SWEN20003
Object Oriented Software Development

Classes and Objects

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne
© University of Melbourne 2020

The Road So Far

Lectures

- Subject Introduction
- A Quick Tour of Java

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Create classes, and give them *properties* and *behaviours*
- Implement and use simple classes
- Identify a series of well-defined classes from a *specification*
- 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.

Lecture 3: Slides 4-34

- Introducing Classes and Objects
- Defining Classes
- Using Classes

Lecture 4: Slides 35-66

- Updating and Accessing Instance Variables
- Static Attributes and Methods
- Standard Methods in Java

Lecture 5: Slides 67-95

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

Introducing Classes and Objects

Introduction

All programming languages support four basic concepts:

- Calculation: constants, variables, operators, expressions
- Selection: `if-else`, `switch`, `?`
- Iteration: `while`, `do`, `for`
- Abstraction: The process of creating self-contained units of software that allows the solution to be parameterized and therefore more general purpose

Abstraction is the fundamental concept that differentiates procedural programming languages such as C from Object Oriented languages such as Java, C++.

Abstraction in Procedural Languages

Abstraction in procedural languages is provided through *functions* or *procedures*.

Functions manipulate external data to by performing operations on them.

Example of a function in C that calculates the average of two floating point numbers:

```
float calculate_average (float a, float b) {
    float result;
    result = (a + b)/2;
    return result;
}
```

Abstraction in Object Oriented Languages

Abstraction in Object Oriented (OO) languages is provided through an*Abstract Data Type (ADT)*, which contains *data* and *functions* that operate on data.

In Java a **Class** is an implementation on an **Abstract Data Type**.

Classes

- A "generalization" of a real world (or "problem world") entity
  - A physical real world thing, like a student or book
  - An abstract real world thing, like a university subject
  - An even more abstract thing like a list or a string (data)
- Represents a template for things that have common properties
- Contains *attributes* and *methods*
- Defines a new **data type**

Keyword

Class: Fundamental unit of abstraction in *Object Oriented Programming*.
Represents an "entity" that is part of a problem.

Objects

- Are an *instance* of a class
- Contain **state**, or dynamic information
- "**X** is of type A", "**X** is an object of the class A", and "**X** is an instance of the class A" are all equivalent

Keyword

Object: A specific, concrete example of a class

Keyword

Instance: An object that exists in your code

Motivating Example

Throughout this topic we will be referring to the following specification:

Develop a system (a set of classes) for a simple Drawing Pad application. The application should allow drawing different types of shapes, such as circles, squares, rectangles, display their geometrical properties: e.g. area, circumference. It should also allow different types of actions such as moving, resizing to the performed on shapes.

How would you develop this, right now? What additional information do you need?

Drawing Pad Application - Classes

What classes can we use for our example problem?

Fundamental:

- `Drawing Pad`
- `Circle`
- `Square`
- Other shapes

Additional:

- `Drawing Tool`
- `Paint Brush`
- `Fill Colour`
- `Fill Type`
- Many more

Identifying Attributes and Methods

Let us consider the `Circle` class.

Can you identify the *attributes* and *methods* of the class?

Attributes:

- `Centre`
- `Radius`
- `Fill Colour`, `Fill Type`
- Many more

Methods (Operations):

- `Compute Circumference`
- `Compute Area`
- `Move`
- `Resize`
- Many more

Object Oriented Features

Following are some key features of the object oriented design paradigm:

- Data Abstraction
- Encapsulation
- Information Hiding
- Delegation
- Inheritance
- Polymorphism

Data Abstraction

Keyword

Data Abstraction: The technique of creating new data types that are well suited to an application by defining new classes.

A class is a special kind of programmer-defined data type.

- For example, by creating classes such as `Circle`, `Drawing Pad`, you are creating new data types, that can be used in applications.

A class is somewhat close to a structure in C but have additional features - attributes and methods.

The class definition determines the types of data (attributes) that an object can contain, as well as the actions (methods) it can perform.

Encapsulation

Keyword

Encapsulation: The ability to group data (attributes) and methods that manipulate the data to a single entity though defining a class.

