Module: Core Java
Session 10: Abstract classes and Interfaces

Interfaces

- Introduction to Interfaces
- Interfaces as APIs
- Interfaces and Multiple Inheritance
- Implementing the Relatable Interface
- Using Interface as a Type
- Rewriting Interfaces

Objective:

At the end of the chapter, you will be able to:

- Know Interfaces in Java and APIs
- Handle Interfaces and Multiple Inheritance
- Implement various Interface
- Use Interface as a Type
- Rewrite Interfaces

Interfaces in Java

Interface is a reference type in the Java programming language. It is similar to a class where only constants, method signatures and nested types can be contained. No method bodies are available. You cannot instantiate interfaces as they can only be implemented by classes or extended by other interface/s.

Note that defining an interface is same as creating a new class:

```
public interface DriveCar {

  // constant declarations, if any

  // method signatures
  int turn(Direction direction,double radius,  // An enum with values
                                      RIGHT, LEFT
        double startSpeed, double endSpeed);
  int changeLanes(Direction direction, double startSpeed, double endSpeed);
  int signalTurn(Direction direction, boolean signalOn);
  int getRadarFront(double distanceToCar, double speedOfCar);
  int getRadarRear(double distanceToCar, double speedOfCar);
       ......
  // more methods
}
```

Have you noticed something here? The thing to notice here is that method signatures have no braces and that they are terminated with a semicolon.

You need to write a class that implements the interface, if you want to use an interface. A method body for each of the methods declared in the interface is provided when an instantiable class implements an interface. Look at the following example:

```
public class DriveHondaAccord implements DriveCar {
  // the DriveCar method signatures, with implementation -- for example:
  int signalTurn(Direction direction, boolean signalOn) {
    //code to turn Honda's LEFT turn indicator lights on
    //code to turn Honda's LEFT turn indicator lights off
    //code to turn Honda's RIGHT turn indicator lights on
    //code to turn Honda's RIGHT turn indicator lights off
  }

  // other members, as needed -- for example, helper classes
  // not visible to clients of the interface
}
```

In the example of the robotic car, the interface will be implemented by the automobile manufactures. These implementations will be very different from each other. For instance, Toyota's implementation will off course differ substantially from that of the Chevrolet. However, both manufacturers will need to adhere to the same interface. For example, systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car will be built by the guidance manufacturers, who are the clients of the interface. How will the guidance systems do that? They will invoke the interface methods like turn, change lanes, brakes, accelerate etc.

Interfaces as APIs

The example of the robotic car shows as interface being used as an industry standard Application Programming Interface, better known as API. In commercial software products also, APIs are commonly used. Typically, a company can sell a software package containing complex methods. This can be wanted by another company to be used in its own software product. Say for instance, package of digital image processing methods that are sold to companies that make end-user graphics programs. A class is written by the image processing company to implement an interface, which is made public to its customers. After this, image-processing methods are invoked by the graphics company, using the signatures and return types defined in the interface. Note that, only the API of the image processing company is made public to its customer, not the implementation of this API. This is a tightly guarded secret. The company might also consider revising the implementation some time later on, as long as the original interface that the customers have relied on is continued to be implemented by the company.

Interfaces and Multiple Inheritance

In Java programming language, interfaces have another very important role. Although interfaces work in combination with classes, they are not part of the class hierarchy.

A class can inherit from only one class in Java. However, note that more than one interface can be implemented by it. Objects can, therefore, have multiple types –i.e. the type of their own class and the types of all the interfaces implemented by them. If a variable is declared to be the type of an interface, its value can reference any object instantiated from any class implementing the interface.

Defining an Interface

Modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body construct an interface declaration. Take a look at the following example:

```
public interface MultiInterface extends Interface1,
                                Interface2, Interface3 {

   // constant declarations
   double E = 2.718282;  // base of natural logs

   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);

}
```

That the interface can be used by any class in any package, is indicated by the public access specifier. The interface will be accessible only to classes defined in the same package as the interface, if it is not specified that the interface is public.

Interfaces can be extended by another interface, just like a class can be extended or sub-classed by another class. The difference here is that a class can extend only one other class, but any numbers of interfaces can be extended by one interface. A comma-separated list of all the interfaces that it extends is included in an interface declaration.

The Interface Body

Method declarations for all the methods included in the interface are contained in the interface method. Within an interface, a method declaration is followed by a semicolon. However, braces are not used here since implementations for the methods declared within an interface, are not provided by the interface.

Constant declarations and method declarations can be contained within an interface. Remember that every constant values defined in an interface are implicitly public, static and final. Also note that, these modifiers can be omitted.

Implementing an Interface

You need to include an implements clause in the class declaration in order to declare a class that implements an interface. More than one interface can be implemented by your class. The implement clause follows the extend clause, if there is one, by convention.

