SWEN20003
Object Oriented Software Development

Classes and Objects - 2

Bach Le
bach.le@unimelb.edu.au

University of Melbourne

The Road So Far

Lectures

- Subject Introduction
- A Quick Tour of Java
- Classes and Objects - 1

Learning Outcomes

Upon completion of this topic, which includes three lectures, you will be able to:

- Explain the difference between a *class* and an *object*
- Create classes, give them *properties* and *behaviours*, implement and use simple classes
- Identify a series of well-defined classes from a *specification*
- Understand the role of *getters*, *setters* and *constructors*
- Understand the differences between *instance*, *static* and *local* variables
- Understand the role of *standard methods* in java
- Explain object oriented concepts: *abstraction, encapsulation, information hiding and delegation*
- Understand the role of *wrapper* classes

Overview

This topic will be delivered through three lectures (Lectures 3, 4 and 5) each covering the following subtopics.

Classes and Objects - 1

- Introducing Classes and Objects
- Defining Classes
- Using Classes

Classes and Objects - 2

- *Getters, Setters and Constructors*
- *Static Attributes and Methods*
- *Standard Methods in Java*

Classes and Objects - 3

- Introducing Java Packages
- Information Hiding
- Delegation through Association
- Wrapper Classes

Getters, Setters and Constructors

Getters and Setters

- Generally initialising/updating/accessing instance variables is done by defining specific methods for each purpose.
- These methods are called **Accessor/Mutator** methods or informally as **Getter/Setter** methods.
- Initialise/update an instance variable using:

```
aCircle.setX(10.0); // mutator method or setter
```

- Access an instance variable using:

```
double x = aCircle.getX(); // accessor method or getter
```

- Usually IDEs such as IntelliJ, Eclipse IDE support automatic code generation for getters and setters.
- You will see better reasons for using getters and setters when we learn topics such as *information hiding, visibility control and privacy*.
  So please be patient if you are not convinced as to why we are doing this!

The Circle Class with Getters and Setters

```java
public class Circle {
    public double centreX, centreY, radius;
    public double getCentreX() {
        return centreX;
    }
    public void setCentreX(double centreX) {
        this.centreX = centreX;
    }
    public double getCentreY() {
        return centreY;
    }
    public void setCentreY(double centreY) {
        this.centreY = centreY;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    } // The rest of the code as before go below
}
```

Using the Circle Class with Getters and Setters

```java
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
  public static void main(String args[]) {
    Circle aCircle = new Circle();
    aCircle.setCentreX(10.0);
    aCircle.setCentreY(20.0);
    aCircle.setRadius(5.0);
    System.out.println("Radius = " + aCircle.getRadius());
    System.out.println("Circum: = " + aCircle.computeCircumference());
    System.out.println("Area = " + aCircle.computeArea());
    aCircle.resize(2.0);
    System.out.println("Radius = " + aCircle.getRadius());
  }
}
```

Program Output:

```
Radius = 5.0
Circum: = 31.41592653589793
Area = 78.53981633974483
Radius = 10.0
```

Initializing Objects

- When objects are created, the initial value of the instance variables are set to default values based on the data type.
- In the previous examples, we set the initial values using the mutator/setter methods.

```java
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.setCentreX(10.0);
        aCircle.setCentreY(20.0);
        aCircle.setRadius(5.0);
    }
}
```

  - What if we have 100 attributes to initialise?
  - What if we have 100 objects to initialise?
  - We need a better... **method**

.

Constructors

How does this actually work?

```java
            Circle aCircle = new Circle();
```

- The right hand side *invokes* (or calls) a class' *constructor*
- Constructors are **methods**
- Constructors are used to initialize objects
- Constructors have the same name as the class
- Constructors cannot return values
- A class can have **one or more** constructors, each with a different set of parameters (called overloading; we'll cover this later)

Keyword

Constructor: A method used to **create** and **initialise** an object.

