**Assignment Questions 4**

<aside> 💡 **Q1.1. Write a program to show Interface Example in java?**

</aside>

<aside> 💡 **Q2.Write a program a Program with 2 concrete method and 2 abstract method in java ?**

</aside>

<aside> 💡 **Q3.Write a program to show the use of functional interface in java?**

</aside>

<aside> 💡 **Q4.What is an interface in Java?**

</aside>

<aside> 💡 **Q5.What is the use of interface in Java?**

</aside>

<aside> 💡 **Q6.What is the lambda expression of Java 8?**

</aside>

<aside> 💡 **Q7.Can you pass lambda expressions to a method? When?**

</aside>

<aside> 💡 **Q8.What is the functional interface in Java 8?**

</aside>

<aside> 💡 **Q9.What is the benefit of lambda expressions in Java 8?**

</aside>

<aside> 💡 [**Q10.Is**](http://Q10.Is) **it mandatory for a lambda expression to have parameters?**

</aside>

Q1. Write a program to show Interface Example in Java.

Ans.

// Interface definition

interface Shape {

double calculateArea(); // Abstract method

double calculatePerimeter(); // Abstract method

}

// Rectangle class implementing the Shape interface

class Rectangle implements Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

public double calculateArea() {

return length \* width;

}

public double calculatePerimeter() {

return 2 \* (length + width);

}

}

// Circle class implementing the Shape interface

class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

public double calculateArea() {

return Math.PI \* radius \* radius;

}

public double calculatePerimeter() {

return 2 \* Math.PI \* radius;

}

}

// Main class

public class InterfaceExample {

public static void main(String[] args) {

Rectangle rectangle = new Rectangle(5, 3);

System.out.println("Rectangle Area: " + rectangle.calculateArea());

System.out.println("Rectangle Perimeter: " + rectangle.calculatePerimeter());

Circle circle = new Circle(4);

System.out.println("Circle Area: " + circle.calculateArea());

System.out.println("Circle Perimeter: " + circle.calculatePerimeter());

}

}

Rectangle Area: 15.0

Rectangle Perimeter: 16.0

Circle Area: 50.26548245743669

Circle Perimeter: 25.132741228718345

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

Ans.

// Abstract class

abstract class AbstractClass {

// Concrete method

public void concreteMethod1() {

System.out.println("Concrete Method 1");

}

// Concrete method

public void concreteMethod2() {

System.out.println("Concrete Method 2");

}

// Abstract method

public abstract void abstractMethod1();

// Abstract method

public abstract void abstractMethod2();

}

// Subclass extending the AbstractClass

class Subclass extends AbstractClass {

// Implementation of abstractMethod1

public void abstractMethod1() {

System.out.println("Abstract Method 1 implementation");

}

// Implementation of abstractMethod2

public void abstractMethod2() {

System.out.println("Abstract Method 2 implementation");

}

}

// Main class

public class AbstractExample {

public static void main(String[] args) {

Subclass subclass = new Subclass();

subclass.concreteMethod1();

subclass.concreteMethod2();

subclass.abstractMethod1();

subclass.abstractMethod2();

}

}

Q.3 Write a program to show the use of a functional interface in Java.

// Functional Interface with a single abstract method

@FunctionalInterface

interface MyFunction {

void performAction();

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

// Lambda expression representing the implementation of the abstract method

MyFunction myFunction = () -> {

System.out.println("Performing action");

};

// Calling the abstract method using the functional interface instance

myFunction.performAction();

}

}

Q.4 // Functional Interface with a single abstract method

@FunctionalInterface

interface MyFunction {

void performAction();

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

// Lambda expression representing the implementation of the abstract method

MyFunction myFunction = () -> {

System.out.println("Performing action");

};

// Calling the abstract method using the functional interface instance

myFunction.performAction();

}

}

Q.5 Q5. What is the use of an interface in Java?

Interfaces in Java have several uses and benefits:

Abstraction: Interfaces allow you to define a contract or a set of rules that a class must follow. By declaring methods in an interface without providing the implementation details, interfaces help achieve abstraction. They define what should be done, without specifying how it should be done.

Multiple Inheritance: Java classes can implement multiple interfaces, unlike inheritance where a class can extend only one superclass. This feature of interfaces enables a class to inherit behaviors from multiple sources, promoting code reuse and flexibility.

Polymorphism: Interfaces can be usedas reference types, allowing for polymorphism. This means that a variable of an interface type can hold objects of any class that implements that interface. This enables writing code that is more generic and flexible, as it can work with different implementations of the same interface.

Contractual Obligation: Interfaces provide a contract that specifies what methods a class must implement. By implementing an interface, a class agrees to fulfill the obligations specified by the interface. This promotes consistency and ensures that classes adhere to a certain set of rules and behaviors.

