UBC OKANAGAN

COSC 121
Computer Programming II

OOP: *Revision*

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

The Basics

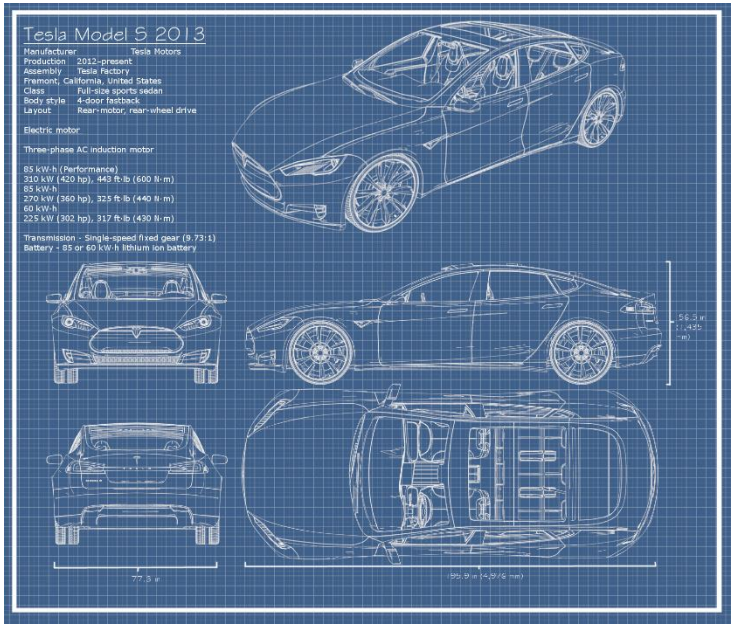
What are ‘software’ objects?

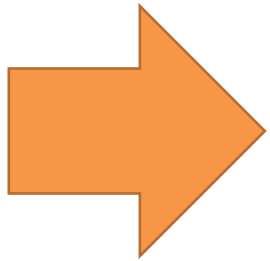
- In a Java program, objects represent **entities in the real-world**
  - Each object has it’s **own space in the memory** to save information about this object.

What entities (objects) do you see in this game?

Coding with objects

- **How are objects created in the real-world?**
  - TWO PHASES. Example: **Cars**.





Phase 1: Blueprint

- **Attributes**
- **Behaviour (Actions)**

Phase 2: Construction

In Java, all objects of a design have the same actions and attributes (although the attribute values can be different).

Phase 1: Designing Objects

- A ***class*** represents the ***blueprint*** of a group of objects of the *same type*.

- This class defines the ***attributes*** and ***behaviors*** for objects.
  - **Attributes**
    - defined as variables inside our class
      - We call them "**instance variables**"
  - **Behavior (actions)**
    - defined as **methods** inside our class

```
String name
double weight
int x, y
```

