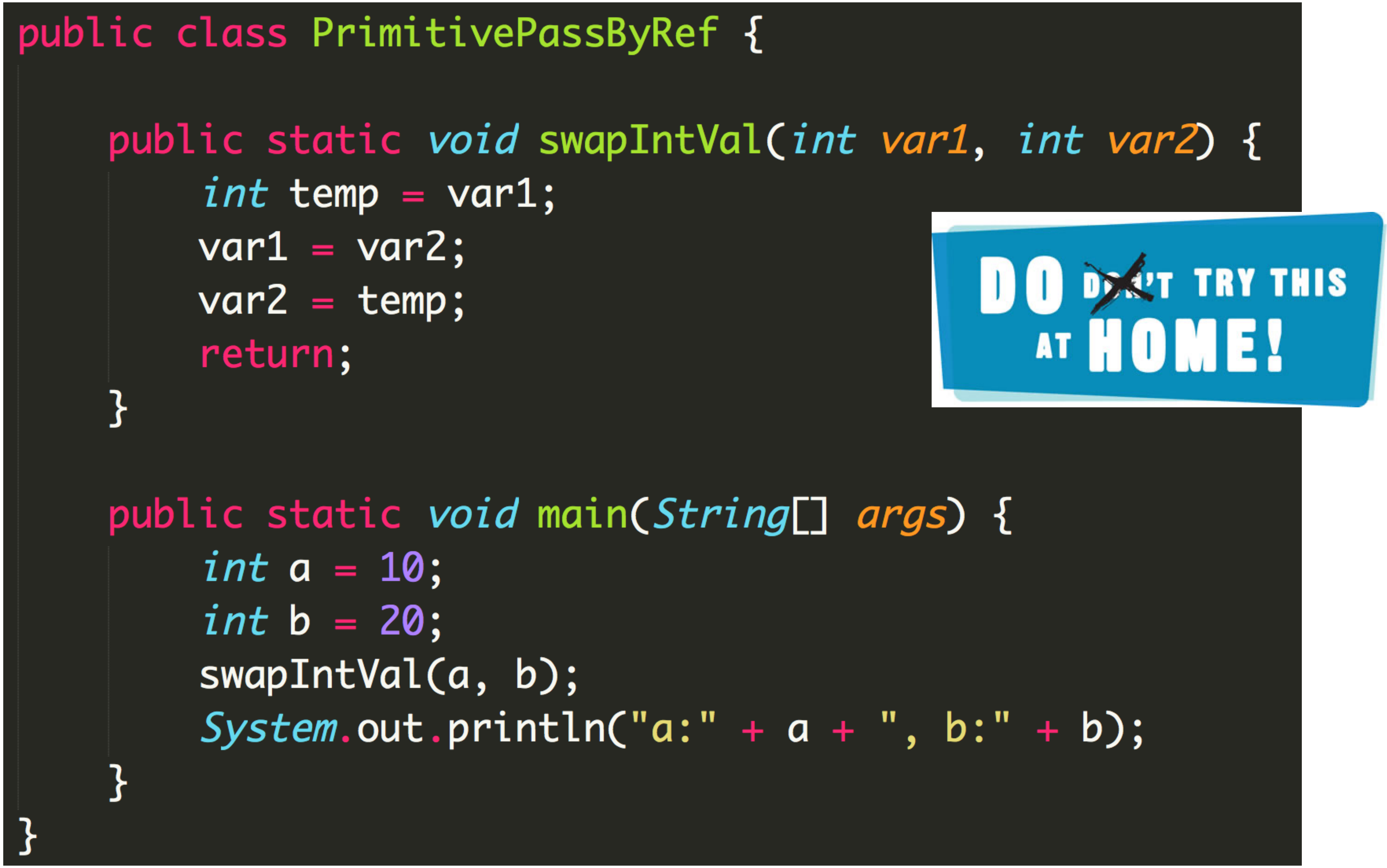
- Arguments must match the parameters in order, number, and compatible type
- Value of the argument is passed to the parameter and variable is not affected
  - Also referred to as pass-by-value

What do we mean by “Pass-by-Value”?

Pass-by-Value

- Java creates a copy of the variable being passed in the method
  - Primitives: relatively straightforward only the value is passed
  - Objects: more tricky, a copy of the reference is created and passed into the method but points to the same memory reference

Example: Primitive Type



```java
public class PrimitivePassByRef {

    public static void swapIntVal(int var1, int var2) {
        int temp = var1;
        var1 = var2;
        var2 = temp;
        return;
    }

    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        swapIntVal(a, b);
        System.out.println("a:" + a + ", b:" + b);
    }
}
```

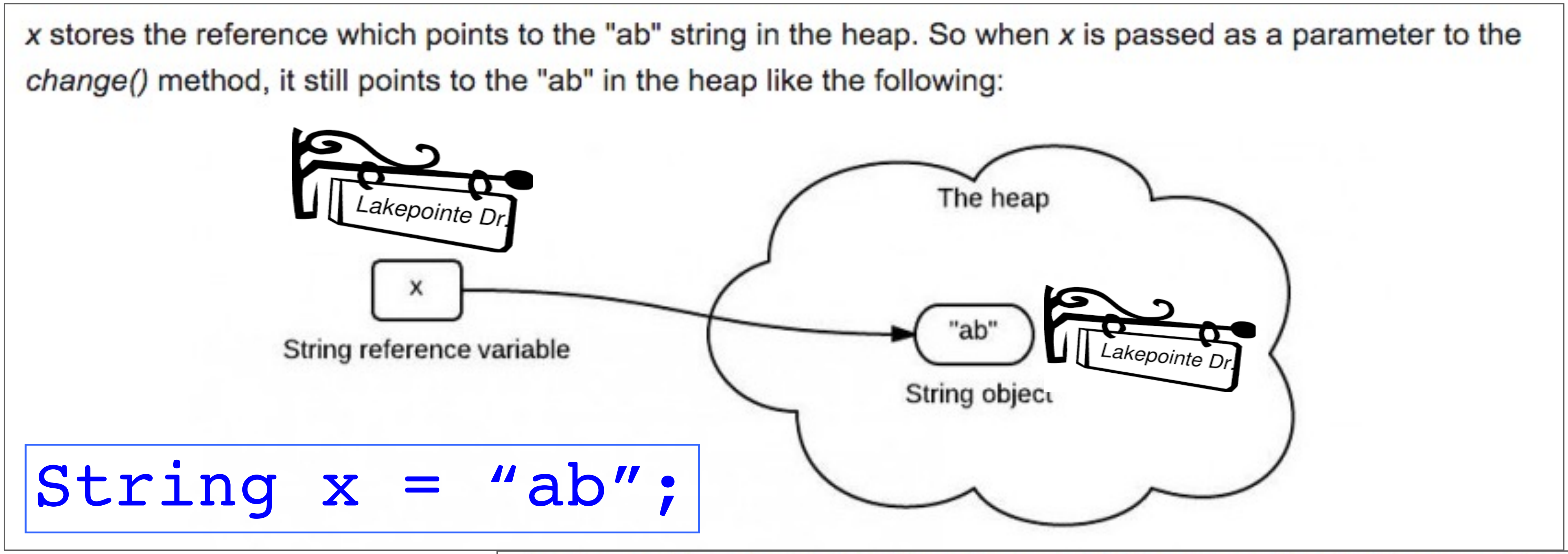
How about Strings?

- See examples in PassByValue.java, uploaded on Canvas under:
  Files > CodeExamples > lectures3-4-passbyvalue >

Tip Run the examples yourself; try modifying the code and experiment with different “What-If” questions.