A class encapsulates data and the methods that operate on the data into a single unit.

This method of encapsulation is unique to OO programming and is not provided by the procedural programming paradigm.

Defining a Class

Defining a Class

Syntax:

```
<visibility modfier> class <ClassName> {
    <attribute declarations>
    <method declarations>
}
```

A bare bone class:

```
// Circle.java - Circle class definition
public class Circle {

}
```

Defining a Class - Adding Attributes

Attribute Syntax:

```
<visibility modifier> <type> <variable name>;
```

Adding attributes (also called data, fields) to the `Circle` class.

```
1       // Circle.java - Circle class definition
2       public class Circle {
3           public double centreX; //centre x coordinate
4           public double centreY; //centre y coordinate
5           public double radius;  //radius
6       }
```

Defining a Class - Adding Attributes

The attributes added to the `Circle` class in the above example are referred to as *instance variables*.

- these attributes maintain the state of the object; i.e. by giving values to `centreX`, `centreY`, `radius` we define a Circle object with particular size and position.

Keyword

Instance Variable: A **property** or **attribute** that is unique to each *instance (object)* of a class.

Defining a Class - Adding Methods

Method Syntax:

```
1       <visibility modifier> <void or typeReturned> myMethod(paramList)
2       {
3           variable declarations
4           statements
5       }
```

- If the method returns data, the data type must be specified in the method definition, otherwise, it is defined as `void`.
- If the method returns data, the method body must contain a return statement, which returns a variable of the specified return type.
- Variables can be declared inside the method - such variables are called *local variables*.
  Note: Local variables are inside the method as opposed to the instance variables (introduced earlier) which are outside the method declaration.

Defining a Class - Adding Methods

Adding methods to `Circle` class.

```
1       // Circle.java
2       public class Circle {
3           public double centreX;
4           public double centreY;
5           public double radius;
6
7           public double computeCircumference () {
8               double circum = 2 * Math.PI * radius;
9               return circum;
10          }
11          public double computeArea () {
12              double area = Math.PI * radius * radius;
13              return area;
14          }
15          public void resize (double factor) {
16              radius = radius * factor;
17          }
18      }
19
```

Using a Class

Using the `Circle` class

Follow the steps below to use the `Circle` class we just created.

- Create a file `Circle.java` and write the code.
  Note: the file name should match the class name.
- You can use an Integrated Development Environment (IDE), such a IntelliJ for this (will be introduced in the workshops), but in this instance you a text editor such as
  `notepad`, `wordpad`, `vim`, `kate` `etc`.
- Compile the class using the following command:

  ```
  javac Circle.java
  ```

  This creates a file `Circle.class`
- `Circle` becomes a derived data type that can be used in a Java program.

Using the Circle class

By creating the Circle class, you have created a new data type Circle - **Data Abstraction**.

- Variables of type Circle can be now defined in a program
- Circle is a **Derived Data Type** (as opposed to a Primitive Data Type such as int, float)

Example:

```
/* CircleTest.java: A test program to test the Circle class */
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle;
        Circle bCircle
    }
}
```