Defining Constructors

```
public <ClassName>(<arguments>) {
    <block of code to execute>
}
```

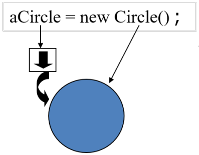
Default `Circle` constructor:

```
        public class Circle {
            public double centreX, centreY, radius;
            public Circle() {
                centreX = 10.0;
                centreY = 10.0;
                radius = 5.0;
            }
            // More code here
        }
```

Using Constructors

Previous Code (without a `Circle` constructor):

```
Circle aCircle = new Circle();
```

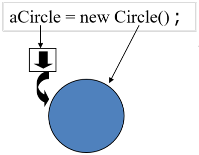


At creation time the center and radius are not defined.

aCricle will have a centre (0.0, 0.0) and radius 0.0 – default values for variables.

New Code (with `Circle` Constructors):

```
Circle aCircle = new Circle();
```



At creation time the constructor with no arguments will be called.

aCricle will have a centre (10.0, 10.0) and radius 5.0 – default values for variables.

Constructor Test - Example

```java
public class CircleConstructorTest {
    public static void main(String args[]) {

        Circle circle_1 = new Circle();
        System.out.println("Defined circle_1 with centre (" +
           circle_1.getCentreX()    + ", " + circle_1.getCentreY()
           + ") and radius " + circle_1.getRadius());
}
```

Program Output:

```
Defined circle_1 with centre (10.0, 10.0) and radius 5.0
```

What is not so good about this constructor?

The good news is:

- constructors can take arguments like any other methods
- a class can have more than one constructor

The `Circle` class with more Constructors

```java
public Circle() {
  public double centreX, centreY, radius;
  public Circle() {
    centreX = 10.0;
    centreY = 10.0;
    radius = 5.0;
  }
  public Circle(double newCentreX, double newCentreY, double newRadius) {
    centreX = newCentreX;
    centreY = newCentreY;
    radius = newRadius;
  }
  public Circle(double newCentreX, double newCentreY) {
    centreX = newCentreX;
    centreY = newCentreY;
  }
  public Circle(double newRadius) {
    radius = newRadius;
  }
  // More code here
}
```

Constructor Test - Example

```java
public class CircleConstructorTest {
    public static void main(String args[]) {

        Circle circle_1 = new Circle();
        System.out.println("Defined circle_1 with centre (" +
            circle_1.getCentreX()   + ", " circle_1.getCentreY()
            + ") and radius " + circle_1.getRadius());

        Circle circle_2 = new Circle(10.0, 20.0, 12.2);
        System.out.println("Defined circle_2 with centre (" +
            circle_2.getCentreX()   + ", " + circle_2.getCentreY() + ")
            and radius " + circle_2.getRadius());
    }
}
```

Program Output:

```
Defined circle_1 with centre (10.0, 10.0) and radius 5.0
Defined circle_2 with centre (10.0, 20.0) and radius 12.2
```

Method Overloading

- Methods have the same name; are distinguished by their signature:
  - number of arguments
  - type of arguments
  - position of arguments
- Any method can be overloaded (Constructors or other methods).
- **Method Overloading:** This is a form of *polymorphism* (same method – different behaviour).
- *Do not to confuse with Method Overriding (coming up soon!).*

Method Overloading and Polymorphism

Keyword

Polymorphism: Ability to process objects differently depending on their data type or class.

Keyword

Method Overloading: Ability to define methods with the same name but with different signatures (argument types and/or numbers).

Pitfall: Constructors

Let us look at our previous definition of the `Circle` Constructor.

```java
public Circle(double newCentreX, double newCentreY, double newRadius) {
    centreX = newCentreX;
    centreY = newCentreY;
    radius = newRadius;
}
```

But what if we did the following instead?

```java
public Circle(double centreX, double centreY, double radius) {
    centreX = centreX;
    centreY = centreY;
    radius = radius;
}
```

How does the code differentiate the two variables?