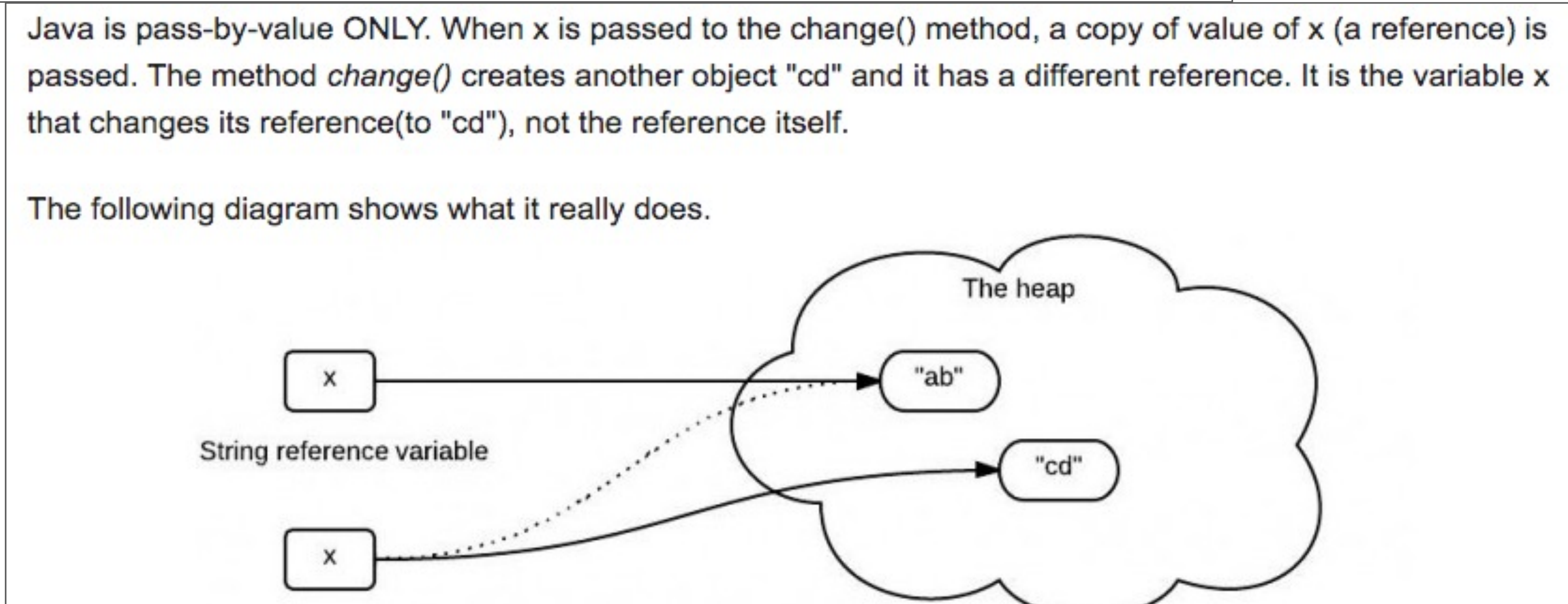
Check if the output matches what you expect. It’s a fun way to learn programming!

How about Strings (non-primitive)?



x stores the reference which points to the "ab" string in the heap. So when *x* is passed as a parameter to the *change()* method, it still points to the "ab" in the heap like the following:

```
String x = “ab”;
```



Java is pass-by-value ONLY. When x is passed to the change() method, a copy of value of x (a reference) is passed. The method *change()* creates another object "cd" and it has a different reference. It is the variable x that changes its reference(to "cd"), not the reference itself.

The following diagram shows what it really does.

https://www.programcreek.com/2013/09/string-is-passed-by-reference-in-java/

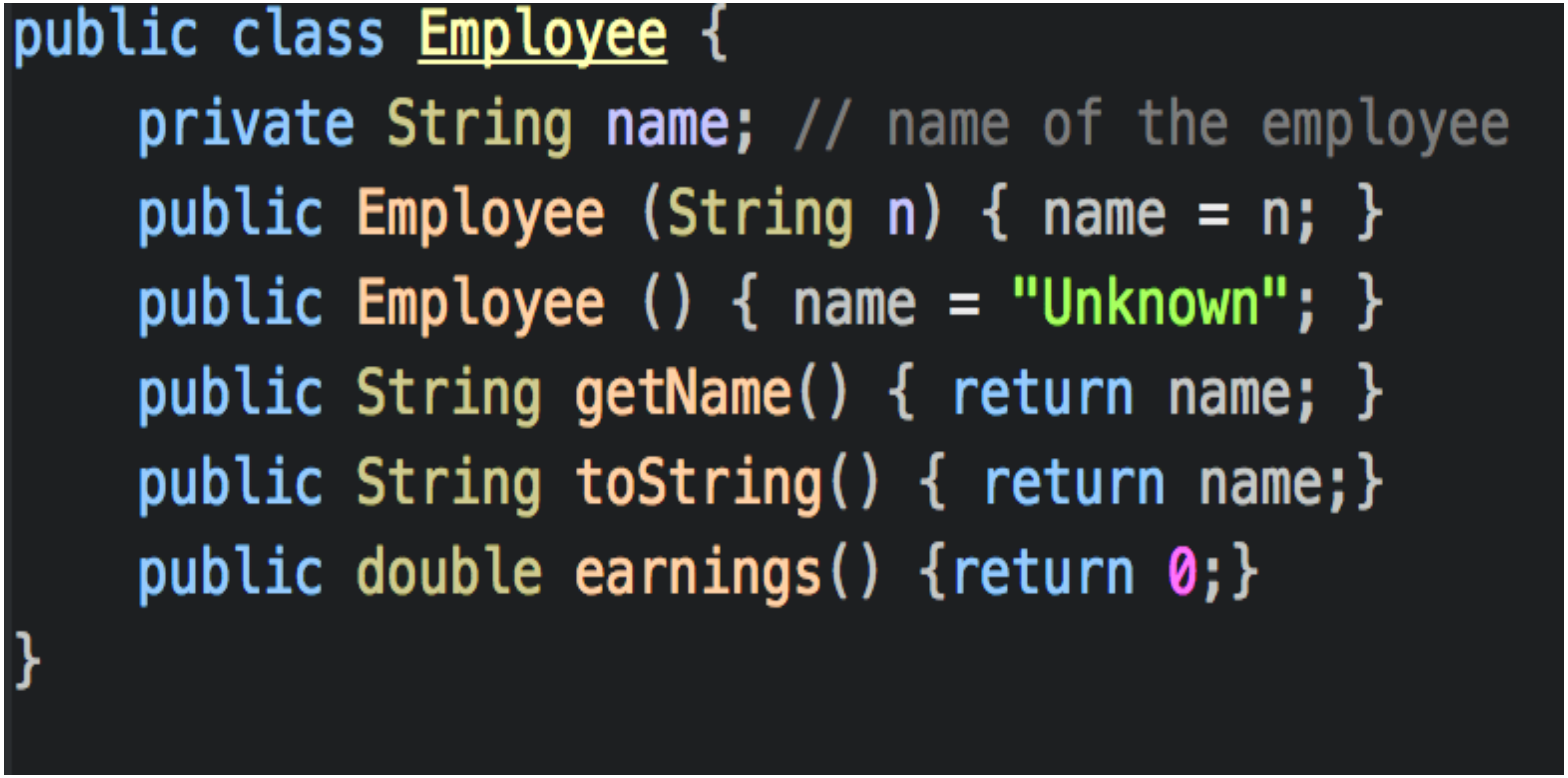
Pass-by-Value: Objects

- Changes are not reflected back if we change the object itself to refer to some other location or object

- However:
  If the reference is not assigned to a new location or object & changes are made to its members, the changes will be reflected back

  - Example? See next slide… /*drum roll*/

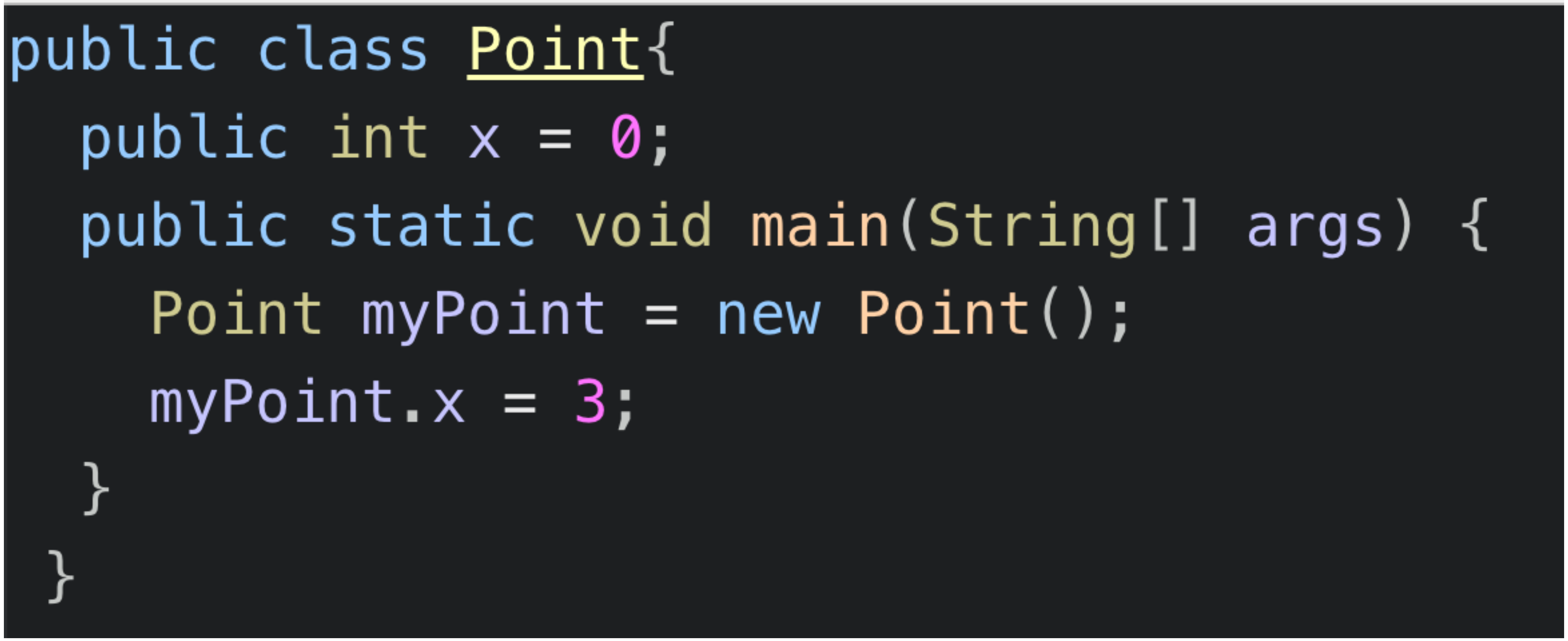
Find the member variable(s) (aka “instance variables”)



```java
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;}
}
```