Using the `Circle` class

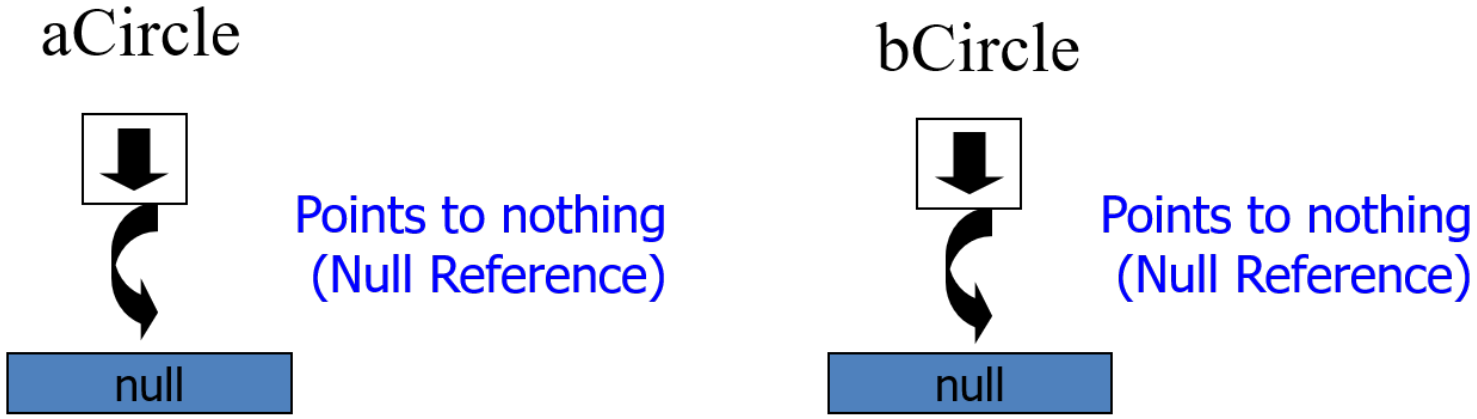
The declarations:

```
Circle aCircle;
Circle bCircle;
```

in the previous example did not create `Circle` objects.

`aCircle` and `bCircle` are simply **references** to `Circle` objects (not objects):

- Currently they point to nothing, hence **null references**.



The `null` Reference

Keyword

null: The Java keyword for "no object here". Null objects can't be "accessed" to get variables or methods, or used in any way.

Instantiating a Class

Objects are **null** until they are *instantiated*.

Keyword

Instantiate: To create an object of a class

```
// Instantiate an Circle object
Circle circle_1 = new Circle();
```

