COSC 111 Computer Programming I

Introduction to Object Oriented Programming

Dr. Abdallah Mohamed

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

Previously…

OO Programming

Object-oriented programming (OOP)

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

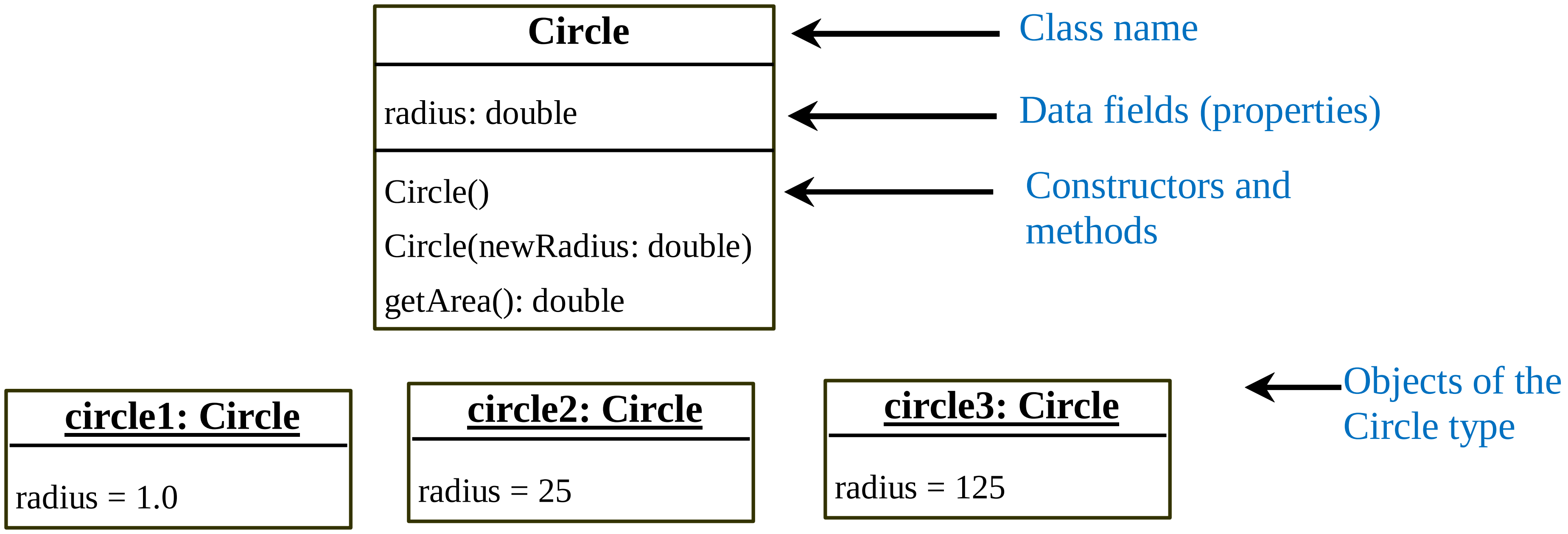
Objects

- 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

UML Notation



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

Data Fields

Data fields (object attributes) 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 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 play the role of initializing objects. The syntax is:

new ClassName(arguments);

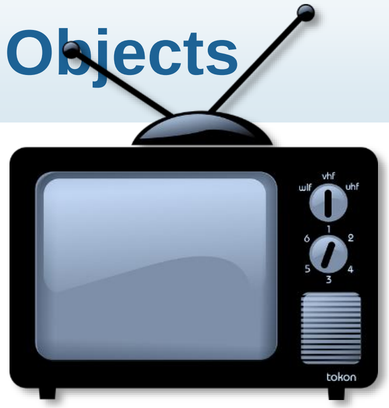
Examples:

```
new Circle();
new Circle(5.0);
```

A constructor with no parameters is referred to as a **no-argument** constructor.

A **default constructor** is provided automatically only if no constructors are explicitly defined in the class.

Example: Defining Classes and Creating Objects



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

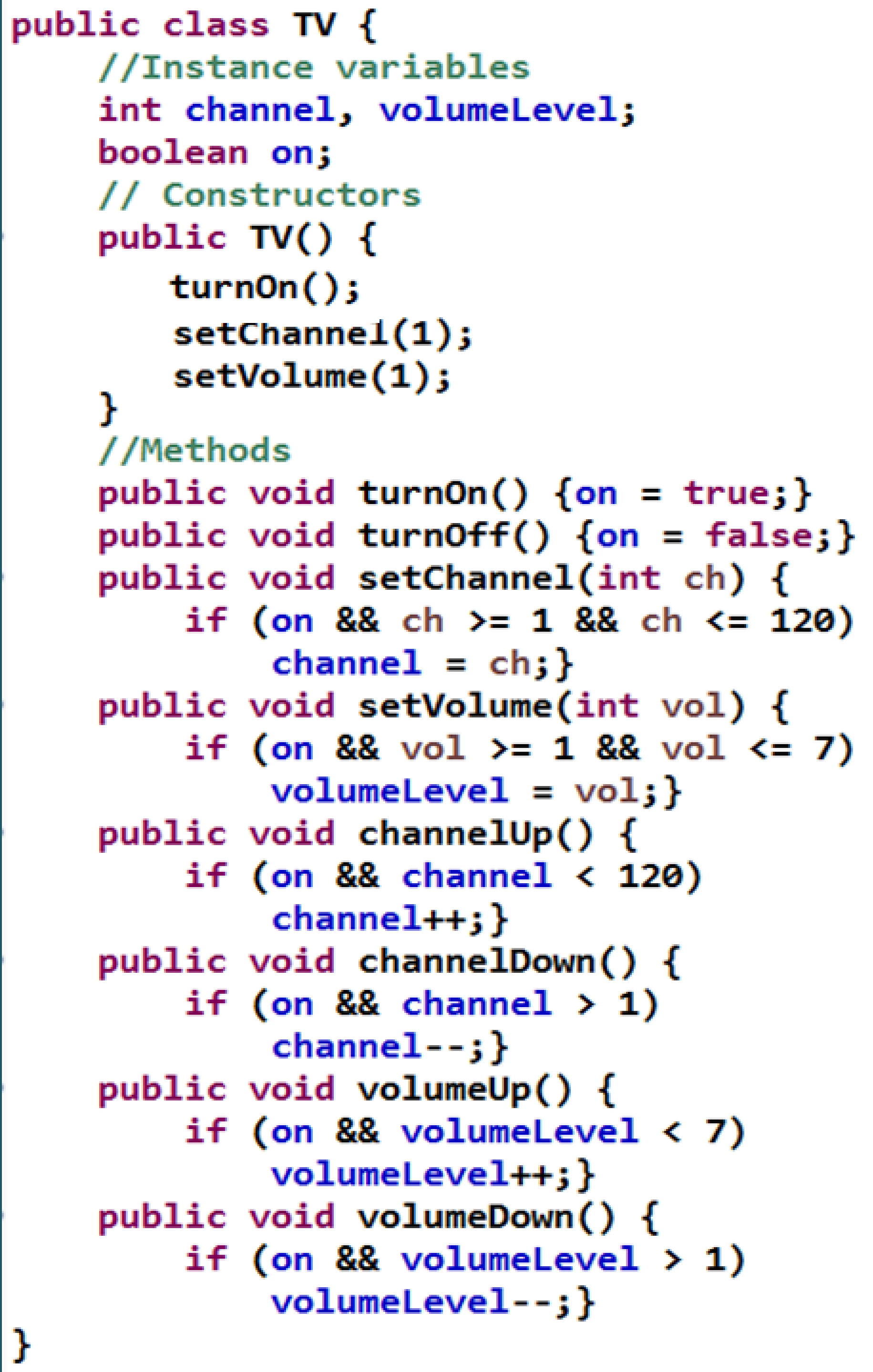
TV	
channel: int	**The current channel (1 to 120) of this TV.**
volumeLevel: int	**The current volume level (1 to 7) of this TV.**
on: boolean	**Indicates whether this TV is on/off.**
TV()	**Constructor(defaults: channel 1, volume=1, turned on**
turnOn(): void	**Turns on this TV.**
turnOff(): void	**Turns off this TV.**
setChannel(newChannel: int): void	**Sets a new channel for this TV.**
setVolume(newVolumeLevel: int): void	**Sets a new volume level for this TV.**
channelUp(): void	**Increases the channel number by 1.**
channelDown(): void	**Decreases the channel number by 1.**
volumeUp(): void	**Increases the volume level by 1.**
volumeDown(): void	**Decreases the volume level by 1.**

Example, cont.

