Assignment 4 Solution | JAVA

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_04;

//Define the interface
interface Vehicle {
	void start();

	void stop();
}

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

	@Override
	public void stop() {
		System.out.println("Car stopped");
	}
}

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

	@Override
	public void stop() {
		System.out.println("Motorcycle stopped");
	}
}

//Main program
public class InterfaceExample_1 {
	public static void main(String[] args) {
		// Create objects of the implementing classes
		Car car = new Car();
		Motorcycle motorcycle = new Motorcycle();
```

```
                    // Use the interface methods
                    car.start();
                    car.stop();

                    motorcycle.start();
                    motorcycle.stop();
          }
}
```

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_04;

//Abstract class with abstract and concrete methods
abstract class Shape {
          // Abstract methods
          public abstract double calculateArea();

          public abstract double calculatePerimeter();

          // Concrete methods
          public void displayArea() {
                    System.out.println("Area: " + calculateArea());
          }

          public void displayPerimeter() {
                    System.out.println("Perimeter: " + calculatePerimeter());
          }
}

//Concrete subclass implementing the Shape abstract class
class Rectangle extends Shape {
          private double length;
          private double width;

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

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

```
        @Override
        public double calculatePerimeter() {
                return 2 * (length + width);
        }
}

//Concrete subclass implementing the Shape abstract class
class Circle extends Shape {
        private double radius;
        private final double PI = 3.14159;

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

        @Override
        public double calculateArea() {
                return PI * radius * radius;
        }

        @Override
        public double calculatePerimeter() {
                return 2 * PI * radius;
        }
}

//Main program
public class ShapeExample_2 {
        public static void main(String[] args) {
                Rectangle rectangle = new Rectangle(5, 7);
                Circle circle = new Circle(3);

                rectangle.displayArea();
                rectangle.displayPerimeter();

                circle.displayArea();
                circle.displayPerimeter();
        }
}
```

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_04;

//Functional interface with a single abstract method
@FunctionalInterface
interface MathOperation {
	int operate(int a, int b);
}

//Main program
public class FunctionalInterfaceExample_3 {
	public static void main(String[] args) {
		// Using lambda expressions to implement the functional interface
		MathOperation addition = (a, b) -> a + b;
		MathOperation subtraction = (a, b) -> a - b;
		MathOperation multiplication = (a, b) -> a * b;

		// Perform operations using the functional interface implementations
		int result1 = performOperation(5, 3, addition);
		System.out.println("Result of addition: " + result1);

		int result2 = performOperation(10, 4, subtraction);
		System.out.println("Result of subtraction: " + result2);

		int result3 = performOperation(6, 8, multiplication);
		System.out.println("Result of multiplication: " + result3);
	}

	// Method that takes a functional interface as a parameter
	public static int performOperation(int a, int b, MathOperation mathOperation) {
		return mathOperation.operate(a, b);
	}
}
```

💡 Q4.What is an interface in Java?

Answer:

An interface in Java is a programming construct that allows the definition of a contract that classes must adhere to. Here are 10 key points about interfaces in Java:

1. An interface is declared using the `**interface**` keyword in Java.
2. An interface can contain abstract methods, default methods, static methods, and constant fields.
3. Abstract methods declared in an interface do not have a method body and are implicitly public and abstract.
4. Classes that implement an interface must provide implementations for all its abstract methods.
5. Interfaces provide a way to achieve multiple inheritance in Java, as a class can implement multiple interfaces.
6. Interfaces are used to define common behaviors or functionality that can be shared among multiple classes.
7. Interfaces establish a contract between the interface and the implementing classes, specifying what methods they must provide.
8. Interfaces can be used to achieve loose coupling and enhance code modularity by programming to interfaces rather than concrete classes.
9. Interfaces can be extended by other interfaces using the `extends` keyword, enabling the creation of interface hierarchies.
10. Java 8 introduced default methods in interfaces, which allow the declaration of method implementations within the interface itself. This feature enables backward compatibility for existing implementations when adding new methods to interfaces.

