5COSC019W - OBJECT ORIENTED PROGRAMMING
Lecture 3: Abstract Classes - Interfaces - Access Specifiers - Polymorphism

Dr Dimitris C. Dracopoulos

Interfaces

An interface is a class which does not have method implementations, but just defines a set of method signatures which have to be implemented by a subclass.
An interface is used:

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

Example

```java
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");

        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;
    }
}
```

```java
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;
    }
}
```

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

Default Methods in Interfaces (Java 8 onwards)

Java 8 introduced **default** and static methods in interfaces.

- A `default` method provides an implementation (body).
- This provides a mechanism for multiple inheritance of implementation (not state) in Java.

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

Default Methods - Rules to resolve ambiguity

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.

Example

```java
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();

        // ... then do something more
    }
}
```

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).

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 Instru

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

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

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.

Example:

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

        p = new PostGraduateStudent("stanford", "peter", "cs");
        p.info();

        System.out.println();

        Instrument instr = new Guitar();
        instr.play();
        instr = new Violin();
        instr.play();

        System.out.println();

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

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
```

Access Specifiers

The details (methods, fields) of the implementation of a class can be hidden from other classes.
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.
- In Java: No access specifier means the *default* access is package. Classes in the same package can access these members.

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

Example:

```
package p1;
public class Book {
    public int numberOfPages;
    protected boolean paperback;
    private String colour;
    String subject;
}
```

```
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
    }
}
```

```java
/* 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
    }
}
```

A bit of caution on “Protected” access!

```
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 */
    }
}
```