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 {
    //Instance variables
    int channel, volumeLevel;
    boolean on;
    // Constructors
    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 <= 120)
            channel = ch;}
    public void setVolume(int vol) {
        if (on && vol >= 1 && vol <= 7)
            volumeLevel = vol;}
    public void channelUp() {
        if (on && channel < 120)
            channel++;}
    public void channelDown() {
        if (on && channel > 1)
            channel--;}
    public void volumeUp() {
        if (on && volumeLevel < 7)
            volumeLevel++;}
    public void volumeDown() {
        if (on && volumeLevel > 1)
            volumeLevel--;}
}
```

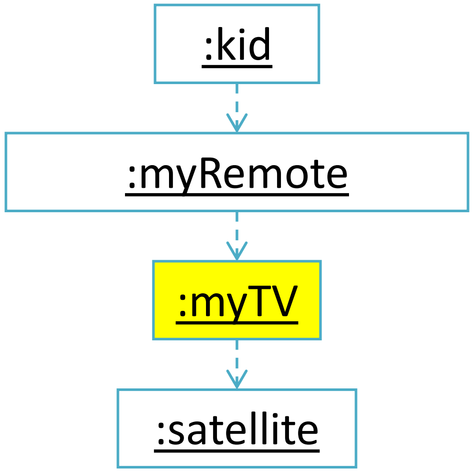
Another Solution

```
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);}
}
```

Example, cont.

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.



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 and getPerimeter that return the area and perimeter respectively.

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

Practice

```java
class Circle {
    //Instance variable
    double radius;
    //Constructors
    Circle() {
        radius = 1;
    }
    Circle(double r) {
        radius = r;
    }
    //Methods
    double getArea() {
        return radius*radius*Math.PI;
    }
    double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    void setRadius(double r) {
        radius = r;
    }
}
```

```java
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 Circle class, and hence Circle will be the main class.

You can put the two classes into one file, but only one class in the file can be a public class. The public class must have the same name as the file name.

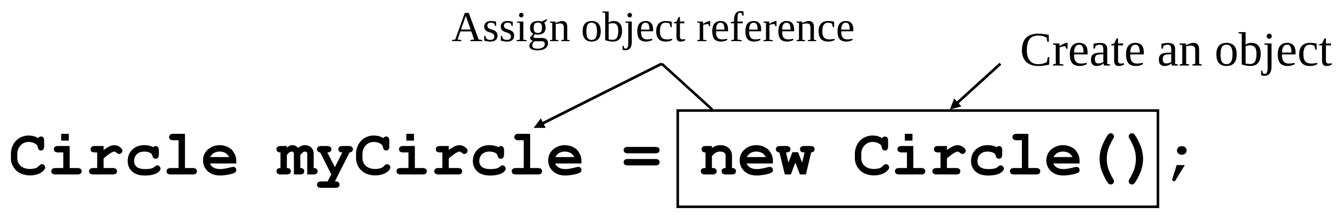
Accessing Objects via Reference Variables

Newly created objects are allocated in the memory. They can be accessed via reference variables:

```
ClassName objectRefVar;
```

- For example:
  - Circle myCircle;

The next statement creates an object and assigns its reference to myCircle:



```
Circle myCircle = new Circle();
```

Accessing Object's Members

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

- Referencing the object's data:

```
objectRefVar.data
```

Example: myCircle.radius

- Invoking the object's method:

```
objectRefVar.methodName(arguments)
```

Example: **myCircle.getArea()**

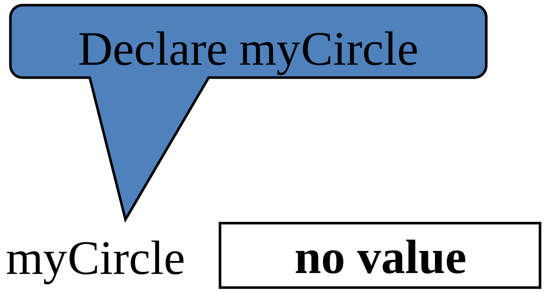
animation

Trace Code

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



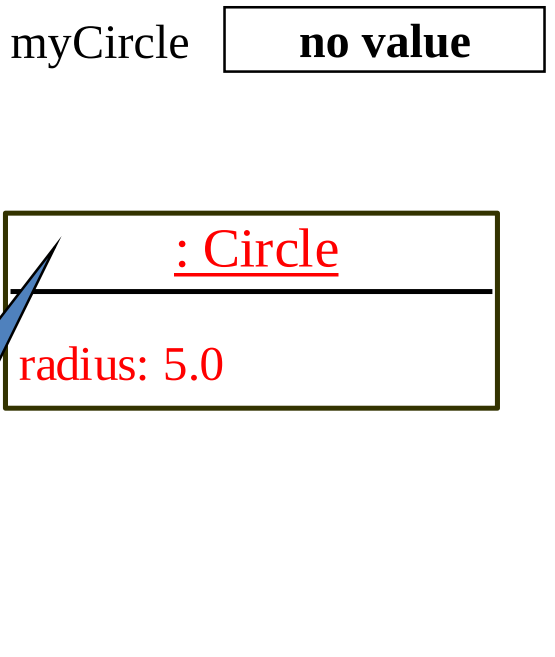
animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



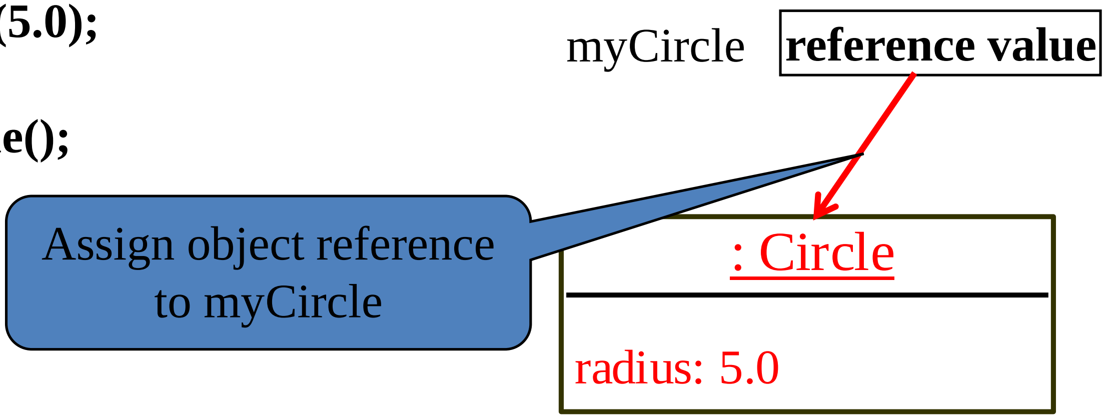
animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



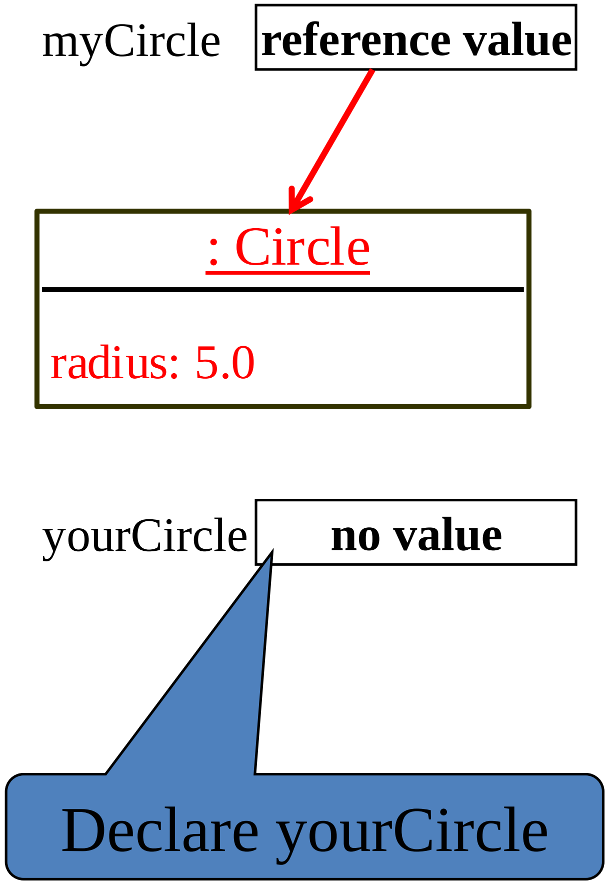
animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



