

COMP20220

Programming II (Conversion)

Michael O'Mahony



Chapter 9 Objects and Classes

Introduction

Previously, solved programming problems using selections, loops, methods, and arrays...

This and subsequent lectures will focus on object-oriented programming.

This lecture – introduce classes and objects.

Next lectures – focus on OOP, inheritance, polymorphism etc.

Introduction

Part I: Fundamentals of Programming

Chapter 1 Introduction to Computers, Programs, and Java



Chapter 2 Elementary Programming



Chapter 3 Selections



Chapter 4 Mathematical Functions, Characters, and Strings



Chapter 5 Loops



Chapter 6 Methods



Chapter 7 Single-Dimensional Arrays



Chapter 8 Multidimensional Arrays

Part II: Object-Oriented Programming

Chapter 9 Objects and Classes



Chapter 10 Thinking in Objects



Chapter 11 Inheritance and Polymorphism



Chapter 12 Exception Handling and Text I/O

Chapter 13 Abstract Classes and Interfaces



OO Programming Concepts

Object-oriented programming (OOP) involves programming using objects.

An *object* represents an entity in the real world that can be distinctly identified:

- For example, a student, a desk, a circle, a button, a loan, a car, a house etc. can all be viewed as objects.

An object has both a *state* and *behaviour*. The *state* defines the object, and the *behaviour* defines what the object does.

OO Programming Concepts

The *state* of an object consists of a set of *data fields* (aka *data members* aka *properties*) with their current values:

- A circle object has a data field `radius`, which is the property that characterizes a circle.
- A rectangle object has the data fields `width` and `height`, which are the properties that characterize a rectangle.

The *behaviour* of an object is defined by methods. To *invoke* a method on an object is to ask the object to perform an action.

- For example, methods named `getArea()` and `getPerimeter()` can be defined for circle objects.
- Then a circle object may invoke `getArea()` to return its area and `getPerimeter()` to return its perimeter.

Classes

Objects of the same type are defined using a common *class*.

A class is a template (or blueprint) for creating objects. It defines what an object's data fields and methods will be.

An object is an *instance* of a class:

- Many instances of a class can be created.
- (Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable.)

The relationship between classes and objects is analogous to that between, for example, an apple-pie recipe and apple pies:

- You can make as many apple pies as you want from a single recipe...

Classes

A Java class uses variables to define data fields (*state*) and methods to define actions (*behaviour*).

Additionally, a class provides methods of a special type, known as *constructors*, which are invoked in order to create a new object (a new instance of a class).

Constructors are typically designed to perform initializing actions, such as initializing the data fields of objects.

Example Class – Circle

```
class Circle {  
    // The radius  
    double radius;
```

← Data field

```
    // Construct a circle object  
    Circle() {  
        radius = 1;  
    }
```

← Constructors

```
    // Construct a circle object  
    Circle(double newRadius) {  
        radius = newRadius;  
    }
```

```
    // Return the area  
    double getArea() {  
        return radius * radius * Math.PI;  
    }  
}
```

← Method

Example #1: Defining Classes and Creating Objects

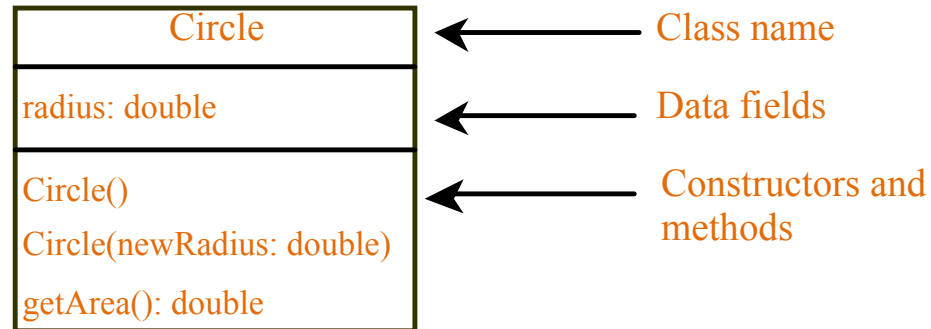
Objective: Demonstrate creating objects, accessing data, and using methods.

