Programare avansata pe obiecte - laborator 5 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

Object oriented programming - lab 5

Interfaces:

- ➢ Similar to a class but can only contain method signatures and fields.
- ➢ An interface is not intended to contain implementations of the methods, only the signature (name, parameters and exceptions) of the method.
- ➢ However, it is possible to provide default implementations of a method in an interface, to make the implementation of the interface easier for classes implementing the interface.
- ➢ An interface is declared using the Java ***interface*** keyword.
- ➢ An interface can be declared public or package scope (no access modifier).
- ➢ Accessing a variable from an interface is similar to accessing a static variable in a class.

```
package com.paolabs.lab5.ex1;

public interface InterfaceExemplul1 {

   public String hello = "Hello!";

   public void sayHello();
}
```

```
package com.paolabs.lab5;

import com.paolabs.lab5.ex1.InterfaceExemplul1;

public class Main {

   public static void main(String[] args) {
       System.out.println(InterfaceExemplul1.hello);
   }
```

```
}
```

- Before you can use an interface, you must implement that interface in some Java class. A class that implements an interface must implement all the methods declared in the interface. The methods must have the exact same signature (name + parameters) as declared in the interface. The class does not need to implement (declare) the variables of an interface.

```
package com.paolabs.lab5.ex1;

public class Exemplul1 implements InterfaceExemplul1 {

   @Override
   public void sayHello() {
       System.out.println(InterfaceExemplul1.hello);
   }
}
```

```
package com.paolabs.lab5;

import com.paolabs.lab5.ex1.Exemplul1;

public class Main {

   public static void main(String[] args) {
       Exemplul1 exemplul1 = new Exemplul1();
       exemplul1.sayHello();
   }
}
```

- Once a Java class implements an Java interface you can use an instance of that class as an instance of that interface.

- You cannot create instances of a Java interface by itself. You must always create an instance of some class that implements the interface, and reference that instance as an instance of the interface.

```
package com.paolabs.lab5;
```

```java
import com.paolabs.lab5.ex1.Exemplul1;
import com.paolabs.lab5.ex1.InterfaceExemplul1;

public class Main {

    public static void main(String[] args) {
        InterfaceExemplul1 interfaceExemplul1 = new Exemplul1();
        interfaceExemplul1.sayHello();
    }
}
```

- ➢ A Java class can implement ***multiple*** Java interfaces. In that case the class must implement all the methods declared in all the interfaces implemented.
- ➢ If a Java class implements multiple Java interfaces, there is a risk that some of these interfaces may contain methods with the same signature (name + parameters). Since a Java class can only implement a method with a given signature once, this could potentially lead to some implementation problems.

```java
package com.paolabs.lab5.ex1;

public interface InterfaceExemplul2 {

    public void sayBye();
}
```

```java
package com.paolabs.lab5.ex1;

public class Exemplul2 implements InterfaceExemplul1, InterfaceExemplul2 {

    @Override
    public void sayHello() {
        System.out.println("Hello!");
    }

    @Override
    public void sayBye() {
        System.out.println("Bye!");
    }
}
```

- A Java interface can contain both variables and constants. All variables in an interface are public, even if you leave out the public keyword in the variable declaration. All methods in an interface are public, even if you leave out the public keyword in the method declaration.
- Before Java 8 Java interfaces could not contain an implementation of the methods, but only contain the method signatures. Java 8 has added the concept of interface default methods to Java interfaces. An interface default method can contain a default implementation of that method. Classes that implement the interface but which contain no implementation for the default interface will then automatically get the default method implementation. We mark a method in an interface as a default method using the default keyword.

```
package com.paolabs.lab5.ex1;

public interface InterfaceExemplul3 {

   default void sayHello() {
       System.out.println("Hello there!");
   }
}
```

```
package com.paolabs.lab5.ex1;

public class Exemplul3 implements InterfaceExemplul3 {
}
```

```
package com.paolabs.lab5;

import com.paolabs.lab5.ex1.Exemplul3;

public class Main {

   public static void main(String[] args) {
       Exemplul3 exemplul3 = new Exemplul3();
       exemplul3.sayHello();
   }
}
```