Introducing the `this` Keyword

Keyword

this: A **reference** to the **calling object**, the object that owns/is executing the method.

```java
public class Circle {
    public double centreX, centreY, radius;

    public Circle() {
        this.centreX = 10.0;
        this.centreY = 10.0;
        this.radius = 5.0;
    }

    public Circle(double centreX, double centreY, double radius) {
        this.centreX = centreX;
        this.centreY = centreY;
        this.radius = radius;
    }
    // More methods go here
}
```

Static Attributes and Methods

Defining Static Variables

How would you count the number of `Circle` objects that you created in your `CircleConstructorTest` program?

```java
public class CircleConstructorTest {
    public static void main(String args[]) {

        Circle circle_1 = new Circle();
        System.out.println("Defined circle_1 with centre (" +
            circle_1.getCentreX()   + ", " circle_1.getCentreY()
            + ") and radius " + circle_1.getRadius());

        Circle circle_2 = new Circle(10.0, 20.0, 12.2);
        System.out.println("Defined circle_2 with centre (" +
            circle_2.getCentreX()   + ", " + circle_2.getCentreY() + ")
            and radius " + circle_2.getRadius());
    }
}
```

Defining Static Variables

Static attributes are shared between objects (only one copy): e.g. count of the number of objects of the type that has been created.

Adding a static attribute, numCircles, to the Circle class.

```java
// Circle.java
public class Circle {
    // static (class) variable - one instance for the Circle class, number of c
    public static int numCircles = 0;
    public double centreX, centreY, radius;

    // Constructors and other methods
    public Circle(double x, double y, double r){
        centreX = x; centreY = y; radius = r;
        numCircles++;
    }
    // Other methods go here
}
```

Using Static Variables

Let us now write a class `CountCircles` to use the static variable.

```java
// CountCircles.java
public class CountCircles {
  public static void main(String args[]) {
    Circle circleA = new Circle( 10.0, 12.0, 20.0);
    System.out.println("Number of Circles  = " + Circle.numCircles);
    Circle circleB = new Circle( 5.0, 3.0, 10.0);
    System.out.println("Number of Circles  = " + Circle.numCircles);
  }
}
```

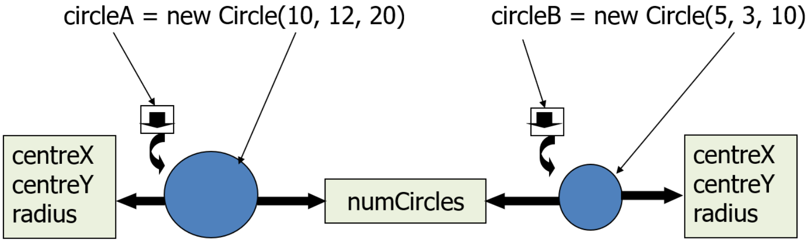
Program Output:

```
Number of Circles  = 1
Number of Circles  = 2
```

Instance vs Static Variables

Instance variables: One copy per object. e.g. centreX, centreY, radius (centre and radius in the circle)

Static variables: One copy per class. e.g. numCircles (total number of circle objects created)


circleA = new Circle(10, 12, 20)
circleB = new Circle(5, 3, 10)
centreX
centreY
radius
numCircles
centreX
centreY
radius

Defining Static Methods

Adding a static method, `printNumCircles`, to the `Circle` class.

```java
// Circle.java
public class Circle {
    // static (class) variable
    public static int numCircles = 0;
    public double centreX, centreY, radius;

    // Constructors and other methods
    public Circle(double x, double y, double r){
        centreX = x; centreY = y; radius = r;
        numCircles++;
    }

    // Static method to count the number of circles
    public static void printNumCircles() {
        System.out.println("Number of circles = " + numCircles);
    }

    // Other methods go here
}

```

Using Static Methods

Using the static method, `printNumCircles()`.

```java
// CountCircles.java
public class CountCircles {
    public static void main(String args[]) {
        Circle circleA = new Circle( 10.0, 12.0, 20.0);
        Circle.printNumCircles();
        Circle circleB = new Circle( 5.0, 3.0, 10.0);
        Circle.printNumCircles();
    }
}
```

Program Output:

```
Number of Circles  = 1
Number of Circles  = 2
```

Using Static Methods

- Static methods can only call other static methods.
- Static methods can only access static data.
- Static methods cannot refer to Java keywords such as, `this` or `super` (will be introduced later) - because they are related to objects (class instances).
- Do not make all methods and attributes in your classes static; if you do that you may end up writing procedural programs using Java as opposed to good OO programs - you will be penalized for doing this in the assignments and exams.

Important: Before you decide to make an attribute or a method `static` think carefully - consider whether it is a class level member or an instance specific member.

