Day8: Interface, Early and late binding, static and default method in java 8, Var-args, Enums

Interface In Java:

An **Interface in Java** is defined as an abstract type used to specify the behaviour of a class. An interface in Java is a blueprint of a class. A Java interface contains static constants and abstract methods.

The interface in Java is *a* mechanism to achieve 100% **abstraction**. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and **multiple inheritance in Java**. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

To declare an interface, use the **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement an interface to a class we use **implements** keyword.

Syntax:

```
interface InterfaceName{

    // declare constant fields
    // declare methods that is abstract by default.
}
```

We save an interface also with the .java extension.

Uses Of an Interface:

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- It is also used to achieve loose coupling.

Note: **The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.**

Example:

```
//Printer.java

public interface Printer{

  int number=10;
  void print();

}

Java compiler converts it as follows:


public interface Printer{
  public static final int number=10;
  public abstract void print();

}
```

```
//ConsolePrinter.java

public class ConsolePrinter implements Printer{

  public void print(){
    System.out.println("Printing on the console.");
  }
}

//FilePrinter.java

public class FilePrinter implements Printer{

  public void print(){
    System.out.println("Printing on the File.");
  }
}


//Main.java

public class Main{

public static void main(String args[]){

  ConsolePrinter cp = new ConsolePrinter();
  Printer p1 = new ConsolePrinter();
  Printer p2 = new FilePrinter();

  cp.print();
  p1.print();
  p2.print();

  }
}
```

I Problem:

lets create an interface by the name Shape with one method draw and implement this interface in Rectangle and Circle classes.

```
//Shape.java

public interface Shape{
  void draw();
}

//Rectangle.java
class Rectangle implements Shape{

  public void draw(){
    System.out.println("drawing rectangle");
  }
}

 //Circle.java
class Circle implements Shape{
  public void draw(){
    System.out.println("drawing circle");
  }
}

//Main.java
class Main{
  public static void main(String args[]){

  Shape shape1=new Rectangle();
  Shape shape2=new Circle();

  shape1.draw();
  shape2.draw();

  }
}
```

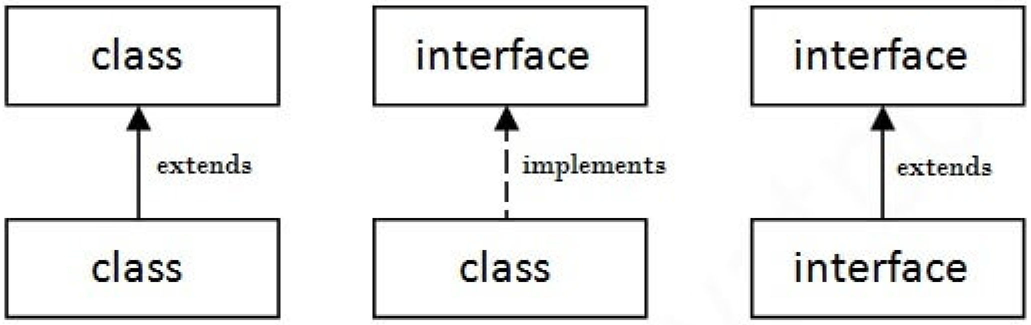
We Problem:

Let’s take a method which will accept the Printer object and now we can call this method by supplying different implementations of the Printer interface.

```
public class Main{

  public static void printSomething(Printer p){
    p.print();
  }

  public static void main(String[] args){

    printSomething(new ConsolePrinter());
    printSomething(new FilePrinter());
    //printSomething(null); // it will raise NullPointerException


  }
}
```

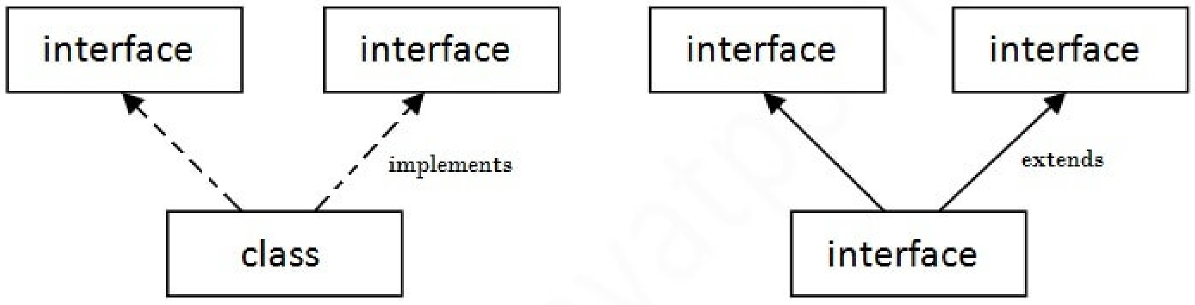
The relationship between classes and interfaces:

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**
.



