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

Answer: Here's an example program that demonstrates the use of interfaces in Java:

```
// Define an interface with a method
interface Vehicle {
    void start();
}

// Implement the interface in a class
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started.");
    }
}
// Implement the interface in another class
class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike started.");
    }
}

// Main class to run the program
public class InterfaceExample {
    public static void main(String[] args) {
        // Create objects of classes that implement the interface
        Car car = new Car ();
        Bike bike = new Bike ();

        // Call the start () method on each object
        car.start(); // Output: Car started.
        bike.start(); // Output: Bike started.
    }
}
```

```
}
```

In this example, we define an interface called `Vehicle` with a single method `start ()`. The `Car` class and the `Bike` class both implement the `Vehicle` interface and provide their own implementation of the `start()` method.

In the `main()` method, we create objects of the `Car` and `Bike` classes and call the `start()` method on each object. Since both classes implement the `Vehicle` interface, we can treat them polymorphically and invoke the `start()` method without worrying about the specific implementation details. The output of the program will be "Car started." and "Bike started".

Interfaces allow you to define a contract or a set of methods that classes must implement. This promotes code reusability, modularity, and loose coupling, as different classes can implement the same interface to provide their own behavior while adhering to a common contract.

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

Answer:

```
abstract class AbstractClass {
   // Abstract method 1
   public abstract void abstractMethod1();

   // Abstract method 2
   public abstract void abstractMethod2();

   // Concrete method 1
   public void concreteMethod1() {
      System.out.println("This is concrete method 1.");
   }

   // Concrete method 2
   public void concreteMethod2() {
      System.out.println("This is concrete method 2.");
   }
}

class ConcreteClass extends AbstractClass {
   // Implementing abstractMethod1
   @Override
```

```
    public void abstractMethod1() {
        System.out.println("Implementation of abstractMethod1.");
    }

    // Implementing abstractMethod2
    @Override
    public void abstractMethod2() {
        System.out.println("Implementation of abstractMethod2.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an instance of ConcreteClass
        ConcreteClass concreteObj = new ConcreteClass();

        // Calling concrete methods
        concreteObj.concreteMethod1();
        concreteObj.concreteMethod2();

        // Calling abstract methods
        concreteObj.abstractMethod1();
        concreteObj.abstractMethod2();
    }
}
```

In this example, we have an abstract class called AbstractClass with two abstract methods (abstractMethod1 and abstractMethod2) and two concrete methods (concreteMethod1 and concreteMethod2). The AbstractClass serves as a blueprint for other classes to inherit from.

The ConcreteClass extends AbstractClass and provides implementations for the abstract methods. The main method in the Main class demonstrates the usage of both concrete and abstract methods by creating an instance of ConcreteClass and calling the methods accordingly.

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

Answer:

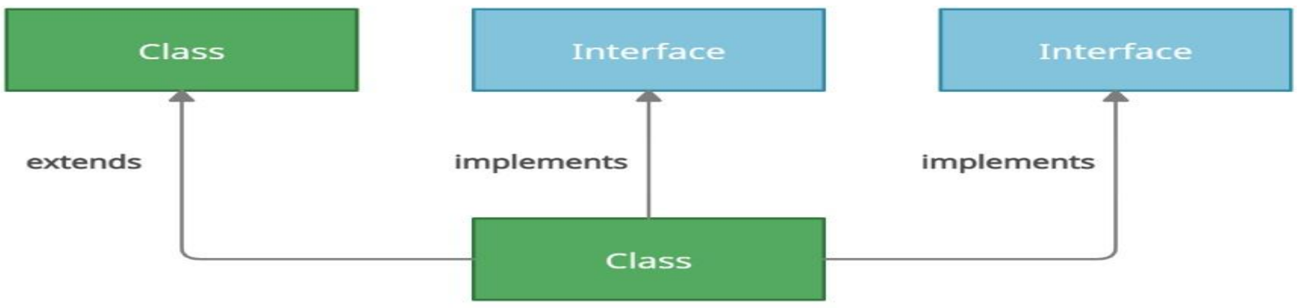
```
interface Calculation {
    int perform (int a, int b);
}
public class Main {
   public static void main (String[] args) {
      Calculation addition = (a, b) -> a + b;
      Calculation subtraction = (a, b) -> a - b;
      Calculation multiplication = (a, b) -> a * b;

      int result1 = addition.perform(5, 3);
      System.out.println("Addition result: " + result1);

      int result2 = subtraction.perform(8, 4);
      System.out.println("Subtraction result: " + result2);

      int result3 = multiplication.perform(6, 2);
      System.out.println("Multiplication result: " + result3);
   }
}
```

Q4. What is an interface in Java?