Phase 1: Designing Objects

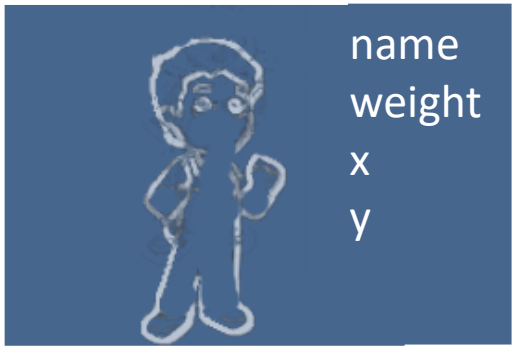
- **Example**: the Farmer class

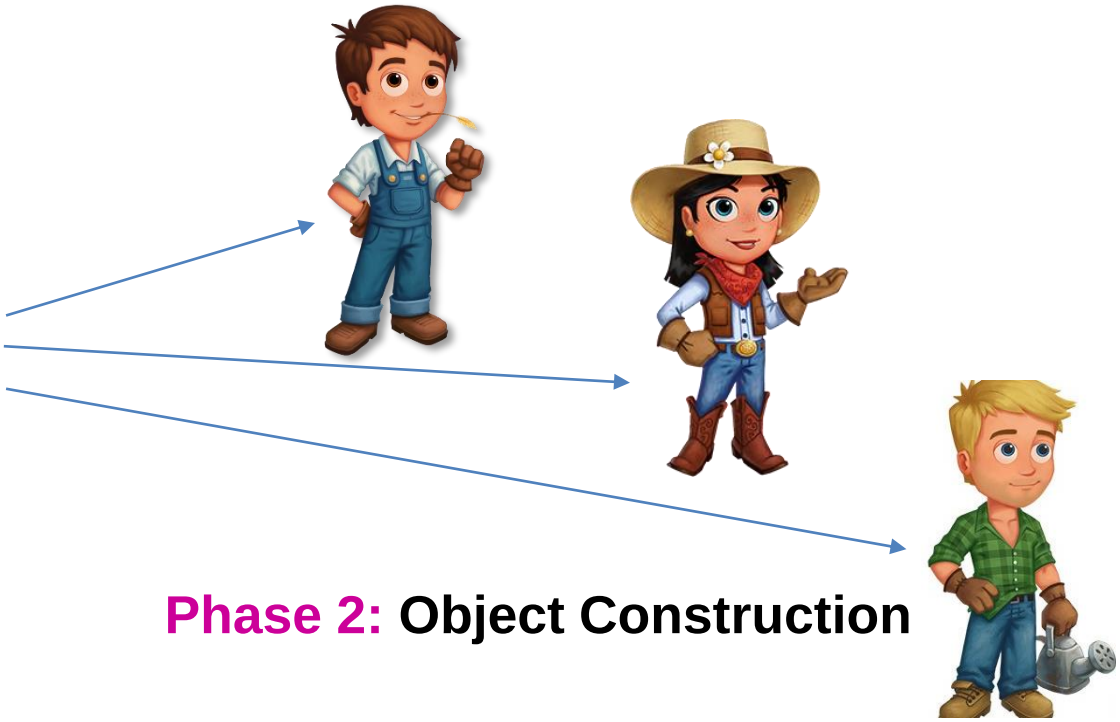
```
class Farmer {
      //instance variables (attributes)
      String name;
      double weight;
      int x, y;

      //methods (actions)
      public void moveUp()    {y++;}
      public void moveDown()  {y--;}
      public void moveRight(){x++;}
      public void moveLeft()  {x--;}
      public void moveTo(int a, int b){
         x = a;     y = b;
      }
}
```

Phase 2: Creating and Using Objects

- Next, we need to create objects based on our class





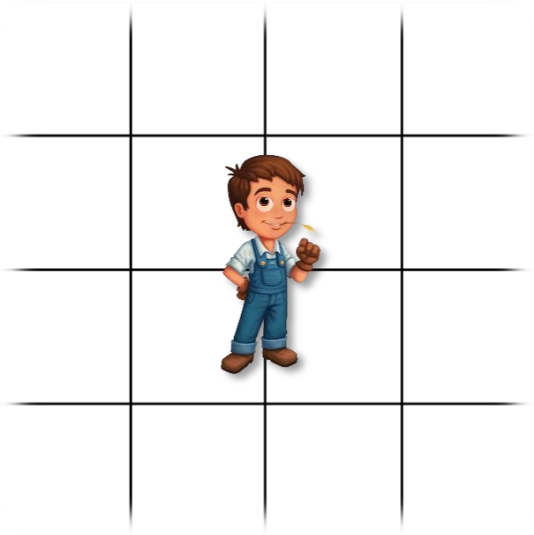
Phase 1: Farmer Class

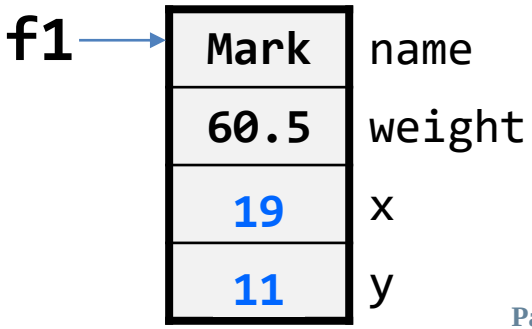
Phase 2: Object Construction

Using the updated design

- Using the **new** keyword
- We will do this inside the **main** method

```
public class FarmerTest {
  public static void main(String[] args) {
      Farmer f1 = new Farmer();
      f1.name = "Mark";
      f1.weight = 60.5;
      f1.x = 20;
      f1.y = 10;
      f1.moveRight();
      f1.moveDown();
      f1.moveTo(19,11);
  }
}
```





