Computer Programming

Introduction to Classes and Objects

Dr. Firas Moosvi

Computer Science

University of British Columbia

UBC OKANAGAN

Acknowledgement: Slides (provided by Dr. Abdallah Mohammed) mainly rely on the materials prepared by Y. D. Liang for the textbook “Introduction to Java Programming, 10th Ed., Pearson Edu. Inc.”.

Previously…

- Java constructs and data types
  - **Variables**: `int`, `double`, `boolean`, …
  - **Displaying output**: `System.out.println()`
  - **Java Library**: `Scanner`, `Math`, `Character`
  - **Selection**: `if`, `if-else`, `switch`
  - **Loops**: `while`, `for`
  - **Methods**
- You used these concepts for simple, interesting programs
  - **Calculations**: area of a shape, unit conversion, …
  - **String processing:** reverse, isPalindrome, count letters, …
  - **Array processing**: sum, max, min, copying, …
  - **Simple games**: guess the number, paper-scissors-rock,…
  - **Project:** Jeopardy game
- **There are more interesting problems…**

Introduction to Objects

- The world consists of objects
  - Cars, people, places, animals, flowers, houses, chairs, etc
- We develop software to address issues in real-world
  - It makes sense to design software in terms of objects

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.

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

What are objects?

- Let's look at one of these objects: the **farmer object**
- Any object:
  - has **attributes**: *define what the object is*
  - can perform **actions**: *define what the object can do*

*We build our software with **objects** that **work together** in order to **achieve the required goal***

Attributes

- …

Actions

- …

Object interaction

Attributes:

- **name**: Mark
- **weight**: 60.5 kg
- **location**: (20, 10)

Actions

- **Move right**
- **Move left**
- **Move up**
- **Move down**
- **Feed animal**

Other Examples of OOP

Games: COSC 111 Project

Objects:

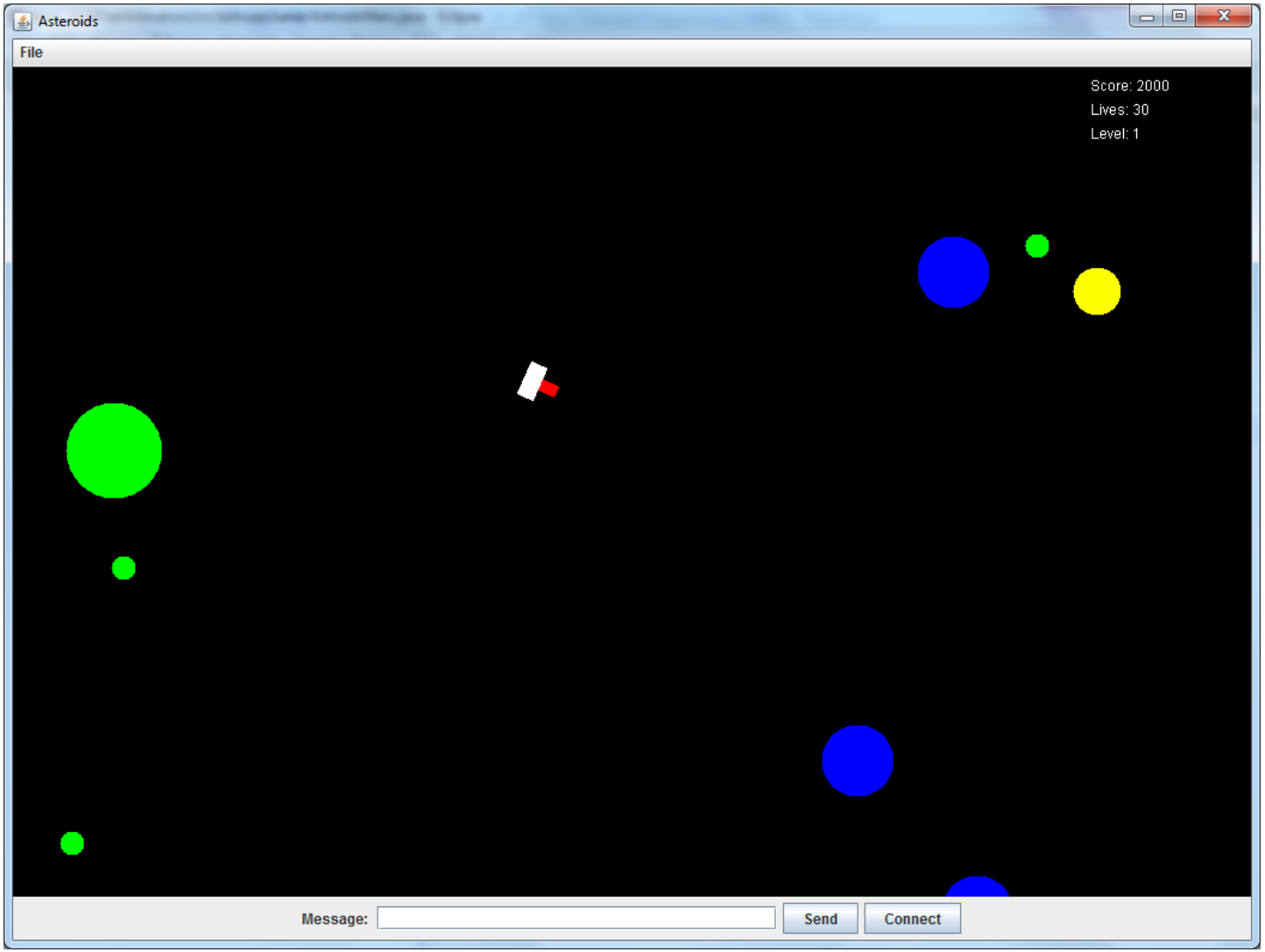
- Players
- Game
- GUI

Other Examples of OOP

Games: Asteroid

Objects:

- Spaceship
- Asteroids
- Bullets
- Game
- GUI

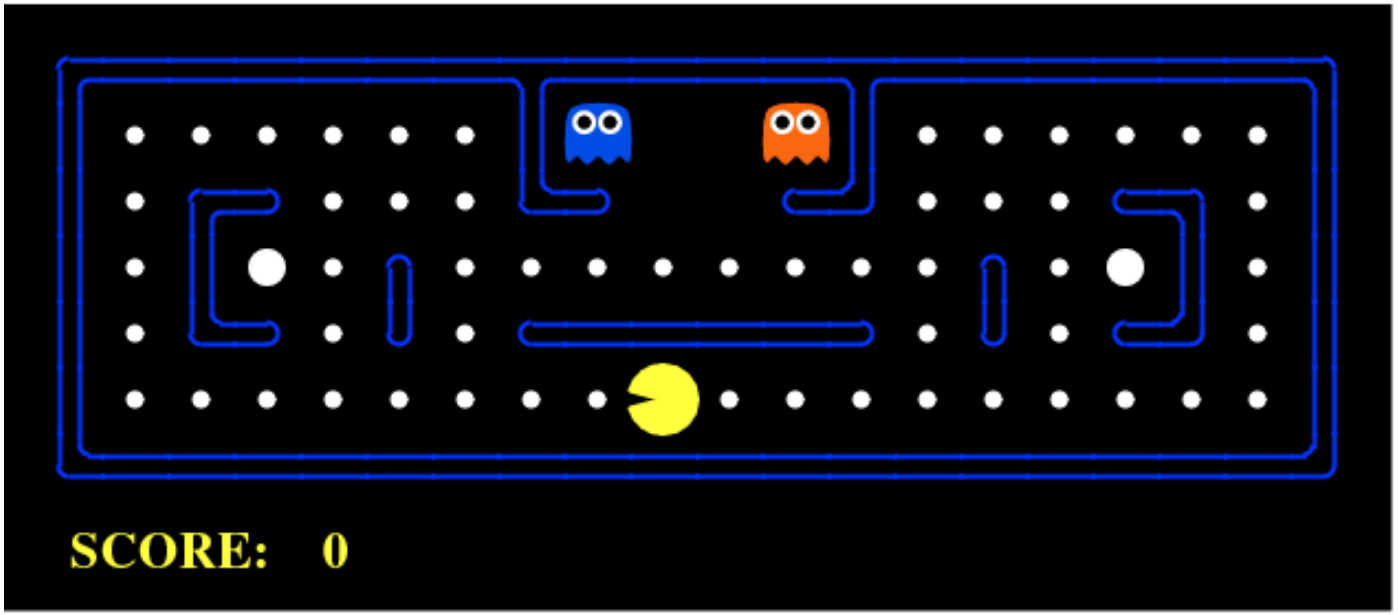


Other Examples of OOP

Games: Pacman

Objects:

- Player
- Ghosts
- Power Pills
- Food items
- Game
- GUI

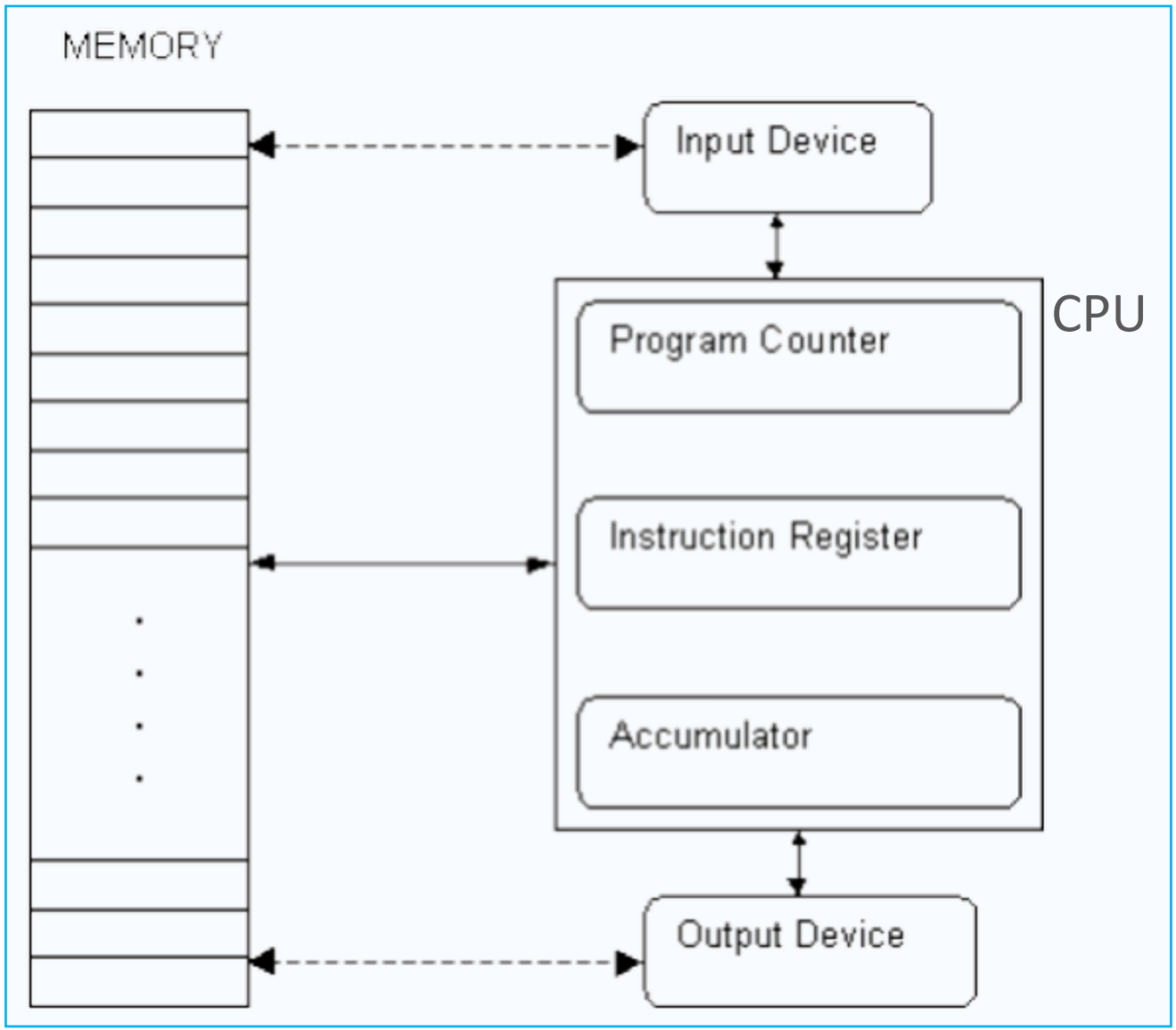


Other Examples of OOP

Computer simulator

Objects:

- Computer
- CPU
- Registers
- I/O devices
- Memory



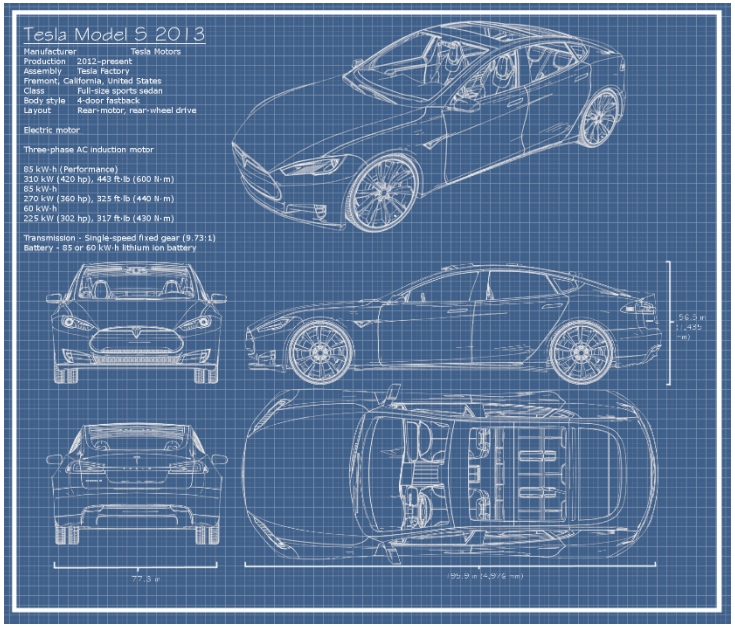
Today…

Aim: Learn how to

- design objects,
- construct objects based on our design, and
- use objects

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).

Learn by Example: The Farmer

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
      - Will discuss them later today!

