Assignment 4 (Core Java)

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

Sol:-

```java
// Interface representing a shape
interface Shape {
    void draw(); // Abstract method to draw the shape

    double calculateArea(); // Abstract method to calculate the area
}

// Class representing a Circle
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public void draw() {
        System.out.println("Drawing a circle");
    }

    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Class representing a Rectangle
class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

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

    public double calculateArea() {
        return length * width;
    }
}

public class example {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        circle.draw();
        System.out.println("Area: " + circle.calculateArea());

        Shape rectangle = new Rectangle(4.0, 6.0);
        rectangle.draw();
        System.out.println("Area: " + rectangle.calculateArea());
    }
}
```

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

Sol:-

```
abstract class AbstractClass {
    public void concreteMethod1() {
        System.out.println("This is concrete method 1.");
    }

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

    public abstract void abstractMethod1();

    public abstract void abstractMethod2();
}

class ConcreteClass extends AbstractClass {
    public void abstractMethod1() {
        System.out.println("This is implementation of abstract method 1.");
    }

    public void abstractMethod2() {
        System.out.println("This is implementation of abstract method 2.");
    }
}

public class example {
    public static void main(String[] args) {
        ConcreteClass concreteObj = new ConcreteClass();

        concreteObj.concreteMethod1();
        concreteObj.concreteMethod2();
        concreteObj.abstractMethod1();
        concreteObj.abstractMethod2();
    }
}
```

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

Sol:-

```
// Functional Interface
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

public class example {
    public static void main(String[] args) {
        // Using Lambda Expression
        Calculator addition = (a, b) -> a + b;
        System.out.println("Addition: " + addition.calculate(5, 3));

        Calculator subtraction = (a, b) -> a - b;
        System.out.println("Subtraction: " + subtraction.calculate(10, 4));

        Calculator multiplication = (a, b) -> a * b;
        System.out.println("Multiplication: " + multiplication.calculate(6, 2));

        Calculator division = (a, b) -> a / b;
        System.out.println("Division: " + division.calculate(20, 5));
    }
}
```

Q4. What is an interface in Java?

Sol:-

=> Interface is a Java Feature, it will allow only abstract methods.

=> In Java applications, for interfaces, we are able to create only reference variables, we are unable to create objects.

=> In the case of interfaces, by default, all the variables are “public static final”.

=> In the case of interfaces, by default, all the methods are “public and abstract”.

=> In Java applications, constructors are possible in classes and abstract classes but constructors are not possible in interfaces.

=> Interfaces will provide more shareability in Java applications when compared with classes and abstract classes.

Important key points of Interface.

1. Whenever we are implementing an interface compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.
2. Whenever we are implementing an interface method, it should be declared as public, otherwise we will get compile time error.
3. In Java applications, it is not possible to extend more than one class to a single class but it is possible to extend more than one interface to a single interface.
4. In Java applications, it is possible to implement more than one interface into a single implementation class.
5. A class can extend a class and can implement any no. Of interfaces simultaneously.
6. An interface can extend any no. Of interfaces at a time.

==========

Q5. What is the use of interface in Java?

Sol:-

1. An interface is used to achieve full abstraction.
2. Using interface is the best way to expose our project’s API to some other projects.
3. Programmers to use interface to customise features of software differently for different object.
4. By using interface we can achieve the functionality of multiple inheritance.

Q6. What is the lambda expression of Java 8?

Sol:-

Lambda expressions are a feature introduced in Java 8 that provide a concise way to represent anonymous functions or function-like constructs. They enable the use of functional programming concepts in Java and facilitate writing more expressive and concise code.

```
(parameters) -> { body }
```

- **Parameters:-** Specifies the input parameters of the lambda expression. It can be empty if there are no parameters, or multiple parameters separated by commas.
- **Arrow Operator:-** Consists of the `->` symbol, which separates the parameters from the body of the lambda expression.
- **Body:-** Contains the implementation of the lambda expression. It can be a single expression or a block of code enclosed in curly braces. If it is a block of code, you may include multiple statements and use a return statement if necessary.

