Object-Oriented Programming (OOPS-3)

What you will learn in this lecture?

- Important keywords and their use.
- Abstraction.
- Interfaces.

Final Keyword

- When a variable is declared with a final keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable.
- If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final list.
- Final keywords can be used to initialise constants.

Initializing a final variable:

```
final int {name_of_variable} = {value};
```

Example:

```
final int pi = 3.14;
```

Refer to the course videos to see the use case and more about the final keyword.

Abstract Classes

An abstract class can be considered as a blueprint for other classes. Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that has a declaration but does not have an implementation. This set of methods must be created within any child classes which inherit from the abstract class. *A class that contains one or more abstract methods is called an* ***abstract class***.

Creating Abstract Classes in Java

- By default, Java does not provide abstract classes.
- A method becomes abstract when decorated with the keyword **`abstract`**.
- An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.
- However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.
- While declaring abstract methods in the class, it is not mandatory to use the **`abstract`** decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Java code uses the **ABC** class and defines an abstract base class:

```
 abstract class ABC{
    int value;
    Abstract int do_something(){ //Our abstract method declaration
        // TO_DO
    }
}
```

We will do it in the following example, in which we define two classes inheriting from our abstract class:

```
class add extends ABC{
    int do_something(){
        return value + 42;
    }
}

class mul extends ABC{
    int do_something(){
        return value * 42;
    }
}

class Test{
    public static void main(String[] args) {
        add x = new add(10);
        mul y = new mul(10);

        System.out.println(x.do_something());
        System.out.println(y.do_something());
    }
}
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

- An abstract method can have an implementation in the abstract class.

- However, even if they are implemented, this implementation shall be overridden in the subclasses.
- If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with **`super()`** call mechanism. (*Similar to cases of "normal" inheritance*).
- Similarly, we can even have concrete methods in the abstract class that can be invoked using **`super()`** call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.
- Consider the given example:

```
abstract class ABC{

    abstract int do_something(){ //Abstract Method
        System.out.println("Abstract Class AbstractMethod");
    }

    int do_something2(){ //Concrete Method
        System.out.println("Abstract Class ConcreteMethod");
    }
}

class AnotherSubclass extends ABC{
    int do_something(){
        //Invoking the Abstract method from super class
        super().do_something();
    }

    //No concrete method implementation in subclass
}

class Test{
    public static void main(String[] args) {
        AnotherSubclass x = new AnotherSubclass()
        x.do_something() //calling abstract method
        x.do_something2() //Calling concrete method
    }
}
```

We will get the output as:

```
Abstract Class AbstractMethod
Abstract Class ConcreteMethod
```

Another Example

The given code shows another implementation of an abstract class.

```
// Java program showing how an abstract class works
abstract class Animal{ //Abstract Class
    abstract move();
}

class Human extends Animal{ //Subclass 1
    void move(){
        System.out.println("I can walk and run");
    }
}

class Snake extends Animal{ //Subclass 2
    void move(){
        System.out.println("I can crawl")
}

class Dog extends Animal{ //Subclass 3
    void move(){
        System.out.println("I can bark")
    }
}

// Driver code
```

```
class Test{
    public static void main(String[] args) {
        Animal R = new Human();
        R.move();
        Animal K = Snake();
        K.move();
        R = Dog();
        R.move();
    }
}
```

We will get the output as:

```
I can walk and run
I can crawl
I can bark
```

Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

An interface is different from a class in several ways:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interface

```
public interface Name_of_interface {
      // body
}
```

Example:

```
public interface VehicleInterface {

      public final static double PI = 3.14;

      public int getMaxSpeed();
      public void print();
}
```

Now we need to implement this interface using a different class. A class uses the implements keyword to implement an interface. The **implements** keyword appears in the class declaration following the extends portion of the declaration.

```
public class Vehicle implements CarInterface{

      @Override
      public void print() {
            // TODO Auto-generated method stub
            // We can implement this function further.
      }

      @Override
      public int getMaxSpeed() {
            // TODO Auto-generated method stub
            return 0;
      }

      @Override
      public String getCompany() {
            // TODO Auto-generated method stub
            return null;
      }
}
```

@**Override** annotation informs the compiler that the element is meant to **override** an element declared in an interface.

We can implement the given overridden functions and instantiate an object of Vehicle class.