SimpleCircle

TestSimpleCircle

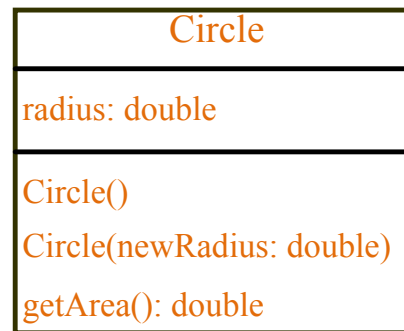
UML Class Diagram

UML Class Diagram



UML Class Diagram

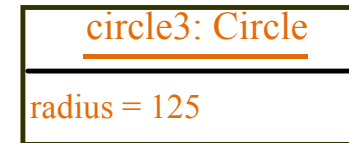
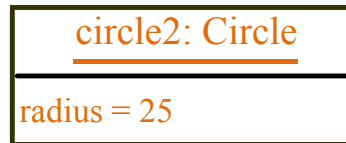
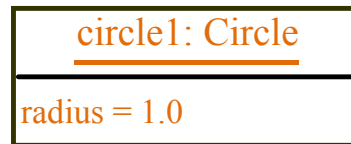
UML Class Diagram



← Class name

← Data fields

← Constructors and methods



← UML notation for objects

Example #2: Defining Classes and Creating Objects

As another example, consider television sets.

Each TV is an object with:

- States (current channel, current volume level, power on or off)
- Behaviours (change channel, adjust volume, turn on/off)

You can use a class to model TV sets. The UML diagram for the class is shown on the next slide

Example #2: Defining Classes and Creating Objects

| TV | |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>channel: int volumeLevel: int on: boolean</pre> | <p>The current channel (1 to 120) of this TV.</p> <p>The current volume level (1 to 7) of this TV.</p> <p>Indicates whether this TV is on/off.</p> |
| <pre>TV() turnOn(): void turnOff(): void setChannel(newChannel: int): void setVolume(newVolumeLevel: int): void channelUp(): void channelDown(): void volumeUp(): void volumeDown(): void</pre> | <p>Constructs a default TV object.</p> <p>Turns on this TV.</p> <p>Turns off this TV.</p> <p>Sets a new channel for this TV.</p> <p>Sets a new volume level for this TV.</p> <p>Increases the channel number by 1.</p> <p>Decreases the channel number by 1.</p> <p>Increases the volume level by 1.</p> <p>Decreases the volume level by 1.</p> |

TV

TestTV

Constructors

Constructors are a special kind of method that are invoked to construct objects:

- A constructor with no parameters is referred to as a *no-arg (no-argument) constructor*.
- Constructors must have the same name as the class itself.
- Constructors do not have a return type (not even `void`).

A class may be defined without constructors:

- In this case, a no-arg constructor with an empty body is implicitly defined in the class.
- This constructor, called a *default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects.

Creating Objects Using Constructors

Examples of constructors:

```
Circle() {  
    radius = 1;  
}
```

```
Circle(double newRadius) {  
    radius = newRadius;  
}
```

Creating objects using constructors:

- General syntax: `new ClassName();`
- Examples:
 - `new Circle();`
 - `new Circle(5.0);`

Declaring Object Reference Variables

To reference an object, assign the object to a reference variable.

To declare a reference variable:

- General syntax: `ClassName objectRefVar;`
- Example: `Circle myCircle;`

Declaring/Creating Objects in a Single Step:

- General syntax: `ClassName objectRefVar = new ClassName();`

Declaring Object Reference Variables

To reference an object, assign the object to a reference variable.

To declare a reference variable:

- General syntax: `ClassName objectRefVar;`
- Example: `Circle myCircle;`

Declaring/Creating Objects in a Single Step:

- General syntax: `ClassName objectRefVar = new ClassName();`
- Example: `Circle myCircle = new Circle();`

Declaring Object Reference Variables

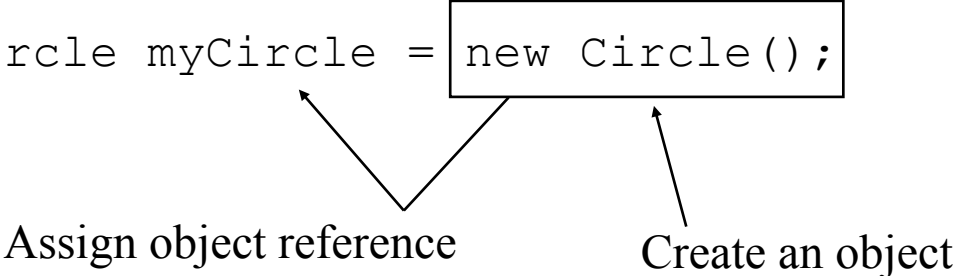
To reference an object, assign the object to a reference variable.