A Sample Interface, Relatable

Consider the following example of comparing the size of objects as defined by interface.

```
public interface Relatable {
          // this (object calling isLargerThan) and
  // other must be instances of the same class
  // returns 1, 0, -1 if this is greater
  // than, equal to, or less than other
  public int isLargerThan(Relatable other);
 }
```

Relatable should be implemented by the class that instantiates the similar objects if you want to compare the size of those objects.

If there is some way to compare the relative “size” of objects instantiated from the class, then any class can implement Relatable. For instance, it could be the number of characters for strings; the number of pages for book; weight for the students and so on. Area for planar geometric objects, and volume for three-dimensional geometric objects can be good choices. The isLargerThan() method can be implented to all the classes like this.

You can compare the size of objects instantiated from the class if you know that a class implements Relatable.

Implementing the Relatable Interface

In the following instance the Rectangle class is written to implement Relatable:

```
public class RectanglePlus implements Relatable {
   public int rect_width = 0;
   public int rect_height = 0;
   public Point rect_origin;

   // four constructors
   public RectanglePlus() {
         rect_origin = new Point(0, 0);
   }
   public RectanglePlus(Point p) {
         rect_origin = p;
   }
   public RectanglePlus(int w, int h) {
         rect_origin = new Point(0, 0);
```

```
                rect_width = w;
                rect_height = h;
        }
        public RectanglePlus(Point p, int w, int h) {
                rect_origin = p;
                rect_width = w;
                rect_height = h;
        }

        // a method for moving the rectangle
        public void move(int x, int y) {
                rect_origin.x = x;
                rect_origin.y = y;
        }

        // a method for computing the area of the rectangle
        public int getArea() {
                return rect_width * rect_height;
        }

        // a method to implement Relatable
        public int isLargerThan(Relatable other) {
                RectanglePlus otherRect = (RectanglePlus)other;
                if (this.getArea() < otherRect.getArea())
                        return -1;
                else if (this.getArea() > otherRect.getArea())
                        return 1;
                else
                        return 0;
        }
}
```

The size of any two RectanglePlus objects can be compared because the RectanglePlus implements Relatable.

Using Interface as a type

What does actually happen when a new interface is defined? In that case a new reference datatype is being defined by you. The interface names can be used wherever any other data type name can be used. In case a reference variable, the type of which is an interface, is defined. Any object assigned to it must be an instance of a class that implements the interface.

In the following example, look at the following method for finding the largest object in a pair of objects, for any objects that are instantiated from a class that implements Relatable:

```
public Object findLargest(Object obj1, Object obj2) {
  Relatable ob1 = (Relatable)obj1;
  Relatable ob2 = (Relatable)obj2;
  if ( (ob1).isLargerThan(ob2)) > 0)
    return ob1;
  else
    return ob2;
    }
```

This can invoke the isLargerThan method by casting the object1 to a Relatable type.

If Relatable is implemented in a wide variety of classes, you can compare the objects instantiated from any of those classes with the findLargest() method. However, note that this is possible only when both objects are of the same class. All of them can also be compared with the following methods:

```
    public Object findSmallest(Object obj1, Object obj2) {
      Relatable ob1 = (Relatable)obj1;
      Relatable ob2 = (Relatable)obj2;
      if ( (ob1).isLargerThan(ob2)) < 0)
        return ob1;
      else
        return ob2;
    }

    public boolean isEqual(Object obj1, Object obj2) {
      Relatable ob1 = (Relatable)obj1;
      Relatable ob2 = (Relatable)obj2;
      if ( (ob1).isLargerThan(ob2)) == 0)
        return true;
      else
        return false;
    }
```

Irrespective of the class inheritance of the “relatable” objects, these methods work for any of them. When Relatable is implemented by these methods, they can be their own class or superclass type, as well as Relatable type. Some advantages of multiple inheritance are thus gained as they can contain behavior both from a superclass and an interface.

Rewriting Interfaces

Suppose an interface called Dott has been developed by you:

```
    public interface DoIt {
      void doSomething(int i, double x);
      int doSomethingElse(String s);
    }
```

Now think of a situation where a third method to DoIt needs to be added. So, now the interface becomes:

```
    public interface DoIt {

      void doSomething(int i, double x);
      int doSomethingElse(String s);
      boolean didItWork(int i, double x, String s);

    }
```

If this change is made then all classes that implement the old DoIt interface will be broken down since the interface is not implemented by them anymore.

Attempting to anticipate all uses of your interface and specifying it from the beginning is a good practice. However, as this is not always possible, the way out is to create more interfaces later. For instance, a DoItPlus interface that extends DoIt can be created:

```
public interface DoItPlus extends DoIt {

   boolean didItWork(int i, double x, String s);

}
```

Your code can now be chosen by the user to continue to use the old interface or to upgrade the new interface.