SWEN20003
Object Oriented Software Development

Classes and Objects - 3

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, 2

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

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 as **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.

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 void setX(double centreX) { this.centreX = centreX;}
    public void setY(double centreY) { this.centreY = centreY;}
    public void setR(double radius) { this.radius = radius;}
    // Other methods
}
```

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

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).

Back to the Circle Class

```java
// 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() {..}
}
```

Can we improve the design of this class?

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)

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**

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

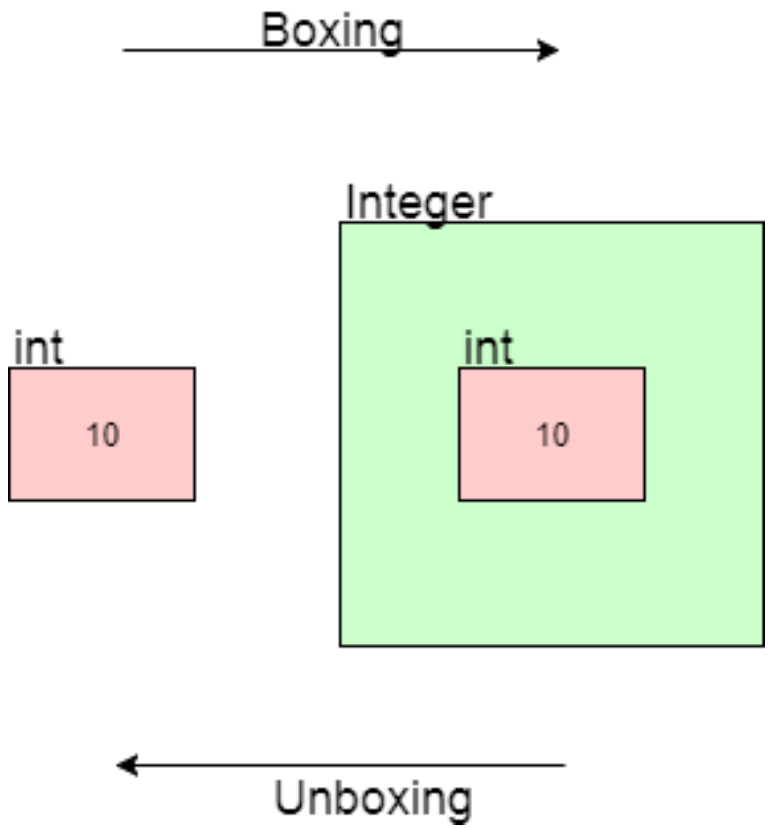
Every wrapper class has a parse function:

- 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:

Topics covered in this lecture:

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

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

References

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