To declare a reference variable:

- General syntax: `ClassName objectRefVar;`
- Example: `Circle myCircle;`

Declaring/Creating Objects in a Single Step:

- General syntax: `ClassName objectRefVar = new ClassName();`

- Example: `Circle myCircle = new Circle();`


Assign object reference

Create an object

Accessing Object's Members

In OOP terminology, an object's *member* refers to its data fields and methods.

After an object is created, use the *dot operator* (.) to access its data fields and to invoke its methods.

To reference the data fields of an object:

- General syntax: `objectRefVar.dataField`
- Example: `myCircle.radius`

To invoke the methods of an object:

- General syntax: `objectRefVar.methodName(arguments)`
- Example: `myCircle.getArea()`

Trace Code

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

animation

Trace Code

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Declare myCircle

myCircle

no value

animation

Trace Code, cont.

Circle myCircle = new Circle(5.0);

myCircle no value

Circle yourCircle = new Circle();

yourCircle.radius = 100;

| |
|-----------------|
| <u>: Circle</u> |
| radius: 5.0 |

Creates a new
Circle object

Trace Code, cont.

Circle myCircle  **new Circle(5.0);**

Circle yourCircle = new Circle();

yourCircle.radius = 100;

myCircle 

Assign object reference
to myCircle

: Circle

radius: 5.0

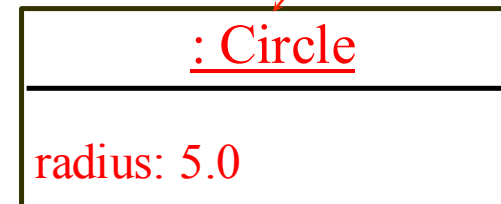
Trace Code, cont.

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

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle no value

Declare yourCircle

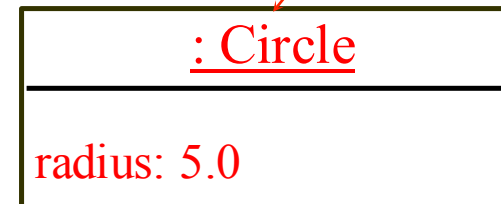
Trace Code, cont.

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

```
Circle yourCircle = new Circle();
```

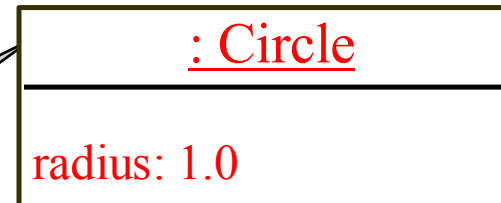
```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle no value

Create a new
Circle object



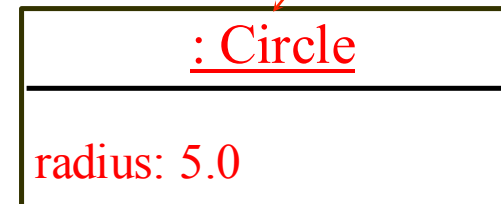
Trace Code, cont.

Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

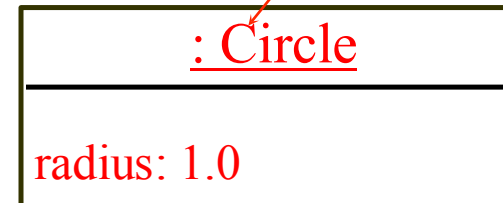
yourCircle.radius = 100;

myCircle reference value



yourCircle reference value

Assign object reference
to yourCircle



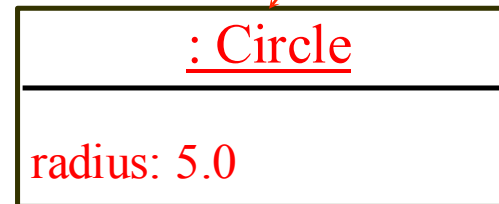
Trace Code, cont.

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

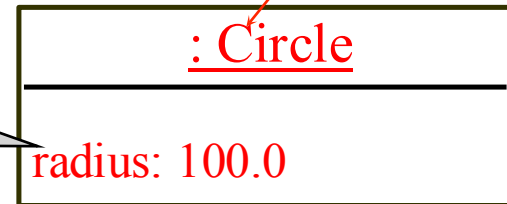
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



Change radius in
yourCircle

Data Fields and Default Values

Data fields can be of reference types.

For example, the following `Student` class contains a data field `name` of the `String` type.