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

Phase 1: Designing Objects, cont'd

- **Example**: the Farmer class

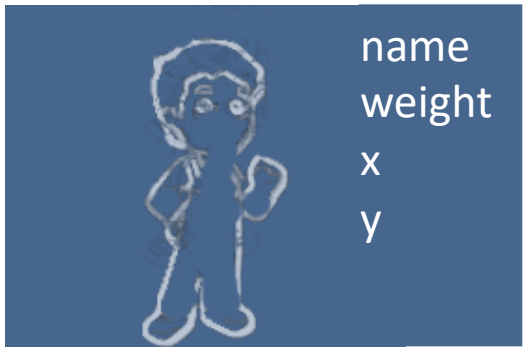
```
class Farmer {
      //instance variables (attributes)



      //methods (actions)

}
```

Famer Blueprint

Phase 1: Designing Objects, cont'd

- **Example**: the Farmer class

```
class Farmer {
      //instance variables (attributes)
      String name;
      double weight;
      int x, y;
      //methods (actions)
      //will add them later
}
```

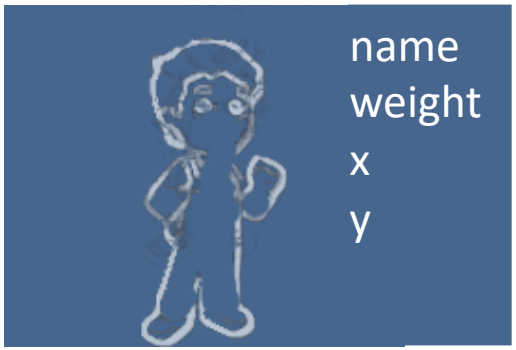


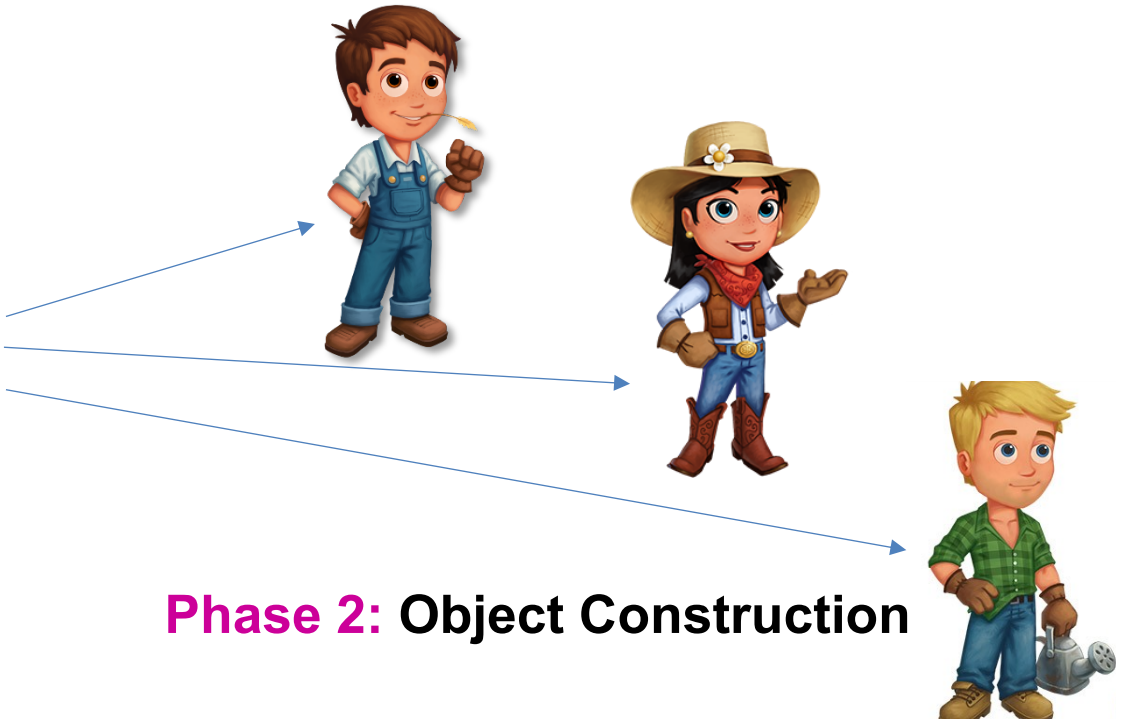
Famer Blueprint

- *Remember*: a class is just a blueprint for creating object
  - Classes store no data an perform no actions for an object
- We need to create objects now!

Phase 2: Creating and Using Objects

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





Phase 1: Farmer Class

Phase 2: Object Construction

Phase 2: Creating objects using `new`

- 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();
  }
}
```

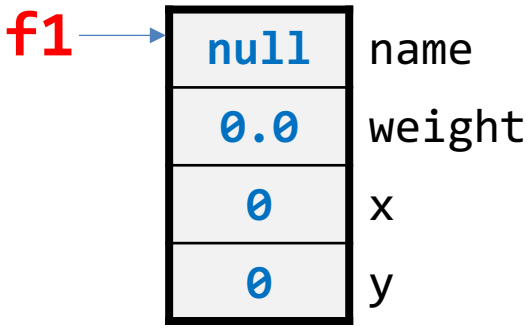
A unique identity for this object

Default values for the attributes

f1

- **name**: null
- **weight**: 0.0 kg
- **location**: (0, 0)

Each object has its own space in ***memory***



Phase 2: Creating and using objects

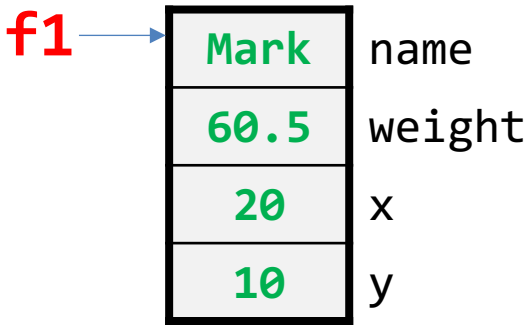
- Using the **new** keyword
- We will do this inside the **main** method

```
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;
  }
}
```

After an object is created, its members can be accessed using the ***dot operator*** (.)

f1

- **name: Mark**
- **weight: 60.5 kg**
- **location : (20 10)**



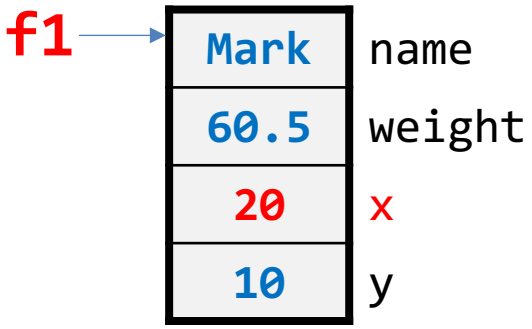
Phase 2: Creating and using objects

- 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;
      System.out.printf( f1.x );
  }
}
```

f1

- **name**: Mark
- **weight**: 60.5 kg
- **location**: (20, 10)



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.

Clicker Question

Consider the Farmer class. Which of the following is a valid instantiation of an object of the type Farmer?

```
class Farmer {
        String name;
        double weight;
        int x, y;
}
```

A. `Farmer f = Farmer();`

B. `Farmer = new Farmer();`

C. `Farmer f = new Farmer();`

D. `Farmer f = new Farmer("Mike");`

E. None of the above

Clicker Question

what is the value of the weight

and name of the object **f** ?

```
class Farmer {
    String name;
    double weight;
}
```

```
public static void main(String[] args){
    Farmer f = new Farmer();
    name = "Mark";
    weight = 30;
}
```

A. `"Mark", 30`

B. `null, 0`

C. `error`

Practice

Assuming that we are developing a farm game where farmers need to feed their animals. An animal must be fed otherwise it becomes dead.