`name`

Find the member variable(s) (aka “instance variables”)

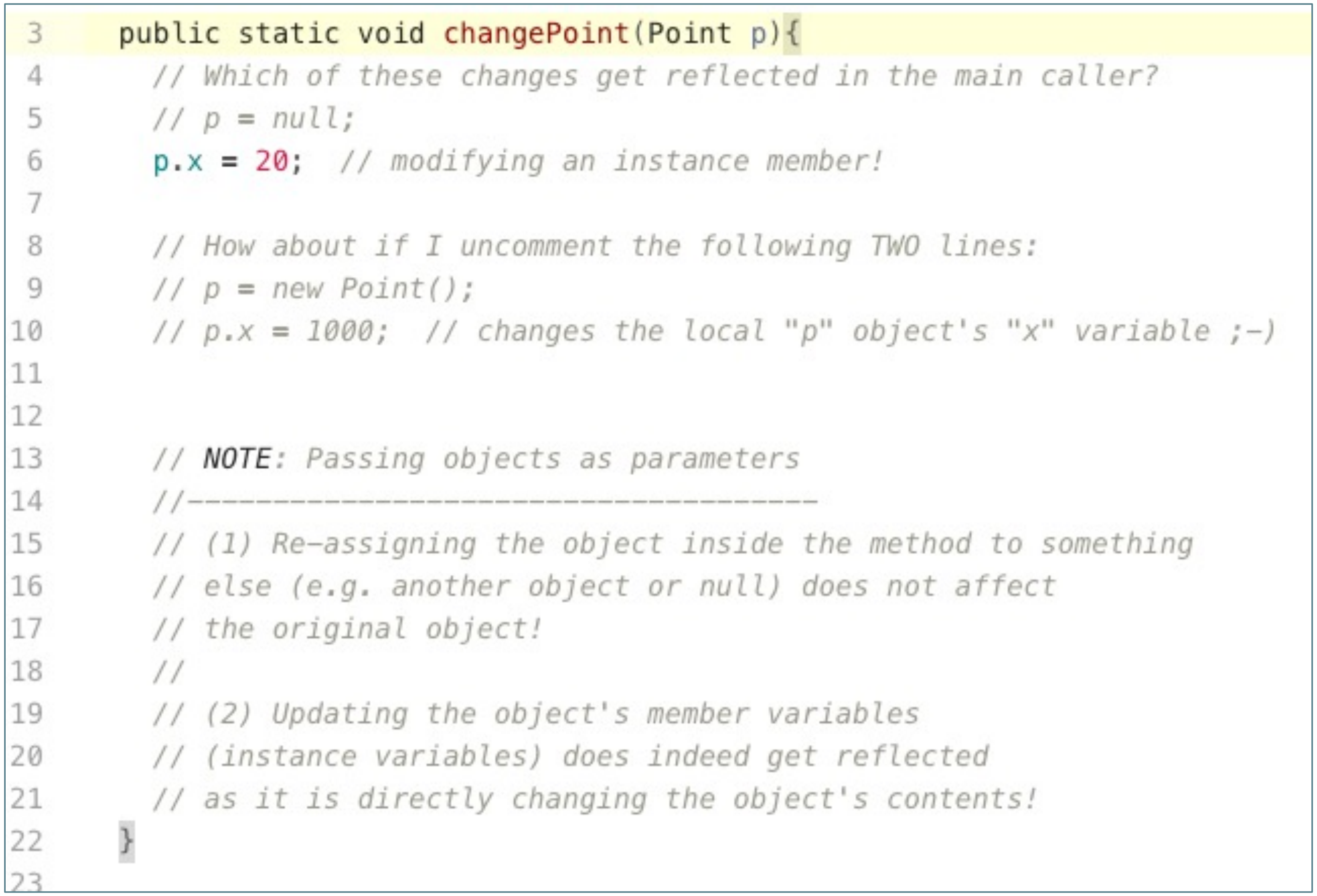


```
public class Point{
  public int x = 0;
  public static void main(String[] args) {
    Point myPoint = new Point();
    myPoint.x = 3;
  }
 }
```

x

Example: Changing a *member* of an Object

See the method changePoint in PointTester.java:

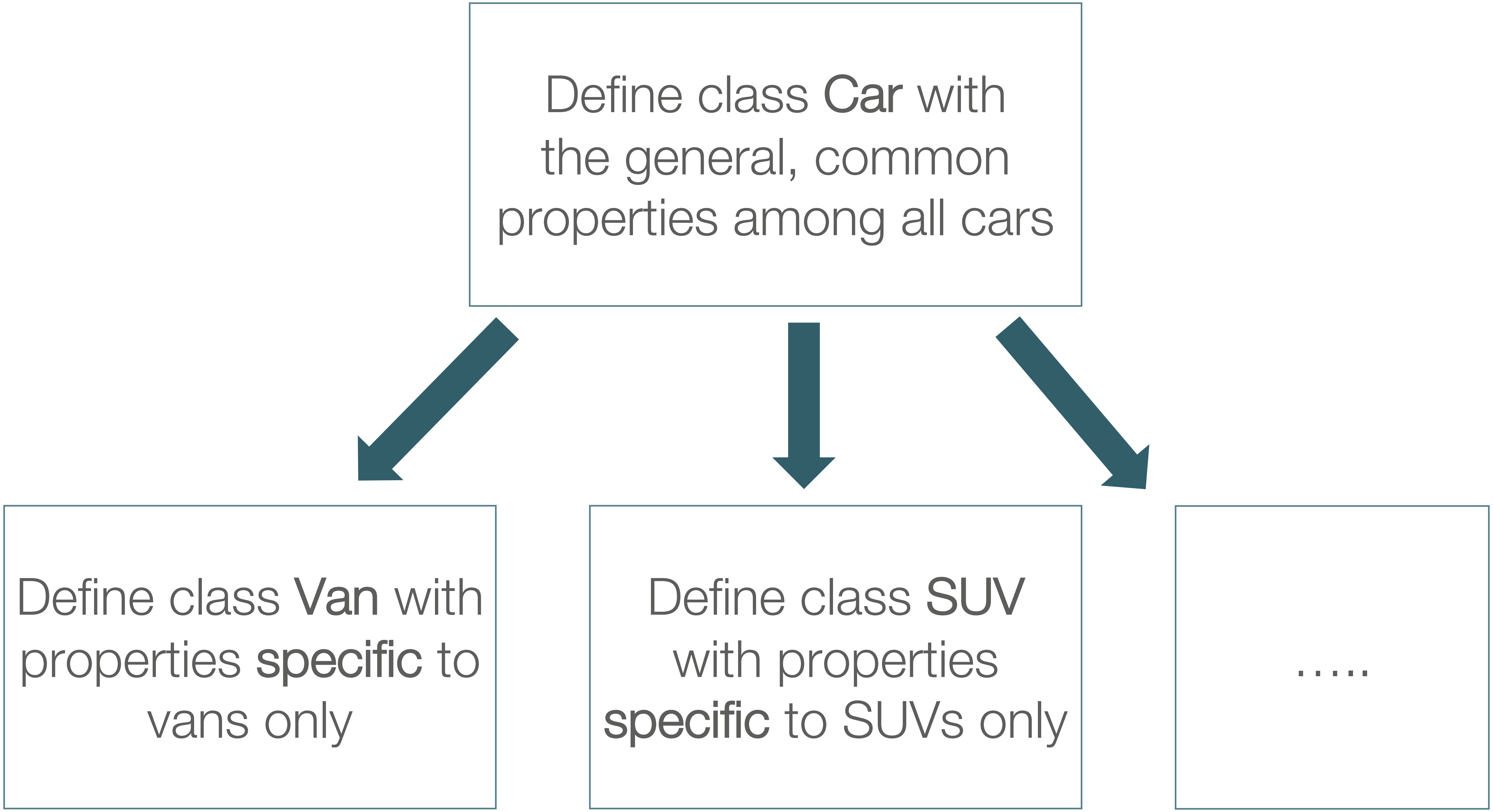


```
    public static void changePoint(Point p){
      // Which of these changes get reflected in the main caller?
      // p = null;
      p.x = 20;  // modifying an instance member!

      // How about if I uncomment the following TWO lines:
      // p = new Point();
      // p.x = 1000;  // changes the local "p" object's "x" variable ;-)


      // NOTE: Passing objects as parameters
      //-------------------------------------
      // (1) Re-assigning the object inside the method to something
      // else (e.g. another object or null) does not affect
      // the original object!
      //
      // (2) Updating the object's member variables
      // (instance variables) does indeed get reflected
      // as it is directly changing the object's contents!
    }
```

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*?