Multiple inheritance in Java by interface:

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
//Printable.java
interface Printable{
  void print();
}

//Showable.java
interface Showable{
  void show();
}

//Demo.java

class Demo implements Printable, Showable{

  public void print(){
    System.out.println("Hello");
  }

  public void show(){
    System.out.println("Welcome");
  }

  public static void main(String args[]){
    Demo obj = new Demo();
    obj.print();
    obj.show();

    Printable p = new Demo();
    p.print();

    Showable s = new Demo();
    s.show();

 }
}
```

Interface inheritance:

A class implements an interface, but one interface extends another interface.

Example:

```
//Printable.java
interface Printable{
  void print();
}

//Showable.java
interface Showable extends Printable{
  void show();
}

 //Main.java
class Main implements Showable{

  public void print(){
    System.out.println("Hello");
  }
  public void show(){
    System.out.println("Welcome");
  }

  public static void main(String args[]){

  Main obj = new Main();

  obj.print();
  obj.show();
 }
}
```

You Problem:

- Create an Interface Animal with one abstract method :

```
public void makeNoise();
```

- Create 2 implementation classes for the above interface Dog and Cat and override the above method accordingly:

  For Dog: System.out.println(”Bark...”);

  For Cat: System.out.println(”Meow”);

- Create a Main class with the following method:

```
public static void getNoise(Animal animal){
    animal.makeNoise();
}
```

- From the main method of Main class call the above getNoise() method 2 times by passing the object of Dog class and the object of Cat class.

New Features Added in Interfaces in JDK 8:

1. **default method inside an interface:**

   Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.

   Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

   The default methods will be inherited inside the implementation classes.

   Example:

```
//Intr.java
interface Intr
{
    //abstract method
    void method1();
    //default method
    default void method2()
    {
        System.out.println("hello");
    }
}

//IntrImpl1.java
class IntrImpl1 implements Intr{

  public void method1(){
    System.out.println("inside method1 of IntrImpl1");
  }

}

//Main.java
class Main{
  public static void main(String[] args){

    IntrImpl1 i1 = new IntrImpl1();
    i1.method1();
    i1.method2();

  }

}
```

Note: The default method of an interface need not override inside the implementation class, but if we want we can override the default method also with some other implementation inside the implementation class.

2. **static method inside an interface:**

Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object.

Note: these methods are not inherited. We always call the static method with the help of Interface name.

```
//Intr.java
interface Intr
```

```
{
    //abstract method
    void method1();

    //default method
    default void method2()
    {
        System.out.println("inside method2");
    }
    //static method
    static void method3()
    {
        System.out.println("inside method3");
    }
}

//IntrImpl1.java
class IntrImpl1 implements Intr{

  public void method1(){
    System.out.println("inside method1 of IntrImpl1");
  }

}

//Main.java
class Main{
  public static void main(String[] args){

    IntrImpl1 i1 = new IntrImpl1();
    i1.method1();
    i1.method2();

    Intr.method3();

  }

}
```

Marker or tagged interface in java:

An interface that has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operations.

Example:

```
public interface Serializable{
}
```

Difference between abstract class and interface:

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	**Interface**
1. An Abstract class can **have abstract and non-abstract** methods.	An Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2. An Abstract class **doesn't support multiple inheritance.** .	An Interface **supports multiple inheritance.** .
3. An Abstract class **can have final, non-final, static and non-static variables** .	An Interface has **only static and final variables** .
4. An Abstract class **can provide the implementation of interface.** .	An Interface **can't provide the implementation of abstract class** .
5. An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
6. A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Simply, an abstract class achieves partial abstraction (0 to 100%) whereas an interface achieves fully abstraction (100%). Not very valid after Java 8.

Which should you use, abstract classes or interfaces

- Consider using abstract classes if any of these statements apply to your situation:
    - You want to share code among several closely related classes.
    - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
    - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
    - You expect that unrelated classes would implement your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.
    - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
    - You want to take advantage of multiple inheritance of type.

An example of an abstract class in the JDK is `AbstractMap`, which is part of the Collections Framework. Its subclasses (which include `HashMap`, `TreeMap`, and `ConcurrentHashMap`) share many methods (including `get`, `put`, `isEmpty`, `containsKey`, and `containsValue`) that `AbstractMap` defines.

An example of a class in the JDK that implements several interfaces is `HashMap`, which implements the interfaces `Serializable`, `Cloneable`, and `Map<K, V>`. By reading this list of interfaces, you can infer that an instance of `HashMap` (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map. In addition, the `Map<K, V>` interface has been enhanced with many default methods such as `merge` and `forEach` that older classes have implemented this interface do not have to define.

Note that many software libraries use both abstract classes and interfaces; the `HashMap` class implements several interfaces and also extends the abstract class `AbstractMap`.

Early and Late Binding in java:

Connecting a method call to the method body is known as binding.

There are two types of binding:

1. Early Binding (also known as Static Binding).
2. Late Binding (also known as Dynamic Binding).

When the type of an object is determined at compile time it is known as early binding, whereas, when a type of the object is determined at run-time it is known as late binding.

In java, each variable has a type, it may be primitive or non-primitive.

Example

int data=30;

Here data variable is a type of int.

Dog d1 = new Dog();

Here d1 is a type of Dog

Early binding example:

When type of the object is determined at compiled time(by the compiler), it is known as early/static binding.

If there is any private, final or static method in a class, there is static binding.

```
class Dog{
   private void eat(){
      System.out.println("dog is eating...");
    }

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Late Binding example:

When type of the object is determined at run-time, it is known as late/dynamic binding.

```
//Animal.java
class Animal{
   void eat(){
      System.out.println("animal is eating...");
    }
}

//Dog.java
class Dog extends Animal{

  void eat(){
    System.out.println("dog is eating...");
  }

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

In the above example, object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So the compiler doesn't know its type, only its base type.

var-args:(Variable Argument):

The var-args allows the method to accept zero or multiple arguments. Before var-args either we use an overloaded method or take an array as the method parameter but it was not considered good practice, because it leads to the maintenance problem. If we don't know how many arguments we will have to pass in the method, var-args is the better approach.

Advantage of var-args:

We don't have to define multiple overloaded methods.

Syntax:

The var-args uses ellipsis i.e. three dots after the data type.

[accessModifier] returntype methodName(datatype... variableName){

}

Example:

```
class Main{

 static void display(String... values){
  System.out.println("display method invoked ");
 }

 public static void main(String args[]){

 display();//zero argument
 display("my","name","is","varargs");//four arguments
 }
}
```

Accessing elements from the var-args:

var-args internally implemented based on the array concept, we can access the elements from a var-args using zero-based index.

Example:

```
class Main{

  static void display(String... values){

    System.out.println("display method invoked ");

     //accessing the elements from var-args
     for(String s:values){
     System.out.println(s);
   }
  }

  public static void main(String args[]){

  display();//zero argument
  display("my","name","is","varargs");//four arguments
  }
}
```

Rules for var-args:

• There can be only one variable argument in the method.

• Variable argument (var-args) must be the last argument if we have any other arguments are there.

Example1:

```
void method(String... a, int... b){}//Compile time error

void method(int... a, String b){}//Compile time error
```

Example2:

```
class Main{

 static void display(int num, String... values){

  System.out.println("number is "+num);

  for(String s:values){
     System.out.println(s);
  }

}

 public static void main(String args[]){

 display(200);//zero var-arg
 display(500,"hello");//one var-args
 display(1000,"my","name","is","varargs");//four var-args
 }
}
```

Enum in Java:

The **Enum in Java** is a data type that contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY), directions (NORTH, SOUTH, EAST, WEST), season (SPRING, SUMMER, WINTER, AUTUMN), colors (RED, YELLOW, BLUE, GREEN, WHITE, BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Declaration of enum in Java:
Enum declaration can be done outside a Class or inside a Class but not inside a Method.

Example: Enum declared at Outside

Here we save an Enum with .java extension, and after compilation, we get a .class file for an enum also.