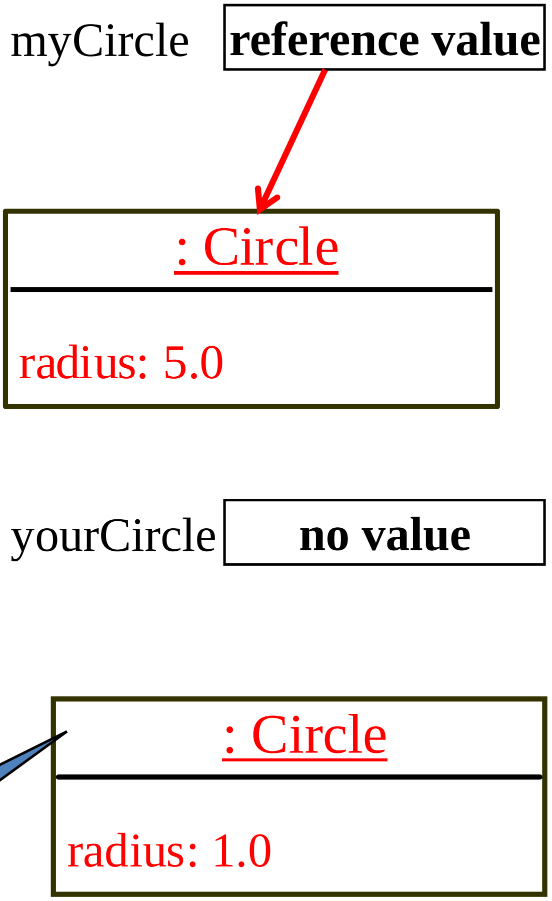
animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



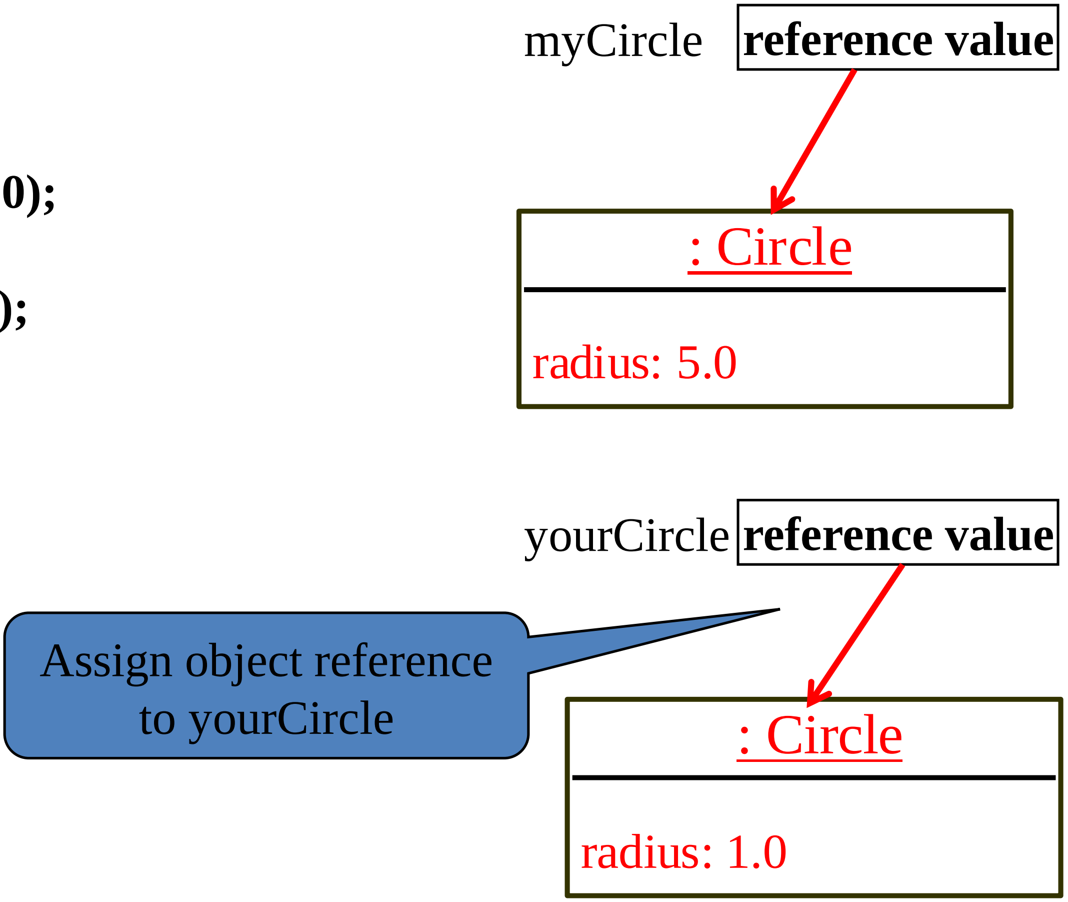
animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



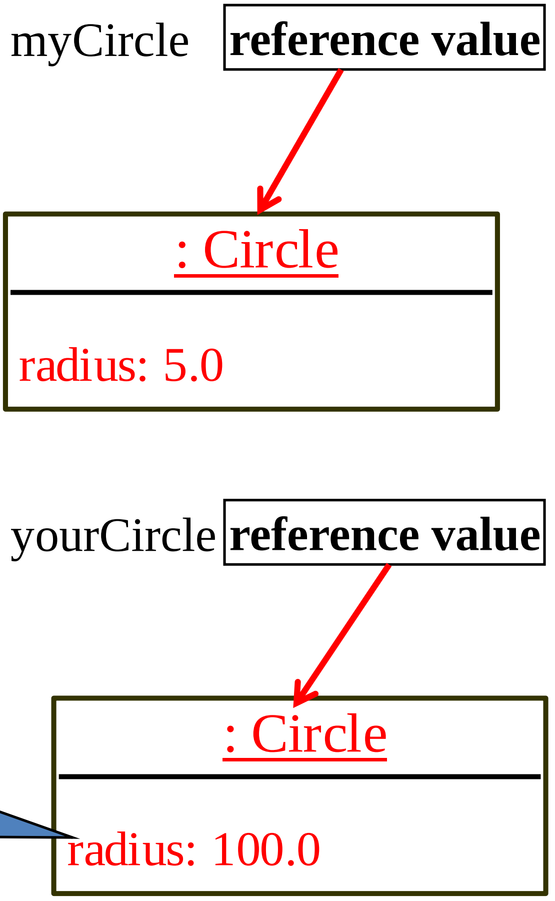
animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Change radius in yourCircle

Caution

Recall that you use Math.methodName(arguments) (e.g., Math.pow(3, 2.5)) to invoke a method in the Math class.

Can we invoke getArea() using SimpleCircle.getArea()?

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, getArea() is non-static. It must be invoked from an object using

- objectRefVar.methodName(arguments)
  - e.g., myCircle.getArea()

More explanations will be given in the section on “Static Variables, Constants, and Methods.”

Intro to 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 more details about visibility modifiers later!

