ASSIGNMENT-4(JAVA CORE MODULE)

Q. Write a program to show Interface Example in Java?

Ans-

```
// Define an interface
interface Animal {
    void sound(); // Abstract method declaration
}

// Implement the interface
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Woof");
    }
}

class Cat implements Animal {
    @Override
    public void sound() {
        System.out.println("Meow");
    }
}

// Main class to test the interface
public class interface_example {
    public static void main(String[] args) {
        Animal dog = new Dog(); // Create an instance of the Dog
class
        dog.sound(); // Invoke the sound() method implemented in the
Dog class

        Animal cat = new Cat(); // Create an instance of the Cat
class
        cat.sound(); // Invoke the sound() method implemented in the
Cat class
    }
}
```

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

Ans-

```
// Abstract class
abstract class Shape {
    // Abstract methods
    public abstract void draw();
    public abstract double calculateArea();

    // Concrete methods
    public void display() {
        System.out.println("This is a shape.");
    }

    public void getColor() {
        System.out.println("The color of the shape is red.");
    }
}

// Concrete class implementing the Shape abstract class
class Circle extends Shape {
    private double radius;

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

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

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

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

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

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

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

// Main class to test the Shape hierarchy
public class ShapeExample {
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        circle.draw();
        double circleArea = circle.calculateArea();
        System.out.println("Circle area: " + circleArea);

        Rectangle rectangle = new Rectangle(4.0, 6.0);
        rectangle.draw();
        double rectangleArea = rectangle.calculateArea();
        System.out.println("Rectangle area: " + rectangleArea);

        circle.display();
        circle.getColor();

        rectangle.display();
        rectangle.getColor();
    }
}
```

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

Ans-

```java
// Define a functional interface
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

// Main class to test the functional interface
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Use lambda expression to implement the functional
interface
        Calculator addition = (a, b) -> a + b;
        int result1 = addition.calculate(5, 3);
        System.out.println("Addition: " + result1);

        Calculator subtraction = (a, b) -> a - b;
        int result2 = subtraction.calculate(8, 4);
        System.out.println("Subtraction: " + result2);

        Calculator multiplication = (a, b) -> a * b;
        int result3 = multiplication.calculate(6, 2);
        System.out.println("Multiplication: " + result3);

        Calculator division = (a, b) -> a / b;
        int result4 = division.calculate(10, 5);
        System.out.println("Division: " + result4);
    }
}
```

Q4. What is an interface in Java?

Ans- In Java, an interface is a reference type that specifies a contract or a set of methods that a class must implement. It defines a collection of abstract methods, constant fields, and default methods. An interface does not provide any implementation details; it only declares the method signatures.

Here are some key points about interfaces in Java:

1. Declaration: Interfaces are declared using the `interface` keyword in Java. For example: `interface MyInterface { ... }`

2. Abstract Methods: Interfaces can contain abstract methods, which are methods without a body (implementation). These methods are implicitly public and abstract. Any class implementing the interface must provide implementations for all the abstract methods.

3. Constant Fields: Interfaces can also declare constant fields, which are implicitly public, static, and final. These fields define values that cannot be changed.

4. Multiple Inheritance: Java supports multiple inheritance of interfaces. A class can implement multiple interfaces by separating them with commas.

5. Implementation: Classes implement an interface using the `implements` keyword. The implementing class must provide concrete implementations for all the abstract methods defined in the interface.

6. Interface Inheritance: Interfaces can also extend other interfaces using the `extends` keyword. This allows for the creation of hierarchical interfaces.

7. Functional Interfaces: Java 8 introduced functional interfaces, which are interfaces that have a single abstract method. Functional interfaces can be used with lambda expressions and method references, enabling the use of functional programming concepts.

Interfaces in Java provide a way to achieve abstraction, decoupling the implementation details from the interface definition. They promote code reusability, flexibility, and good software design by defining a contract that classes can adhere to without specifying the implementation. Interfaces are widely used in Java for defining APIs, defining callback mechanisms, and implementing various design patterns.

Q5. What is the use of interface in Java?