==========

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

Sol:-

Yes, in Java, lambda expressions can be passed as arguments to methods. This capability is often referred to as "passing behavior as an argument" or "behavior parameterization." It allows you to define the desired behavior in the form of a lambda expression and pass it to a method that expects a functional interface as a parameter.

==========

Q8. What is the functional interface in Java 8?

Sol:-

In Java 8, a functional interface is an interface that has exactly one abstract method. It is also known as a Single Abstract Method (SAM) interface or a functional interface because it represents a single unit of behavior.

The concept of functional interfaces is closely tied to the introduction of lambda expressions in Java 8. Lambda expressions provide a concise way to implement the abstract method of a functional interface.

Functional interfaces serve as the foundation for utilizing lambda expressions and functional programming concepts in Java. They enable the use of functional-style programming paradigms, such as passing behavior as arguments, method references, and composing functions.

The **@FunctionalInterface** annotation is used to mark an interface as a functional interface. While not strictly required, it is a good practice to use this annotation to indicate the intended usage of the interface.

==========

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

Sol:-

1.Concise and Readable Code:- Lambda expressions allow you to write more concise code by reducing boilerplate code. They provide a more compact syntax for expressing behavior, making the code easier to read and understand.

2.Functional Programming:- Lambda expressions facilitate functional programming in Java. They enable you to treat behavior as a first-class citizen by passing functions as arguments, returning functions as results, and storing functions in variables. This allows for writing more expressive and declarative code, focusing on what needs to be done rather than how to do it.

3.Enhanced APIs:- Lambda expressions improve the APIs in Java by enabling the use of functional interfaces. Functional interfaces, combined with lambda expressions, provide a more intuitive and flexible way to work with APIs that require behavioral customization. This leads to cleaner and more natural API designs.

4.Code Reusability and Flexibility:- Lambda expressions promote code reusability and flexibility by allowing behavior to be passed as arguments. This enables the creation of generic methods and functions that can be customized with different behaviors. It reduces code duplication and promotes modular and reusable code.

5.Improved Performance:- Lambda expressions can improve performance in certain scenarios by allowing more efficient use of resources. For example, they enable lazy evaluation and enable certain optimizations in stream processing operations, leading to better performance.

6.Multithreading and Parallelism:- Lambda expressions are well-suited for working with concurrent and parallel programming paradigms. They can be used with functional interfaces in the java.util.concurrent package and with parallel stream operations, simplifying the implementation of parallel algorithms.

7.Enhanced Event Handling:- Lambda expressions provide a concise and expressive way to handle events and callbacks. They are commonly used in event-driven programming, GUI frameworks, and callback-based APIs, reducing the verbosity and improving the readability of event handling code.

==========

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

Sol:-

No, it is not mandatory for a lambda expression to have parameters. The presence or absence of parameters in a lambda expression depends on the functional interface it is implementing.

If the functional interface has no parameters in its abstract method, you can define a lambda expression without any parameters. Here's an example:-

```
@FunctionalInterface
interface NoParamInterface {
    void performAction();
}

public class example {
    public static void main(String[] args) {
        // Lambda expression with no parameters
```

```
        NoParamInterface action = () -> System.out.println("Performing an action");
        action.performAction();
    }
}
```

In the example above, we have a functional interface called **NoParamInterface** with a single abstract method that takes no parameters (**performAction()**). We create an instance of this interface using a lambda expression without any parameters. The lambda expression defines the implementation of the abstract method, which in this case is simply printing a message.

However, if the functional interface has parameters in its abstract method, you need to specify the parameters in the lambda expression. For example, if the abstract method has two parameters of type **int**, the lambda expression would need to include those parameters:-

```
@FunctionalInterface
interface ParamInterface {
    int performOperation(int a, int b);
}

public class example {
    public static void main(String[] args) {
        // Lambda expression with parameters
        ParamInterface operation = (a, b) -> a + b;
        int result = operation.performOperation(5, 3);
        System.out.println("Result: " + result);
    }
}
```

In this example, the **ParamInterface** functional interface has an abstract method **performOperation()** that takes two **int** parameters. The lambda expression used to implement this interface includes the parameters **(a, b)** and specifies the addition operation.