1DV532 – Starting Out with Java

Lesson 13

Interfaces, Abstract Classes, and Polymorphism

Dr. Nadeem Abbas
nadeem.abbas@lnu.se

Interface

- An **Interface**, in Java, is a *reference type*, similar to a class, that can contain only constants, abstract methods, default methods, and static methods.
  - An **abstract method** is a method that is declared without an implementation (without braces, and followed by a semicolon) as shown below:

```
int get(int pos);        // abstract method
int size();              // abstract method
```

  - A **default method** is a method that is defined with a keyword `default` at the beginning of a method signature, as shown below:

```
public default void show() {
 System.out.println("Just an example of a default method");
}
```

  - A **static method** is one that is defined with a keyword *static* and is associated with the type (interface/class) in which it is defined rather than objects of the type.

```
public static void hello()
 {
  System.out.println("Just an example of a static method");
 }
```

- **Note:** Most of the interfaces that we use in practice have only *abstrac*t methods, use of the *default* and *static* method is very rare.

Defining an Interface

- As shown in the example below, the syntax for defining an interface is similar to that of defining a class
  - Except the keyword **`interface`** is used in place of **`class`**
- **Interface Definition – Example:**

```
public interface IntList {
    void add(int n); // append integer n to list
    int get(int pos); // returns integer at position pos
    boolean contains(int n); // true if list contains n
    int size(); // returns number of stored integers

    // a default method
    public default void show() {
            System.out.println("Just an example of a default method");
    }

    // a static method
    public static void hello(){
            System.out.println("Just an example of a static method");
    }
}
```

Using an Interface – Interface Implementation

- We cannot instantiate an Interface, i.e., we can not create objects of an Interface as we can do for Classes.

- To use an interface, we need to write one or more Classes that ***implement*** the interface, as shown in the example on next slide.

- A class that *implements* an interface is required to provide complete definition for all the abstract methods defined in the interface.

Interface Implementation – Example

```
public class IntListImpl implements IntList {
    int length = 4; //initial list length or size
    int top = 0;
    private int[] values; // an integer array to store the list values

    public IntListImpl() {    // constructor
            values = new int[length];
    }

    // implementation of the size method
    public int size() {
            return top;
    }
    // implementation of the add method
    public void add(int n) {
            if (top == length)
                    resize();
            values[top++] = n;
    }
    // implementation of the other abstract methods, see the Example
    // Program, IntListImpl.java for complete code
}
```

Interfaces and Classes

- An interface definition separates "*what a class does*" from "*how it does*"
- An Interface definition specifies a a *contract* that all the Classes which implement the interface are required to fulfill.
  - The contact is specified in the form of *abstract methods*, which all the classes that implements the interface are required to define having same name return type and input parameters.

Interfaces and Inheritance

- The Java programming language supports **multiple inheritance of type**, which is the ability of a class to implement more than one interfaces.
  - Java allows a class to implement more than one interfaces.
- Based on the concept of **multiple inheritance of type**, an object can have multiple types, i.e., the type of its own class and the types of all the interfaces that the class implements.
  - This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface.

Abstract Classes

- An abstract class is a class declared using ***abstract*** keyword

  [access modifier] ***abstract*** **class** ClassName {

  //Member fields and methods

  }

- The abstract class generally contains **abstract methods**, but may have non-abstract methods as well
  - *If a class contains one or more abstract methods, it must be declared abstract*

- Abstract class cannot be instantiated.
  - Abstract classes serve as *templates* that define abstract general behavior using abstract methods and require *subclasses* to implement the abstract methods, i.e., provide view specific details/implementation,

