5COSC001W - OBJECT ORIENTED PROGRAMMING

Lecture 3: Abstract Classes - Interfaces - Access Specifiers - Polymorphism

Dr Dimitris C. Dracopoulos

email: `d.dracopoulos@westminster.ac.uk`

1 Interfaces

An interface is a "abstract" class which does not have method implementations, and just defines a set of method signatures which have to be implemented by a subclass. Therefore, no objects can be created out of an interface.

An interface is used:

- to define the set of public methods (operations) that an implemented class provides.

A class which "implements" the methods outlined in an interface has to use the `implements` keyword.

Interfaces have the following characteristics:

1. They do not provide an implementation for any of their methods (from Java 8 onwards *default* methods which is an exception to this rule can also be included).
2. All of their data and methods are public.
3. All of the fields are static and final.

Java 8 has extended interfaces so that they can also include "default" methods and also static and private methods (static and private methods have a block body implementation of the corresponding methods). The default methods provide an implementation (body) which is inherited by classes which implement the interface and by any interfaces which extend the interface. Static methods in Java 8 interfaces also provide an implementation but they are not inherited by classes implementing the interface. The `default` keyword should be used in front of the return type of a method to make the method default.

In this way, Java 8 provides the means to implement multiple inheritance as a class which implements an interface (or more than one) and extends a class could inherit the same method from 2 or more different sources (the parent class and/or the parent interfaces). This is a multiple inheritance of implementation rather than multiple inheritance of state (fields) which could be done in C++ (inheritance of the same field name from more than one parent classes). Because in Java all fields in an interface are constant (final) and static, the developer does not have to worry about problems that result from multiple inheritance of state.

When the same method is inherited more than once in Java 8, the compile attempts to infer which version to use by the following rules:

1. A class (or abstract class) wins over an interface. If the same method is inherited from a parent class (or an abstract class) and an interface (the default method), the method of the class will be used.
2. A subtype wins. If a method is overridden in an interface then the parent interface implementation is "forgotten".
3. If two or more independently defined default methods conflict, or a default method conflicts with an abstract method (i.e. a non-default method), then the Java compiler produces a compiler error. In this case, you have to:
    - override the method in the class which implements the interfaces. In the overriden implementation you could choose to invoke one of the methods in the parent interfaces by using the `super` keyword. I.e. if the name of one of the parent interfaces is `Animal` you could invoke `Animal's` implementation of the `foo` method in the overriden `foo` by using the following:

      ```
      interface Animal {
          default void foo() {
              // ... implementation
          }
      }

      interface Mammal {
          default void foo() {
              // ... implementation
          }
      }

      class Dog implements Animal, Mammal {
          public void foo() {
              // invoke the Animal's version of foo()
              Animal.super.foo();
          }
      }
      ```

Example:

According to the following interface `Game`, every concrete game (concrete class derived from `Game`) should have methods `computeScore()` and `printInstructions()` implemented.

```
import java.util.*;

interface Game {
    int computeScore();
    void printInstructions();  // how to play the game
}

class BlackJack implements Game {
    public int computeScore() {
        System.out.println("BlackJack computeScore() called");

        // calculate and return a radom score
        Random randomGenerator = new Random();
        int score = randomGenerator.nextInt(22);

        return score;
    }

    public void printInstructions() {
        System.out.println("BlackJack printInstructions() called");
    }

    public int getNumberOfCards() {
        return 52;
    }
}

class ComputerGame implements Game {
    String platform;

    public ComputerGame(String platform) {
        this.platform = platform;
    }


    public int computeScore() {
        System.out.println("ComputerGame computeScore() called");

        return 100;  // buggy game, always returns 100
    }

    public void printInstructions() {
        System.out.println("ComputerGame printInstructions() called");
    }

    public String getPlatform() {
        return platform;
    }
}
```

```
public class GameTest {
    public static void main(String[] args) {
        // Game g = new Game();  // Error! Cannot create an object out of an interface

        ComputerGame g1 = new ComputerGame("Linux");
        System.out.println("Score: " + g1.computeScore());
        g1.printInstructions();

        BlackJack g2 = new BlackJack();
        System.out.println("Score: " + g2.computeScore());
        g2.printInstructions();
    }
}
```

The above program displays:

```
ComputerGame computeScore() called
Score: 100
ComputerGame printInstructions() called
BlackJack computeScore() called
Score: 11
BlackJack printInstructions() called
```

Note that a class can implement more than one interfaces. For example, it is possible to have:

```
interface Vehicle { /* details omitted */ }
interface Sellable { /* details omitted */ }
interface Repairable { /* details omitted */ }

class Car implements Vehicle, Repairable, Sellable {
    // ... implementation omitted
}
```

2 Abstract Classes

An abstract class is a class which cannot be instantiated (i.e. objects of it cannot be created), and has one or more methods without implementation (abstract methods).

As with interfaces, choosing a class to be abstract or not is a design issue.

