Set 24: Abstract Classes and Interfaces

Skill 24.01: Explain the purpose of an abstract class
Skill 24.02: Implement an abstract class
Skill 24.03: Explain the purpose of an interface
Skill 24.04: Implement an interface

Skill 24.01: Explain the purpose of an abstract class

Skill 24.01 Concepts

Suppose that for some reason, we had a programming task that was to model animals. For example, we want to have a class for a cat, a class for a dog, a class for a horse, and so forth. As you can imagine, there could be lots of redundancy among those classes because animals have some common characteristics. For instance, all animals have, say, two eyes, or two ears, and make sounds, and so forth. Or at least all the animals that we would like to model do.

We could create a super class called animal to capture all these common characteristics and behaviors. And then we could have all the specific animal classes derived from the animal class. So far so good…

The question though is, what would new animal mean? In other words, if we create an object of the animal class, what kind of object would that be? What would we try to model by creating an object of the animal class?

You may be thinking there is no point in creating objects from the animal class, so there is no need to ever invoke the animal constructor. But subclasses could benefit from extending such a class.

Java prohibits, under some circumstances, creating objects from specific classes. Such classes are referred to as abstract classes. Abstract classes are classes that cannot be instantiated – that is, objects cannot be created from abstract classes. While the methods abstract classes carry may be implemented, methods declared as abstract cannot be implemented by the abstract class, they must instead be implemented by the subclass. Such methods can provide a framework for a subclass.

- Objects cannot be created from abstract classes

- Methods declared as abstract cannot be implemented by the abstract class, they must instead be implemented by the subclass

Skill 24.02: Implement an abstract class

Skill 24.02 Concepts

Abstract classes are implemented using the keyword *abstract*

In the below example, we have created an abstract class called *animal*

```
public abstract class Animal
{

//methods and fields

}
```

As previously mentioned, abstract classes can provide the framework for a subclass. For example, consider the following subclasses which inherit the animal class,

```
public class Cow extends Animal
{

//methods and fields

}

public class Dog extends Animal
{

//methods and fields

}

public class Cat extends Animal
{

//methods and fields

}
```

Each of the classes that inherit the parent class, Animal, have some common characteristics. For example, a Cow, a Dog, and a Cat, all have two eyes, two ears, and make sounds, and so forth. But these characteristics are different for each animal we want to model.

Suppose we want to ensure that every animal we model can speak. To do this, we would write an abstract method in our abstract class.

```
public abstract class Animal
{

public abstract void speak();

}
```

Notice we have only declared the abstract method, but we did not implement it. In fact, Java does not allow for this. However, the classes that inherit the Animal class *must* implement the speak method.

This is illustrated below,

Abstract class Animal

```java
public abstract class Animal
{

    public abstract String speak();
    public abstract String type();
}
```

Class that inherits Animal

```java
public class Cat extends Animal {
   private String name;

   public Cat(String n){
       name = n;
   }
   //overriden
   public String speak(){
       return "Meow!";
   }

   public String type(){
       return "Mammal";
   }

   public String toString(){
       String msg = "My name is " + name
                              + " " + speak();
        return msg;
   }
}
```

The above illustrates how abstract classes can serve as a framework for subclasses. In addition to abstract methods, abstract classes can also include regular methods. For example, all the animals in our subclasses have four legs, therefore we can create a method to return the number of legs in the parent class.

Abstract class with non-abstract method

```java
public abstract class Animal
{

    public abstract String speak();

    public abstract String type();

    public String getLegs(){
        return "Four";
    }
}
```

Class that inherits Animal

```java
public class Cat extends Animal {
   private String name;

   public Cat(String n){
       name = n;
   }
   //overriden
   public String speak(){
       return "Meow!";
   }

   public String type(){
       return "Mammal";
   }

   public String toString(){
       String msg = "My name is " + name
                                + " " + speak();
        return msg;
   }
}
```

Main.java

```java
class Main {
    public static void main(String[] args) {
      Animal[] myPets = new Animal[4];
      myPets[0] = new Cat("Garfield");

      System.out.println(myPets[0]);

    }
}
```

Output

```
My name is Garfield Meow!
```

https://www.youtube.com/watch?v=_KvtWtBnsqE

Skill 24.02: Exercises 1 & 2

Skill 24.03: Explain the purpose of an interface

Skill 24.03 Concepts

An interface is an abstract super class that contains only abstract methods. Consider the following example.