```
public class Student {  
    String name; // default value null  
    int age; // default value 0  
    boolean isScienceMajor; // default value false  
    char gender; // default value '\u0000'  
}
```

The default value of data fields is `null` for reference types, `0` for numeric types, `false` for boolean types, and `'\u0000'` for char types.

- `null` is a literal for reference types just like, for example, `true` and `false` are literals for Boolean types.

Default Values for Data Fields

What is the output of the following code?

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " + student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```

Output:

Default Values for Data Fields

What is the output of the following code?

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " + student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```

Output:

name? null

age? 0

isScienceMajor? false

gender?

Caution

Java does not assign a default value to (local) variables inside a method...

```
public class Test {  
    public static void main(String[] args) {  
        int x;  
        String y;  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```


Caution

Java does not assign a default value to (local) variables inside a method...

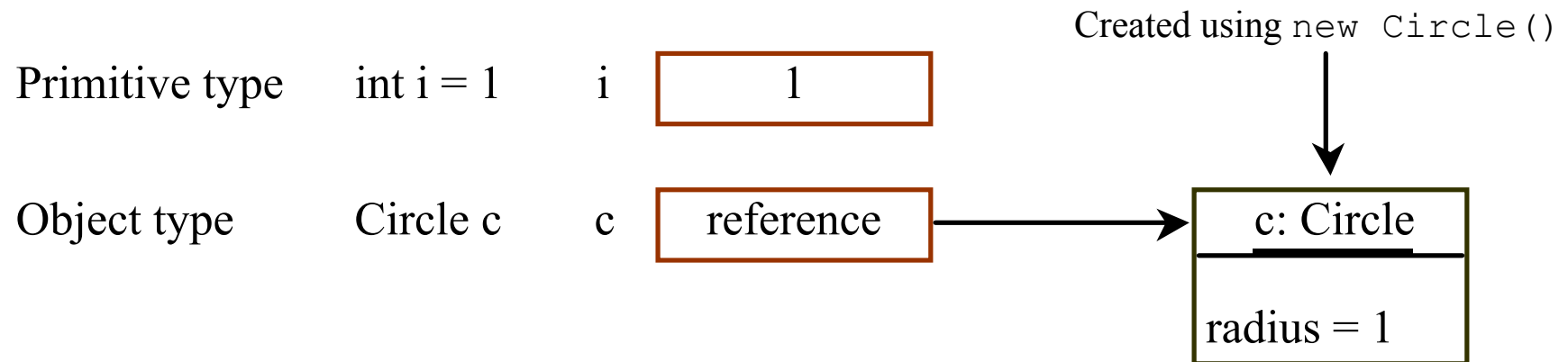
```
public class Test {  
    public static void main(String[] args) {  
        int x;  
        String y;  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```



Compile error: variables not
initialized

Differences between Variables of Primitive Data Types and Object Types

Differences between Variables of Primitive Data Types and Object Types



Copying Variables of Primitive Data Types

```
int i = 1;
```

```
int j = 2;
```

```
i = j; // primitive type assignment
```

Copying Variables of Primitive Data Types

```
int i = 1;
```

```
int j = 2;
```

```
i = j; // primitive type assignment
```

Before:

i 1

j 2

After:

i 2

j 2

Copying Variables of Object Types

```
Circle c1 = new Circle(5);
```

```
Circle c2 = new Circle(9);
```

```
c1 = c2; // object type assignment
```

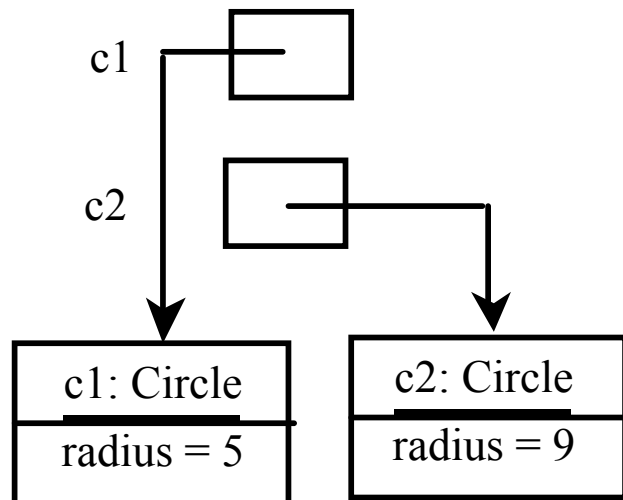
Copying Variables of Object Types

