Abstract Classes

As we move up the inheritace hierarchy, classes become more general and probably more abstract. At some point, the ancestor class becomes so general that you think of it more as a basis for other classes than a class which you want to instantiate.

Consider for example car, in real life we can have different implementation of car like **TeslaCar**, **AudiCar** etc. It is good to create a more general **Car** class and to put all the general methods and instance variables in that class because the basic prototype for every car is same. All that change is how a particular manufacturer implement those basic functionalities.

A point to look on is that the existence of **Car** object dosen’t make any sense, all that matter is the existence of those that implemented it. In our case **TeslaCar** and **AudiCar**.

This is where **Abstract Classes** comes into play. When you make a class **abstact** then you are ensuring that no one can create an object of that class. Along with **Abstract Classes** we have **Abstract Methods**. By marking a method **abstract** we are ensuring that we are just providing method declaration and no definition.

Whenever we add an **abstract** method in a class then we must also declare that class **abstract**. But the vice versa is not true if a class is **abstract** then it may or may not have **abstract** method. Many programmers think that **Abstract Classes** are house to **Abstarct Methods**, no they are so much more !!. Along with abstract methods they can have instance variables as well as concrete methods.

Abstract Methods act as a placeholder for methods that are implemented in subclasses. When we extend an **Abstact Class** we have two choices either to define all the **Abstact Methods** and make a **Concrete SubClass** or to leave some (or all) of the **Abstract Methods** undeclared and mark the subclass **abstract** as well.

Note: By Concrete Class I mean the class which can be instantiated, ie. The class whose objects can be created.

A very interesting thing to note about **Abstract Classes** is that, although we can not instantiate them but we still can create object references of **Abstract Class**. These object references must refer to an object of a **Concrete SubClass**. For example,

```
Car c = new TeslaCar("Model S", 4, 250.0);
```

I hope by now, the differences between object and object references must have been clear.

Now let us define our **Abstract Class** Car about which we were discussing so long.

```
/**
 * This program demonstrate Abstract Class.
 * @version 1.01- 17-08-2018
 * @author Abhey Rana
 */
```

```
import java.util.Calendar;
import java.util.Date;

public abstract class Car{

        // Abstract Class can also have instance variables .....

        private String modelName;
        private int passengerCapacity;
        private double topSpeed;
        private Date manufacturingDate;

        // Abstract Class can have constructor just like other clas

        public Car(String modelName, int passengerCapacity, double
                this.modelName = modelName;
                this.passengerCapacity = passengerCapacity;
                this.topSpeed = topSpeed;
                this.manufacturingDate = Calendar.getInstance().get
        }

        // Abstract Methods

        public abstract void accelerate();
        public abstract void brake();
        public abstract String getDescription();

        // Accessor Methods (See Abstract Class can also have concr

        public String getModelName(){
                return this.modelName;
        }

        public int getPassengerCapacity(){
                return this.passengerCapacity;
        }

        public double getTopSpeed(){
                return this.topSpeed;
        }

        public Date getManufacturingDate(){
                return this.manufacturingDate;
        }

        // Mutator Methods

        public void setModelName(String modelName){
                this.modelName = modelName;
        }

        public void setPassengerCapcity(int passengerCapacity){
                this.passengerCapacity = passengerCapacity;
        }

        public void setTopSpeed(double topSpeed){
                this.topSpeed = topSpeed;
        }

}
```

Now let's define our concrete subclasses namely **TeslaCar** and **AudiCar**

```
/**
 * This program demonstrate Concrete SubClasses of Car Abstract Cla
 * @version 1.01- 17-08-2018
 * @author Abhey Rana
 */

class TeslaCar extends Car{

        public TeslaCar(String modelName, int passengerCapacity, do
                super(modelName, passengerCapacity, topSpeed);
        }

        // Overriding Abstract methods

        public void accelerate(){
                System.out.println("Tesla's specialized acceleratio
        }

        public void brake(){
                System.out.println("Tesla's specialized braking sys
        }

        public String getDescription(){
                return "Tesla Car, Model Name: " + this.getModelNam
        }

}

class AudiCar extends Car{

        public AudiCar(String modelName, int passengerCapacity, dou
                super(modelName, passengerCapacity, topSpeed);
        }

        // Overriding Abstract methods

        public void accelerate(){
                System.out.println("Audi's specialized acceleration
        }

        public void brake(){
                System.out.println("Audi's specialized braking syst
        }

        public String getDescription(){
                return "Audi Car, Model Name: " + this.getModelName
        }
```

```
 }
```

Finally let’s create our AbstractTest Class which will act as a driver class and will illustrate all the concepts disucssed till now.

```
 /**
  * This program demonstrate AbstractTest driver class
  * @version 1.01- 17-08-2018
  * @author Abhey Rana
  */

  public class AbstractTest{

         public static void main(String args[]){

                 Car[] showroom = new Car[3];

                 showroom[0] = new TeslaCar("Model S", 4, 250.0);
                 showroom[1] = new AudiCar("Audi A3", 4, 240.0);
                 showroom[2] = new TeslaCar("Model X", 4,210.0);

                 for(Car c: showroom)
                         System.out.println(c.getDescription());

         }

  }
```

Why to make methods of Car class abstract ? Can we not just leave their method definition empty ? Will it not serve the purpose ?

Yeah we can leave the method definiton of those abstract method empty and surely it will serve the purpose in current scenario. But by making a method **abstract** we are ensuring that every **Concrete SubClass** that extends **Car** class will override these methods which will further ensure that none of those classes are using the default implementation of any of the **abstract** methods of Car class.