Create a class **`Cow`**:

- A cow has the attributes
  - **nickname** (`String`)
  - **stomach** (`int`) that represents the percentage (0 to 100) of food in cow's stomach
  - **isFull** (`boolean`) that indicates whether the cow is full

Write a program to

- Create two Cow instances (objects) set their attributes to any values
- Display the information of the two Cow instances.

Adding Behaviour to Our Design

Updating Our Design

- The ***blueprint*** of a group of objects of the *same type* is represented by a ***class***.

- The 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

Attributes

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

Actions

Move right
Move left
Move up
Move down

Adding Behaviour to Our Design

- **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--;}
}
```

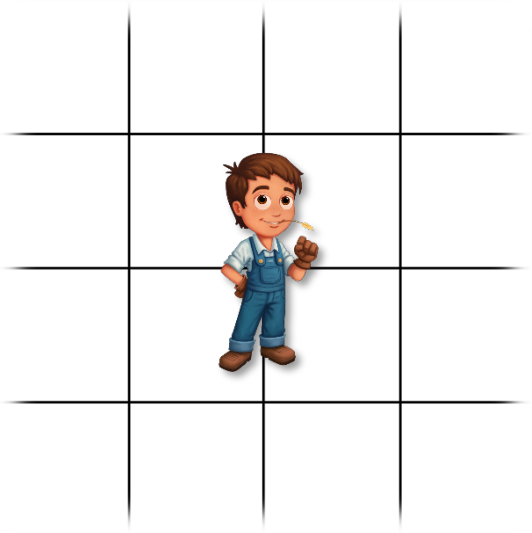
Note: methods inside a class can reference instance variables without (.)

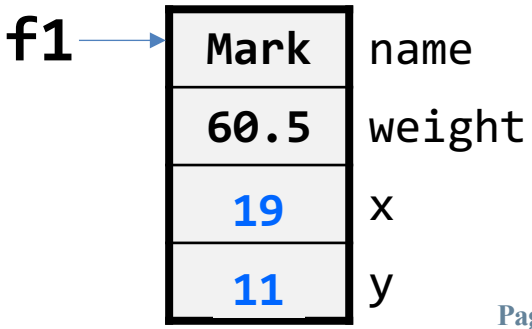
Using the updated design

- 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);
  }
}
```

Once implemented, the farmer will learn this new action (see next slide)





Adding Behaviour to Our Design

- **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;
      }
}
```

Practice

(1) Modify your **`Cow`** class to include the following two methods:

- `void eat(int amount)` that increments food in `stomach` by the given amount.
- `void say(String msg)` that causes the animal to display the given `msg` on the console preceded by its `nickname`.
  For example, if the nickname is "`Bolt`" and `msg` is "`Hi`", the output is
  - **`Bolt says: Hi!`**

Attributes

- String nickname
- int stomach
- boolean isFull

Methods

- eat(…)
- say(…)

(2) Modify `eat` method such that `stomach` is never larger than a 100 at which `isFull` is set to true. Also, make sure the cow can't eat anymore if it is full (i.e., `isFull = true`).

Caution

Recall that you can invoke a method directly from the Math class using *Math.methodName*, e.g., Math.random().

In the previous example, can we use **Cow.say()?**

- The answer is **no**. All the methods used before this chapter are **static methods**, which are defined using the static keyword. Static methods can be invoked directly from their class.
- However, say() is **not static.** It **must be invoked from an object** using: objectRefVar.methodName(arguments)
  - e.g., Cow c1 = new Cow; c1.say();
- More details on this later, in the section "Static Variables, Constants, and Methods."

Constructors

Another Solution

```
public class TV {
  //attributes
  private int channel, volumeLevel;
  private boolean on;
  //constructor
  public TV() {turnOn(); setChannel(1); setVolume(1); }
  //methods
  public void turnOn()   {on = true;}
  public void turnOff()  {on = false;}
  public void setChannel(int newChannel) {
    if(!on)  System.out.println("Cannot change channel. TV is off!");
    //change case below to better control channels instead of displaying an error
    else if(newChannel<1 || newChannel>120)
          System.out.println("Invalid channel value!");
    else   channel = newChannel;
  }
  public void setVolume(int newVolLevel) {
    if(!on)       System.out.println("Cannot change volume. TV is off!");
    else if(newVolLevel<1) volumeLevel = 1;
    else if(newVolLevel>7) volumeLevel = 7;
    else                   volumeLevel = newVolLevel;
  }
  public void channelUp()  {setChannel(channel + 1);}
  public void channelDown()  {setChannel(channel - 1);}
  public void volumeUp()   {setVolume(volumeLevel+1);}
  public void volumeDown()   {setVolume(volumeLevel-1);}
}
```

Constructors

- What if I want to initialize objects as I create them?
  - Example:

```
Farmer f1 = new Farmer();

f1.name = "Mark";
f1.weight = 60.5;
f1.x = 20;
f1.y = 10;
```

```
Farmer f1 = new Farmer("Mark", 60.5, 20, 10);
```

Constructors, cont'd

- 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**.

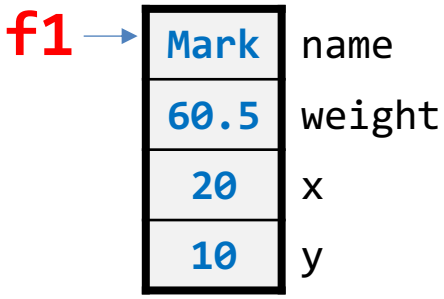
Constructors: Example

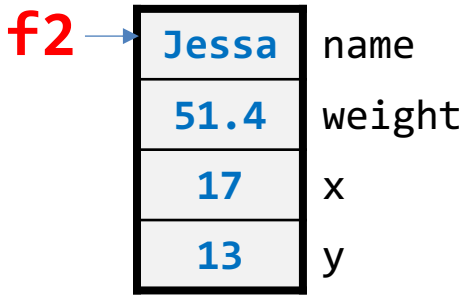
- **Example**: the Farmer class

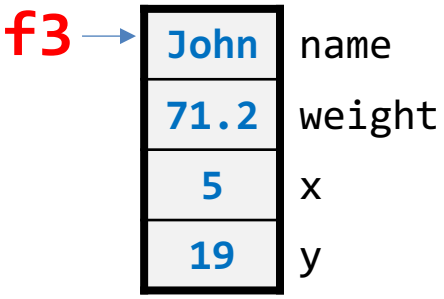
```
class Farmer {
      //instance variables
      String name;
      double weight;
      int x, y;
      //constructors
      Farmer(String aName, int aWeight, int x1, int y1){
         name = aName;
         weight = aWeight;
         x = x1;
         y = y1;
      }
      //methods
      public 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);
   }
}
```







Try this now…

```
class Farmer {
      //instance variables
      ...
      //constructors
      Farmer(String aName, int aWeight, int x1, int y1){
         name = aName;
         weight = aWeight;
         x = x1;
         y = y1;
      }
      //methods
      ...
      ...
      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();  // ERROR!! WHY??
      }
}
```

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.

Problem Fixed!!