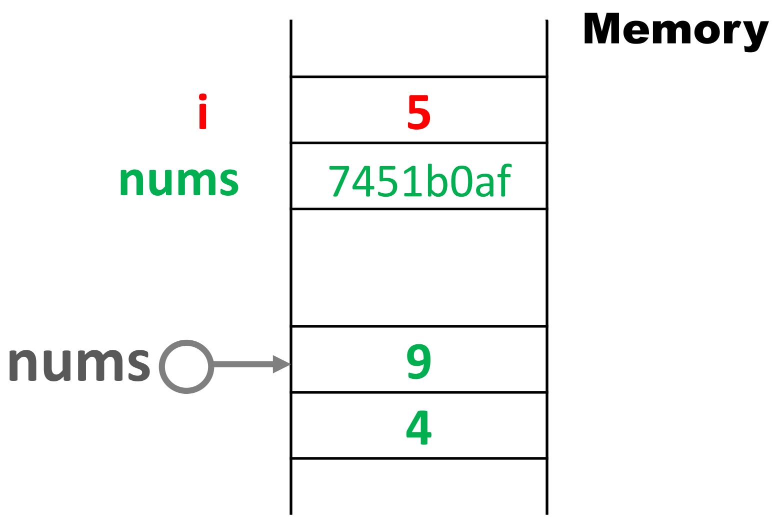
Garbage Collection

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 the object in the memory.
  - Such a variable is said to **refer to an object** in the program.
- Exampe:

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



Garbage Collection

Consider the following code:

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

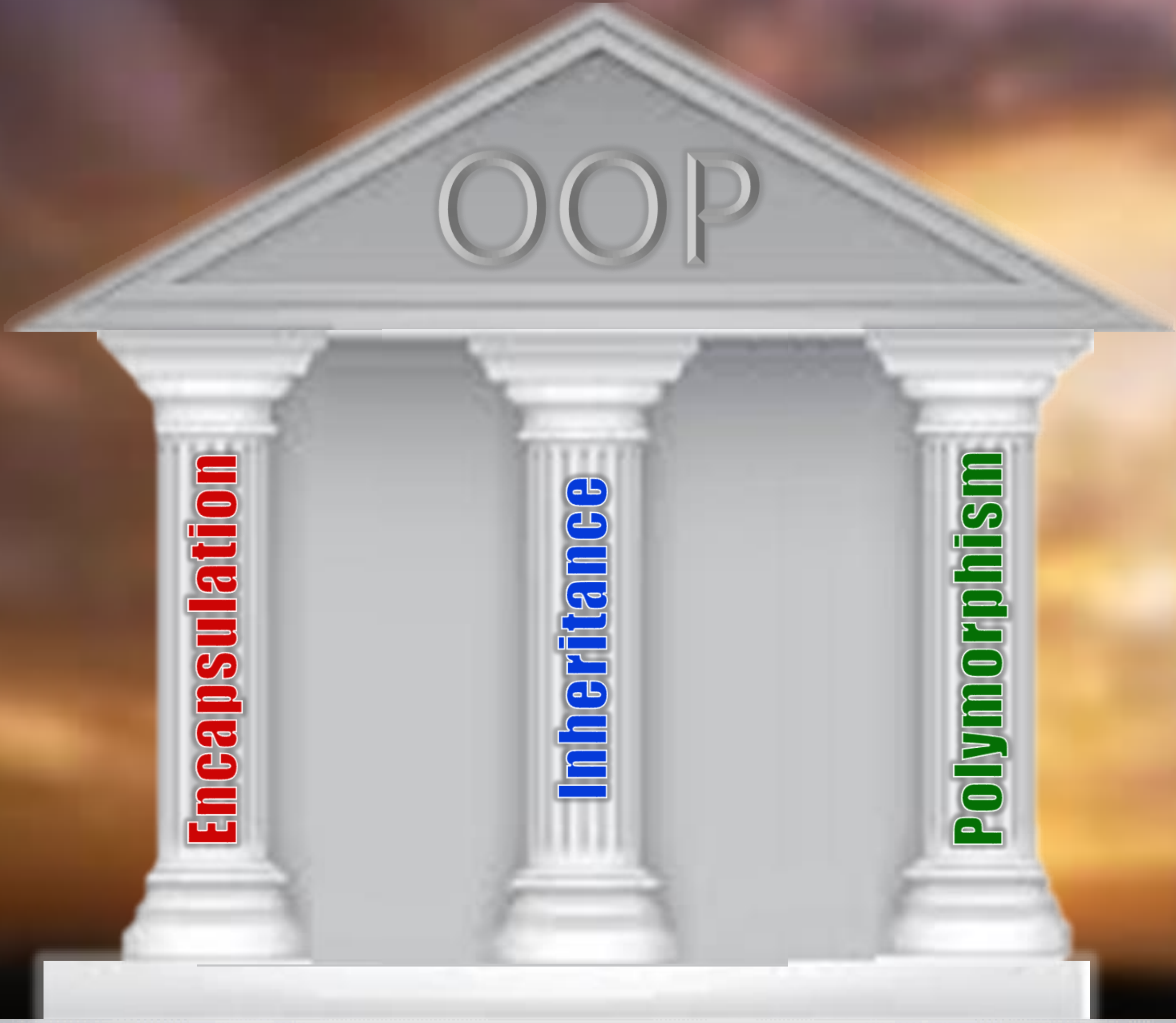
In this example, c1 points to the same object referenced by c2.

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.

The Three Pillars of OOP



The 1st pillar

Data Field Encapsulation

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).
- 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.

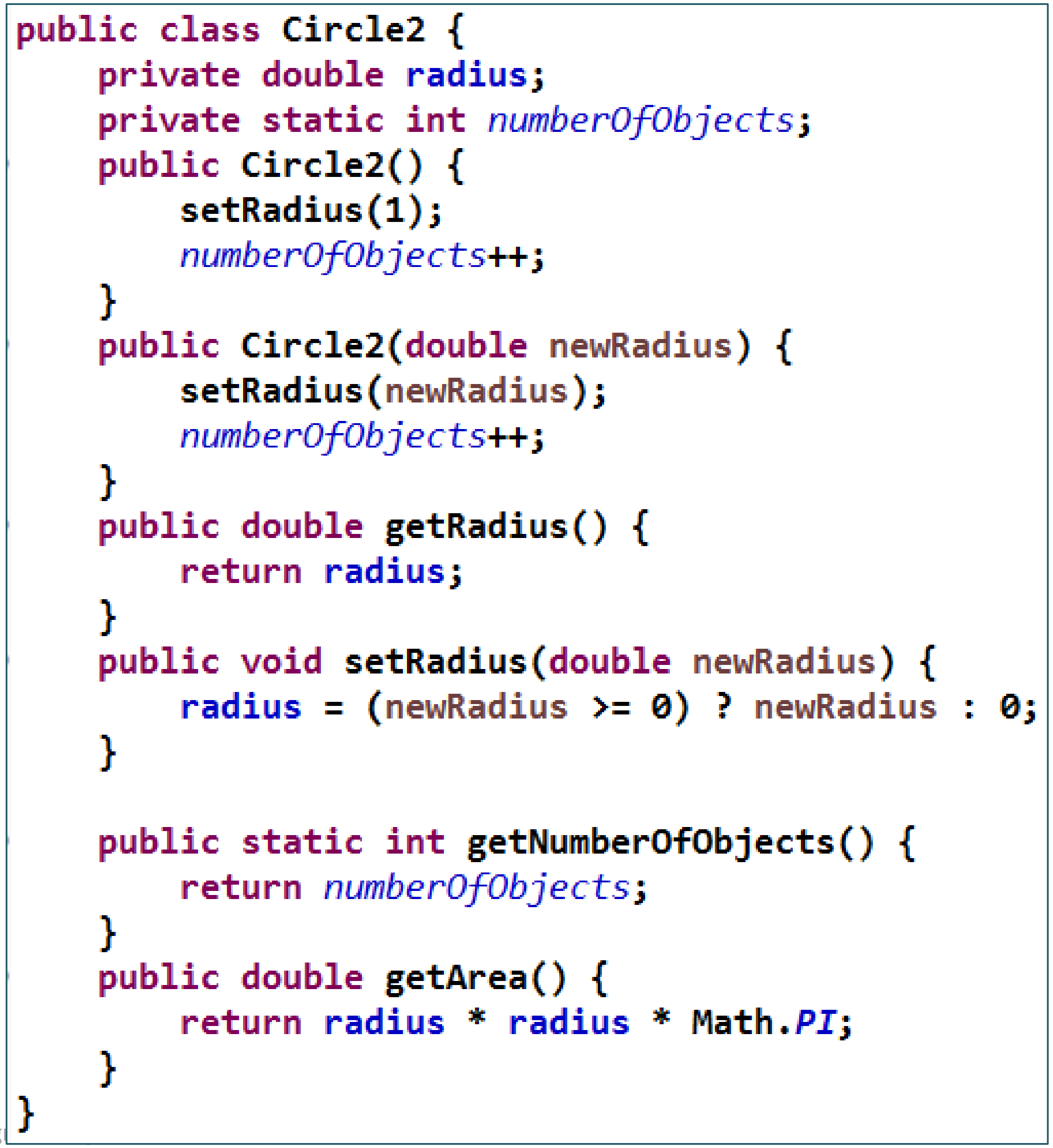
Example of Data Field Encapsulation

Circle2	
- radius: double	The radius of this circle (default: 1.0).
- numberOfObjects: int	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.
+ getNumberOfObjects(): int	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