- An interface can have ***static methods***. Static methods in a Java interface must have implementation.
- Static methods do not depend on the need to create object of a class. You can refer them by the class name itself.
- Static methods in interfaces can be useful when you have some utility methods you would like to make available, which fit naturally into an interface related to the same responsibility.

```
package com.paolabs.lab5.ex1;

public interface InterfaceExemplul4 {

   public static void print(String message) {
       System.out.println(message);
   }
}
```

```
package com.paolabs.lab5;

import com.paolabs.lab5.ex1.InterfaceExemplul4;

public class Main {

   public static void main(String[] args) {
       InterfaceExemplul4.print("Hello!");
   }
}
```

- It is possible for an interface to inherit from another interface, just like classes can inherit from other classes. By using the ***extends*** keyword.
- Unlike classes, interfaces can actually inherit from multiple superinterfaces. You specify that by listing the names of all interfaces to inherit from, separated by comma. A class implementing an interface which inherits from multiple interfaces must implement all methods from the interface and its superinterfaces.

```
package com.paolabs.lab5.ex1;
```

```
public interface InterfaceExemplul5 extends InterfaceExemplul1 {

}
```

```
package com.paolabs.lab5.ex1;

public class Exemplul5 implements InterfaceExemplul5 {

   @Override
   public void sayHello() {

   }
}
```

- ➢ If two interfaces contain the same method signature (name + parameters) and one of the interfaces declare this method as a default method, a class cannot automatically implement both interfaces.

Generic Interfaces:

- ➢ A generic Java interface is an interface which can be *typed* - it can be specialized to work with a specific type when used.

```
package com.paolabs.lab5.ex2;

public class Car {

   private String name;
   private String year;

   public Car() {
   }

   public String getName() {
       return name;
   }

   public void setName(String name) {
```

```
        this.name = name;
    }

    public String getYear() {
        return year;
    }

    public void setYear(String year) {
        this.year = year;
    }
}
```

```
package com.paolabs.lab5.ex2;

public interface Producer {

    public Object produce();
}
```

```
package com.paolabs.lab5.ex2;

public class CarProducer implements Producer {

    @Override
    public Object produce() {
        return new Car();
    }
}
```

```
package com.paolabs.lab5.ex2;

public class Main {

    public static void main(String[] args) {
        Producer carProducer = new CarProducer();

        Car car = (Car) carProducer.produce();
    }
```

```
}
```

The object returned from the carProducer.produce() method call has to be cast to a Car instance, because the produce() method return type is Object.

Using Java Generics you can type the MyProducer interface so you can specify what type of object it produces when you use it:

Generics add a way to specify concrete types to general purpose classes and methods that operated on Object before.

```
package com.paolabs.lab5.ex2;

public interface Producer<T> {

   public T produce();
}
```

```
package com.paolabs.lab5.ex2;

public class CarProducer<T> implements Producer<T> {

   @Override
   public T produce() {
       return (T) new Car();
   }
}
```

```
package com.paolabs.lab5.ex2;

public class Main {

   public static void main(String[] args) {

       Producer<Car> carProducer = new CarProducer<Car>();
       Car car = carProducer.produce();

       Producer<String> myStringProducer = new CarProducer<String>();
       String produce = myStringProducer.produce();
```

```
    }
}
```

You can also lock down the generic type of the Producer interface already when you implement it, in the specific producer class:

```
package com.paolabs.lab5.ex2;

public class Bike {

    private String description;

    public Bike() {
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

```
package com.paolabs.lab5.ex2;

public class BikeProducer implements Producer<Bike> {

    @Override
    public Bike produce() {
        return new Bike();
    }
}
```

```
package com.paolabs.lab5.ex2;

public class Main {
```

```
    public static void main(String[] args) {
        Producer<Bike> bikeProducer = new BikeProducer();
        Bike bike = bikeProducer.produce();
    }
}
```

Java Functional Interfaces:

- ➢ The term Java functional interface was introduced in Java 8.
- ➢ A functional interface in Java is an interface that contains only a single abstract (unimplemented) method.
- ➢ A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method.

```
package com.paolabs.lab5.ex3;

public interface FunctionalInterfaceExemplul1 {

    public void execute();
}
```

```
package com.paolabs.lab5.ex3;

public interface FunctionalInterfaceExemplul2 {

    public void execute();

    public default void print(String text) {
        System.out.println(text);
    }

    public static void print(String text, String name) {
        System.out.println(name + ": " + text);
    }
}
```

- Java functional interface can be implemented by a Java Lambda Expression.
- A Java lambda expression implements a single method from a Java interface. In order to know what method the lambda expression implements, the interface can only contain a single unimplemented method => the interface must be a Java functional interface.