Employee.java – Abstract Class Example

```
public abstract class Employee extends Person  {
        private int payPerHour;
        public Employee(String name, int
        paymentPerHour) {
                super.setName(name);
                this.payPerHour = paymentPerHour;
        }
        //abstract method
        public abstract int getSalary();

        //non-abstract methods
        public int getPaymentPerHour() {
                return payPerHour;
        }
        public void setPaymentPerHour(int
        paymentPerHour) {
                this.payPerHour = paymentPerHour;
        }

}
```

```
public class ContractEmployee extends Employee {
        private int workingHours;
        public ContractEmployee(String name, int
        paymentPerHour, int workingHours) {
                super(name, paymentPerHour);
                this.workingHours = workingHours;
        }
        @Override
        public int getSalary() {
        return getPaymentPerHour() * workingHours;
        } }
```

```
public class RegularEmployee extends Employee {
        private int workinghours=8;
        public RegularEmployee(String name, int
        paymentPerHour) {
                super(name, paymentPerHour);
        }
        @Override
        public int getSalary() {
          return getPaymentPerHour() * workinghours;
        }
}
```

Java Interfaces Vs. Abstract Classes

Similarities

- Both support abstraction and polymorphism
- Both may contain a mix of abstract and non-abstract methods
- Both cannot be instantiated

Differences

Interfaces

- With Interfaces, all fields are automatically public, static, and final, and all methods declared as default methods are public
- A class can *implement* any number of interfaces
- An interface can *extend* any number of interfaces

Abstract Classes

- With Abstract classes, we can declare fields that are not static and final, and define public, protected, and private methods
- A class can *extend* only one abstract (or non-abstract) class,
- An abstract or non-abstract class may *implement* any number of interfaces.

Polymorphism

- **Basic Idea:** *"One interface – multiple behavior"*
- The word "polymorphism" is derived from two *Greek* words: **poly + morphos**
  - **poly** means *many* and **morphos** means *forms*

    => ***polymorphism*** means ***many forms***.
- Polymorphism represents a concept in type theory according to which a single identifier for a type may represent or refer to objects of many different types that are related by some common type such as superclass or interface.
- It helps to reduce complexity by allowing an Interface or Superclass to be used to specify a general class of action.

Polymorphism and Method Binding

- In OOP, polymorphism refers to language's ability to process a method call differently depending on type (class) of the object on which a methods is invoked.
  - This is achieved through a special method binding mechanism called ***Late or Dynamic Binding***
- The process of associating a method definition with a method invocation or call is called ***method binding***

Polymorphism and Late Binding

- Java distinguish between two types of method bindings:
  1. **Dynamic or Late Binding**
  2. **Static or Early Binding**
- If a method definition is associated with its invocation when the code is compiled, this is called ***early binding***
  - This type of binding is often referred as ***static or compile time binding***
  - All **static and overloaded methods** have static bindings
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding*
  - This type of binding is often referred as ***dynamic or runtime binding***
  - **All non-static methods** and **overridden methods** have dynamic bindings.

Polymorphism - Example

```
public interface Animal {
   public void sound();
}
public class Cat implements Animal {
  public void sound() {
    System.out.println("Meow");
  }
}
public class Cow implements Animal {
   public void sound() {
     System.out.println("Mooo");
   }
}
public class Horse implements Animal {
   public void sound() {
     System.out.println(”Neighh");
   }
}
```

```
public class PolymorphismDemo {
      public static void main(String[] args) {
            Animal[] animals = new Animal[3];
            animals[0] = new Cat();
            animals[1] = new Cow();
            animals[2] = new Horse();

            for (Animal a : animals)
                  a.sound();
      }
}
```

- **Which sound method is called?**
- Run the program to find an answer!
  - You will see various sound methods will be invoked depending type of the object on which the method gets invoked.

Suggested Readings

- Absolute Java, Global Edition, 6/E by Walter J. Savitch, Chap 8 Polymorphism and Abstract Classes, Chap 13 Interfaces and Inner Classes (Interfaces part only)
- Introduction to Java Programming, Brief Version, Global Edition, 11/E Liang, Chapter 11 Inheritance and Polymorphism (Polymorphism part only), Chap 13 Abstract Classes and Interfaces
- Java Tutorials
  - https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
  - https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
  - https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Lnu.se