```
class Farmer {
      //instance variables
      ...
      //constructors
      Farmer(){    //now we have a zero-arg constructor
      }
      Farmer(String fname, int fweight, int fx, int fy){
         name = fname;
         weight = fweight;
         x = fx;
         y = fy;
      }
      //methods
      ...
      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();  // No error!!
      }
}
```

Practice

1) Add two constructors to your **`Cow`** class:

- A zero-argument constructor to set the `stomach` to 50 and `nickname` to "Anonymous".
- A two-argument constructor to set the cow's `nickname` and `stomach` to given values. Make sure `stomach` doesn't get a value larger than 100.

Attributes

- String nickname
- int stomach
- boolean isFull

Methods

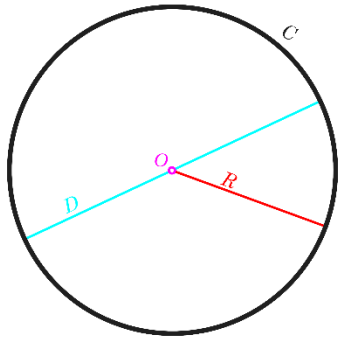
- Cow()
- Cow(…)
- eat(…)
- say(…)

Q: Should we also create a 3-arg constructor (nickname, stomach, isFull)?

Answer: NO, `isFull` should be set based on value of `stomach`.

2) Test your class by creating a Cow instance with (`stomach = 30, nickname=Bolt`), make it eat 10 food units, and then make it say something like "Hi".

Try this at home!



- Write a class **`Circle`** which has:
  - an instance variable (attribute) `double radius`.
  - a no-argument constructor that sets `radius` to 10.
  - a one-argument constructor that sets `radius` to a given value.
  - a method `setRadius` that changes the radius to a given value.
  - two methods `getArea` and `getPerimeter` that return the area and perimeter respectively.

- Test your class by creating three instances of `Circle` and invoke their different methods.

UML Notation

UML Notation

- UML stands for Unified Modeling Language
- UML diagrams are one method for representing and communicating a *model* of the software being developed.

Farmer

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

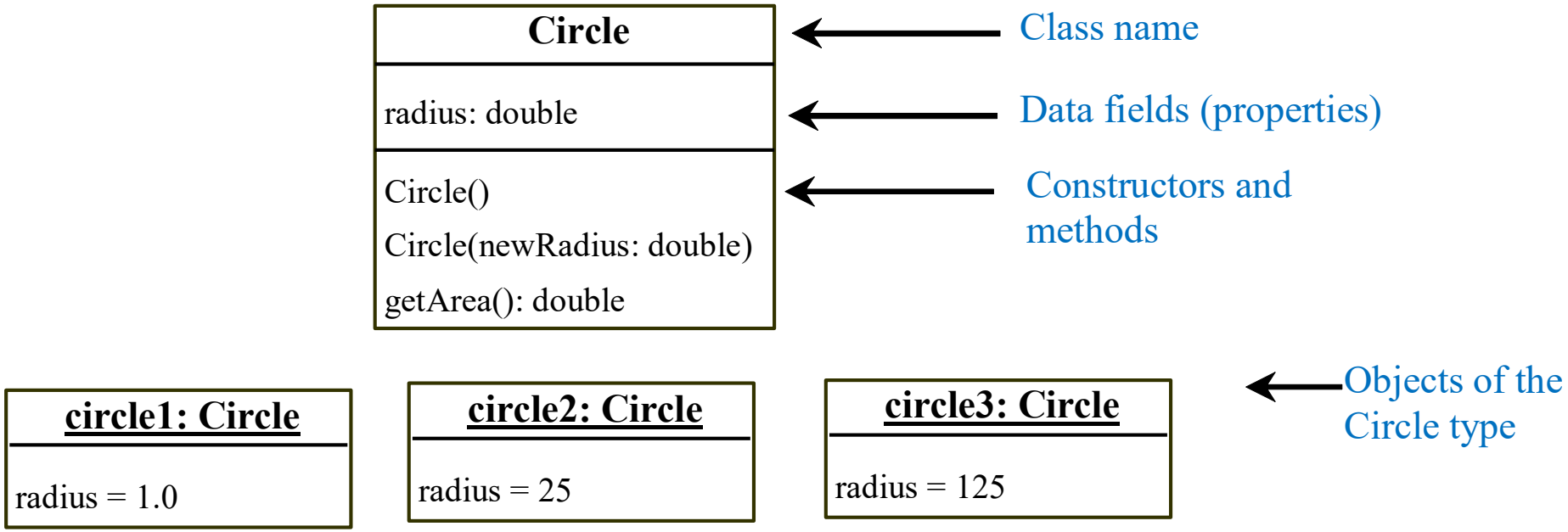
`Farmer()`
`Farmer(name: String, weight: double, x:int, y:int)`
`moveUp(): void`
`moveDown(): void`
`moveRight(): void`
`moveLeft(): void`
`moveTo(int x, int y): void`

← Class name

← Attributes (data fields)

← Constructors and Methods

UML Notation, cont'd



In the class diagram, the data field is denoted as

dataFieldName: dataFieldType

The constructor is denoted as

ClassName(parameterName: parameterType)

The method is denoted as

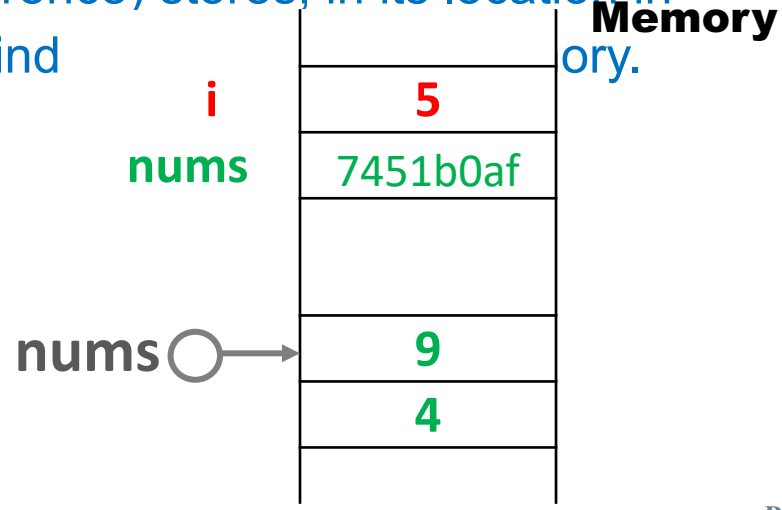
methodName(parameterName: parameterType): returnType

Garbage Collection and OOP Advantages

Remember: Primitive vs. Reference Types

- Java's types are divided into:
  - 1. Primitive types
    - Includes `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
    - A primitive-type variable stores, in its location in memory, **a value** of its declared type.
  - 2. Reference types
    - Includes all non-primitive types, (e.g., **Arrays**, Strings, Scanner, etc.)
    - A reference-type variable (or a reference) stores, in its location in memory, data which Java uses to find ory.
    - Such a variable is said to **refer to an object** in the program.
  - Example:

```
int i = 5;
int [] nums = {9,4};
```



Garbage Collection

- Consider the following code:

```
Circle c1 = new Circle();
Circle c2 = new Circle();
c1 = c2
```

- In this example, c1 points to the same object referenced by c2.

- The object previously referenced by c1 is **no longer referenced**. This object is known as **garbage**. Garbage is automatically collected by JVM.

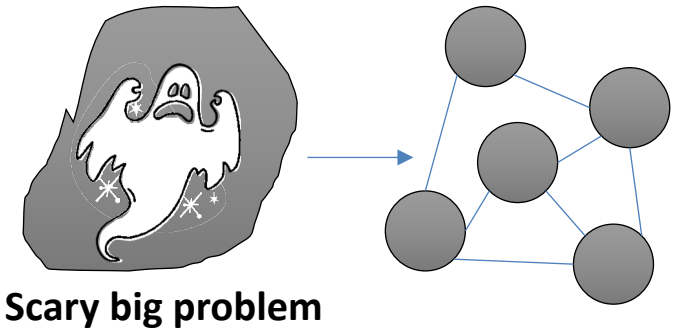
- **TIP**: If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object.

- **Bottom line:** JVM will automatically collect the space if the object is not referenced by any variable.

Advantages of Object Oriented Programming (OOP)

Modularization

- Big problem into smaller subproblems.
- Improves understandability

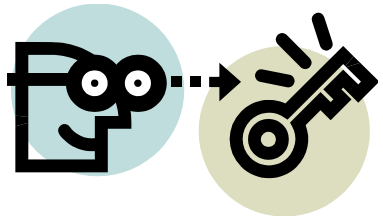


Encapsulation and Reuse

- Hide complexity and protect low-level functionality.
- Reuse code in other programs

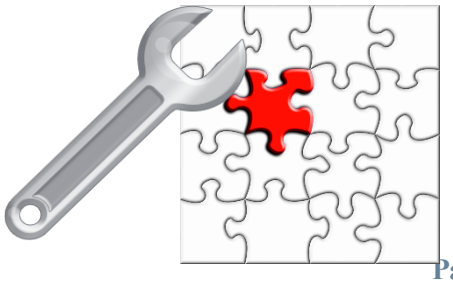
Understandability (abstraction)

- Composability (big objects are built off smaller ones)
  - More about this later



Maintenance

- Easier to change code of individual modules



Summary of what we covered so far...

Remember: key concepts

Object-oriented programming (OOP)

- It is a programming paradigm based on the concept of "objects"

Objects

- An object is an **entity in the real world**. An object has
  - a unique **identity**,
  - **state** (also known as **properties** or **attributes**).
  - **behavior (methods):** what the **object can do**.

Classes

- Objects of the same type are defined using a common class.
  - A class is a template or blueprint that defines the properties and behaviors for objects.
- A Java class uses
  - **variables** to define the state
  - **methods** to define behaviors.
  - **Constructors** to perform initializing actions

Summary so far

- Terminology:
  - Object-oriented programming (OOP), classes, objects, instance variable, class methods.
- Creating a class with instance variables and methods
- Purpose, use, and definition of constructors.
- Creating objects using `new`
- Calling object's methods using the dot (.) operator

More on Basic OOP

What is next…

- In this part, we will discuss more topics related to basic OOP programming, specifically
  - 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. We shall study two modifiers:

  `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.

- If no access modifier is used, then a class member can be accessed by any class in the same package.

- ***We will discuss other visibility modifiers later!***

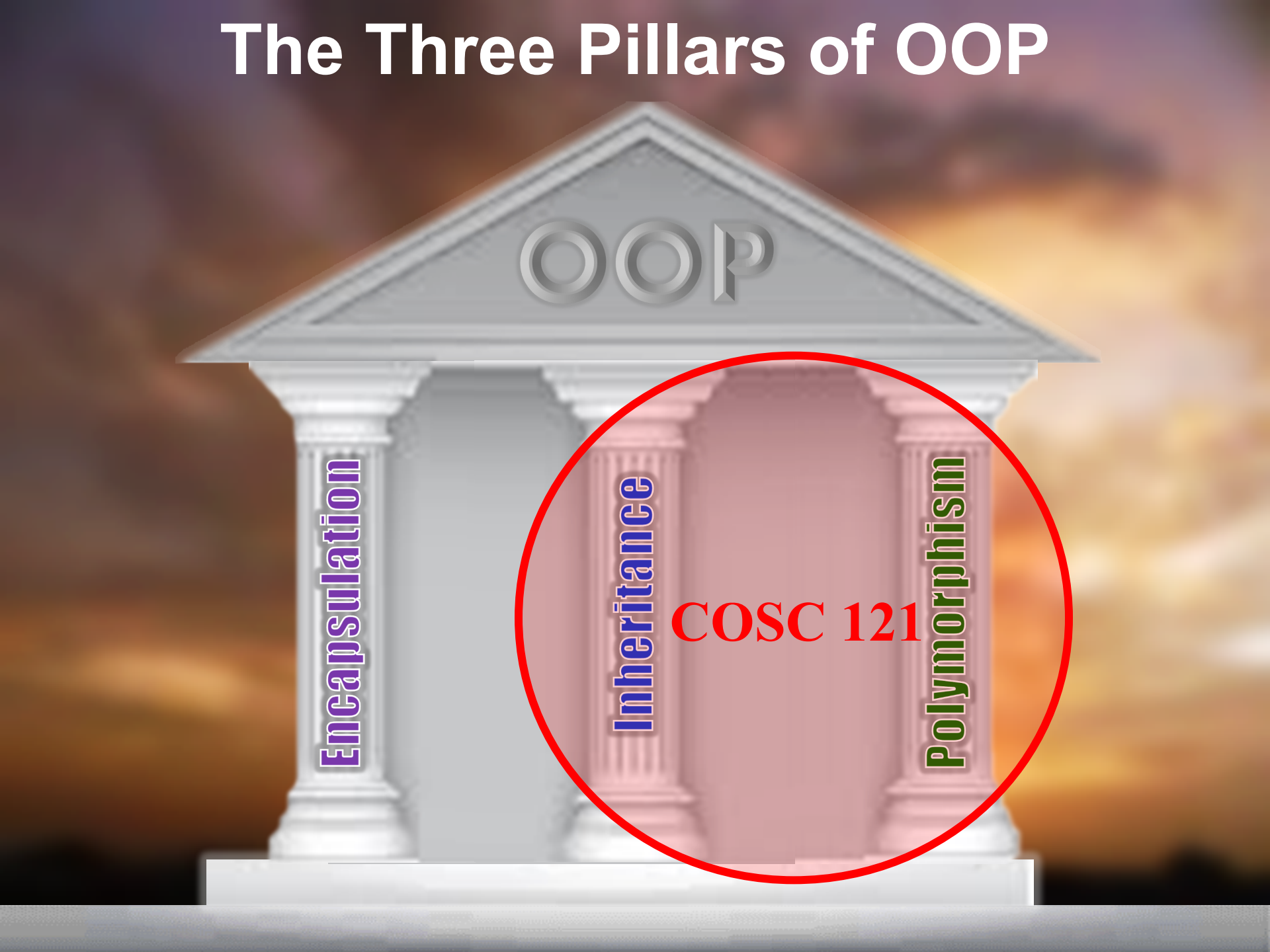
Data Field Encapsulation

- 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



Practice

Animation 

- Write a public class **SimpleCircle** which has:
  - an instance variable double radius.
  - a no-argument constructor that sets radius to 10.
  - a one-argument constructor that sets radius to a given value.
  - a method setRadius that changes the radius to a given value.
  - two methods getArea that returns the area.

- Test your class by creating three instances of SimpleCircle and invoke their different methods.

Solution