Creating several objects

```
public class FarmerTest {
  public static void main(String[] args) {
      Farmer f1 = new Farmer();
      Farmer f2 = new Farmer();
      Farmer f3 = new Farmer();
      ... //change attribute values for f1,f2,f3
  }
}
```

f1		f2		f3	
Mark	name	Jessa	name	John	name
60.5	weight	51.4	weight	71.2	weight
20	x	17	x	5	x
10	y	13	y	19	y

Each object has its own memory space

Default values

- Data fields (object attributes or instance variables) can be of the following types:
  - **primitive**
    - e.g., `int`, `double`, etc
    - **Default values**:
      - **0** for a numeric type,
      - **false** for a **boolean** type, and
      - **\u0000** for a **char** type.
  - **reference types**.
    - e.g., String, arrays, or other class types.
    - **Default values**:
      - **`null`**, which means that the data field does not reference any object.

Constructors

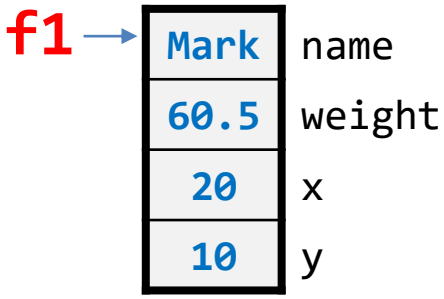
- Constructors play the role of **initializing objects**.
- Constructors are a **special kind of method.**
- They have 3 peculiarities:
  - Constructors must have the **same name as the class itself**.
  - Constructors **do not have a return type** -- not even void.
  - Constructors are invoked using the **`new`** operator **when an object is created**.

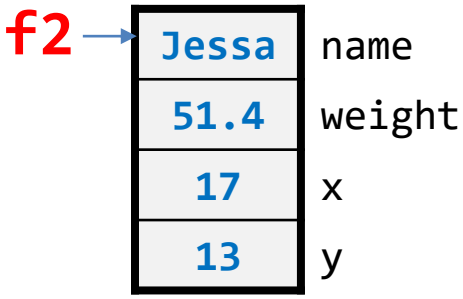
Constructors: Example

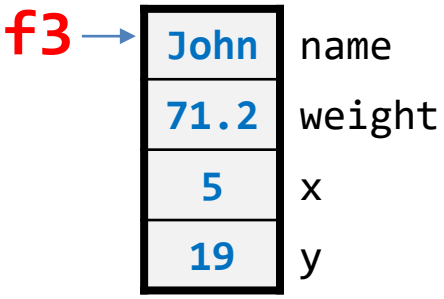
```
class Farmer {
      //instance variables
      String name;
      double weight;
      int x, y;
      //constructors
      Farmer(String aName, double aWeight, int x1,int y1){
         name = aName;
         weight = aWeight;
         x = x1;
         y = y1;
      }
      //methods
      public void moveUp()    {y++;}
      public void moveDown()  {y--;}
      public void moveRight(){x++;}
      public void moveLeft()  {x--;}
      public void moveTo(int a, int b) { x = a; y = b; }
}
```

Constructors: Example

```
public class FarmerTest {
   public static void main(String[] args) {
       Farmer f1 = new Farmer("Mark", 60.5, 20, 10);
       Farmer f2 = new Farmer("Jessa", 51.4, 17, 13);
       Farmer f3 = new Farmer("John", 71.2, 5, 19);
   }
}
```







The Default Constructor

- A **default constructor** is provided automatically only if no constructors are explicitly defined in the class.
- It sets the attributes to their default values:
  - String → null
  - Numeric → zero
  - Boolean → false

- In the previous example, the programmer included a four-argument constructor, and hence the default constructor was not provided.

More on Basic OOP

In this section…

- *public* / *private* Visibility Modifiers
- Data Field Encapsulation
- *this* keyword
- *static* modifier
- Passing Objects to Methods
- Array of Objects

`public/private` Visibility Modifiers

- **Access modifiers** are used for controlling levels of access to class members in Java:

  `public`,

  - The class, data, or method is visible to any class in any package.

  `private`:

  - The data or methods can be accessed only by the declaring class.