```
Circle c1 = new Circle(5);
```

```
Circle c2 = new Circle(9);
```

```
c1 = c2; // object type assignment
```

Before:



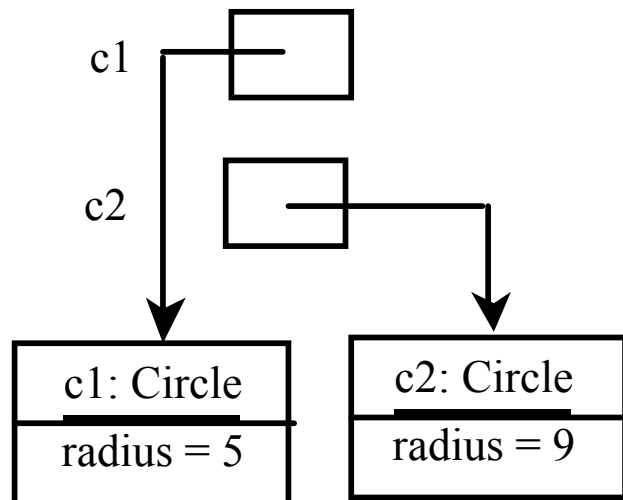
Copying Variables of Object Types

```
Circle c1 = new Circle(5);
```

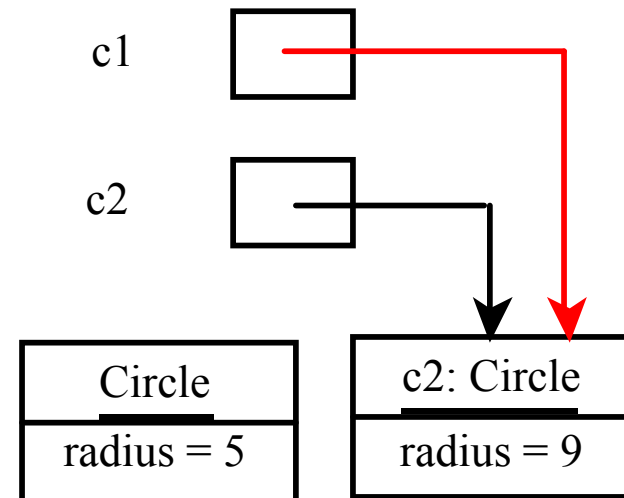
```
Circle c2 = new Circle(9);
```

```
c1 = c2; // object type assignment
```

Before:



After:



Garbage Collection

As shown in the previous example, after the assignment statement $c1 = c2$, $c1$ points to the same object referenced by $c2$.

The object previously referenced by $c1$ is no longer referenced.

This object is known as *garbage* – i.e. when it is no longer referenced by any reference variable.

Garbage is automatically collected by the JVM.

Example Class – Circle

```
class Circle {  
    // The radius  
    double radius;
```

← Data field

```
    // Construct a circle object  
    Circle() {  
        radius = 1;  
    }
```

← Constructors

```
    // Construct a circle object  
    Circle(double newRadius) {  
        radius = newRadius;  
    }
```

```
    // Return the area  
    double getArea() {  
        return radius * radius * Math.PI;  
    }  
}
```

← Method

Instance Variables and Methods

The data field `radius` in the `circle` class is known as an *instance variable*.

An instance variable is tied to a specific instance of the class – it is not shared among objects of the same class.

For example, suppose that you create the following objects:

```
Circle c1 = new Circle(2);  
Circle c2 = new Circle(5);
```

The radius in `c1` is independent of the radius in `c2`.

Changes to `c1`'s radius do not affect `c2`'s radius, and vice versa.

Instance variables and instance methods are accessed via a reference variable – for example `c1.radius`, `c1.getArea()`.

Static Variables, Constants, Methods

If you want all the instances of a class to share data, use *static variables* (aka *class variables*).

Static variables are shared by all objects of the class. If one object changes the value of a static variable, all objects of the same class are affected. Static methods are not tied to a specific object.

Constants in a class are shared by all objects of the class. Constants should be declared as `final static`. For example, the constant `PI` in the `Math` class is defined as:

```
final static double PI = 3.14159...;
```

Static variables, constants and methods should be accessed from their class name – for example `Math.PI`, `Math.pow(2, 3)`

To declare static variables, methods, and constants, use the `static` modifier.

Scope of Instance and Static Variables

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

Typically, instance and static variables are declared at the beginning of a class.

Example Using Instance and Static Variables and Methods

Objective:

- Demonstrate the roles of instance and static variables and their uses.
- This example adds a static variable `numberOfObjects` and a static method `getNumberOfObjects()` to track the number of `Circle` objects created.

[CircleWithStaticMembers](#)

[TestCircleWithStaticMembers](#)

Visibility Modifiers

Visibility modifiers can be used to specify the visibility of a class and its members.

If no visibility modifier is used, then classes, data fields and methods can be accessed by any class in the same *package*.

`public` visibility modifier – classes, data fields and methods can be accessed by any class in any package.

`private` visibility modifier – data fields and methods can be accessed only by the declaring class.

Visibility Modifiers

```
package p1;  
  
class C1 {  
    ...  
}
```

```
package p1;  
  
public class C2 {  
    can access C1  
}
```

```
package p2;  
  
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

If a class is not defined as `public`, it can be accessed only within the same package.