The super class,

```
public abstract class Parent
{
public abstract void method1();
public abstract void method2();
public abstract int method3(double d);
}
```

And now the subclass,

```
public class Child extends Parent
{
public void method1(){ … some code … }
public void method2(){ … some code … }
public int method3()(double c) { … some code … }
}
```

Notice that in the above example, the super class does nothing. In other words, all methods in the super class are abstract. Moreover, there are no variables which need to be accessed. The only purpose of the super class is to force the subclass to implement its methods. If this is all a particular super class does, then the “abstract” declaration can be replaced with an interface as shown below,

```
public interface Parent
{
void method1();
void method2();
int method3(double d);
}
```

Notice, that with the methods above, it would be legal to start their signatures with public abstract, however, even if we leave them off, they are automatically public and abstract… all because it is an interface. It is conventional in interfaces *not* to use public and abstract in the signatures.

Finally, because the parent class has been changed to an interface, the subclass must also be changed,

```
public class Child implements Parent
{
public void method1(){ … some code … }
public void method2(){ … some code … }
public int method3()(double c) { … some code … }
}
```

In the subclass, the word **extends** has been replaced with **implements**. Everything else is the same. Notice, that all the interface does here is to force us to implement those methods in the subclass… big deal! Actually, it is a very big deal, as the interface provides the structure that the subclasses must adhere to.

- All the methods in an interface are abstract
- An interface provides the structure for the subclass. That is, all methods in the interface, must be implemented in the subclass

Skill 24.04: Implement an interface

Skill 24.04 Concepts

An interface is implemented using the keyword **implements**

Consider the interface below,

RobotArm.java

```
public interface RobotArm
{
    void moveUp( double rate, double howFar );
    void moveDown( double rate, double howFar );
    void twistLeft( double deg );
    void twistRight( double deg );
}
```

The robot interface created above provides the “glue” that holds together several cooperating classes shown below, specifically two different industrial robots supplied by two different robot manufacturers. They are the Lexmar 234 and the General Robotics 56A.

Lexmare234.java

```
public class Lexmar234 implements RobotArm{

   public Lexmar234(){  /* constructor */ }

   public void moveUp(double rate, double howFar){ /* some code */ }

   public void moveDown(double rate, double howFar) { /* some code */ }

   public void twistLeft( double deg ) { /* some code */ }

   public void twistRight( double deg ) { /* some code */ }
}
```

GR56A.java

```
public class GR56A implements RobotArm {

   public GR56A() { /* constructor */ }

   public void moveUp(double rate, double howFar){ /* some more code */ }

   public void moveDown(double rate, double howFar) { /* some more code */ }

   public void twistLeft( double deg ) { /* some more code */ }

   public void twistRight( double deg ) { /* some more code */ }
}
```

So far there is no difference from the implementation of the abstract class we previously discussed. In other words, just like the abstract class, the interface forces us to write code for those methods in classes where we specifically implement *RobotArm*.

Now let's find out what is different about interfaces. In the main Robot class below, we will create objects from the *RobotArm* interface… recall this was not allowed with abstract classes.

Robot.java

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

    RobotArm lx = new Lexmar234();
    RobotArm gr = new GR56A();

    //Do something with the Lexmar robot
    lx.moveDown(3, 27.87);
    lx.twistRight(22.0);

    //Do something with the General Robotics machine
    gr.moveUp(22.2, -34.0);
    gr.twistLeft(18);
    }
}
```

It is significant that nowhere in the above class did we say implements in the code. Also notice, for example, that when we declare,

```
RobotArm lx = new Lexmar234();
```

That lx is a type RobotArm even though RobotArm is not a class, it is an interface. This is specified by the left side of the above statement, and it means that lx can only use methods given in the RobotArm interface. The object lx will use these methods as implemented in the Lemar234 class. Notice this is specified on the right side of the above statement.

- Objects can be created from interfaces but not from abstract classes
- Classes can implement many interfaces, but can only extend one parent class

For more information on interfaces, check out the video below,

https://www.youtube.com/watch?v=xOcEuIR4pNU&t=3s

Skill 24.04: Exercises 1 & 2