Idea

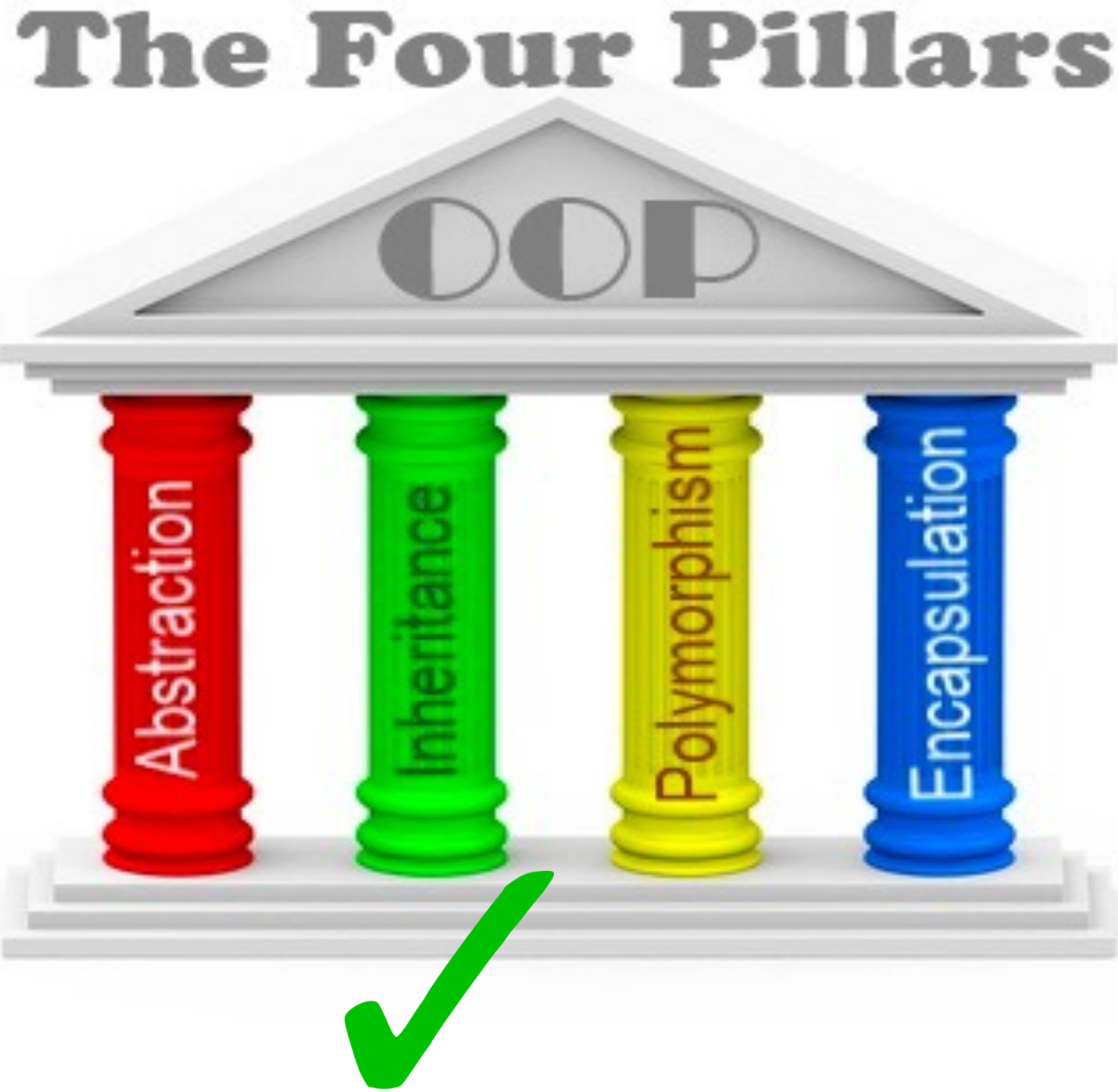


Define class **Car** with the general, common properties among all cars

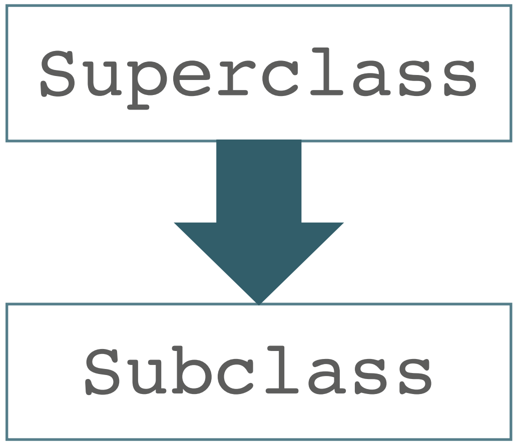
Define class **Van** with properties **specific** to vans only

Define class **SUV** with properties **specific** to SUVs only

.....


The Four Pillars
OOP
Abstraction
Inheritance
Polymorphism
Encapsulation
✓

Inheritance to the rescue



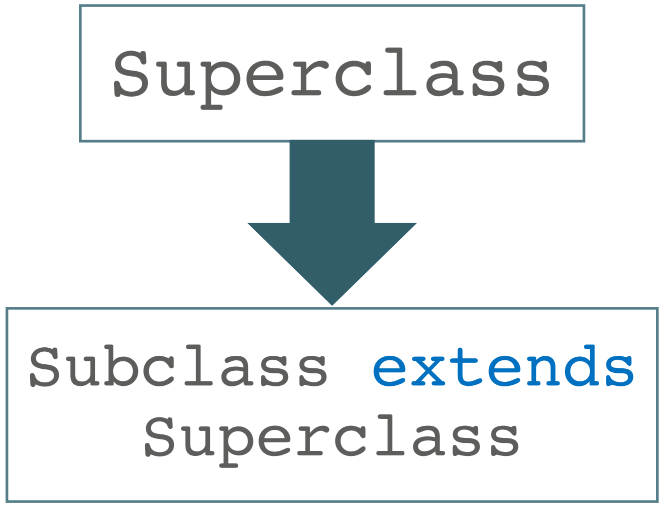
Superclass

Subclass

- A subclass inherits all fields 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`*



```
Superclass
```

```
Subclass extends
Superclass
```

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

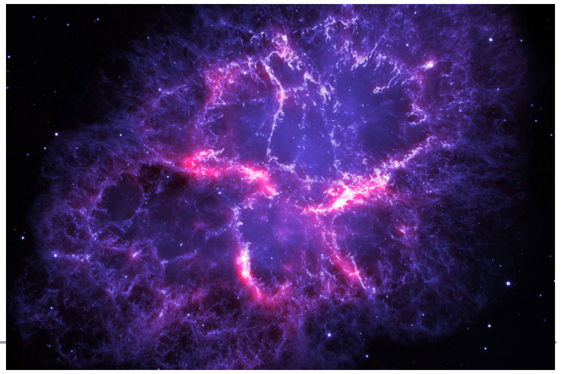
HTTP://PRESSTOREPEAT.TUMBLR.COM
LOIS LOIS LOIS LOIS. MOM MOM MOM MOM
MOMMY MOMMY MOMMY MOMMY . MOMMA MOMMA MOMMA