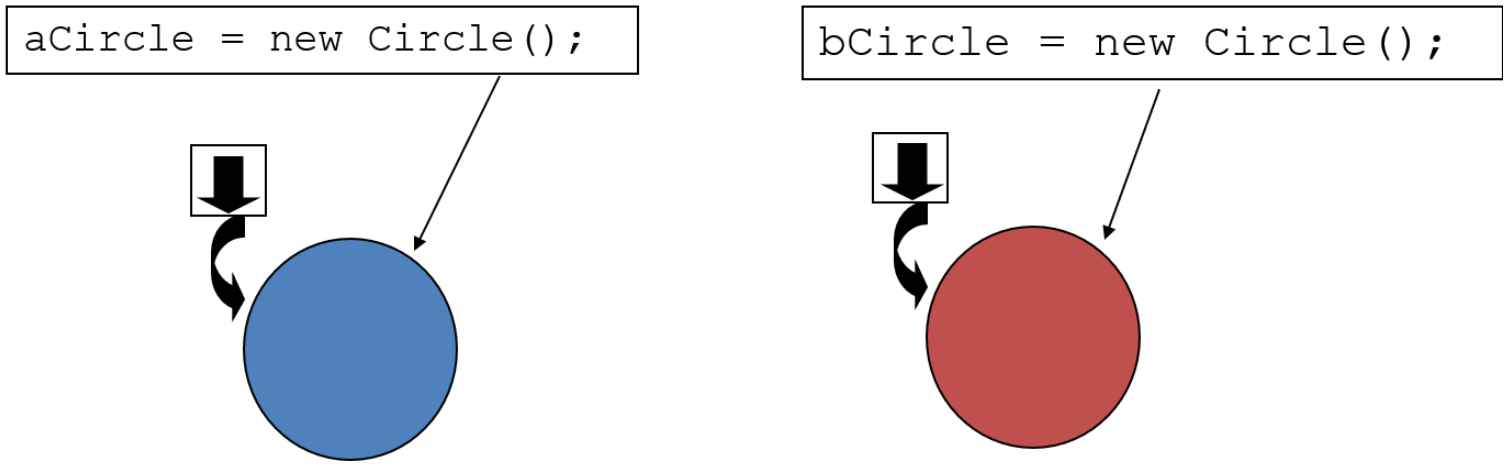
Keyword

new: Directs the JVM to allocate memory for an object, or *instantiate* it

Creating Objects

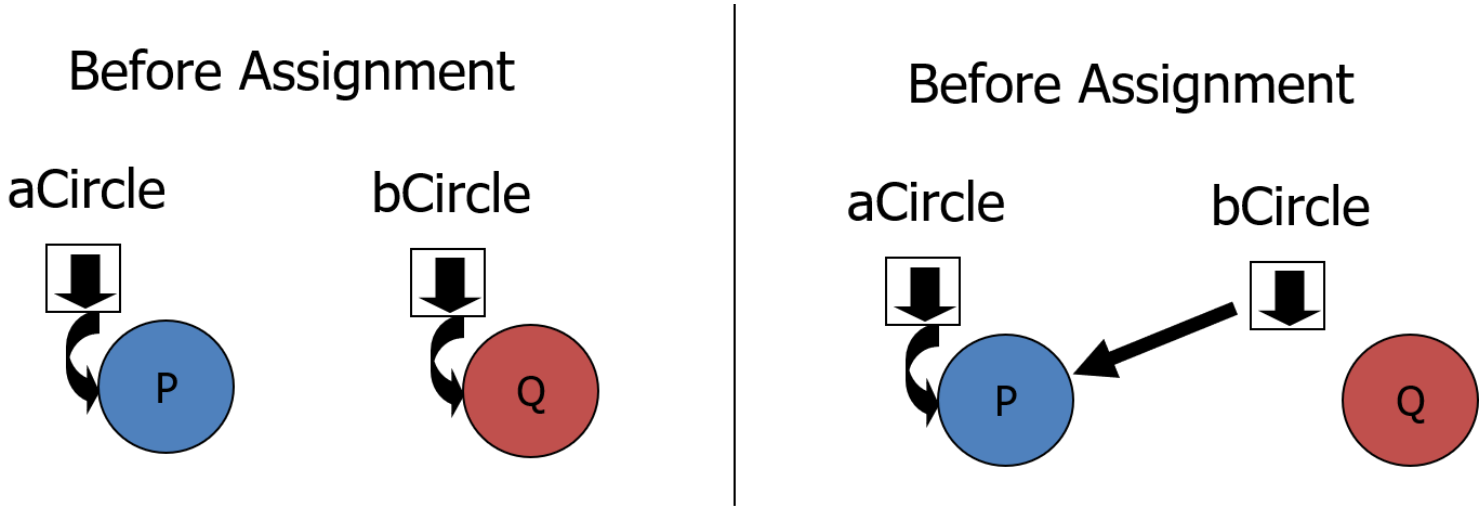
Objects are created dynamically using the `new` keyword.

```
// CircleTest.java: A test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle, bCircle;
        aCircle = new Circle(); //aCircle now points to an object
        bCircle = new Circle(); //bCircle now points to an object
    }
}
```



Assigning References of a Class

```
// CircleTest.java: A test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle, bCircle;
        aCircle = new Circle(); //aCircle now points to an object
        bCircle = new Circle(); //bCircle now points to an object
        bCircle = aCircle; // Assining a class reference
    }
}
```



Garbage Collection in Java

- In the previous example, object **Q** does not have a valid reference and, therefore, cannot be used in future.
- The object becomes a candidate for **Java Automatic Garbage Collection**.
  - Java automatically collects garbage periodically, and frees the memory of unused objects and makes this memory available for future use; you do not have to do this explicitly in the program.

Using Instance Variables and Methods

Syntax:

```
<objectName>.<varibaleName>;
<objectName>.<methodName>(<arguments>);
```

Syntax is similar to C syntax for accessing data defined in a structure.

Example:

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

// Initialize centre and radius
aCircle.centreX = 2.0;
aCircle.centreY = 2.0;
aCircle.radius = 1.0;

//Invoking methods or sending a "message" to methods
area = aCircle.computeArea();
aCircle.resize(2.0);
```

Using the Circle Class - Example

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

Program Output:

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

Back to the `main` method

- A program in Java is just a class that has a `main` method.
- When you give a command to run a Java program, the run-time system invokes the `main` method.
- The `main` is a `void` method, as indicated by its heading:

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

}
```

- `static` - it is still to come - please wait!