```
class SimpleCircle {
  //Attributes
  private double radius;
  //Constructors
  public SimpleCircle() {
    setRadius(1);
  }
  public SimpleCircle(double radius){
    setRadius(radius);
  }
  //Methods
  public double getRadius() {
    return radius;
  }
  public void setRadius(double r){
    if (radius >= 0)
        radius = r;
  }
  public double getArea() {
    return radius*radius*Math.PI;
  }
}
```

```
public class TestCircle {
  public static void main(String[] args) {
    Circle c1 = new Circle();
    Circle c2 = new Circle(5.0);

    System.out.println("For a circule of radius "
      + c1.radius+", the area is " + c1.getArea());
    System.out.println("For a circule of radius "
      + c2.radius+", the area is " + c2.getArea());
    c1.setRadius(100);
    System.out.println("For a circule of radius "
      + c1.radius+", the area is " + c1.getArea());
  }
}
```

TestCircle is the main class. Its sole purpose is to test the second class.

We could include the main method in SimpleCircle class, and hence SimpleCircle will be the main class.

The `this` Keyword

- The `this` keyword is the name of a reference that an object can use to refer to itself.

- **Uses:**
  - 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!

Practice

Given this Cow class → (which we created before in a practice question):

- Q1: change all attributes to `private` and all methods and constructors to `public`.
- Q2: create setters and getters for `nickname` and `stomach` attributes. make sure that:
  - `nickname` starts with a letter and it is at least 4 characters
  - `stomach` value is always between 0 and 100 (inclusive). If a value >100 is given, set `stomach` to 100.
- Q3: create a getter for `full` (but not a setter).
  - `full` can only be set based on `stomach`
- Q4: make any necessary changes to
  - reduce code redundancy and properly use the setters and getters in your class
  - in setters, give your arguments the same name of the attributes to which they are related.

```
public class Cow {
  String nickname;
  int stomach;
  boolean full;

  Cow(){nickname = "Anonymous"; stomach = 50;}
  Cow(String n, int st){
    nickname = n;
    if(st >= 0) {
      stomach = st;
      if(stomach >= 100) {
        stomach = 100;
        full = true;
      }
    }
  }

  void eat(int amount) {
    if(amount>0) {
      stomach += amount;
      if(stomach >= 100) {
        stomach = 100;
        full = true;
      }
    }else {
      System.out.println("invalid food amount.");
    }
  }
  void say(String msg) {
    System.out.println(nickname + " says: " + msg);
  }
}
```

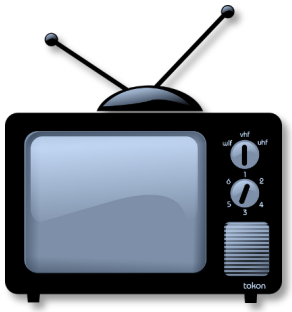
Solution

```
public class Cow {

  private String nickname;
  private int stomach;
  private boolean full;


  public Cow2(){this("Anonymous", 50);}
  public Cow2(String nickname, int stomach){
    setNickname(nickname);
    setStomach(stomach);
  }


  public void eat(int amount) {
    if(amount >= 0)
      setStomach(stomach + amount);
    else
      System.out.println("invlaid food amount.");
  }

  public void say(String msg) {
    System.out.println(nickname + " says: " + msg);
  }
```

```
  //SETTERS
  public void setNickname(String nickname) {
    char firstchar = nickname.charAt(0);
    int len = nickname.length();
    if(len>=4 && Character.isLetter(firstchar))
      this.nickname = nickname;
    else
      System.out.println("invalid nickname.");
  }
  public void setStomach(int stomach) {
    if(stomach >= 0) {
      stomach = stomach>100? 100 : stomach;
      full = stomach >= 100;
    }else {
      System.out.println("invalid stomach value.");
    }
  }
  //'full' attribute is a read only -> no setFull()

  //GETTERS
  public String getNickname() {return nickname;}
  public int getStomach() {return stomach;}
  public boolean isFull() {return full;}
}
```

Practice

Write a class **TV** according to the following UML diagram.



TV
channel: int volumeLevel: int on: boolean
TV() turnOn(): void turnOff(): void setChannel(newChannel: int): void setVolume(newVolumeLevel: int): void channelUp(): void channelDown(): void volumeUp(): void volumeDown(): void

The current channel (1 to 120) of this TV.

The current volume level (1 to 7) of this TV.

Indicates whether this TV is on/off.

Constructor(defaults: channel 1, volume=1, turned on

Turns on this TV.

Turns off this TV.

Sets a new channel for this TV.

Sets a new volume level for this TV.

Increases the channel number by 1.

Decreases the channel number by 1.

Increases the volume level by 1.

Decreases the volume level by 1.

Solution

The constructor and methods in the **TV** class are defined public so they can be accessed from other classes.

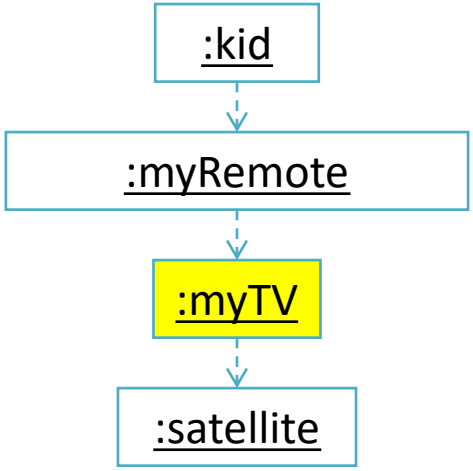
Note that

- the channel and volume level are not changed if the TV is not on.
- Before either of these is changed, its current value is checked to ensure that it is within the correct range.