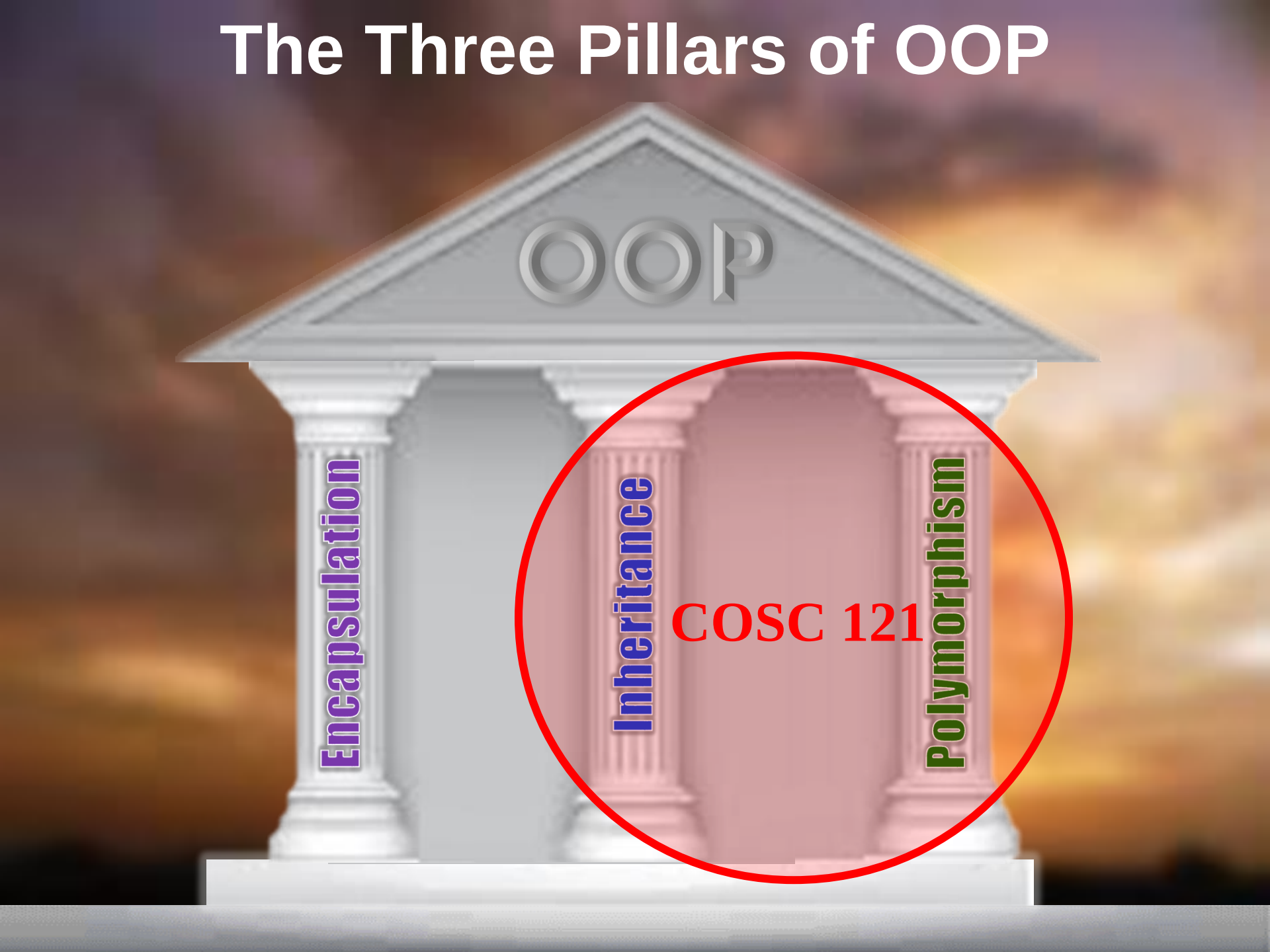
Data Field Encapsulation

- It is preferred to declare the data fields **private** in order to
  - protect data from being mistakenly set to an invalid value
    - e.g., c1.radius = -5 //this is logically wrong
  - make code easy to maintain.

- You may need to provide two types of methods:
  - A **getter method** (also called an **'accessor'** method):
    - Write this method to make a private data field accessible.
  - A **setter method** (also called a **'mutator'** method)
    - Write this method to allow changes to a data field.

- Usually, constructors and methods are created public unless we want to “hide” them.

The Three Pillars of OOP



The `this` Keyword

- `this` is used inside a method or a constructor to refer to the current object, whose method/constructor is being called.
- Use `this` to avoid naming conflicts in the method/constructor of your object.