Back to the `main` method

When a Java program is executed the Java virtual machine invokes the `main` method, which is a `static` method.

```java
// HelloWorld.java: Display "Hello World!" on the screen
import java.lang.*;
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Static Members

Keyword

Static Members: Methods and attributes that are not specific to any object of the class.

Keyword

Static Variable: A variable that is shared among all objects of the class; a single instance is shared among all objects of the class. Such an attribute is accessed using the class name.

Keyword

Static Method: A method that does not depend on (access or modify) any instance variables of the class. Such a method is invoked (called) using the class name.

Standard Methods in Java

Standard Methods

There are some methods, that are frequently used, that are provided as standard methods in every class.

We will look at three such methods:

- the `equals` method
- the `toString` method
- the copy constructor

Standard Methods - *equals*

```
    public boolean equals(<ClassName>  var) {
        return <boolean expression>;
    }
```

- It is useful to be able to compare if two objects are **equal**
- Doing the equality test with == operator will only check if references are equal as opposed to checking if objects are equal
- How to determine if objects are equal is up to you; use **one or more** properties of the objects
- This is version one; we'll "improve" it as we go

Adding `equals` to `Circle` Class

We will now add the `equals` method to the `Circle` class.

How would you compare a `Circle` object to another `Circle` object?

```java
public boolean equals(Circle circle) {
    return Double.compare(circle.centreX, centreX) == 0 &&
           Double.compare(circle.centreY, centreY) == 0 &&
           Double.compare(circle.radius, radius) == 0;
}
```

Standard Methods - `toString`

- What you if you want to print the attributes of the `Circle` class - is there an easy way?
- What would happen if you have:
  - `System.out.println(c_1);` - `c_1` is a reference to a `Circle` object
- The `toString` method which returns a `String` **representation** of an object is the way to go:
  - It is automatically called when the object is asked to act like a `String`, e.g. **printing** an object using: `System.out.println(c_1);`

```
      public String toString() {
          return <String>;
      }
```

Adding the toString method to the Circle Class

We will now add the toString method to the Circle class.

```java
public class Circle {
    // Other attributes and methods

    public String toString() {
            return "I am a Circle with {" + "centreX=" + centreX +
                    ", centreY=" + centreY +
                    ", radius=" + radius +'}';
    }
}
```

Now, if your program has the following lines of code:

```java
    Circlce aCircle = new Circle(5.0, 5.0, 40.0);
    System.out.println(aCircle);
```

Program Output:

```
I am a Circle with {centreX=5.0, centreY=5.0, radius=40.0}
```

Standard Methods - *Copy Constructor*

```
public <ClassName>(<ClassName> var) {
    <block of code to execute>
}
```

- Is a constructor with a single argument of the same type as the class
- Creates a **separate copy** of the object sent as input
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object
- In case some of the instance variables are references to other objects, a new object with the same state must be created using its copy constructor - *deep copy*

Adding a Copy Constructor to the Circle Class

```java
public class Circle {
    public  double centreX, centreY, radius;
    // Copy Constructor
    Circle (Circle aCircle) {
        if (aCircle == null) {
            System.out.println("Fatal Error."); //Not a valid circle
            System.exit(0);
        }
        this.centreX = aCircle.centreX;
        this.centreY = aCricle.centreY;
        this.radius = aCircle.radius;
    }
    // Other methods
}
```

```java
Circle c1 = new Circle(10.0, 10.0, 5.0); //s new object
Circle c2 = c1; //a reference to the same object pointed by c1
Circle c3 = new Circle(c1); //a new object - state is same as c1
```

Learning Outcomes:

Topics covered in this lecture:

- Getters, Setters and Constructors
- Static Attributes and Methods
- Standard Methods in Java

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Create classes, give them *properties* and *behaviours*, implement and use simple classes
- Identify a series of well-defined classes from a *specification*
- Understand the role of *getters*, *setters* and *constructors*
- Understand the differences between *instance*, *static* and *local* variables
- Understand the role of *standard methods* in java
- Explain object oriented concepts: *abstraction and encapsulation*

References

- Absolute Java by Water Savitch (Fourth Edition), Chapters 4 & 5