Overall, interfaces provide a **powerful mechanism for abstraction, polymorphism, and defining contracts in Java**, enabling code **reusability** and **promoting good software design** practices.

💡 Q5.What is the use of interface in Java?

Answer:

Interfaces in Java serve several important purposes and have various uses. Here are 10 key points highlighting the use of interfaces in Java:

1. **Define Contracts**: Interfaces allow you to define contracts that classes must adhere to. By implementing an interface, a class commits to providing the methods declared in the interface, ensuring a certain level of functionality.

2. **Achieve Multiple Inheritance:** Java does not support multiple inheritance with classes, but interfaces allow a class to implement multiple interfaces, enabling it to inherit behavior from multiple sources.

3. **Encourage Polymorphism**: Interfaces facilitate polymorphism, allowing objects of different classes that implement the same interface to be treated uniformly, providing flexibility and modularity in code design.

4. **Promote Loose Coupling:** Programming to interfaces rather than concrete classes promotes loose coupling between components, making code more flexible, maintainable, and easier to test and refactor.

5. **Enable API Design and Versioning:** Interfaces are instrumental in designing APIs by providing a clear specification of methods and their expected behaviour. They also enable backward compatibility when new methods are added in subsequent versions of an API through the use of default methods.

6. **Encapsulate Common Behaviours:** Interfaces allow the definition of common behaviours or functionality that can be shared among unrelated classes. This promotes code reuse and reduces duplication.

7. **Enable Dependency Injection**: Interfaces play a crucial role in implementing dependency injection patterns, where components depend on abstractions (interfaces) rather than concrete implementations. This improves code modularity, testability, and flexibility.

8. **Facilitate Callbacks and Event Handling**: Interfaces are often used to define callback mechanisms or event handlers, allowing objects to register and respond to events or notifications.

9. **Enable Service Providers**: Interfaces are used to define service contracts in scenarios where different implementations can be dynamically loaded at runtime. This is commonly seen in frameworks and libraries that support plug-in architectures.

10. **Support Strategy and Template Patterns:** Interfaces provide the foundation for the strategy and template design patterns, allowing classes to encapsulate different algorithms or provide customizable behaviour by implementing specific interfaces.

Overall, interfaces are a fundamental part of the Java language and are widely used to define **contracts, promote abstraction, support polymorphism, encourage modularity, and enable flexible and extensible software design**.

💡 Q6.What is the lambda expression of Java 8?

Answer:

In Java 8, lambda expressions were introduced as a new language feature. A lambda expression is a concise way to represent a functional interface, which is an interface with a single abstract method. Lambda expressions provide a more concise and expressive way to write functional-style code in Java.

The syntax for a lambda expression consists of the following elements:

(parameters) -> expression

or

(parameters) -> { statements; }

Here's a breakdown of the elements:

- **Parameters**: The input parameters of the lambda expression, which can be empty or contain one or more parameters.
- **Arrow operator (`->`):** Separates the parameters from the lambda body.
- **Expression**: Represents the single expression to be executed as the body of the lambda. This expression is implicitly returned.
- **Statements**: A block of statements enclosed in curly braces, representing the body of the lambda. If there is more than one statement, a return statement must be used explicitly.

Lambda expressions are often used in conjunction with functional interfaces, allowing you to pass behaviour as an argument to methods or define behaviour inline. They promote the use of functional-style programming and provide a concise alternative to anonymous inner classes.

Here's an example to illustrate the usage of lambda expressions:

```
package in.ineuron.pptAssignmentJAVA_04;
interface MathOperation_ {
        int operate(int a, int b);
}
public class LambdaExample_6 {
        public static void main(String[] args) {
                MathOperation addition = (a, b) -> a + b;
                MathOperation subtraction = (a, b) -> a - b;

                int result1 = addition.operate(5, 3);
                System.out.println("Result of addition: " + result1);

                int result2 = subtraction.operate(10, 4);
                System.out.println("Result of subtraction: " + result2);
        }
}
```

Lambda expressions have revolutionized the way Java developers write code by providing a more concise and expressive syntax for functional programming paradigms.