Interface

Interfaces in Java are a way of describing **what** classes should do without specifiying **how** they should do it. In the Java programming language, an interface is not a class but a set of **requirements** that the implementing classes must conform to.

Typically, the supplier of some service states:

> If your class conforms to a particular interface,
> then I will perform the service

In layman terms you can think of an interface as a contract, which once you sign then you have to abide by it’s terms and conditions. And the terms and conditions in case of Java interface is to override all the methods of interface.

In Java Interface all the methods are by default **public** and **abstract**. Even if you mark a method as **public abstract** then also Java compiler won’t complain. But it you try to make method **private** or **protected** then Java compiler will raise an error.

Just like Abstract Classes, Interfaces can’t be instantiated but they can have still have Object References which will refer to objects of implementing classes.

A major difference between Java Interfaces and Abstract Classes is that, interfaces can’t have instance members and concrete methods(like constructor, accessor methods, mutator methods etc).

Now let’s create an interface of our own, Repairable which will act as contract for RepairingGuy class. If any class implements this Repairable interface then the ReairingGuy will repair objects of that class.

```
public interface Repairable{
        boolean isRepairable();
        void repair();
}
```

Now if we want to give the functionality of Repairing to TeslaCar then we must implement this interface. So now our TeslaCar will look something like this

```
import java.util.Calendar;
import java.util.Date;

class TeslaCar extends Car implements Repairable{

        /*
                Previous code in TeslaCar class will remain same.
                All we have to do is to override methods of Repairal
        */

        public boolean isRepairable(){
                // Tesla Car has a warranty time and can be repaired
                Date currentDate = Calendar.getInstance().getTime()
                if(currentDate.getTime() - this.getManufacturingDate
                        return false;
                return true;
        }

        public void repair(){
                System.out.println("Procedure for repairing Tesla Ca
        }

}
```

Why to create a new interface with these two abstract method ? Can't we just include those abstract methods in Car Abstract class ?

Yes you can but again that will not be a good design practice. Because TeslaCar or AudiCar are not the only things that can be repaired. Let say we have another completely unrelated class Music Player then how will you provide the functionality of repairing to it.

Let's see how Repairable interface provides functionality of repairing to a SonyMusicPlayer.

```
public class SonyMusicPlayer implements Repairable{

        // Some code for Music Player .....

        public boolean isRepairable(){
                // Let's assume that our music player comes with li
                return true;
        }

        public void repair(){
                System.out.println("Procedure for repairing Sony Mu
        }

}
```

Now let's see the implementation of RepairingGuy class the generic class responsible for all repair related operations.

```
public class RepairingGuy{

        // Static doRepair method of RepairingGuy ....

        public static void doRepair(Repairable object){
                if(!object.isRepairable())
                        System.out.println("Sorry I can't repair thi
                else{
                        object.repair();
                        System.out.println("Your item has been succe
                }
        }

}
```

Finally we have InterfaceTest driver class which will illustrate all the concepts of interface discussed till now.

```
public class InterfaceTest{

        public static void main(String args[]){

                Repairable[] item_to_repair = new Repairable[3];

                item_to_repiar[0] = new TeslaCar("Model S", 4, 250.
                item_to_repair[1] = new SonyMusicPlayer();
                item_to_repair[2] = new TeslaCar("Model X", 4,210.0

                for(Repairable item: item_to_repair)
                        RepairingGuy.doRepair(item);

        }

}
```

A more real world example.

Yayy!! It’s festival time in your house and you’re busy with all the arrangements.

```
}
```

Now there are a lot of people that are involved in the various chores in the house. These people include the electrician, plumber, painter, cook etc.

As such, every person is given the responsibility to do **his own work** with utmost dedication. Obviously, every person has a job and is not concerned with the job of any other person.

> Hint: Unrelated entities that we discussed in class.

Therefore, we can think of an interface Workable that has a structure as follows:

```
public interface Workable {
        void doWork();
}
```

Every person should implement this interface in his own way, because **they all do their own work which is not related to the other in any way.**

Coming to the various **Concrete Implementations** :

```
class Plumber implements Workable {

        /**
                This method is overriden.
        */
        public void doWork() {
                System.out.println("I am here to fix the toiletries
        }

        // Other methods.
}
```

```
class Cook implements Workable {

        /**
                This method is overriden.
        */
        public void doWork() {
                System.out.println("I am here to cook delicious foo
        }

        // Other methods.
}

class Electrician implements Workable {

        /**
                This method is overriden.
        */
        public void doWork() {
                System.out.println("I am here to fix all electrical
        }

        // Other methods.
}

class Painter implements Workable {

        /**
                This method is overriden.
        */
        public void doWork() {
                System.out.println("I am here to paint the house wi
        }

        // Other methods.
}
```

In this article we have tried to explain when to use Abstract Classes and when to use Interfaces. For first timers many things might not make any sense. It is only after when you read a lot of good code that these thing will start making sense. As greatly said -

> To write good code you must read good code.

Interfaces and Abstract Classes are highly confusing concepts, So in case of any doubt feel free to reach any of the Softablitz coordinator.

Raw file of all the codes in this article can be found here.

I hope by the above example you got the gist of interfaces. In case of any doubts, feel free to contact us. We are happy to help. :)

Contributors

The above explaination is one of the best explainations of Abstract class and Interfaces available. This was compiled by our immediate seniors.