```
//Color.java
enum Color {
  RED,
  GREEN,
  BLUE;
}

//Main.java
public class Main {

  public static void main(String[] args)
  {
    Color c1 = Color.RED;
    System.out.println(c1);
  }
}
```

Example: Enum declared inside a class

```
//Main.java
public class Main {
  enum Color {
    RED,
    GREEN,
    BLUE;
  }

  public static void main(String[] args)
  {
    Color c1 = Color.RED;
    System.out.println(c1);
  }
}
```

Note:

• Every enum is internally implemented by using Class.

• Every enum constant represents an **object** of type enum.

```
//Color.java
enum Color {
  RED,
  GREEN,
  BLUE;
}

the above enum Color will be internally converted as :

final class Color extends Enum
{
     public static final Color RED = new Color();
     public static final Color BLUE = new Color();
     public static final Color GREEN = new Color();
}
```

We can declare the **main() method** inside the enum. and we can run an enum file directly from   the Command Prompt.

```
//Color.java
enum Color {
  RED,
  GREEN,
  BLUE;

  public static void main(String[] args)
  {
    Color c1 = Color.RED;
    System.out.println(c1);
  }
}

>javac Color.java
>java Color
```

Enum and Inheritance:

- All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
- **toString() method** is overridden in **java.lang.Enum class**, which returns enum constant name.
- enum can implement many interfaces.

values() and ordinal() methods:

- These methods are present inside **java.lang.Enum**.
- **values() method** can be used to return all values present inside the enum.
- Order is important in enums.By using the **ordinal() method**, each enum constant index can be found, just like an array index.

Example:

```
public class Test {
  enum Color {
    RED,
    GREEN,
    BLUE;
  }

  public static void main(String[] args)
  {
    // Calling values()
    Color arr[] = Color.values();


    for (Color col : arr) {
      // Calling ordinal() to find index of color
      System.out.println(col + " at index " + col.ordinal());
    }

  }
}

Output
RED at index 0
GREEN at index 1
BLUE at index 2
```

We can define constructors, methods, and fields inside enum types, which makes them very powerful.

method inside an enum:

example:

```
//Item.java

public enum Item {

  SUGER,RICE,SALT;

  public void info(){
    System.out.println("This is grocerry item");
  }
}

//Main.java

public class Main{

  public static void main(String[] args) {

    Item[] itr= Item.values();

    for(Item item:itr){
      item.info();
    }
  }
}
```

If we want we can override the method inside any item also.

example:

```
//Item.java
public enum Item {

  SUGER,RICE{  // overwritten method

    public void info(){
      System.out.println("This is Rice");
    }
  },SALT;

  public void info(){
    System.out.println("This is grocery item");
  }

}
```

Enum can contain a constructor and it is executed separately for each enum constant at the time of enum class loading.

 We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

The constructor of enum type is private. If you don't declare private, compiler internally creates private constructor.

Example:

```
enum Season{
  WINTER(10),SUMMER(20);

  private int value;

  Season(int value){
    this.value=value;
  }
}
```

Example2:

```
//City.java
public enum City {

  DELHI{

    public void message(){  // overriden method
      System.out.println("welcome user..");
      System.out.println("you are in capital");
    }

  },
  MUMBAI,
  CHENNAI("50 towers"),
  KOLKATA;

  //variables in enum
  public String numberofTowers;

  //constructor
  City(){
    this.numberofTowers="100 towers";
  }

  City(String numberofTowers){   //overloaded constructor
```

```
    this.numberofTowers=numberofTowers;
  }

  public void message(){
    System.out.println("welcome user..");
  }
}



//Main.java

public class Main{

  public void printCity(City city){

    if(city != null){
      System.out.println("our service is available ");

      city.message();

      System.out.println(city.numberofTowers);


    }
    else
      System.out.println("invalid city");

  }

  public static void main(String[] args) {

    Main m1=new Main();

    m1.printCity(City.CHENNAI);
  }
}
```

Enum in switch case :

From Java 5 onwards we can take enum inside the switch case also.

Example:

```
class Main{

enum Day{
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
  }

public static void main(String args[]){

    Day day=Day.MONDAY;

    switch(day){
    case SUNDAY:
     System.out.println("sunday");
     break;
    case MONDAY:
     System.out.println("monday");
     break;
    default:
    System.out.println("other day");
    }
  }
}
```

References:

https://www.javatpoint.com/

https://www.geeksforgeeks.org/

https://www.programiz.com/

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html