Overview

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

Lecture 3: Slides 4-34

- Introducing Classes and Objects
- Defining Classes
- Using Classes

Lecture 4: Slides 35-66

- Updating and Accessing Instance Variables
- Static Attributes and Methods
- Standard Methods in Java

Lecture 5: Slides 67-95

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

Updating and Accessing Instance Variables

Updating and Accessing Instance Variables

- 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); // mutator method or setter
```

- Access an instance variable using:

```
aCircle.getX(10); // accessor method or setter
```

- 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

```
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

```
// 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 using Constructors

- 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.

```
// 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?

```
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 Circle() {
    centreX = 10.0;
    centreY = 10.0;
    radius = 5.0;
}
```

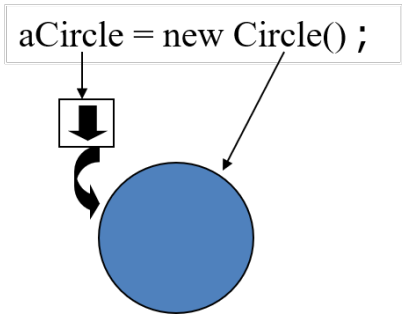
More useful `Circle` constructor:

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

Using Constructors

Previous Code (without a `Circle` constructor):

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

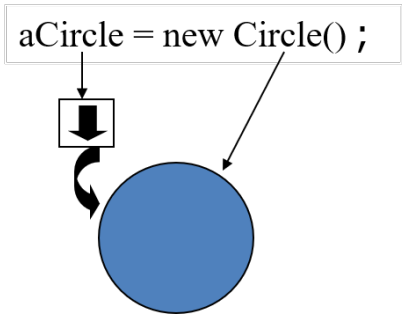


At creation time the center and radius are not defined.

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

```
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
```

The Circle class with more Constructors

```
public class 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
}
```

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.

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

But what if we did the following instead?

```
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.

```
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

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 classes. 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.

Defining Static Variables

Static attribute 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.

```
// Circle.java
pulic class Circle {
    // static (class) variable - one instance for the Circle class, num
    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.

```
// 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 );
    }
}
```

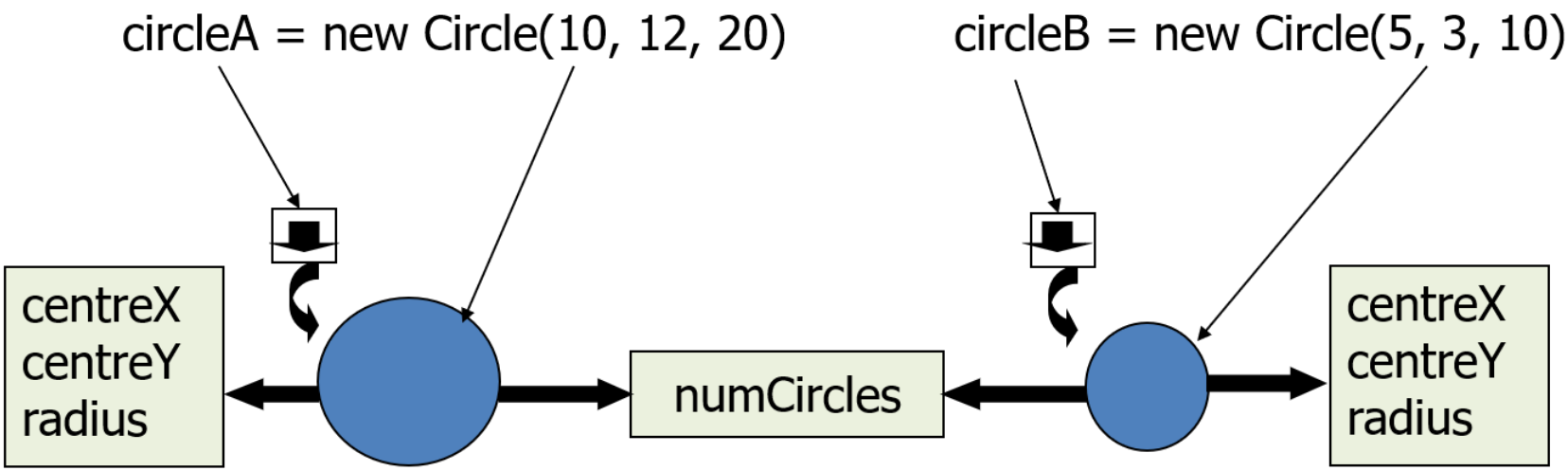
Program Output:

```
    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)



Defining Static Methods

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

```
// Circle.java
pulic 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()`.

```
// 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.

```
// 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!");
    }
}
```

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?

```
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.

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

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

```
    System.out.println(new Circle(5.0, 5.0, 40.0));
```

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

```
public class Circle {
    public  double centreX, centreY, radius;
    // Copy Constructor
    Circle (Circle aCircle) {
        if (aCicle == 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
}
```

```
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
```

Overview

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

Lecture 3: Slides 4-34

- Introducing Classes and Objects
- Defining Classes
- Using Classes

Lecture 4: Slides 35-66

- Updating and Accessing Instance Variables
- Static Attributes and Methods
- Standard Methods in Java

Lecture 5: Slides 67-95

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

Introducing Java Packages

Packages in Java

Keyword

Package: Allows to group classes and interfaces (will be introduced later) into bundles, that can then be handled together using an accepted naming convention.

Why would you group classes into packages?

- Works similar to libraries in C; can be developed, packaged, imported and used by other Java programs/classes.
- Allows reuse, rather than rewriting classes, you can use existing classes by importing them.
- Prevents naming conflicts.
  - Classes with the same name can be used in a program, uniquely identifying them by specifying the package they belong to.
- Allows access control - will learn more when we learn *Information Hiding/Visibility Control*.
- It is another level of **Encapsulation**.

Creating Java Packages

- To place a class in a package, the first statement in the Java class must be the `package` statement with the following syntax:

```
package <directory_name_1>.<directory_name_2>;
```

  - This implies that the class in `directory_2`, which is a sub-directory of `directory_1`.

Example:

```
package utilities.shapes;

public class Circle {
    // Code for Circle goes here
}
```

  - `Circle.class` must be in directory `shapes`, which is a sub-directory of directory `utilities`

Using Java Packages

- To use classes in a package, the `import` statement, which can take one of the following forms must be used:

```
import <packageName>.*; // Imports all classes in the package
import <packageName>.<className>; // Imports the particular class
```

  - Once imported the, the class importing the package, can use the class.
  - The parent directory where the classes are placed must be in the `CLASSPATH` environment variable - similar to PATH variable.

Example:

```
import utilities.shapes.Circle;
public class CircleTest {
    public static void main(String aargs[]) {
          Circle my_circle = new Circle();
    }
}
```

  - The parent directory of `utilities` directory, must be in the `CLASSPATH` environment variable.

The Default Package

- All the classes in the current directory belong to an unnamed package called the `default` package - no package statement is needed.
- As long as the current directory (.) is part of the CLASSPATH variable, all the classes in the default package are automatically available to a program.
- If the CLASSPATH variable is set, the current directory must be included as one of the alternatives; otherwise, Java may not even be able to find the .class files for the program itself.
- If the CLASSPATH variable is not set, then all the class files for a program must be put in the current directory.

This was a very brief introduction to packages; if you want to use packages you will have to read up more. Here is one good link.

Information Hiding

Information Hiding

- The OO design paradigm allows information related to classes/objects (i.e. attributes and methods) to be grouped together - **Encapsulation**.
- Actions on objects can be performed through methods of the class **interface** to the class.
- The OO design paradigm also supports **Information Hiding**; some attributes and methods can be hidden from the user.
- Information Hiding is also referred to a **Visibility Control**.

Keyword

Information Hiding: Ability to "hide" the details of a class from the outside world.

Keyword

Access Control: Preventing an outside class from **manipulating** the properties of another class in **undesired** ways.

Information Hiding

Java provides control over the **visibility (access)** of variables and methods through **visibility modifiers**:

- This allows to safely seal data within the capsule of the class
- Prevents programmers from relying on details of class implementation
- Helps in protecting against accidental or wrong usage
- Keeps code elegant and clean (easier to maintain)
- Enables to provide access to the object through a clean interface

Visibility Modifiers

Keyword

public: Keyword when applied to a class, method or attribute makes it available/visible everywhere (within the class and outside the class).

Keyword

private: Keyword when applied to a method or attribute of a class, makes them only visible within that class. Private methods and attributes are not visible within *subclasses*, and are not inherited.

Keyword

protected: Keyword when applied to a method or attribute of a class, makes them only visible within that class, *subclasses* and also within all classes that are in the same package as that class. They are also visible to *subclasses* in other packages.

Note: We will learn about *subclasses* when we learn Inheritance.

Visibility Modifiers

Modifier	Class	Package	Subclass	Outside
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`default`	Y	Y	N	N
`private`	Y	N	N	N

The `Circle` Class with Visibility Modifiers

- Attributes of the class must be made `private` and accessed through getter/setter methods, which are `public`.
- Methods that other classes do not call must be defined as `private`.

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

    //Methods to get and set the instance variables
    public double getX() { return centreX;}
    public double getY() { return centreY;}
    public double getR() { return radius;}
    public double setX(double centreX) { this.centreX = centreX;}
    public double setY(double centreY) { this.centreY = centreY;}
    public double setR(double radius) { this.radius = radius;}
    // Other methods
}
```

Mutability

Keyword

Mutable: A class that contains public mutator methods or other public methods that can change the instance variables is called a *mutable class*, and its objects are called *mutable objects*.

Keyword

Immutable: A class that contains no methods (other than constructors) that change any of the instance variables is called an *immutable class*, and its objects are called *immutable objects*.

Back to the Circle Class

```
// Circle.java
public class Circle {
    private double centreX, centreY, radius;
    private static int numCircles;

    public Circle(double newCentreX, double newCentreY, double newRadius) {
        public double getCentreX() {.. }
        public void setCentreX(double centreX) {.. }
        public double getCentreY() {.. }
        public void setCentreY(double centreY) {.. }
        public double getRadius() {..}
        public void setRadiusd(doube radius) {..}
        public double computeCircumference() {..}
        public double computeArea() {..}
        public void resize(double factor) {..}
        public static int getNumCircles() {..}
    }
```

Is this an immutable class?

How would you create an immutable Circle class?

Creating an Immutable Class

```
// ImmutableCircle.java
public class ImmutableCircle {
    private final double centreX, centreY, radius;
    private static int numCircles;

    public ImmutableCircle(double newCentreX, double newCentreY,
                           double newRadius) {..}
    public double getCentreX() {.. }
    public double getCentreY() {.. }
    public double getRadius() {..}
    public double computeCircumference () {..}
    public double computeArea () {..}
    public static int getNumCircles() {..}
}
```

Delegation through Association

Delegation

- A class can **delegate** its responsibilities to other classes.
- An object can **invoke methods** in other objects through **containership**.
- This is an **Association** relationship between the classes (will be explained in more detail later).

Delegation - Example

We will demonstrate the Association relationship and Delegation through a `Point` class contained within the `Circle` class.

```
public class Point {
    private double xCoord;
    private double yCoord;

    // Constructor
    ....

    public double getXCoord() {
        return xCoord;
    }

    public double getYCoord() {
        return yCoord;
    }
}
```

Delegation - Example

```
public class Circle {
    private Point centre;
    private double radius;

    public Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }
    public double getX() {
        return centre.getXCoord();
    }
    public double getY() {
        return centre.getYCoord();
    }
    // Other methods go here
}
```

A `Point` object is contained in the `Circle` object; methods in a `Circle` object can call methods in the `Point` object using the reference to the object, `centre`.