Ans- Interfaces in Java serve several purposes and provide various benefits. Here are some of the key uses of interfaces in Java:

1. Abstraction and Polymorphism: Interfaces allow you to define a contract of methods that a class must implement, without specifying the implementation details. This promotes abstraction by separating the interface from the concrete implementation. It also enables polymorphism, as objects of different classes that implement the same interface can be treated interchangeably through the interface reference.

2. API Design: Interfaces are often used to define APIs (Application Programming Interfaces) in Java. By providing an interface, you define a clear and standardized way for users to interact with your

code. Interfaces can define methods that represent the actions or behaviors expected from implementing classes, providing a consistent and well-defined interface for developers to work with.

3. Code Reusability: Interfaces enable code reuse by allowing multiple classes to implement the same interface. This promotes a design principle called "programming to an interface," which encourages writing code that depends on interfaces rather than concrete implementations. This makes it easier to swap implementations and ensures that different classes can be used interchangeably if they adhere to the same interface.

4. Multiple Inheritance: Interfaces in Java support multiple inheritance, unlike classes that can only inherit from a single superclass. A class can implement multiple interfaces, enabling it to inherit behavior and functionality from multiple sources. This flexibility allows for greater code reuse and the combination of different functionalities from various interfaces.

5. Callback Mechanism: Interfaces are commonly used in Java to implement callback mechanisms. A class can define an interface that represents a callback contract, and other classes can implement that interface to provide the necessary callback behavior. This allows for loose coupling and dynamic behavior, as one class can invoke methods on the interface reference without knowing the concrete implementation.

6. Functional Programming: With the introduction of functional interfaces in Java 8, interfaces can be used to define lambda expressions and support functional programming concepts. Functional interfaces have a single abstract method and can be used with lambda expressions or method references, enabling concise and expressive code for functional-style programming.

Overall, interfaces play a crucial role in Java by providing a mechanism for defining contracts, promoting abstraction and polymorphism, enabling code reuse and modularity, and supporting API design and functional programming. They contribute to writing flexible, maintainable, and extensible code in Java.

Q6. What is the lambda expression of Java8?

Ans- In Java 8, lambda expressions were introduced as a new language feature. A lambda expression is a concise way to represent an anonymous function, i.e., a function without a name. It allows you to write more compact and expressive code, especially when working with functional interfaces.

The syntax of a lambda expression consists of three main parts:

1. Parameter list: It specifies the input parameters of the function. If the function does not take any parameters, an empty set of parentheses is used. For example: `()`, `(int x, int y)`, etc.

2. Arrow token: It is represented by the `->` symbol and separates the parameter list from the body of the lambda expression. It denotes the assignment of input parameters to the function body. For example: `->`, `-> { ... }`, etc.

3. Function body: It contains the code that implements the functionality of the lambda expression. It can be a single expression or a block of statements enclosed in curly braces `{}`. For example: `x + y`, `{ ... }`, etc.

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

Ans- Yes, in Java, you can pass lambda expressions as arguments to methods. This capability allows you to pass behavior or functionality as a parameter to a method, which is useful for implementing callbacks, event handling, or performing operations on collections.

Lambda expressions can be passed to methods when:

1. Functional Interfaces are Involved: If the method parameter type is a functional interface, you can pass a lambda expression as an argument. A functional interface is an interface with a single abstract method, also known as a SAM (Single Abstract Method) interface. Lambda expressions can provide a concise implementation of that single abstract method.

2. Callback Mechanisms: When a method expects a callback, such as an event listener or a callback function, you can pass a lambda expression to specify the behavior to be executed when the callback is triggered. The lambda expression serves as a compact way to define the callback logic inline.

3. Stream Operations: Java 8 introduced the Stream API, which allows you to perform operations on collections in a declarative and functional manner. Many methods in the Stream API accept lambda expressions as arguments to specify operations to be performed on elements of the stream, such as filtering, mapping, or reducing.

Q8. What is the functional interface in Java8?

Ans- 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. Functional interfaces are a key concept in functional programming and serve as the foundation for using lambda expressions and method references.

Here are some characteristics of functional interfaces in Java 8:

1. Single Abstract Method (SAM): A functional interface has only one abstract method. This method represents the behavior or functionality that the interface defines.