Example of Data Field Encapsulation, cont.



```
public class Circle2 {
    private double radius;
    private static int numberOfObjects;
    public Circle2() {
        setRadius(1);
        numberOfObjects++;
    }
    public Circle2(double newRadius) {
        setRadius(newRadius);
        numberOfObjects++;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double newRadius) {
        radius = (newRadius >= 0) ? newRadius : 0;
    }

    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```

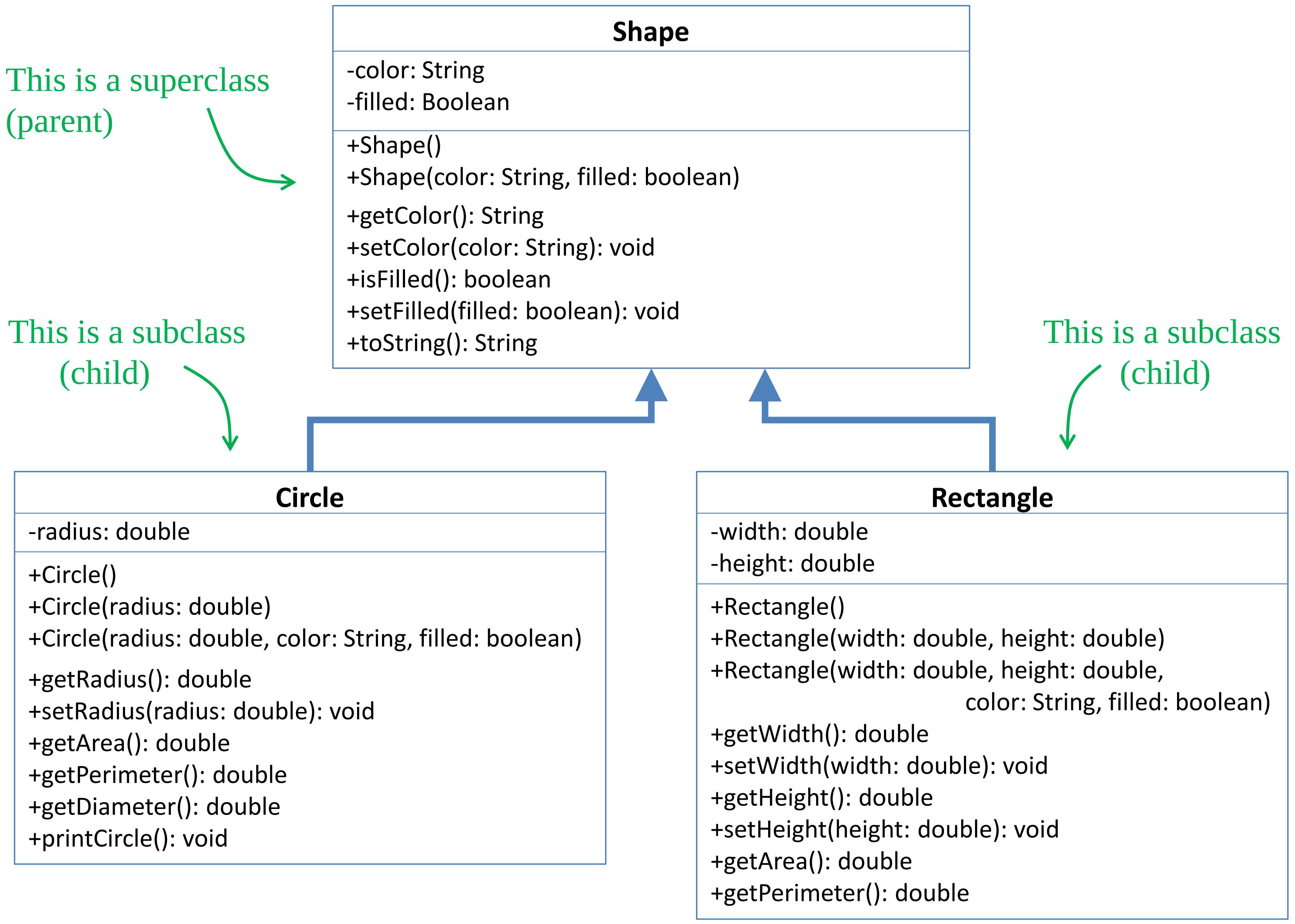
The 2nd pillar

Inheritance

What is inheritance?

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

Inheritance Example


This is a superclass (parent)
This is a subclass (child)
This is a subclass (child)

Shape

-color: String
-filled: Boolean

+Shape()
+Shape(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+toString(): String

Circle

-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double,
color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Inheritance Example, cont.

Shape is the superclass (parent)

```
public class Shape {
    private String color;
    private boolean filled;

    public Shape() {color = "white";}
    public Shape(String color, boolean filled) {
        this.color = color;
        this.filled = filled;
    }
    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 String toString() {
        return "Color is " + color + ". Filled? " + filled;
    }
}
```

Inheritance Example, cont.

Rectangle is the subclass.
Its parent is Shape

```
public class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle() { this(1.0, 1.0); }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle(double w, double h, String color, boolean filled) {
        setWidth(w);
        setHeight(h);
        setColor(color);
        setFilled(filled);
    }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
    public double getArea() { return width * height; }
    public double getPerimeter() { return 2 * (width + height); }
}
```

Inheritance Example, cont.

This is a test program!

```
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(1);
        System.out.println("A circle\n" + circle.toString());
        System.out.println("The color is " + circle.getColor());
        System.out.println("The radius is " + circle.getRadius());
        System.out.println("The area is " + circle.getArea());
        System.out.println("The diameter is " + circle.getDiameter());

        Rectangle rectangle = new Rectangle(2, 4);
        System.out.println("\nA rectangle\n" + rectangle.toString());
        System.out.println("The area is " + rectangle.getArea());
        System.out.println("The perimeter is " + rectangle.getPerimeter());
    }
}
```

The output

```
A circle
Color is white. Filled? false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0

A rectangle
Color is white. Filled? false
The area is 8.0
The perimeter is 12.0
```

What can you do in a subclass?

A subclass inherits from a superclass. You can:

- **Use** inherited class members (properties and methods).
- **Add** new class members.
- **Override** instance methods of the superclass
  - to modify the implementation of a method defined in the superclass
  - the method must be defined in the subclass using the same signature and the same return type as in its superclass.
- **Hide** static methods of the superclass
  - By writing a new *static* method in the subclass that has the same signature as the one in the superclass.
- **Invoke** a superclass constructor from within a subclass constructor
  - either implicitly or by using the keyword super

Overriding vs. Overloading

Overridden methods are in different classes related by inheritance; **overloaded** methods can be either in the same class or different classes related by inheritance.

Overridden methods have the same signature and return type; **overloaded** methods have the same name but a different parameter list.

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

`this` and `super` keywords

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);

The `super` Keyword

The keyword `super` refers to the superclass of the class in which super appears.

Uses:

- To reference class members in the superclass.
  - Example:
    - super.amethod(3, 5);
    - super.toString();
- To enable **a constructor to invoke another constructor** of the superclass.
  - A constructor can only be invoked from within another constructor
  - Examples:
    - super(10, 5);

2) Calling a superclass constructor, cont.

CAUTION

- You must use the keyword super to call the superclass constructor. Invoking a superclass constructor’s name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

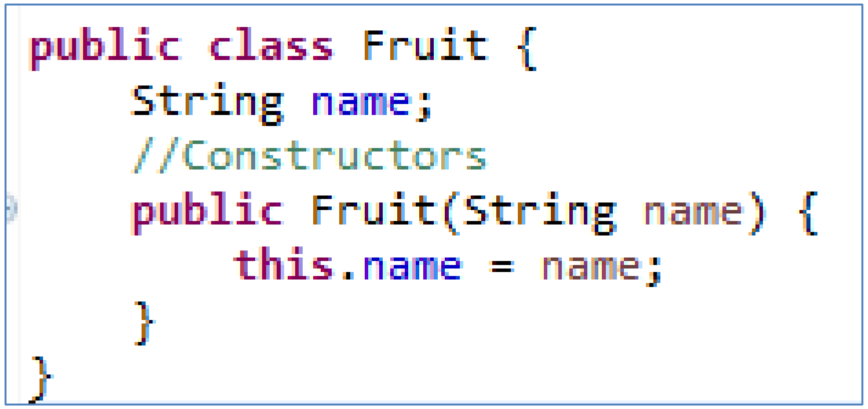
Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as **constructor chaining**.

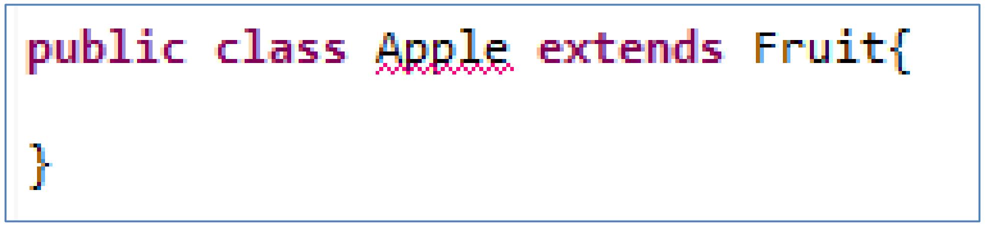
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Example on the Impact of a Superclass without no-arg Constructor

What is wrong with the code below?