- **For what 'items' can I use `this`?**
  - To reference class members within the class.
    - Class members can be referenced from anywhere within the class
    - Examples:
      - this.x = 10;
      - this.amethod(3, 5);
  - To enable **a constructor to invoke another constructor** of the same class.
    - A constructor can only be invoked from within another constructor
    - Examples:
      - this(10, 5);

Practice

Code these two classes in Java

- Make sure that no invalid values are assigned to the attributes.
- Use the “this” keyword whenever possible.

Circle
-radius: double -color: String -filled: Boolean
+Circle() +Circle(radius: double) +Circle(radius: double, color: String, filled: boolean) +getters/setters for all attributes +getArea(): double +getPerimeter(): double +toString(): void

Rectangle
-width: double -height: double -color: String -filled: Boolean
+Rectangle() +Rectangle(width: double, height: double) +Rectangle(width: double, height: double, color: String, filled: boolean) +getters/setters for all attributes +getArea(): double +getPerimeter(): double +toString(): void

- The - sign indicates private modifier
- The + sign indicates public modifier

Solution

```
public class Circle {
    // attributes
    private String color;
    private boolean filled;
    private double radius;
    // constructors
    public Circle() { this(1,"Black",true); }
    public Circle(double radius) { this(radius, "Black", true);}
    public Circle(double radius, String color, boolean filled) {
        setRadius(radius);
        setColor(color);
        setFilled(filled);
    }
    // methods
    public double getArea() {return Math.PI*radius*radius;}
    public double getPerimeter(){return 2*Math.PI*radius;}
    // setters/getters
    public String getColor()              { return color;}
    public void setColor(String color)    { this.color=color;}
    public boolean isFilled()             { return filled;}
    public void setFilled(boolean filled){ this.filled=filled;}

    public double getRadius()             { return this.radius;}
    public void setRadius(double radius){
        if(radius >= 0) this.radius = radius;
    }
    // to string
    public String toString() {
        return "radius="+radius+",color="+color+",filled="+filled;
    }
}
```

```
public class Rectangle {
    // attributes
    private String color;
    private boolean filled;
    private double width,height;
    // constructors
    public Rectangle() { this(1,1,"Black",true);}
    public Rectangle(double width,double height) { this(width, height,"Black",true); }
    public Rectangle(double width, double height, String color, boolean filled) {
        setWidth(width);   setHeight(height);
        setColor(color);
        setFilled(filled);
    }
    // methods
    public double getArea() {return width * height;}
    public double getPerimeter() {return 2 * (width + height);}

    // setters/getters
    public String getColor()              {return color;}
    public void setColor(String color)    { this.color = color;}
    public boolean isFilled()             {return filled;}
    public void setFilled(boolean filled) { this.filled = filled;}
    public double getWidth()              {return width;}
    public void setWidth(double width)    { if(width >= 0) this.width = width;}
    public double getHeight()             {return height;}
    public void setHeight(double height) {if(height >= 0) this.height = height;}

    // to string
    public String toString() {
        return "color="+color+", filled="+filled+", width="+width+", height="+height;
    }
}
```

Note how much code redundancy we have!
Inheritance can solve this!

The `static` Modifier

- **`Static` class members:**
  - Static variables (also known as **class variables**) are **shared** by all the instances (objects) of the class.
  - Static methods (also known as **class methods**) are not **tied to a specific object** (they carry out a general function)
    - Example: Math.max(3, 5);

- **Remember that, *unlike* static class members:**
  - Instance variables belong to a specific instance (i.e. object).
  - Instance methods are invoked by an instance of the class

Passing Objects to Methods

- Remember: Java uses pass-by-value for passing arguments to methods:
  - **Passing primitive variable**:
    - the value is passed to the parameter, which means we will have two distinct primitive variables.
    - i.e. changes that happens inside the method do not influence the original variable.
  - **Passing reference variable**:
    - the value is the reference to the objects, which means the two references (the argument and the parameter) will refer to the **same object**. **Changes that happen inside the method using the passed reference are applied to that object.**

Example

```
public static void main(String[] args) {
  int x = 0;
  Circle c = new Circle(0);

  System.out.printf("Before foo: x is %d, c.radius is %.0f\n",x,c.getRadius());
  foo(x, c);
  System.out.printf("After foo: x is %d, c.radius is %.0f\n",x,c.getRadius());
}

public static void foo(int a, Circle b) {
  a = 7;
  b.setRadius(7);
}
```

```
class Circle {
  private double radius;
  public Circle(double radius){
    setRadius(radius);
  }
  public double getRadius() {
    return radius;
  }
  public void setRadius(double r){
    if (radius >= 0)
        radius = r;
  }
}
```