```
public class TV {
  //Attributes
  private int channel, volume;
  private boolean on;
  //Constructor
  public TV()
  {turnOn();setChannel(1);setVolume(1);}
  //Methods
  public void turnOn()  {on = true;}
  public void turnOff() {on = false;}
  public void setChannel(int ch) {
    if(on && ch>=1 && ch<=121)
      channel = ch;
  }
  public void setVolume(int vol) {
    if(on && vol>=0 && vol<=7)
      volume = vol;
  }
  public void channelUp()
    {setChannel(channel + 1);}
  public void channelDown()
    {setChannel(channel - 1);}
  public void volumeUp()
    {setVolume(volume+1);}
  public void volumeDown()
    {setVolume(volume-1);}
}
```

Interactions between Objects

- In the previous example, you have seen the code TV class. Once you create an object of the type TV (highlighted below), other objects could call the TV methods perform certain actions.
  - Example scenario:
    - The Kid presses the ON button on `myRemote` Object (an event).
    - `myRemote` calls a method `myTV.turnOn()` which cause the TV object to turn on.



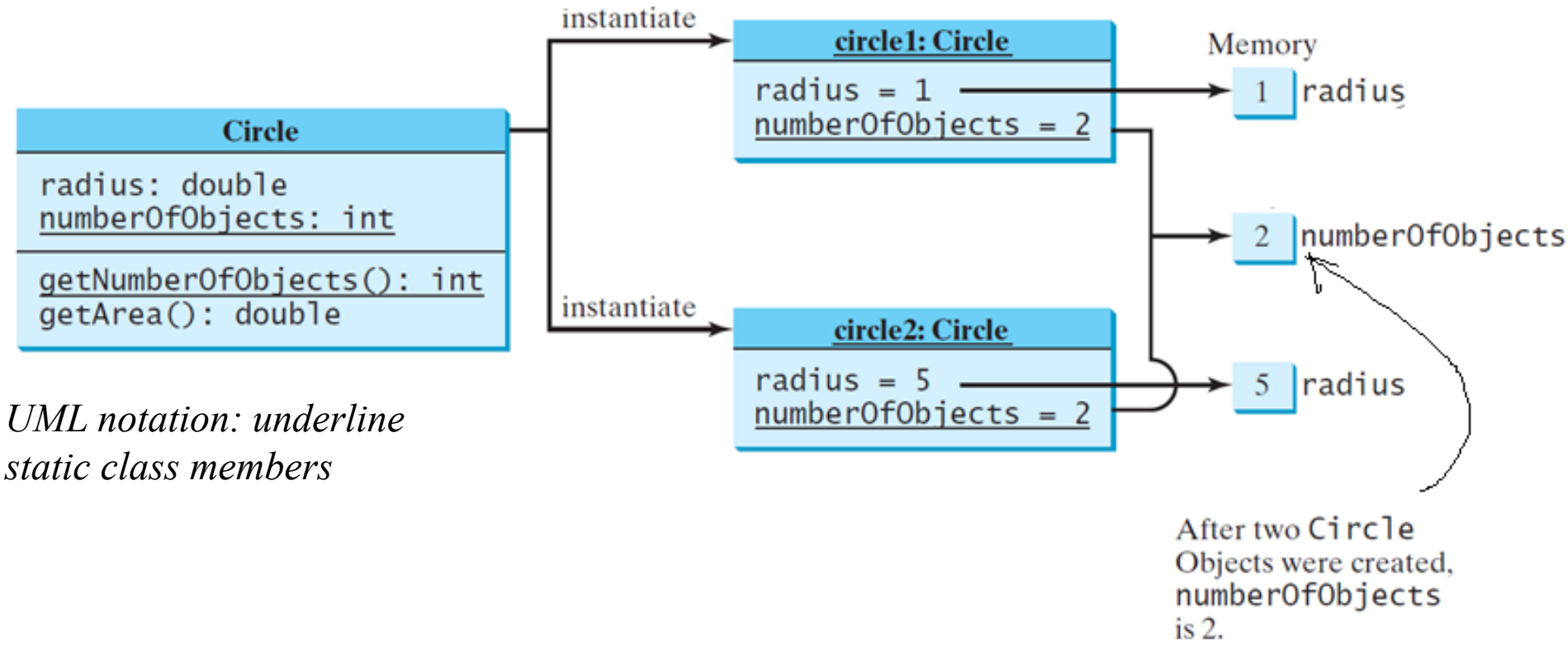
The `static` Modifier

- **`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

The `static` Modifier, cont'd

- Assume we modify **Circle** class, which originally defines the instance variable **radius,** and add a static variable **numberOfObjects** to count the number of circle objects created. We also add static method **getNumberOfObjects**.
  - See the example on the next slide.



Practice

Circle	
- radius: double	The radius of this circle (default: 1.0).
- <u>numberOfObjects: int</u>	The number of circle objects created.
+ Circle2()	Constructs a default circle object.
+ Circle2(radius: double)	Constructs a circle object with the specified radius.
+ getRadius(): double	Returns the radius of this circle.
+ setRadius(radius: double): void	Sets a new radius for this circle.
+ <u>getNumberOfObjects(): int</u>	Returns the number of circle objects created
+ getArea(): double	Returns the area of this circle.

The + sign indicates public modifier

The - sign indicates private modifier

Underlined text is static

Solution

```
public class Circle {
    private double radius;
    private static int numberOfObjects;
    public Circle() { this(1);}
    public Circle(double radius) {
        setRadius(radius);
        numberOfObjects++;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        if(radius>=0) this.radius = radius;
    }
    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```

Solution, cont'd

```
public class CircleTest {
   public static void main(String[] args) {
     System.out.print("Number of Circle objects: ");
     System.out.println(Circle.getNumberOfObjects());

     //Create two circles
     Circle c1 = new Circle();
     Circle c2 = new Circle(9);

     // Changing the radius of c1
     c1.setRadius(18);

     System.out.print("Number of Circle objects: ");
     System.out.println(Circle.getNumberOfObjects());
   }
}
```

It is better to Reference static members by their class name.

Scope of Variables

- **instance** and **static** variables
  - Scope is the entire class.
  - They can be declared anywhere inside a class.

- **local** variables
  - Scope starts from its declaration and continues to the end of the block that contains the variable.
  - A local variable must be initialized explicitly before it can be used.

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:

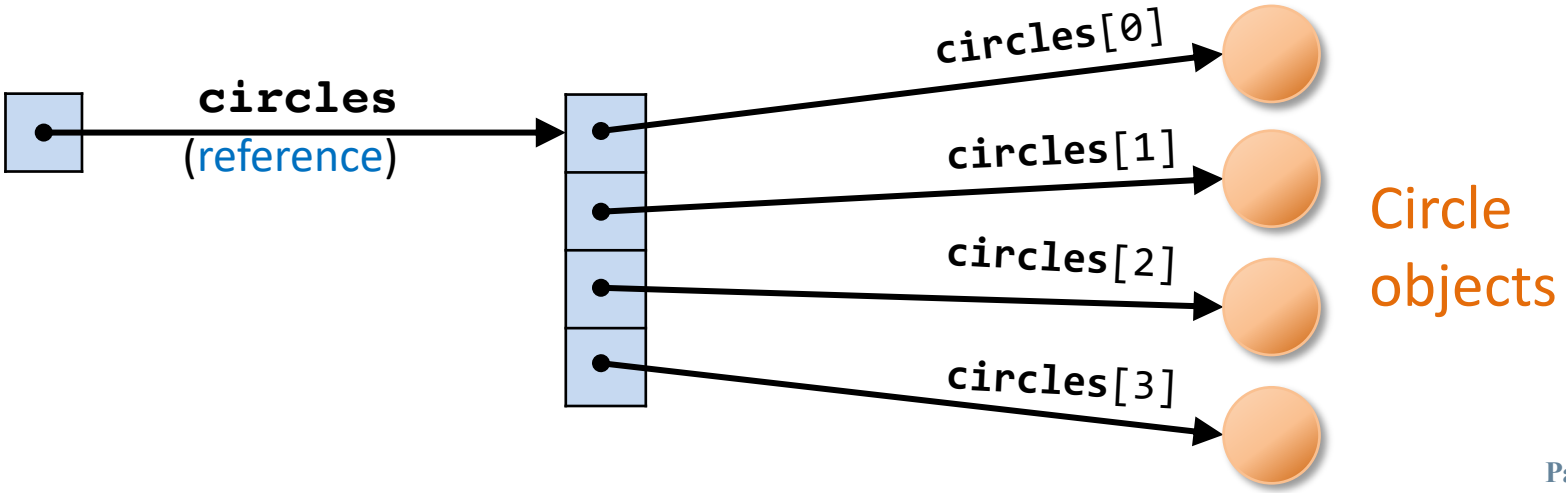
1. Declaration of reference variables:
   - You can create an array of objects, for example,

```
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;
   }
}
```