```java
public class Fruit {
    String name;
    //Constructors
    public Fruit(String name) {
        this.name = name;
    }
}
```



```java
public class Apple extends Fruit{

}
```

Objects, Methods, and Arrays

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.
- **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.

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,

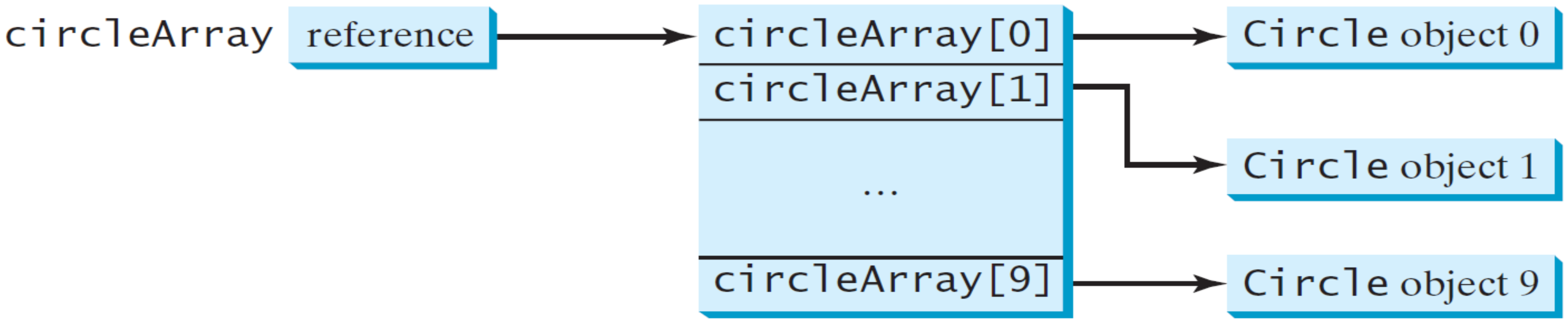
```
Circle[ ] circles = new Circle[10];
```

- 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.

`static` and `final` modifiers

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);

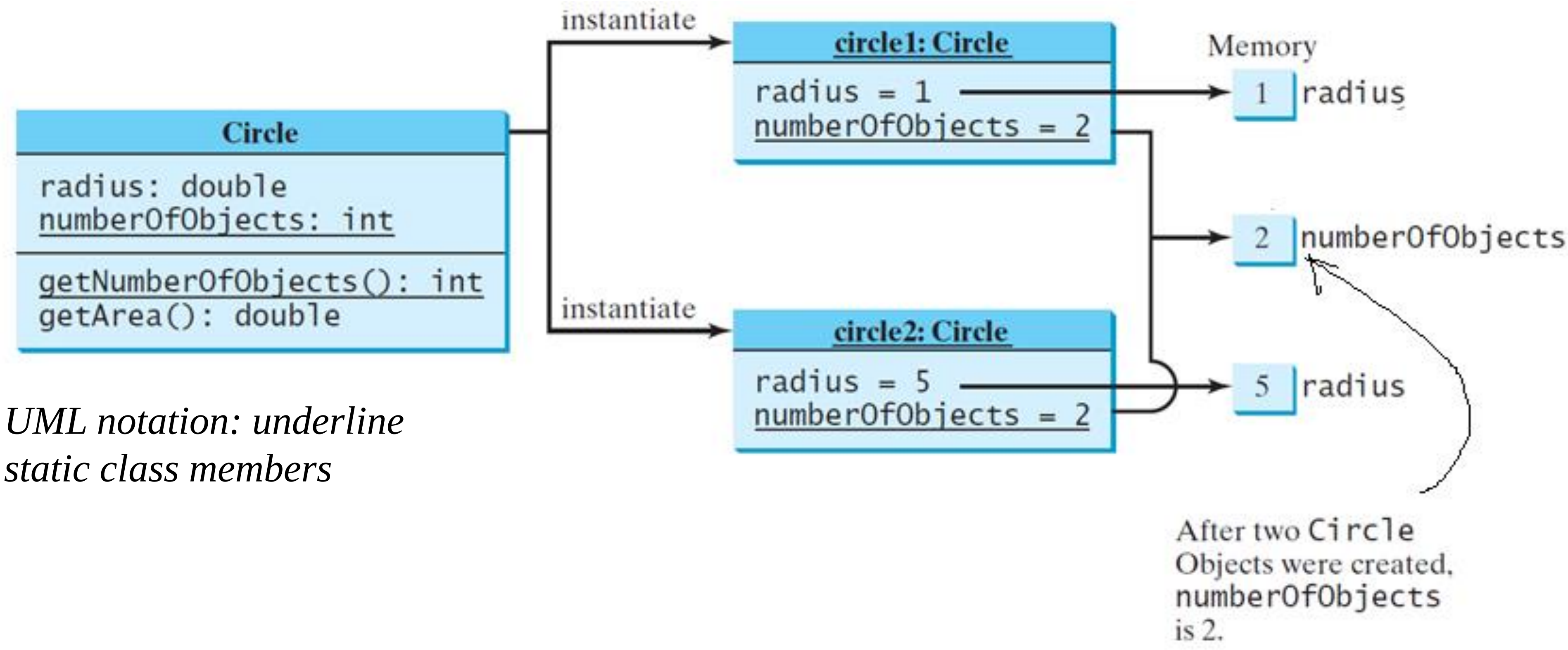
Remember that, *unlike* static class members:

- Instance variables belong to a specific instance.
- Instance methods are invoked by an instance of the class

The `static` Modifier

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**.

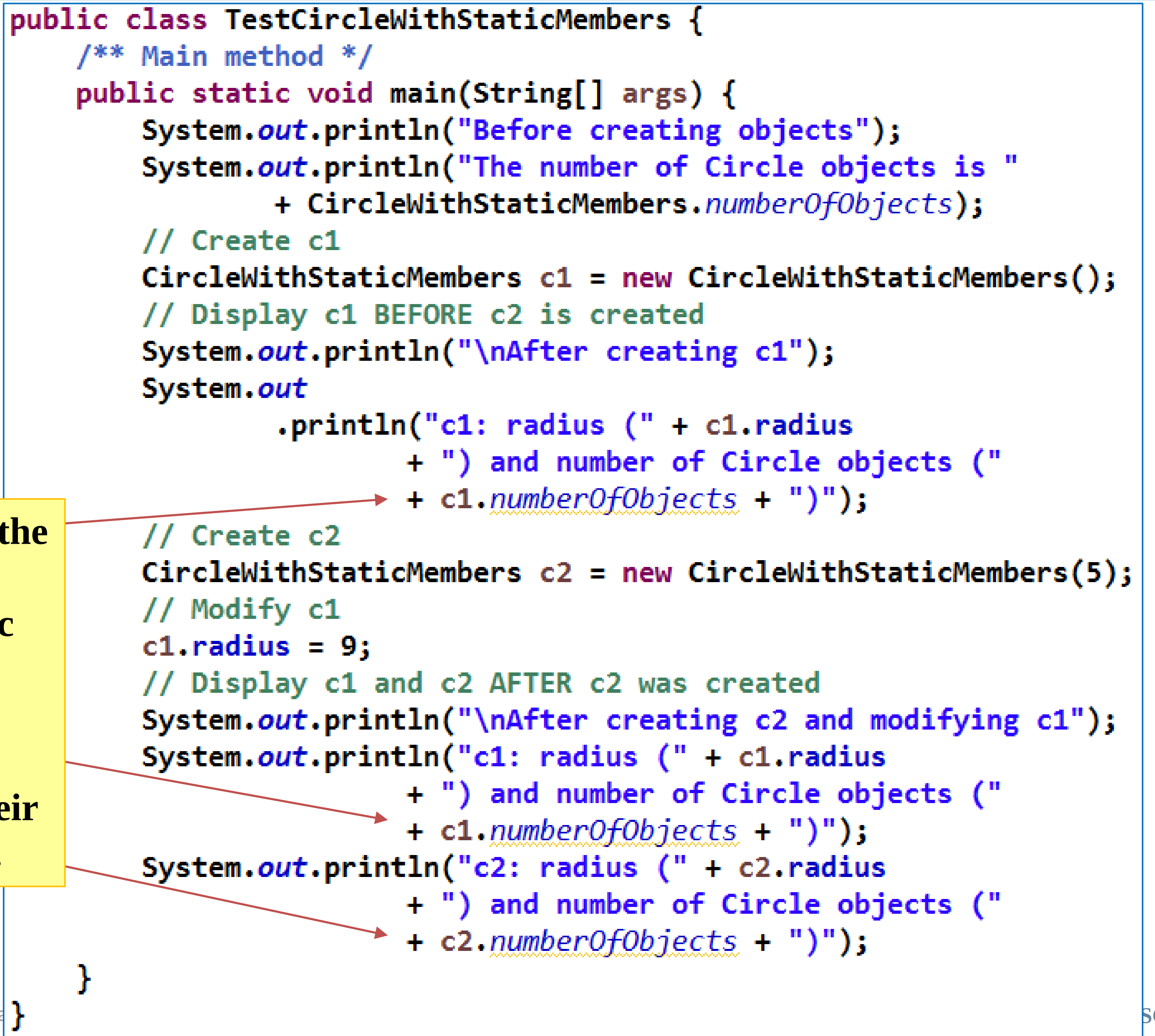
- See the code is in the next slide.



Instance vs. Static Class Members, cont.