extends and super Keywords

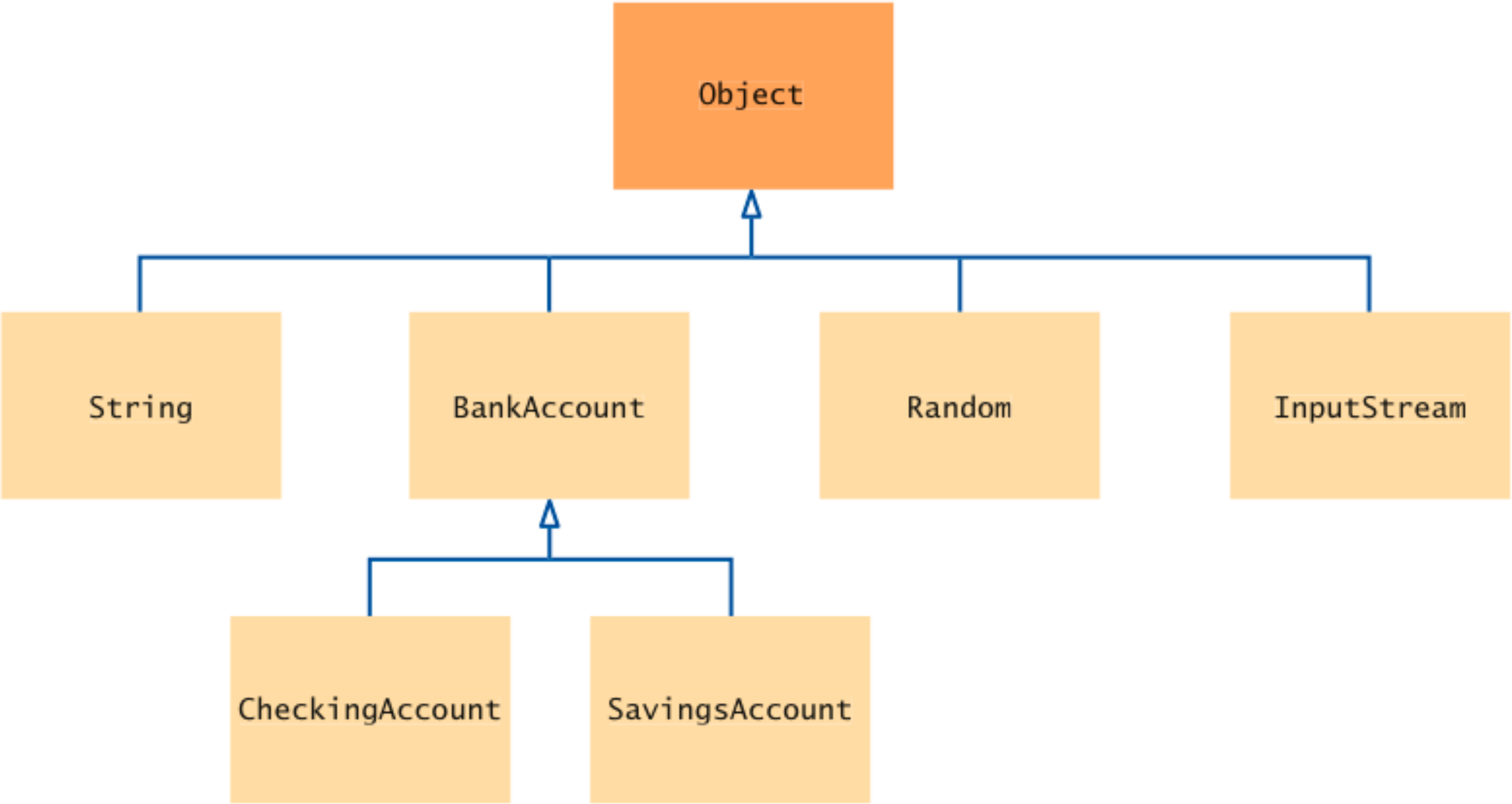
- `extends` keyword indicates that one class (subclass) inherits from other class ➔
  `public class Child extends ParentClass`

- `super` refers to the superclass and can be used in a few ways:
  - Call a superclass constructor ➔ `super(x,y);`
  - Call a superclass method➔ `super.foo();`
  - Access a superclass public/protected data field
    ➔ `super.name;`

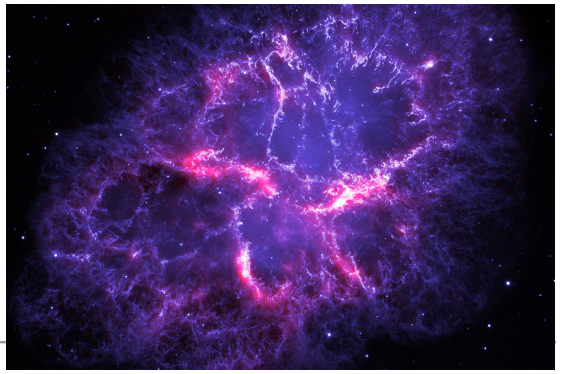
Object:
The Cosmic Superclass



- 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 Methods

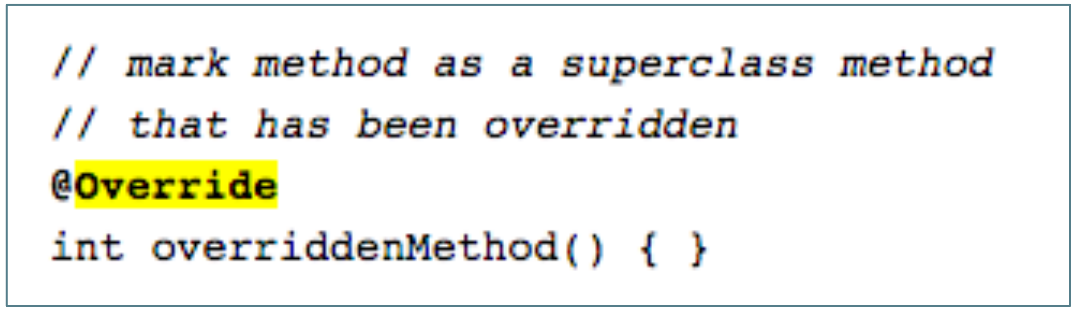
- Subclass can modify the implementation of a method defined in the superclass — known as method overriding
  - Same exact signature (method name and parameter types) as a method in the superclass

It’s like when I inherited my grandmother’s blueberry muffins recipe but decided to make my own changes to it (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)