API Design: Interfaces are widely used in API design. They define a set of methods that should be implemented by classes to use a specific functionality or service. By providing interfaces, API designers can create a clear and well-defined way for developers to interact with their code.

Loose Coupling: Interfaces promote loose coupling between classes. By depending on interfaces rather than concrete classes, you can write code that is more modular and independent of specific implementations. This enhances flexibility, maintainability, and testability of the code.

Overall, interfaces in Java play a crucial role in achieving abstraction, enabling multiple inheritance, promoting polymorphism, defining contracts, designing APIs, and facilitating loose coupling between classes. They are a fundamental part of the object-oriented programming paradigm in Java.

Q6. What is the lambda expression in Java 8?

In Java 8, lambda expressions were introduced as a new feature to provide a concise way of representing anonymous functions. A lambda expression is a block of code that can be treated as an object. It is often used to implement functional interfaces, which are interfaces that contain a single abstract method.

The syntax for a lambda expression in Java is as follows:

cssCopy code

(parameters) -> { body }

Here, **parameters** represent the input parameters of the lambda expression, and **body** represents the code to be executed.

Lambda expressions allow you to write shorter and more readable code compared to traditional anonymous inner classes. They provide a functional programming style in Java by enabling the use of functional interfaces and promoting the use of methods as first-class citizens.

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

Yes, you can pass lambda expressions as arguments to methods in Java. Lambda expressions can be used as a convenient way to represent implementations of functional interfaces.

When passing a lambda expression as a method argument, the method parameter should be of a functional interface type. The lambda expression should match the signature (parameter types and return type) of the single abstract method in the functional interface.

Here's an example that demonstrates passing a lambda expression to a method:

In this example, we have a functional interface named **MathOperation** with a single abstract method called **operate()**. The **Calculator** class has a static method **calculate()** that takes two integers and a **MathOperation** as arguments.

We pass a lambda expression **(a, b) -> a + b** to the **calculate()** method. The lambda expression represents the addition operation, which matches the signature of the **operate()** method in the **MathOperation** interface.

The **calculate()** method invokes the **operate()** method on the provided lambda expression, and the result is returned and printed.

Q8. What is a functional interface in Java 8?

In Java 8, a functional interface is an interface that contains only a single abstract method, known as the functional method. It can also contain default methods and static methods, but the presence of a single abstract method is what makes it a functional interface.

Functional interfaces play a significant role in enabling the use of lambda expressions in Java 8. Lambda expressions can be used to provide the implementation of the single abstract method in a functional interface, allowing concise and expressive code.

Java 8 introduced the **@FunctionalInterface** annotation to provide a hint to the compiler that an interface is intended to be a functional interface. While the annotation is not mandatory for a functional interface, it helps in avoiding accidental addition of multiple abstract methods.

Functional interfaces can be used wherever a functional programming style is desired, such as with lambda expressions, method references, and the Stream API.

Q9. What are the benefits of lambda expressions in Java 8?

Lambda expressions in Java 8 provide several benefits, including:

1. Concise Syntax: Lambda expressions offer a more compact and concise syntax compared to traditional anonymous inner classes. They allow you to write shorter and more readable code, reducing boilerplate code and improving code expressiveness.
2. Functional Programming: Lambda expressions facilitate functional programming in Java. They enable the use of functional interfaces, which are interfaces with a single abstract method, and promote the use of methods as first-class citizens. This allows for a more functional programming style, focusing on behavior rather than object-oriented structures.
3. Code Readability and Maintainability: By providing a more expressive and concise syntax, lambda expressions can enhance code readability. They allow developers to focus on the essential behavior and logic of the code, making it easier to understand and maintain.
4. Improved API Design: Lambda expressions enable the design of APIs that accept functional interfaces, making it easier for developers to provide custom behavior or implementations. This promotes code reuse, modularity, and extensibility.
5. Performance Optimization: Lambda expressions can improve performance in certain scenarios by allowing for more efficient and optimized code execution. They enable better utilization of resources, such as avoiding unnecessary object creation and reducing overhead.
6. Enhanced Multithreading and Parallelism: Lambda expressions facilitate the use of functional programming techniques for concurrent and parallel programming. They can be used with streams and parallel streams in the Stream API to achieve efficient parallel processing and take advantage of multi-core processors.

Overall, lambda expressions in Java 8 provide powerful and flexible features that enhance code expressiveness, readability, maintainability, and performance, while promoting functional programming concepts.

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

No, it is not mandatory for a lambda expression to have parameters. Lambda expressions can be parameterless, depending on the functional interface they are used with.

If the functional interface has a single abstract method that takes no arguments, a lambda expression representing that method can also be parameterless.

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

interface MyFunction {

void performAction();

}

public class LambdaExpressionExample {

public static void main(String[] args) {

MyFunction myFunction = () -> {

System.out.println("Performing action");

};