# **Assignment 4**

Q1. Write a program to show Interface Example in java?

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

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.

# **Example**

1. **interface** Bank{
2. **float** rateOfInterest();
3. }
4. **class** SBI **implements** Bank{
5. **public** **float** rateOfInterest(){**return** 9.15f;}
6. }
7. **class** PNB **implements** Bank{
8. **public** **float** rateOfInterest(){**return** 9.7f;}
9. }
10. **class** TestInterface2{
11. **public** **static** **void** main(String[] args){
12. Bank b=**new** SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. } }

Q2. Write a program with 2 concrete methods and 2 abstract methods in java?

// Abstract class example

**abstract** **class** AbstractExample {

**abstract** **void** display(); // Abstract method

**void** show() // Concrete method

{

System.out.println("Concrete method of abstract class");

}

}

// Subclass of abstract class

**class** SubClass **extends** AbstractExample {

// Implementing the abstract method

**void** display()

{

System.out.println("Abstract method implemented");

}

}

**public** **class** AbstractClass{

**public** **static** **void** main(String args[])

{

// Creating an object of the subclass

SubClass obj = **new** SubClass();

// Calling the abstract method

obj.display();

// Calling the concrete method

obj.show();

} }

Q3. Write a program to show the use of functional interface in java?

## Java Functional Interfaces

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods.

### @FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message.

# **Example**

// implement a user defined functional interface.

@FunctionalInterface

interface Square {

int calculate(int x);

}

class Test {

public static void main(String args[])

{

int a = 5;

// lambda expression to define the calculate method

Square s = (int x) -> x \* x;

// parameter passed and return type must be

// same as defined in the prototype

int ans = s.calculate(a);

System.out.println(ans);

}

}

Q4. What is an interface in Java?

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not 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.

Since Java 9, we can have **private methods** in an interface.

Q5. What is the use of interface in Java?

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

### Syntax:

**interface** <interface\_name>{

// declare constant fields

// declare methods that abstract

// by default.

}

# **Example**

**interface** printable{

**void** print();

}

**class** A **implements** printable{

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

**public** **static** **void** main(String args[]){

A obj = **new** A();

obj.print();

}

}

Q6. What is the lambda expression of Java 8?

Lambda Expression is just an anonymous (nameless) function. That means the function which doesn’t have the name, return type and access modifiers.

Lambda Expression also known as anonymous functions or closures.

// A sample functional interface (An interface with single abstract method)

interface FuncInterface

{

// An abstract function

void abstractFun(int x);

// A non-abstract (or default) function

default void normalFun()

{

System.out.println("Hello");

}

}

class Test

{

public static void main(String args[])

{

FuncInterface fobj = (int x)->System.out.println(2\*x);

// This calls above lambda expression and prints 10.

fobj.abstractFun(5);

}

}

Q7. Can you pass lambda expressions to a method? When?

A **lambda expression** passed in a method that has an argument of type of **functional interface**. If we need to pass a lambda expression as an argument, the type of parameter receiving the lambda expression argument must be of a functional interface type.

In the example below, the lambda expression can be passed in a method which argument's type is "**TestInterface**".

## Example

interface TestInterface {

boolean test(int a);

}

class Test {

// lambda expression can be passed as first argument in the check() method

static boolean check(TestInterface ti, int b) {

return ti.test(b);

}

}

public class LambdaExpressionPassMethodTest {

public static void main(String arg[]) {

boolean result = Test.check((x) -> (x%2) == 0, 10);

System.out.println("The result is: "+ result);

} }

Q8. What is the functional interface in Java 8?

Functional Interfaces If an interface contains only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

## Some Built-in Java Functional Interfaces

1) Runnable -It contains only run() method

2) Comparable -It contains only compareTo() method

3) ActionListener -It contains only actionPerformed()

4) Callable -It contains only call() method

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

**Java SE 8 included four main kinds of functional interfaces** which can be applied in multiple situations as mentioned below:

**Consumer**

**Predicate**

**Function**

**Supplier**

## 1. Consumer

The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value. It returns nothing. There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.

## 2. Predicate

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate. Similarly, in the Java programming language, a predicate functional interface of Java is a type of function that accepts a single value or argument and does some sort of processing on it, and returns a boolean (True/ False) answer. The implementation of the Predicate functional interface also encapsulates the logic of filtering (a process that is used to filter stream components on the base of a provided predicate) in Java.

## 3. Function

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. There are many versions of Function interfaces because a primitive type can’t imply a general type argument, so we need these versions of function interfaces. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long. The different sequences of these primitive types are also used in the argument.

## 4. Supplier

The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output. This type of functional interface is generally used in the lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream. generate method, which is implemented by the Supplier functional Interface.

Q9. What is the benefit of lambda expressions in Java 8?

A **lambda expression** is an inline code that implements a **functional interface** without creating a concrete or anonymous class. A lambda expression is basically an **anonymous method**.

## Advantages of Lambda Expression

* **Fewer Lines of Code** − One of the most benefits of a lambda expression is to **reduce the amount of code**. We know that lambda expressions can be used only with a **functional interface**. For instance, **Runnable** is a functional interface, so we can easily apply lambda expressions.
* **Sequential and Parallel execution support by passing behavior as an argument in methods** − By using **Stream API** in Java 8, the functions are passed to collection methods. Now it is the responsibility of collection for processing the elements either in a sequential or parallel manner.
* **Higher Efficiency** − By using **Stream API** and **lambda expressions**, we can achieve higher efficiency (**parallel execution**) in case of bulk operations on collections. Also, lambda expression helps in achieving the internal iteration of collections rather than external iteration.

**Without Lambda Expression**

1) interface Interf {

2) public void methodOne() {}

3) public class Demo implements Interface {

4) public void methodOne() {

5) System.out.println(“method one execution”);

6) }

7) public class Test {

8) public static void main(String[] args) {

9) Interfi = new Demo();

10) i.methodOne();

11) }

12) }

**Above code With Lambda expression**

1) interface Interf {

2) public void methodOne() {}

3) class Test {

4) public static void main(String[] args) {

5) Interfi = ()  System.out.println(“MethodOne Execution”);

6) i.methodOne();

7) }

8) }

Q10. Is it mandatory for a lambda expression to have parameters?

No, it is mandatory for a lambda expression to have parameters

A lambda expression is similar to a method that has an **argument**, **body**, and **return type**. It can also be called an **anonymous function** (method without a name).

We need to follow some rules while using formal parameters in a lambda expression.

* If the abstract method of functional interface is a **zero-argument** method, then the **left-hand side** of the arrow (->) must use **empty parentheses**.
* If the abstract method of functional interface is a **one-argument** method, then the **parentheses** are not mandatory.
* If the abstract method of functional interface is a **multiple argument** method, then the **parentheses** are mandatory. The formal parameters are **comma-separated** and can be in the same order of the corresponding functional interface.
* The mentioning type of **formal parameters** is not mandatory. In case we have not mentioned the type of formal parameters, then it's type can be determined by the compiler from the corresponding **Target Type**.

**Type 1:** No Parameter.

**Syntax:**

() -> System.out.println("Hello");

**Type 2:** Single Parameter.

**Syntax:**

(p) -> System.out.println(p);

**Type 3:** Multi parameters

(p1, p2) -> System.out.println(p1 + " " + p2);

It is not mandatory to use parentheses if the type of that variable can be inferred from the context.