```
package com.paolabs.lab5.ex3;

public class Main {

   public static void main(String[] args) {

       FunctionalInterfaceExemplul1 functionalInterfaceExemplul1 = () -> {
           System.out.println("Executing...");
       };

       functionalInterfaceExemplul1.execute();
   }
}
```

Comparable and Comparator:

- Java provides two interfaces to sort objects using data members of the class: Comparable and Comparator

Using Comparable Interface:

- A comparable object is capable of comparing itself with another object. The class itself must implement the java.lang.Comparable interface to compare its instances. This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.
- ***public int compareTo(Object obj)***: It is used to compare the current object with the specified object. It returns:
    - positive integer, if the current object is greater than the specified object.
    - negative integer, if the current object is less than the specified object.
    - zero, if the current object is equal to the specified object.

```
package com.paolabs.lab5.ex4;
```

```java
public class Movie implements Comparable<Movie>{

   private String name;
   private String type;
   private int year;
   private double rating;

   public Movie(String name, String type, int year, double rating) {
       this.name = name;
       this.type = type;
       this.year = year;
       this.rating = rating;
   }

   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
   }

   public String getType() {
       return type;
   }

   public void setType(String type) {
       this.type = type;
   }

   public int getYear() {
       return year;
   }

   public void setYear(int year) {
       this.year = year;
   }

   public double getRating() {
       return rating;
   }
```

```
    public void setRating(double rating) {
        this.rating = rating;
    }

    @Override
    public int compareTo(Movie movie) {
        /*
         * positive integer, if the current object is greater than the
specified object.
         * negative integer, if the current object is less than the
specified object.
         * zero, if the current object is equal to the specified object.
         */
        return this.year - movie.year;
    }
}
```

```
package com.paolabs.lab5.ex4;

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {
        Movie forceAwakens = new Movie("Force Awakens", "SF", 2015, 10);
        Movie starWars = new Movie("Star Wars", "SF",1977, 10);
        Movie empireStrikesBack = new Movie("Empire Strikes Back", "SF",
1980, 10);

        Movie[] movies = {forceAwakens, starWars, empireStrikesBack};
        Arrays.sort(movies);

        System.out.println("Movies after sorting : ");
        for (Movie movie: movies) {
            System.out.println(movie.getName() + "-"+ movie.getYear());
        }

    }
}
```

```
Movies after sorting :
Star Wars-1977
Empire Strikes Back-1980
Force Awakens-2015
```

Suppose we want to sort movies by their rating and names also. When we make a collection element comparable(by having it implement Comparable), we get only one chance to implement the compareTo() method.

Using Comparator Interface:

- ➢ Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.
- ➢ To compare movies by Rating, we need to do 3 things :
    - Create a class that implements Comparator (and the compare() method that does the work previously done by compareTo()). The **compare()** method in Java compares two class specific objects (x, y) given as parameters. It returns the value:
        - 0: if (x==y)
        - -1: if (x < y)
        - 1: if (x > y)
    - Make an instance of the Comparator class.
    - Call the overloaded sort() method, giving it both the array and the instance of the class that implements Comparator.

```
package com.paolabs.lab5.ex4;

import java.util.Comparator;

public class NameCompare implements Comparator<Movie> {

   @Override
   public int compare(Movie movie1, Movie movie2) {
       return movie1.getName().compareTo(movie2.getName());
   }
}
```

```
package com.paolabs.lab5.ex4;

import java.util.Arrays;

public class Main {

   public static void main(String[] args) {
       Movie forceAwakens = new Movie("Force Awakens", "SF", 2015, 10);
       Movie starWars = new Movie("Star Wars", "SF",1977, 10);
       Movie empireStrikesBack = new Movie("Empire Strikes Back", "SF",
1980, 10);

       Movie[] movies = {forceAwakens, starWars, empireStrikesBack};

       System.out.println("Sorted by name: ");
       NameCompare nameCompare = new NameCompare();
       Arrays.sort(movies, nameCompare);
       for (Movie movie: movies) {
           System.out.println(movie.getName() + "-"+ movie.getYear());
       }
   }
}
```