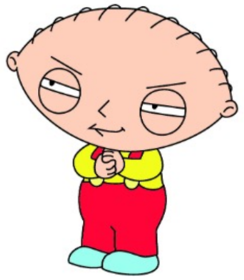


```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

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

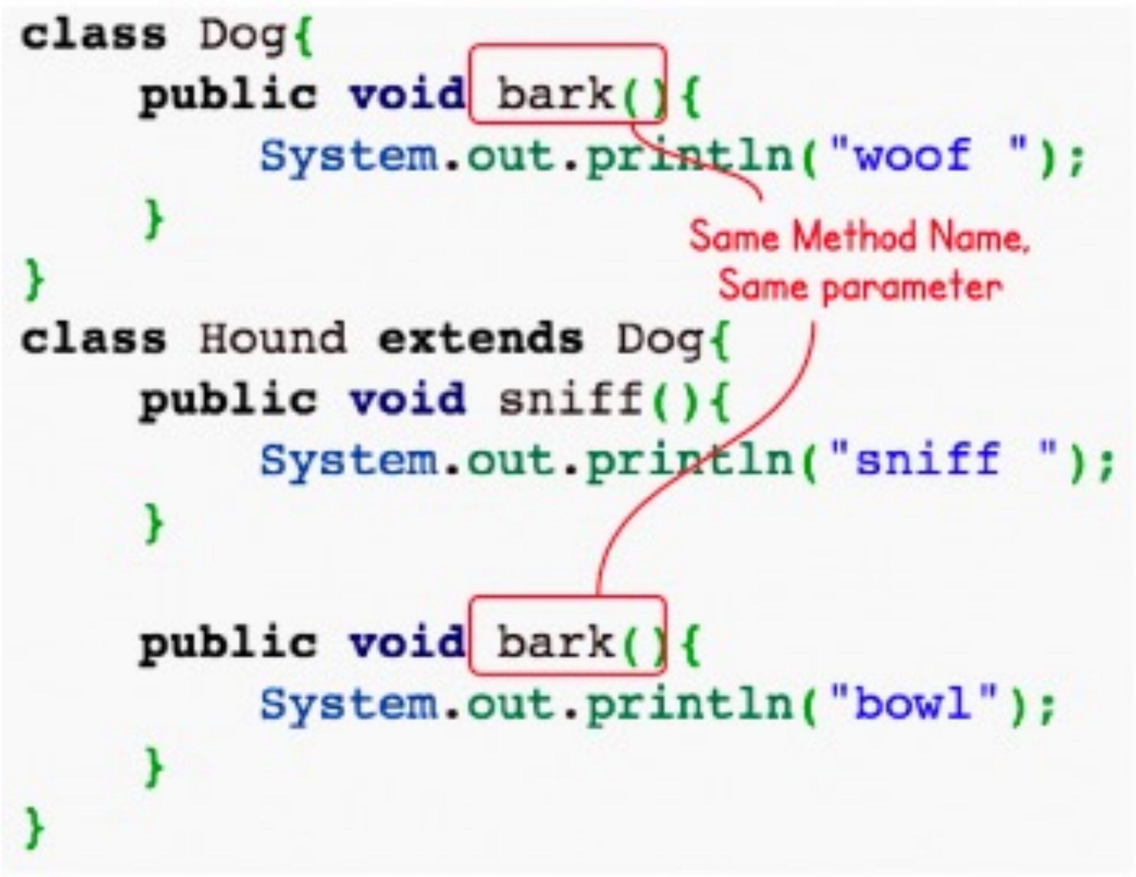
Overloading vs Overriding

- 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*



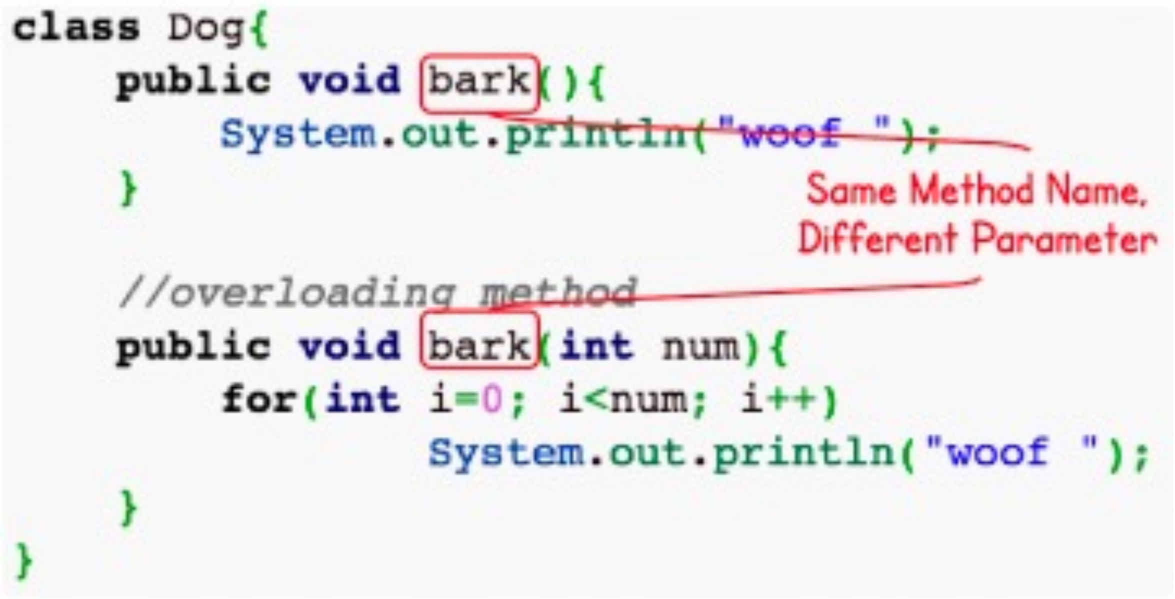
  - Useful for incorporating additional information into the methods supported by the basic API of the superclass

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

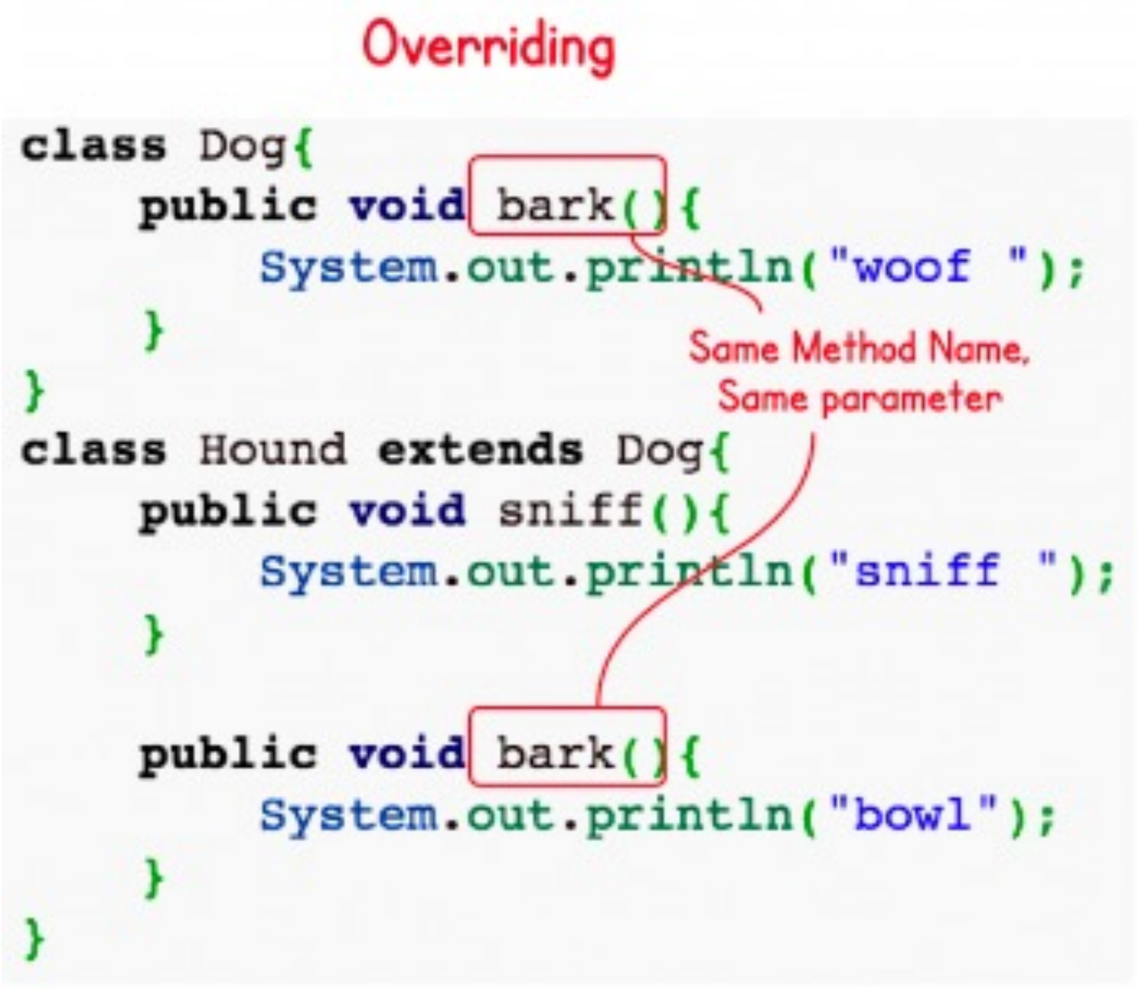


```java
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

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

Same Method Name,
Different Parameter

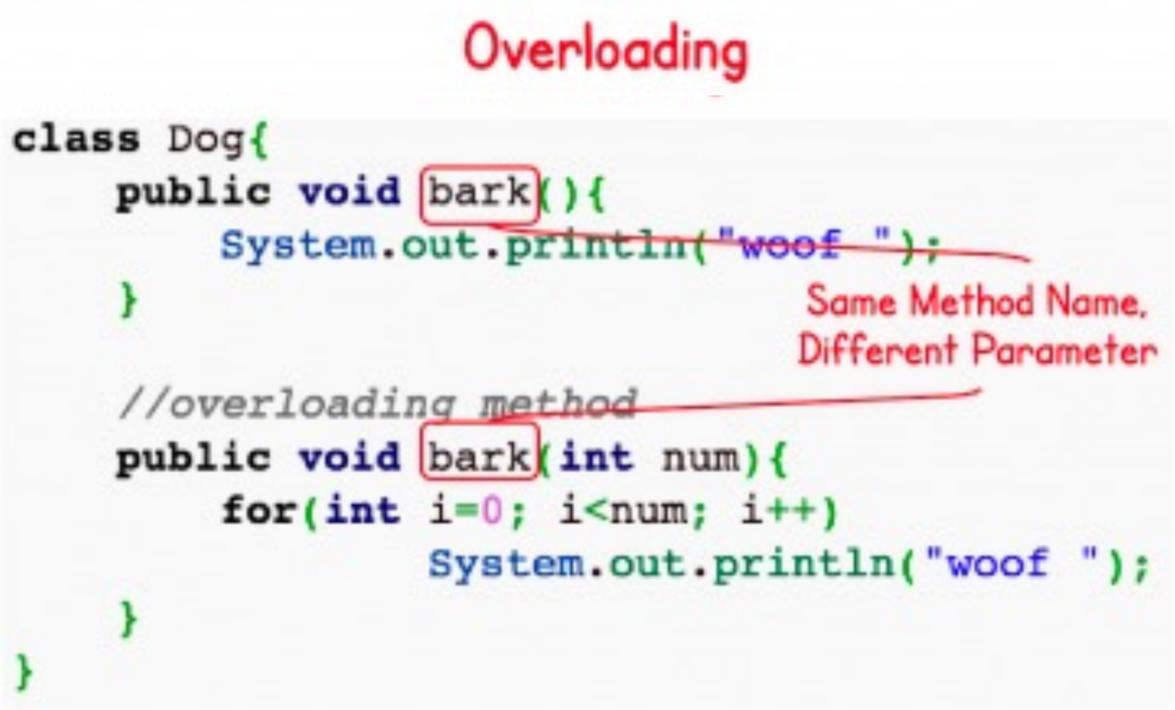
https://www.programcreek.com/2009/02/overriding-and-overloading-in-java-with-examples/