```
public class CircleWithStaticMembers {
      double radius;
      static int numberOfObjects = 0;
      CircleWithStaticMembers() {
        radius = 1.0;
        numberOfObjects++;
      }
      CircleWithStaticMembers(double newRadius) {
        radius = newRadius;
        numberOfObjects++;
      }
      static int getNumberOfObjects() {
        return numberOfObjects;
      }
      double getArea() {
        return radius * radius * Math.PI;
      }
    }
```

Instance vs. Static Class Members, cont.



```
public class TestCircleWithStaticMembers {
    /** Main method */
    public static void main(String[] args) {
        System.out.println("Before creating objects");
        System.out.println("The number of Circle objects is "
                + CircleWithStaticMembers.numberOfObjects);
        // Create c1
        CircleWithStaticMembers c1 = new CircleWithStaticMembers();
        // Display c1 BEFORE c2 is created
        System.out.println("\nAfter creating c1");
        System.out
                .println("c1: radius (" + c1.radius
                        + ") and number of Circle objects ("
                        + c1.numberOfObjects + ")");
        // Create c2
        CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);
        // Modify c1
        c1.radius = 9;
        // Display c1 and c2 AFTER c2 was created
        System.out.println("\nAfter creating c2 and modifying c1");
        System.out.println("c1: radius (" + c1.radius
                        + ") and number of Circle objects ("
                        + c1.numberOfObjects + ")");
        System.out.println("c2: radius (" + c2.radius
                        + ") and number of Circle objects ("
                        + c2.numberOfObjects + ")");
    }
}
```

This is not the best way to access static members. Better to reference them by their class name.

Instance vs. Static Class Members, cont.

```java
public class TestCircleWithStaticMembers {
    /** Main method */
    public static void main(String[] args) {
        System.out.println("Before creating objects");
        System.out.println("The number of Circle objects is "
                + CircleWithStaticMembers.numberOfObjects);
        // Create c1
        CircleWithStaticMembers c1 = new CircleWithStaticMembers();
        // Display c1 BEFORE c2 is created
        System.out.println("\nAfter creating c1");
        System.out
                .println("c1: radius (" + c1.radius
                        + ") and number of Circle objects ("
                        + CircleWithStaticMembers.getNumberOfObjects() + ")");
        // Create c2
        CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);
        // Modify c1
        c1.radius = 9;
        // Display c1 and c2 AFTER c2 was created
        System.out.println("\nAfter creating c2 and modifying c1");
        System.out.println("c1: radius (" + c1.radius
                        + ") and number of Circle objects ("
                        + CircleWithStaticMembers.getNumberOfObjects() + ")");
        System.out.println("c2: radius (" + c2.radius
                        + ") and number of Circle objects ("
                        + CircleWithStaticMembers.getNumberOfObjects() + ")");
    }
}
```

Note the difference

Scope of Variables

The scope of **instance and static variables** is the entire class. They can be declared anywhere inside a class.

The scope of **a local variable** 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.

The `final` Modifier

A `final` local variable is a constant inside a method.

The `final` class cannot be extended:

```
final class Math {
    ...
}
```

The `final` method cannot be overridden by its subclasses.

Visibility Modifiers Revisited

Visibility Modifiers

Access modifiers are used for controlling levels of access to class members in Java. We shall study two modifiers:

`public,`

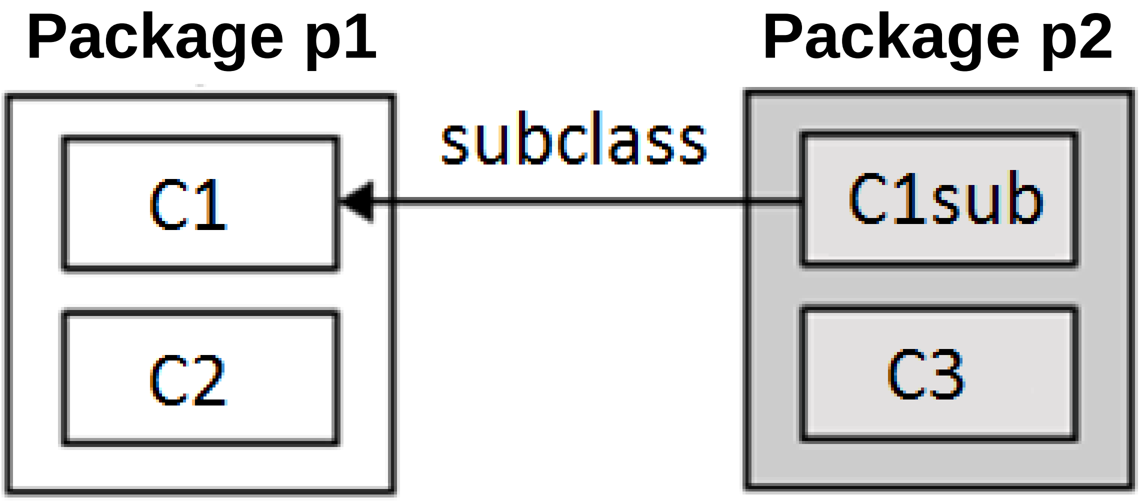
- The class, data, or method is visible to any class in any package.

`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.

Visibility Modifiers

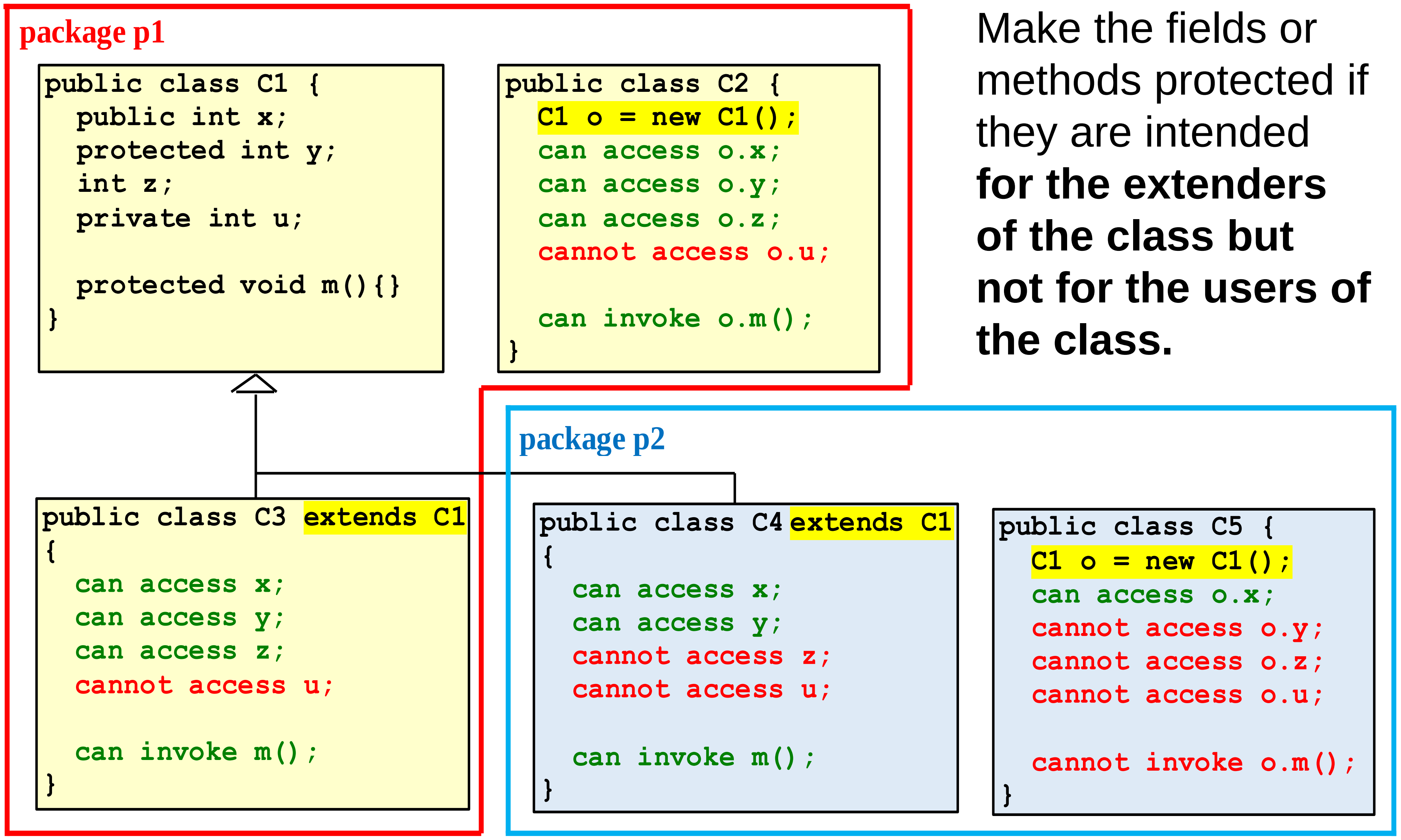


Visibility of a class member in C1

Modifier	C1	C2	C1sub	C3
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
Private	Yes	No	No	No

NOTE
Java 9 introduces a new feature: Java modules, which allows for more accessibility levels (e.g. public to module only instead of to all) but we won't discuss it in this class.

Visibility Modifiers



```
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m(){}
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3 extends C1
{
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

```
public class C4 extends C1
{
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

Make the fields or methods protected if they are intended **for the extenders of the class but not for the users of the class.**

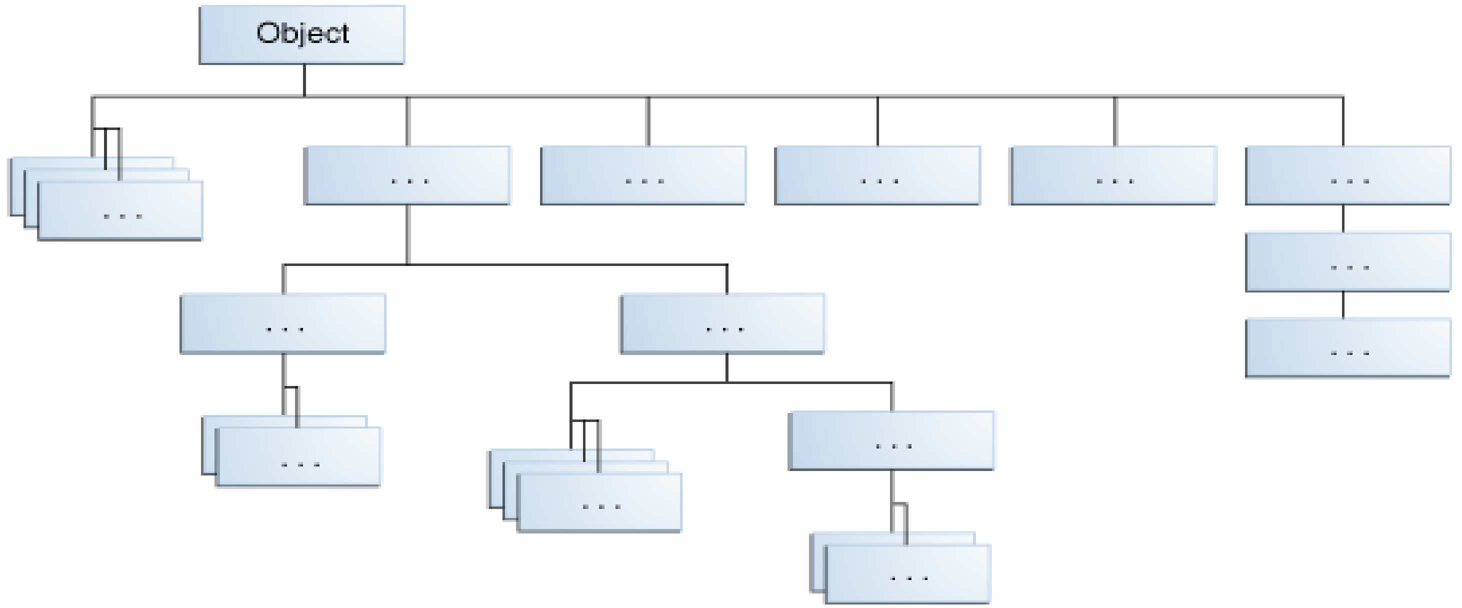
A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The `Object` Class and Its Methods

The `Object` class

Classes in Java are descendants of `java.lang.Object` class



Source: oracle.com

Several methods are inherited from `Object` such as:

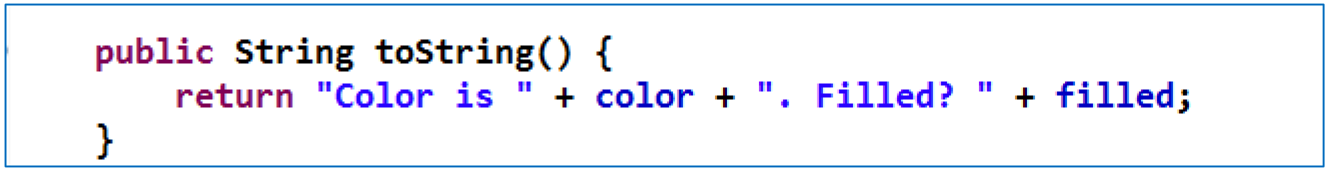
- `public String toString()`
  - Returns a string representation of the object.
- `public boolean equals(Object obj)`
  - Indicates whether some other object is "equal to" this one
- …

The `toString()` method

The `toString()` method returns a string representation of the object.

Usually you should **override the `toString`** method so that it returns a descriptive string representation of the object.

- For example, the `toString` method in the `Object` class was overridden in the `Shape` class presented earlier as follows:



```
public String toString() {
    return "Color is " + color + ". Filled? " + filled;
}
```