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

Answer:

Yes, in Java, you can pass lambda expressions as arguments to methods. This is particularly useful when working with functional interfaces, as lambda expressions can be used to provide implementations for the abstract methods defined in those interfaces.

Lambda expressions can be passed as arguments to methods in various scenarios, including:

1. **Functional Interfaces:** When a method expects an object of a functional interface as a parameter, you can pass a lambda expression as an argument to provide the implementation of the abstract method(s) of that interface.

2. **Callbacks and Event Handling:** Methods that expect callback functions or event handlers as parameters can receive lambda expressions as arguments. The lambda expression represents the behaviour to be executed when the specified event or condition occurs.

3. **Higher-Order Functions:** Higher-order functions are functions that can accept other functions as arguments or return functions as results. In Java, you can achieve a similar effect by passing lambda expressions as arguments to methods, allowing for more flexible and dynamic behaviour.

Here's an example to illustrate passing lambda expressions as method arguments:

```
package in.ineuron.pptAssignmentJAVA_04;

interface StringTransformer {
        String transform(String input);
}

class TextProcessor {
        public void processString(String input, StringTransformer transformer) {
                String result = transformer.transform(input);
                System.out.println("Transformed string: " + result);
        }
}

public class LambdaMethodExample_7 {
        public static void main(String[] args) {
                TextProcessor processor = new TextProcessor();

                // Passing lambda expressions as method arguments
                processor.processString("Hello, World!", str -> str.toUpperCase());
                processor.processString("Java is fun!", str -> str.replaceAll("fun", "awesome"));
        }
    }
```

Passing lambda expressions as method arguments allows for more flexibility and modularity in code, as it enables the dynamic specification of behaviour at runtime. It is one of the key features that make lambda expressions powerful in functional programming paradigms.

💡 Q8.What is the functional interface in Java 8?

Answer:

In Java 8, a functional interface is an interface that contains exactly one abstract method. It is also known as a **single abstract method** (SAM) interface. Here are 10 key points about functional interfaces in Java 8:

1. **Definition**: A functional interface is an interface that has only one abstract method, but it can have multiple default or static methods.

2. **Purpose**: Functional interfaces are primarily used to enable the use of lambda expressions and method references, which provide concise and expressive ways to represent behaviour.

3. **Lambda Expression Compatibility:** Since functional interfaces have only one abstract method, they can be implemented using lambda expressions, providing a more concise alternative to anonymous inner classes.

4. **`@FunctionalInterface` Annotation**: The `@FunctionalInterface` annotation is an optional annotation that can be used to indicate that an interface is intended to be a functional interface. It helps detect accidental addition of extra abstract methods.

5. **SAM Conversion**: Functional interfaces can be converted into lambda expressions or method references, and vice versa. This conversion allows the interface's abstract method to be represented by the lambda expression or method reference.

6. **Built-in Functional Interfaces**: Java 8 introduced several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Function`, `Consumer`, and `Supplier`, to provide common functional programming patterns.

7. **Predicate**: The `Predicate` functional interface represents a predicate (boolean-valued function) of one argument and provides methods like `test()`, `and()`, `or()`, and `negate()`.

8. **Function**: The `Function` functional interface represents a function that takes one argument and produces a result. It provides methods like `apply()`, `compose()`, `andThen()`, and `identity()`.

9. **Consumer**: The `Consumer` functional interface represents an operation that takes one argument and returns no result. It provides methods like `accept()`, `andThen()`, and `compose()`.

10. **Supplier**: The `Supplier` functional interface represents a supplier of results. It has no arguments but produces a result. It provides a single method `get()`.

Functional interfaces provide the foundation for functional programming in Java 8 and beyond. They enable the use of lambda expressions and method references, making code more expressive, concise, and modular. The built-in functional interfaces simplify common functional programming patterns and provide standardized ways to represent behavior.