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");
    }
}
```



Overloading

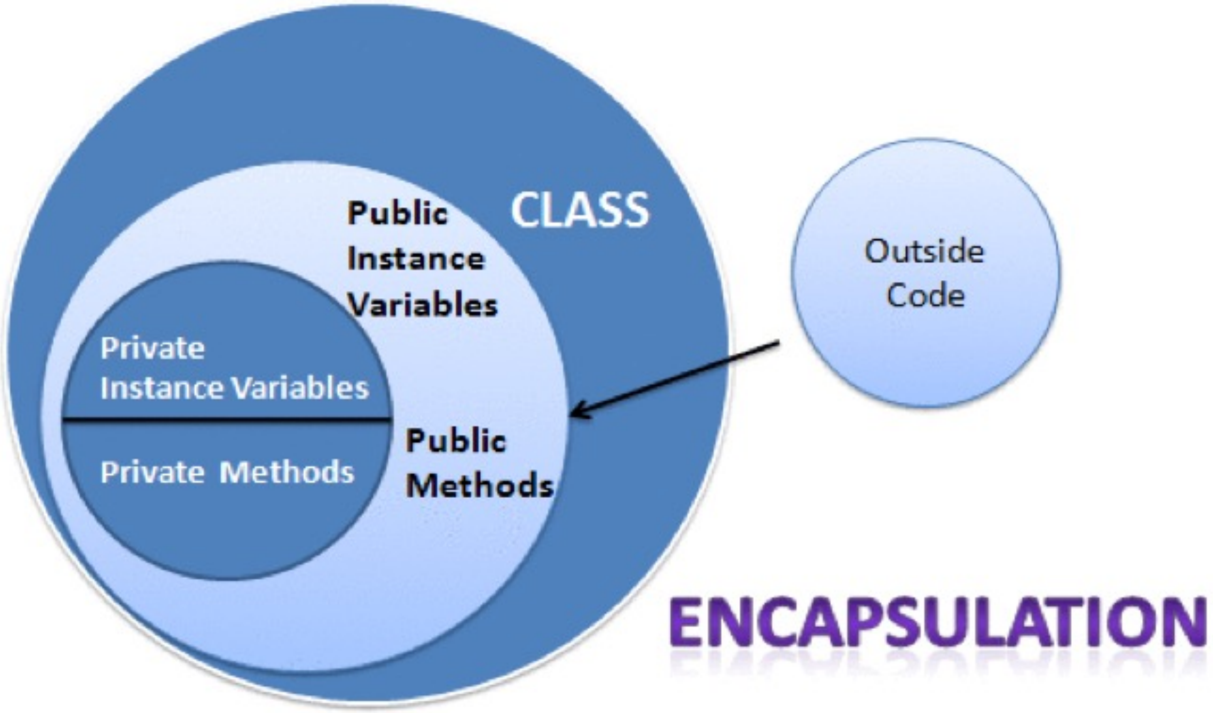
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

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

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

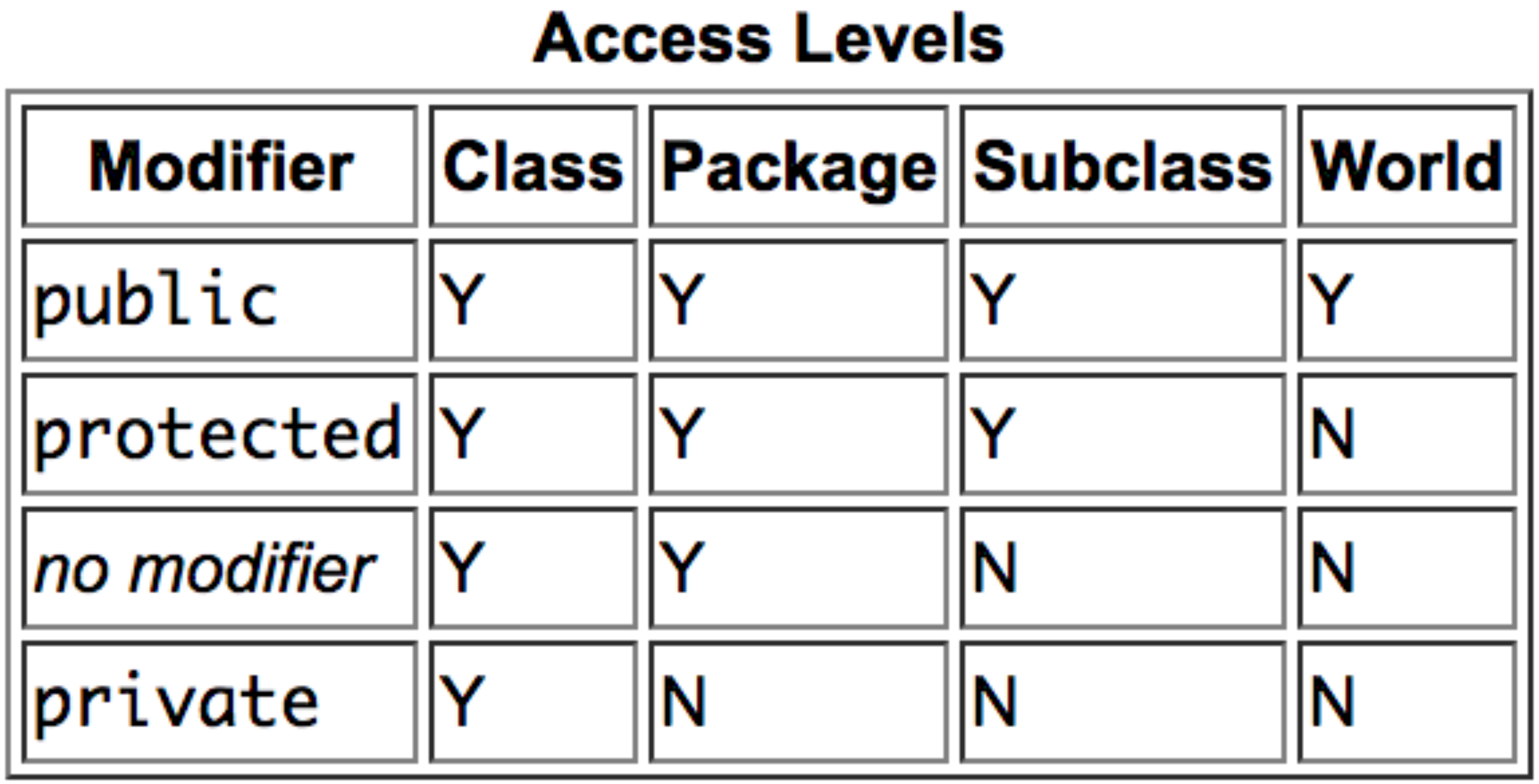
The Four Pillars
OOP
Abstraction
Inheritance
Polymorphism
Encapsulation
✓

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



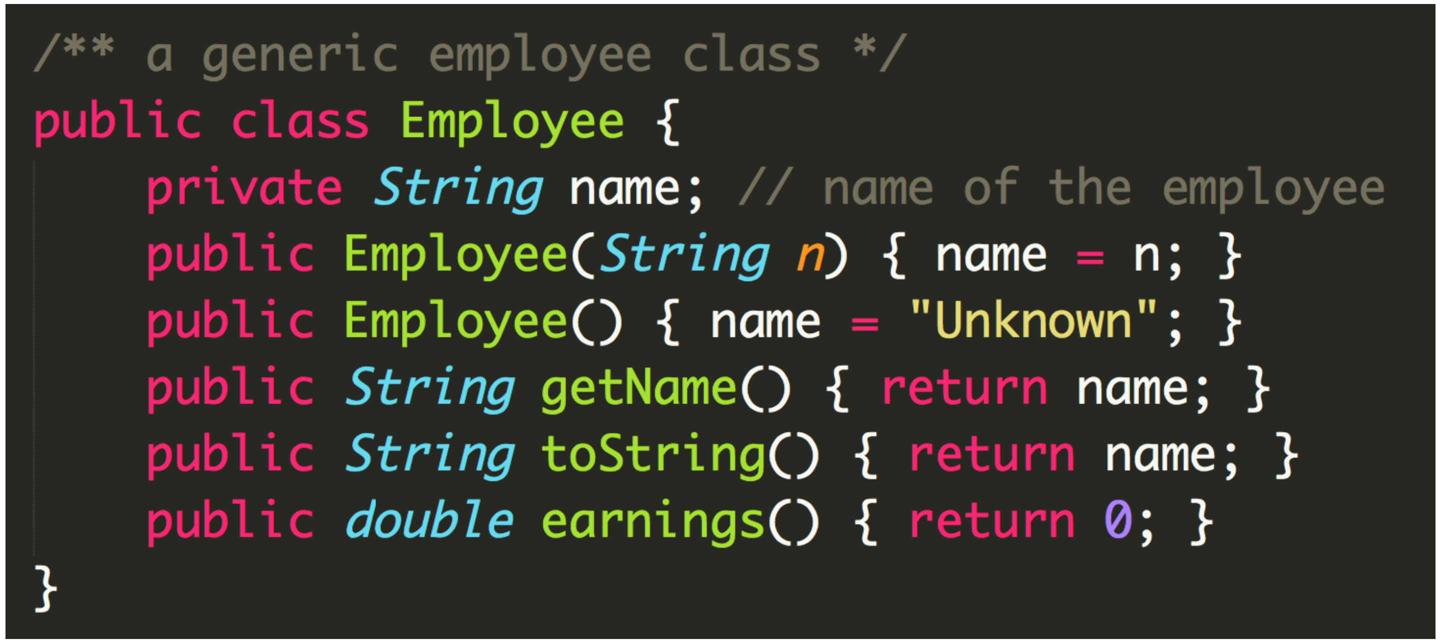
Access Levels

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

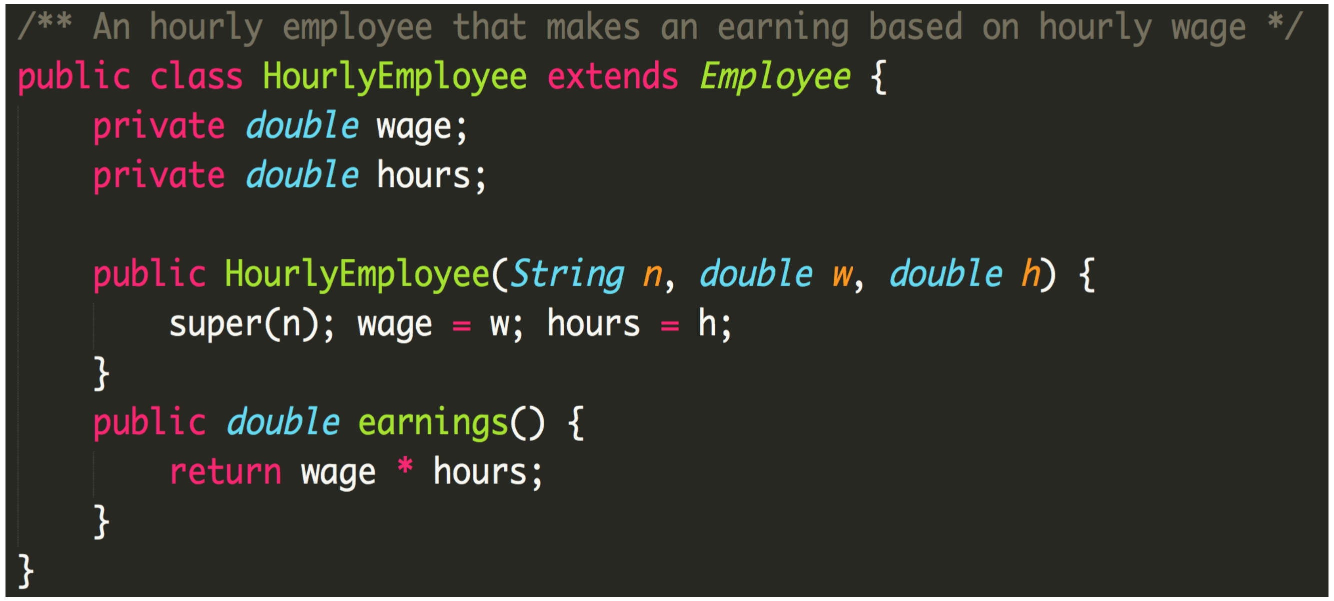
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Employee example

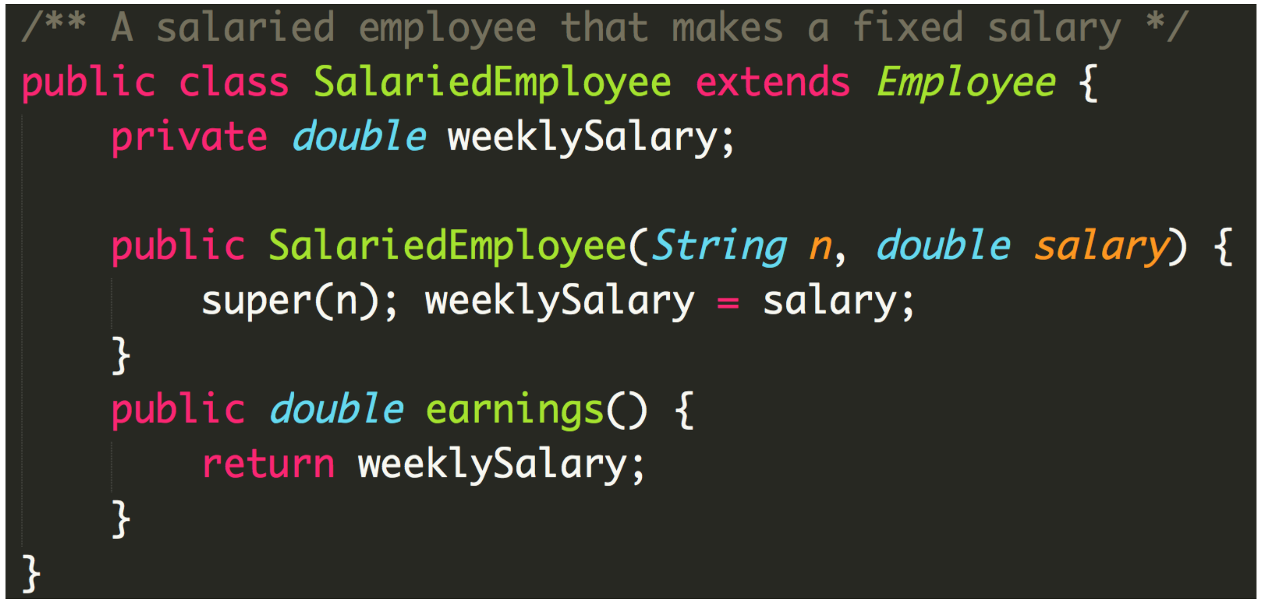
[Source code will be available on Canvas]