Class `C1` can be accessed from `C2` but not from `C3`.

Visibility Modifiers

```
package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}
```

```
package p1;

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

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}
```

```
package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}
```

The `private` modifier restricts access to within a class, the default modifier restricts access to within a package, and the `public` modifier enables unrestricted access.

Data Field Encapsulation

Making data fields `private` makes code robust and classes easy to maintain.

To prevent direct modifications of data fields, data fields should be declared `private`, using the `private` modifier.

This is known as *data field encapsulation*.

Getter and Setter Methods

A private data field cannot be accessed by an object from outside the class that defines it.

However, a client program often needs to retrieve and/or modify data fields:

- To make a private data field accessible, use a *getter* (aka *accessor*) method.
- To enable a private data field to be updated, use a *setter* (aka *mutator*) method.
- Each data field has its own getter/setter methods.

```
public class Circle {  
    private double radius;  
  
    public Circle() {  
        radius = 1;  
    }  
  
    public double getRadius() { // getter method  
        return radius;  
    }  
  
    public void setRadius(double newRadius) { // setter method  
        radius = newRadius;  
    }  
}
```

Getter and Setter Methods

A getter method has the following signature:

- `public returnType getPropertyName()`
- **Example:** `public double getRadius()`

A setter method has the following signature:

- `public void setPropertyName(dataType propertyValue)`
- **Example:** `public void setRadius(double radius)`

Data Field Encapsulation

Why is it good practice?

- Suppose you want to modify the `Circle` class to ensure that the radius is non-negative – after other programs have already used the class.
- Need to change not only the `Circle` class but also the programs that use it, because these programs may have modified the radius directly (e.g., `c1.radius = -5`).

Make all data fields private.

Provide getter/setter methods when you wish to allow a client program to retrieve/modify data fields.

this Reference

this is a reference that refers to an object itself.

There are two common uses for the reference `this` ...

this Reference

Using `this` to reference a class's *hidden instance data fields*:

- A data field name is often used as the parameter name in a setter method – in such cases, the data field is said to be *hidden* in the setter method
- A hidden instance variable is accessed using the keyword `this`

A *hidden static variable* is accessed using `ClassName.variable`

this Reference

Using `this` to reference a class's *hidden instance data fields*:

- A data field name is often used as the parameter name in a setter method – in such cases, the data field is said to be *hidden* in the setter method
- A hidden instance variable is accessed using the keyword `this`

A *hidden static variable* is accessed using `ClassName.variable`

```
public class F {  
    private int j = 1;  
    private static int k = 2;  
  
    public void setJ(int newJ) {  
        j = newJ;  
    }  
  
    public static void setK(int newK) {  
        k = newK;  
    }  
    ...  
}
```


this Reference

Using `this` to reference a class's *hidden instance data fields*:

- A data field name is often used as the parameter name in a setter method – in such cases, the data field is said to be *hidden* in the setter method
- A hidden instance variable is accessed using the keyword `this`

A *hidden static variable* is accessed using `ClassName.variable`

```
public class F {  
    private int j = 1;  
    private static int k = 2;  
  
    public void setJ(int newJ) {  
        j = newJ;  
    }  
  
    public static void setK(int newK) {  
        k = newK;  
    }  
    ...  
}
```

≡

```
public class F {  
    private int j = 1;  
    private static int k = 2;  
  
    public void setJ(int j) {  
        this.j = j;  
    }  
  
    public static void setK(int k) {  
        F.k = k;  
    }  
    ...  
}
```

this Reference

Using `this` to enable a constructor to invoke another constructor of the same class.

this Reference

Using `this` to enable a constructor to invoke another constructor of the same class.

```
public class Circle {  
    private double radius;  
  
    public Circle() {  
        radius = 1.0;  
    }  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    ...  
}
```

this Reference

Using `this` to enable a constructor to invoke another constructor of the same class.

```
public class Circle {  
    private double radius;  
  
    public Circle() {  
        radius = 1.0;  
    }  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    ...  
}
```