Output:

```
Before foo: x is 0, c.radius is 0
After foo: x is 0, c.radius is 7
```

Note how the primitive variable x didn't change while the object c has changed

Array of Objects

To create an array of objects, you need to follow two steps:

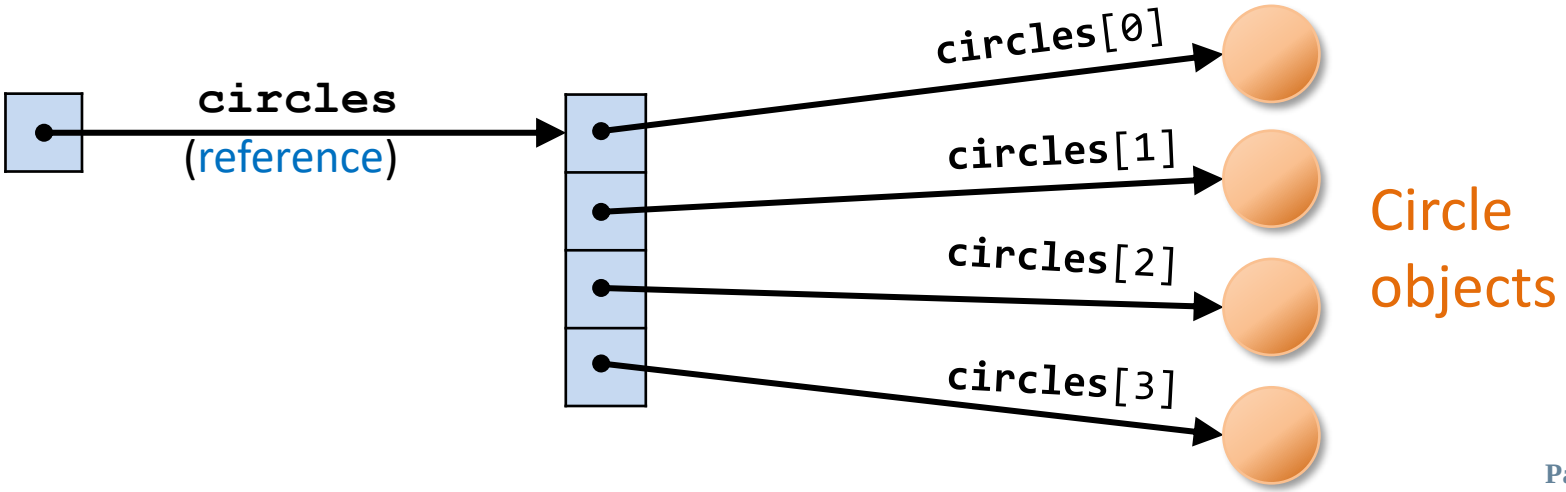
1. Declaration of reference variables:
   - You can create an array of objects, for example,

```
Circle[ ] circles = new Circle[4];
```

   - An array of objects is actually an array of reference variables. We don't have any objects created yet.

2. Instantiation of objects:
   - To initialize **circles**, you can use a **for** loop like this one:

```
for (int i = 0; i < circles.length; i++)
        circles[i] = new Circle();
```



Array of Objects, cont.

- You may then invoke any method of the Circle objects using a syntax similar to this:
  - `circles[1].setRadius(1);`
- , which involves two levels of referencing:
  - **circles** references to the entire array, and
  - **circles[1]** references to a Circle object.

Example

```
//create circles array
Circle[ ] circles = new Circle[4];
for (int i = 0; i < circles.length; i++)
  circles[i] = new Circle(i);

//randomize radius
for (int i = 0; i < circles.length; i++)
  circles[i].radius = Math.random()*10;

//print areas of all circles
for (int i = 0; i < circles.length; i++)
  System.out.println(circles[i].getArea());
```

```
class Circle {
   private double radius;
   public Circle(double radius){
      setRadius(radius);
   }
   public double getRadius() {
      return radius;
   }
   public void setRadius(double r){
      if (radius >= 0)
         radius = r;
   }
   public double getArea() {
      return radius*radius * Math.PI;
   }
}
```