```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; }
}
```



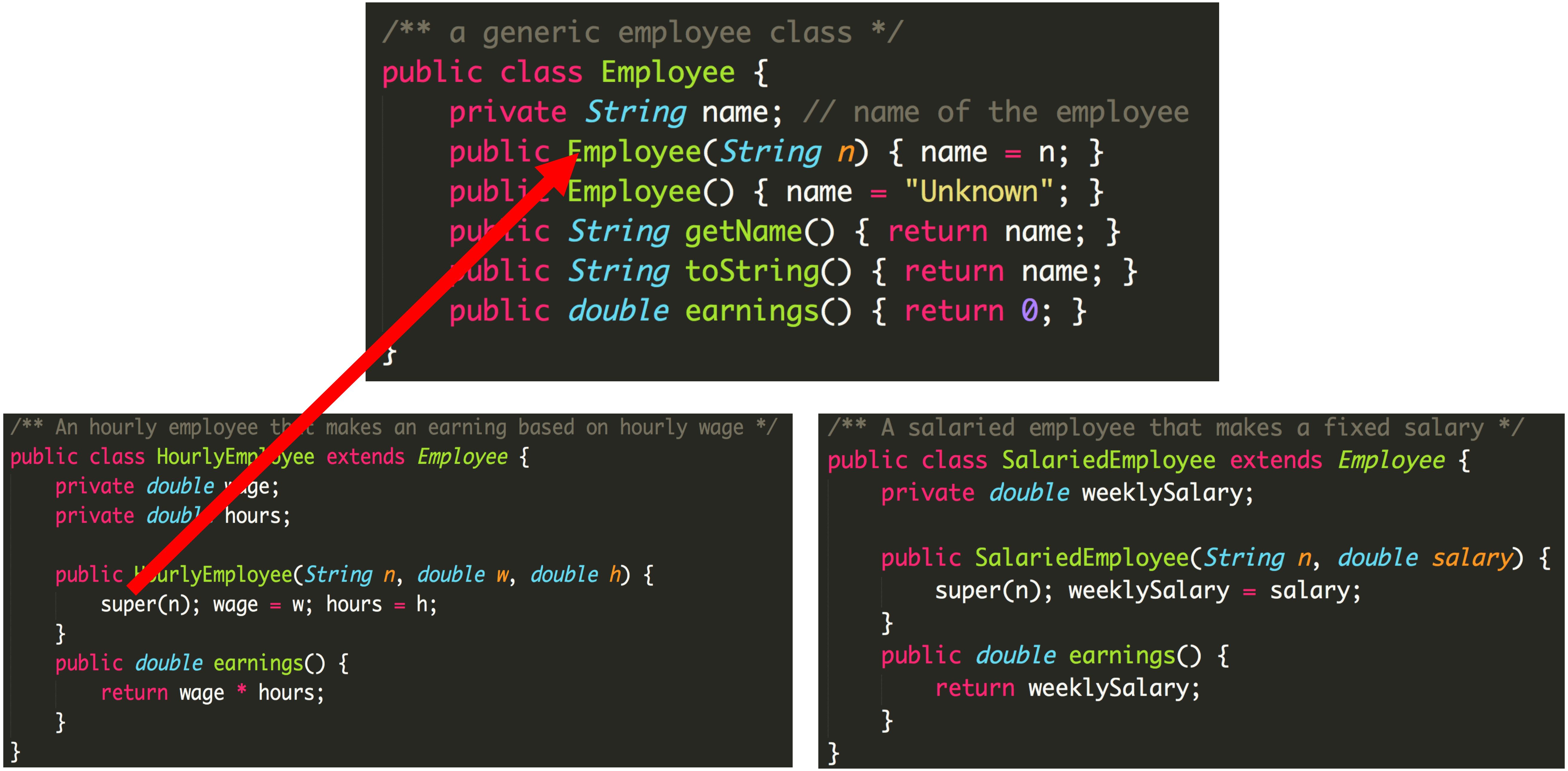
```java
/** An hourly employee that makes an earning based on hourly wage */
public class HourlyEmployee extends Employee {
    private double wage;
    private double hours;

    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 will be 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;
    }
}
```