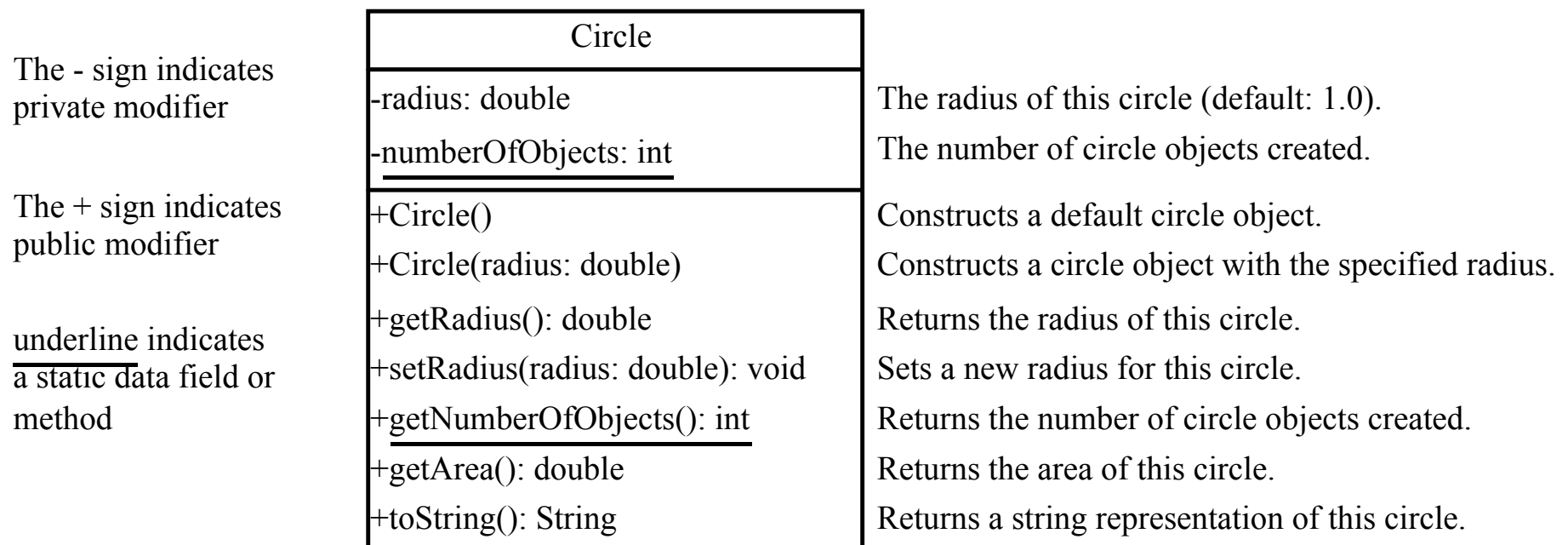


```
public class Circle {  
    private double radius;  
  
    public Circle() {  
        this(1.0);  
    }  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    ...  
}
```

Example

Bringing it all together – example using getter (accessor) and setter (mutator) methods and using the `this` reference.

UML diagram for class `Circle`:



Circle

TestCircle

Immutable Objects and Classes

Normally, you create an object and allow its contents to be changed at a later time as needed (e.g. updating the radius of a `Circle` object).

Occasionally it is desirable to create an object whose contents cannot be changed once the object has been created.

To this end, *immutable classes* can be defined to create *immutable objects*. The contents of immutable objects cannot be changed.

For example, the `String` class is designed to be immutable.

Immutable Objects and Classes

If the setter method in the `Circle` class was deleted, then the class would be immutable (because `radius` is private and cannot be changed without a setter method).

```
public class Circle {  
    private double radius;  
  
    public Circle() {  
        radius = 1;  
    }  
  
    public double getRadius() { // getter method  
        return radius;  
    }  
  
    public void setRadius(double radius) { // setter method  
        this.radius = radius;  
    }  
    ...  
}
```

Immutable Objects and Classes

However – a class with all private data fields and without setter methods is not necessarily immutable.