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

Answer:

Lambda expressions introduced in Java 8 bring several benefits to the language. Here are 10 key points highlighting the benefits of lambda expressions:

1. **Concise Syntax:** Lambda expressions provide a more compact and readable syntax compared to anonymous inner classes, reducing boilerplate code.

2. **Functional Programming:** Lambda expressions promote functional programming paradigms by allowing behaviour to be treated as a first-class citizen, enabling the use of higher-order functions and functional-style programming.

3. **Code Clarity:** Lambda expressions make code more expressive and self-documenting by emphasizing the intent of the behaviour being passed or implemented.

4. **Enhanced Readability**: With lambda expressions, the code focuses on the "what" rather than the "how," making it easier to understand the purpose and logic of the code.

5. **Improved API Design:** Lambda expressions facilitate the design of APIs by allowing the passing of behaviour as arguments to methods, promoting flexibility and customization.

6. **Enable Functional Interfaces:** Lambda expressions enable the use of functional interfaces, which define contracts for behaviour, allowing for greater code modularity and reusability.

7. **Simplified Multithreading**: Lambda expressions can simplify the implementation of concurrent and parallel programming by providing a concise syntax for expressing tasks and operations on collections.

8. **Encourages Modularization:** Lambda expressions promote modular design and composition, as they can be combined and chained together to create complex behaviour with minimal code.

9. **Facilitates Stream API:** Lambda expressions are an integral part of the Stream API in Java 8, enabling powerful and concise operations on collections and sequences of data.

10. **Improved Developer Productivity:** The concise syntax and expressive nature of lambda expressions allow developers to write code more quickly and efficiently, boosting productivity.

Overall, lambda expressions in Java 8 offer a more modern and functional approach to writing code. They provide benefits such as increased code readability, concise syntax, modular design, improved API design, and enhanced developer productivity, making Java a more expressive and powerful language.

💡 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. Here are 10 points regarding lambda expressions and parameters:

1. **Parameterless Lambda:** Lambda expressions can be written without any parameters when the expected functional interface has no input arguments.

2. **Empty Parameter List**: A lambda expression without parameters is denoted by an empty parameter list `()`.

3. **() -> Expression:** The lambda expression syntax with no parameters consists of empty parentheses `()` followed by the arrow operator `->` and an expression to be evaluated.

4. **No Access to Parameters**: In a parameterless lambda expression, there are no parameters available for use within the expression. It is typically used when the behaviour doesn't depend on any input.

5. **Supplier Example**: A common use case for parameterless lambdas is with the `Supplier` functional interface, which represents a supplier of values with no input. For example, `() -> "Hello, World!"` can be used as a `Supplier<String>` lambda expression.

6. **Runnable Interface:** The `Runnable` interface, which represents a task that can be executed, is another example where a parameterless lambda expression can be used. For instance, `() -> System.out.println("Task executed")` can be used as a `Runnable` lambda expression.

7. **Functional Interface Compatibility**: A lambda expression must match the functional interface's method signature, meaning the number and types of arguments must align. Thus, if the functional interface expects no arguments, the lambda expression must also have an empty parameter list.

8. **Method References**: In some cases, when a lambda expression has no parameters, it can be replaced with a method reference. Method references provide an alternative syntax for referring to an existing method without invoking it.

9. **Flexibility**: The presence or absence of parameters in a lambda expression depends on the requirements of the functional interface being implemented. Lambda expressions provide flexibility in capturing and representing behaviour, including the ability to omit parameters when they are not needed.

10. **Functional Interface Design**: When designing functional interfaces, it is essential to consider the appropriate number and types of parameters based on the intended usage. This ensures compatibility with lambda expressions that implement the interface.

In summary, while lambda expressions typically involve parameters to represent behaviour with input arguments, it is not mandatory for a lambda expression to have parameters. The decision to include or omit parameters depends on the requirements of the functional interface being implemented and the specific use case.