Answer: 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.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

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.



Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- 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.

Q5. What is the use of interface in Java?

Answer: Use of Interface in Java

- Abstraction is achieved using an **Interface in Java**.
- To achieve loose coupling.
- Supports the functionality of multiple inheritances in Java.

Things to keep in mind when using an Interface in Java

- The same signature must be used when overriding interface methods.
- Interface methods must be implemented in classes that are not abstract.
- Zero, one, or more interfaces can be implemented by a class at a time.
- An interface in Java can extend another interface just like a class extends another class.

Q6. What is the lambda expression of Java 8?

Answer: In Java, Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface). Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value. Lambda Expressions are recently included in Java SE 8.

Functionalities of Lambda Expression in Java
Lambda Expressions implement the only abstract function and therefore implement functional interfaces lambda expressions are added in Java 8 and provide the below functionalities.

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed.

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

Answer: Yes, we can pass lambda expressions as arguments to methods in Java. This feature is often used when working with functional interfaces or when you need to provide a concise implementation of a specific behavior to a method.

Lambda expressions can be passed as arguments in situations where the target method expects a functional interface as a parameter. Since functional interfaces have a single abstract method, lambda expressions can be used to provide an implementation for that method.

Q8. What is the functional interface in Java 8?

Answer: An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Example:

```
interface sayable{

    void say(String msg);

}

public class FunctionalInterfaceExample implements sayable{

    public void say(String msg){

        System.out.println(msg);

    }

    public static void main(String[] args) {

        FunctionalInterfaceExample fie = new FunctionalInterfaceExample();

        fie.say("Hello there");

    }

}
```

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

Answer: Java 8 introduced a number of new features to the programming language, one of the most significant being the lambda expression. Lambda expressions provide a concise way to express a method that can be passed around as an argument to another method, enabling functional programming paradigms in Java.

Advantages of lambda expressions

- **Conciseness**

Functional interfaces can be succinctly represented using lambda expressions. Functional interfaces, which are interfaces with a single abstract method, are the fundamental units of Java 8's lambda expressions. We can more succinctly explain the implementation of the functional interface using a lambda expression than we can with an unnamed inner class.

- **Readability**

Lambda expressions make the code more readable by eliminating the need for boilerplate code that would typically be required for anonymous inner classes. This makes the code more expressive and easier to understand, as the programmer can focus on the intent of the code rather than the implementation details.

- **Flexibility**

We are able to develop more adaptable code with lambda expressions. We may quickly give behaviour to a method as a parameter by utilising functional interfaces and lambda expressions. As a result, our code is more modular and can be applied to various situations.

- **Parallelism**

The idea of streams, a potent tool for working with collections, was introduced in Java 8. The simplicity with which streams can be parallelized is one of its benefits. We may make use of multi-core processors and boost the efficiency of our programmes by using lambda expressions to define the behaviour of a stream operation.

- **Improved API**

Java API designers are now able to give developers a more useful API thanks to lambda expressions. The API can offer methods that take functions as parameters by employing lambda expressions, which makes the API more flexible and user-friendly. As a result, the API has become more simplified and expressive and is simpler to understand and use.

Some Other Benefits of lambda expressions in Java

- Reduces code bloat.
- Eliminates shadow variables.
- Encourages functional programming.
- Code reuse
- Enhanced iterative syntax.
- Simplified variable scope.
- Less boilerplate codes.
- JAR file size reductions

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

Answer: No, it is not mandatory for a lambda expression to have parameters. In Java, lambda expressions are used to represent functional interfaces, which are interfaces with a single abstract method.

A lambda expression can have zero or more parameters, depending on the signature of the abstract method in the functional interface. If the abstract method does not require any parameters, we can use an empty parameter list in the lambda expression.

Here's an example of a lambda expression without parameters:

```
Runnable runnable = () -> {
    // Code to be executed
    System.out.println("Hello, lambda expression without parameters!");
};

// Execute the lambda expression
runnable.run();
```

In this example, we have a lambda expression assigned to a Runnable interface. The Runnable interface has a single abstract method called run() that takes no parameters. Therefore, we can use an empty parameter list () in the lambda expression. The code within the lambda expression is executed when run() is called

Note that even though the lambda expression does not have any parameters, we still need to include the empty parameter list () to maintain the syntax and indicate that it is a lambda expression.