Example:

```
abstract class Instrument {
    String manufacturer;
    public abstract void play();

    public void setManufacturer(String m) {
        manufacturer = m;
    }


    public String getManufacturer() {
        return manufacturer;
    }
}

class Violin extends Instrument {
    public void play() {
        System.out.println("Violin's melody");
    }
}

class Guitar extends Instrument {
    public void play() {
        System.out.println("Guitar music");
    }
}

public class InstrumentTest {
    public static void main(String[] args) {
        // Instrument i1 = new Instrument();   // Error! Cannot instantiate Instrument

        Violin i2 = new Violin();
        Guitar i3 = new Guitar();

        i2.play();
        i3.play();
    }
}
```

When the above program is run, it displays:

```
Violin's melody
Guitar music
```

3 Polymorphism

A reference variable of type `X` (which can be a concrete base class, an interface or an abstract class) can hold an object of any subclass of `X`.

- Because of *late binding*, the specific class of the object held by the reference variable will be determined at run time, and calls to the appropriate methods will be produced.

  Thus assuming that `a` is a reference variable of a base class, an abstract class or an interface the call:

  ```
  a.foo()
  ```

  will invoke a different `foo()` method if the object that `a` points to is an object of the base class or an object of a subclass.

This behaviour is called *polymorphism* (capability of assuming different forms). Calling a method with the same name and signature on a reference variable, corresponds (at run time) to the invocation of the appropriate method according to the actual class of the object that the reference variable points to.

Example:

The following code uses the classes and interfaces of previous examples:

```
public class PolymorphismTester {
    public static void main(String[] args) {
        Person p = new Student("wmin", "tom");
        p.info();  // calls info() of class Student

        p = new PostGraduateStudent("stanford", "peter", "cs");
        p.info();  // calls info() of class PostGraduateStudent

        System.out.println();

        Instrument instr = new Guitar();
        instr.play();  // calls play of class Guitar
        instr = new Violin();
        instr.play();  // calls play of class Violin

        System.out.println();

        Game game = new BlackJack();
        // game.getNumberOfBalls();  // Error! getNumberOfBalls not in Game
        game.printInstructions();  // call printInstructions() of BlackJack
        game = new ComputerGame("unix");
        game.printInstructions();  call printInstructions() of ComputerGame
    }
}
```

The output of the above program is:

```
name: tom
school: wmin

name: peter
school: stanford
firstDegree: cs

Guitar music
Violin's melody

BlackJack printInstructions() called
ComputerGame printInstructions() called
```

4 Access Specifiers

The details (methods, fields) of the implementation of a class can be hidden from other classes.

The access specifiers `public`, `protected`, `private` can be used to determine the access of classes to the individual methods and fields of another class.

The members of a class (i.e. data and methods) can be:

- *Public*: Everybody can access them.
- *Private*: Only the class itself can access them.
- *Protected*: Only the class itself, its subclasses, and the classes in the same package can access them.

The capability of classes to hide information from other classes is one of the fundamental characteristics of object oriented programming, called *encapsulation*.

If a class member (data or method) does not have an access specifier, then by default it has *package access*, i.e. only the class itself and other classes in the same package can access it.

Besides members of classes, the `public` access specifier can precede the definition of a class. Similarly with class members, a `public` class can be accessed by any other class, whether they are in the same package or not. If no `public` precedes the definition of the class, then the class has package access, and it can only be accessed by classes in the same package. The `protected` and `private` specifiers cannot be used in front of a class definition.

Example:

The following classes are defined in two separate files `Book.java` and `AccessSpecifiersExample.java`:
File `Book.java` contains:

```
package p1;

public class Book {
    public int numberOfPages;
    protected boolean paperback;
    private String colour;
    String subject;
}
```

File `AccessSpecifiersExample.java` contains:

```
package p1;

class TextBook extends Book {
    void modifyData() {
        numberOfPages = 100;  // OK
        paperback = true;  // OK
        colour = "blue";  // Error! colour is private in superclass
        subject = "computer science";  // OK
    }
}

/* class AccessSpecifierExample is in the same package p1
   with class Book */
public class AccessSpecifiersExample {
    public static void main(String[] args) {
        Book b = new Book();
        b.numberOfPages = 200;  // OK
        b.paperback = false;  // OK
        b.colour = "yellow";  // Error! colour is private in Book
        b.subject = "engineering";  // OK
    }
}
```

In the code above, the three classes `Book`, `TextBook`, `AccessSpecifierExample` are defined in the same package `p1`.

Class `TextBook` is a subclass of `Book`. Therefore, inside method `modifyData`, it can access the fields `numberOfPages`, `paperback`, `subject` of the parent class, but not `colour` which is `private` in `Book`.

Class `AccessSpecifierExample` is in the same package with `Book`. Therefore inside method `main`, it can access the fields `numberOfPages`, `paperback`, `subject` of `Book`, but not `colour` which is `private` in `Book`.

Note what it means that a subclass has access to the `protected` members of the parent class (assuming that the subclass and the parent class are in different packages - if they are in the same package the example above applies). The real meaning of this is, that the inherited protected members in an object of the subclass, can be accessed by the methods of that object. It does not mean, that the methods of an object of the subclass can access the protected members of an object of the base class:

```
package p2;
```

```
import p1.Book;  // import class A from package p1

public class D extends Book {
    public static void main(String[] args) {
        D d1 = new D();
        d1.paperback = true;  // OK

        Book b2 = new Book();
        b2.paperback = true;  // No! Error! paperback has protected access in p1.Book
    }
}
```