For example, the following class `Student` has all private data fields and no setter methods, but it is mutable.


```
public class Student {  
    private int id;  
    private BirthDate birthDate;  
  
    public Student(int ssn,  
        int year, int month, int day) {  
  
        id = ssn;  
        birthDate =  
            new BirthDate(year, month, day);  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public BirthDate getBirthDate() {  
        return birthDate;  
    }  
}
```

```
public class Student {  
    private int id;  
    private BirthDate birthDate;  
  
    public Student(int ssn,  
        int year, int month, int day) {  
  
        id = ssn;  
        birthDate =  
            new BirthDate(year, month, day);  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public BirthDate getBirthDate() {  
        return birthDate;  
    }  
}
```

```
public class BirthDate {  
    private int year;  
    private int month;  
    private int day;  
  
    public BirthDate(int year,  
        int month, int day) {  
  
        this.year = year;  
        this.month = month;  
        this.day = day;  
    }  
  
    public void setYear(int year) {  
        this.year = year;  
    }  
}
```

```
public class Student {
    private int id;
    private BirthDate birthDate;

    public Student(int ssn,
        int year, int month, int day) {

        id = ssn;
        birthDate =
            new BirthDate(year, month, day);
    }

    public int getId() {
        return id;
    }

    public BirthDate getBirthDate() {
        return birthDate;
    }
}
```

```
public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int year,
        int month, int day) {

        this.year = year;
        this.month = month;
        this.day = day;
    }

    public void setYear(int year) {
        this.year = year;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}
```

Immutable Objects and Classes

For a class to be immutable, it must meet the following requirements:

- All data fields must be private.
- No setter (mutator) methods for data fields.
- No getter (accessor) methods that can return a reference to a data field that is mutable.

Passing Objects to Methods

Recall – Java uses exactly one mode of passing arguments to methods: pass-by-value.

Passing an object to a method is to pass the *reference* of the object to the method.

(Same as passing an array to a method – the array reference variable is passed.)

Examples – illustrate pass-by-value and the difference between passing a primitive type and an object to a method.

TestPassObject1

TestPassObject2

Arrays of Objects

An array can hold objects as well as primitive type values.

Recall – to declare and create a primitive type array of 10 integers:

```
int[] array = new int[10];
```

The following statement declares and creates an array of 10 `Circle` objects:

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

To initialize `circleArray`, you can use a `for` loop like this for example:

```
for (int i = 0; i < circleArray.length; i++) {  
    circleArray[i] = new Circle(i + 1);  
}
```

Array of Objects

An array of objects is actually an *array of reference variables*.

Invoking `circleArray[1].getArea()` involves two levels of referencing:

- `circleArray` references the entire array
- `circleArray[1]` references a `Circle` object

Array of Objects

An array of objects is actually an *array of reference variables*.

Invoking `circleArray[1].getArea()` involves two levels of referencing:

- `circleArray` references the entire array.
- `circleArray[1]` references a `Circle` object.

Example:

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


Array of Objects

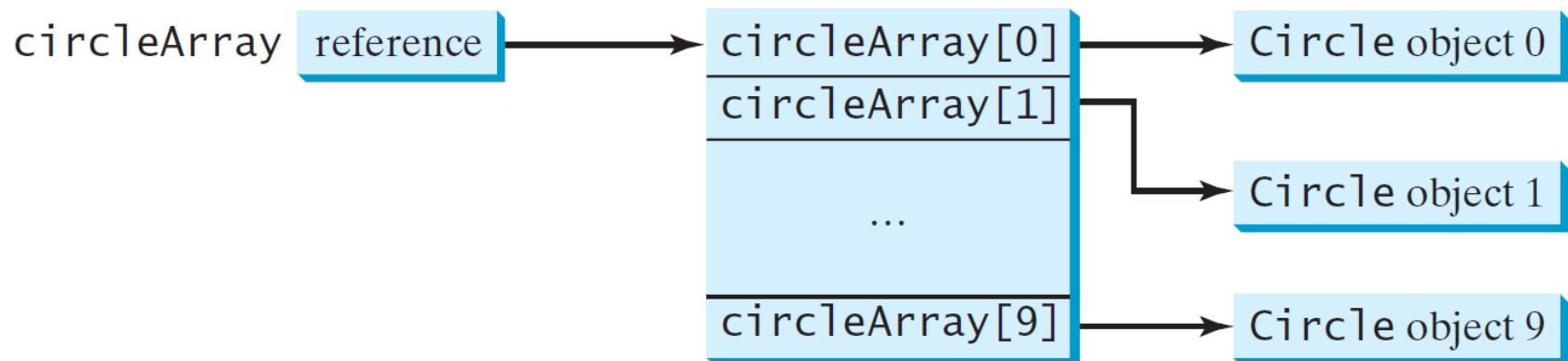
An array of objects is actually an *array of reference variables*.

Invoking `circleArray[1].getArea()` involves two levels of referencing:

- `circleArray` references the entire array.
- `circleArray[1]` references a `Circle` object.

Example:

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



Array of Objects

Example – using arrays of objects and passing/returning arrays of objects to/from methods.



TotalArea

Next topics...

This lecture: concluded Chapter 9 – introducing classes and objects.

Next lectures: Chapter 10 – designing programs using the object-oriented paradigm.