Wrapper Classes

Back to Primitive Data Types

Primitives like `int` and `double`:

- Contain **only** data
- Do **not** have attributes or methods
- Can't "perform actions" like parsing

Keyword

Primitive: A unit of information that contains only data, and has no attributes or methods

Wrapper Classes

- Java provides "wrapper" classes for primitives
- Allows primitive data types to be "packaged" or "boxed" into objects
- Allows primitives to "pretend" that they are classes (this is important later)
- **Provides extra functionality for primitives**

Keyword

Wrapper: A class that gives extra functionality to primitives like `int`, and lets them behave like objects

Wrapper Classes

Primitive	Wrapper Class
`boolean`	Boolean
`byte`	Byte
`char`	Character
`int`	Integer
`float`	Float
`double`	Double
`long`	Long
`short`	Short

Integer Class

Provides a number of methods such as:

- Reverse: `Integer.reverse(10)`
- Rotate Left: `Integer.rotateLeft(10, 2)`
- Signum: `Integer.signum(-10)`
- **Parsing**: `Integer.parseInt("10")`

```
Integer x = Integer.parseInt("20");
int y = x;
Integer z = 2*x;
```

Parsing

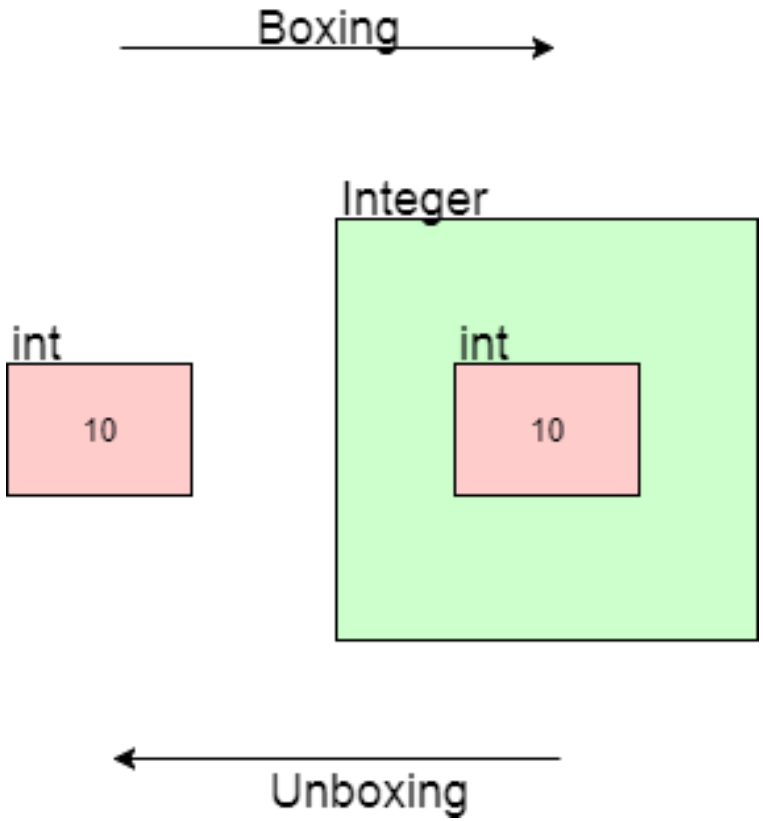
Every wrapper class has a parse function:

- xxx var = XXX.parseXXX(<string>);
- `int i = Integer.parseInt("1");`
- `double d = Double.parseDouble("1");`
- `boolean b = Boolean.parseBoolean("TruE");`

Keyword

Parsing: Processing one data type into another

Automatic Boxing/Unboxing



Keyword

(Un)Boxing: The process of converting a primitive to/from its equivalent wrapper class

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Create classes, and give them *properties* and *behaviours*
- Implement and use simple classes
- Identify a series of well-defined classes from a *specification*
- Explain object oriented concepts: abstraction, encapsulation, information hiding and delegation
- Understand the role of Wrapper classes

References

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