2. Default and Static Methods: Functional interfaces can have default and static methods in addition to the single abstract method. These methods provide default implementations or utility methods that can be used by implementing classes.

3. Lambda Expressions and Method References: Functional interfaces are designed to be used with lambda expressions and method references. A lambda expression can be used to provide a concise implementation of the single abstract method defined in the functional interface.

4. @FunctionalInterface Annotation: Java provides the `@FunctionalInterface` annotation to explicitly declare an interface as functional. While not mandatory, using this annotation can help enforce the contract of a functional interface and catch accidental addition of extra abstract methods.

Functional interfaces are widely used in Java 8 and beyond, especially in the context of the Stream API, which facilitates functional-style programming on collections. By providing a concise and expressive way to define behavior, functional interfaces enable the use of lambda expressions and support functional programming paradigms in Java.

Java 8 also introduced several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Function`, `Consumer`, and `Supplier`, which cover common use cases for functional programming and provide pre-defined functional interfaces for various types of operations. These built-in functional interfaces can be used directly or as a basis for creating custom functional interfaces.

Q9. What is the benefit of lambda expression in Java?

Ans-

Lambda expressions introduced in Java 8 offer several benefits and advantages, enhancing the expressiveness and flexibility of the language. Here are some key benefits of lambda expressions in Java 8:

1. Concise and Readable Code: Lambda expressions allow you to write more compact and concise code compared to traditional anonymous inner classes. They eliminate the need for boilerplate code, such as explicit class definitions, method names, and return statements. This leads to improved code readability and easier understanding of the intent.

2. Functional Programming Support: Lambda expressions promote functional programming concepts in Java. They enable writing code in a declarative and functional style, focusing on the "what" rather than the "how." This leads to more expressive and maintainable code that is often easier to reason about and test.

3. Enhanced APIs: The addition of lambda expressions has allowed for the design of APIs that accept functional interfaces as parameters. This enables developers to pass behavior or functions as arguments, providing greater flexibility and customization when working with APIs.

4. Collection Stream Operations: Lambda expressions are closely tied to the Stream API introduced in Java 8. The Stream API provides a functional programming approach to work with collections, allowing for expressive and efficient stream operations such as filtering, mapping, reducing, and more. Lambda expressions are used to define the behavior for these operations, resulting in concise and readable code for data manipulation.

5. Improved Performance: In some cases, lambda expressions can lead to improved performance. The use of lambda expressions can enable the Java compiler to generate more efficient bytecode and optimize the execution of certain operations, resulting in potential performance gains.

6. Code Reusability and Flexibility: Lambda expressions promote code reusability and modularity by providing a way to pass behavior as an argument. They allow for the creation of more generic and flexible code that can be easily adapted to different use cases by providing different lambda expressions.

7. Parallel Processing: Lambda expressions can facilitate parallel processing by enabling easy parallelization of operations on collections. The Stream API, coupled with lambda expressions, provides a convenient way to parallelize operations and leverage multi-core processors for improved performance.

Overall, the introduction of lambda expressions in Java 8 brings many benefits, including more expressive and readable code, improved APIs, enhanced support for functional programming, and performance optimization. They have significantly expanded the capabilities of the Java language, making it more modern and suitable for a wide range of programming styles and paradigms.

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

Ans-

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 associated with and the specific requirements of the functionality being implemented.

Lambda expressions can have zero or more parameters. When a lambda expression has zero parameters, an empty set of parentheses is used to indicate that there are no input parameters.

Here are some examples to illustrate lambda expressions with different parameter counts:

1. Lambda expression with no parameters:

```java
Runnable runnable = () -> {
   // Code to be executed
};
```

2. Lambda expression with one parameter:

```java
Consumer<String> consumer = (str) -> {
   System.out.println(str);
};
```

3. Lambda expression with multiple parameters:

```java
BinaryOperator<Integer> sum = (a, b) -> a + b;
```

4. Lambda expression with inferred parameters (Java 8 and later):

```java
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;
```

As you can see, lambda expressions can be defined with or without parameters, depending on the functional interface and the requirements of the specific code being implemented. The decision to include or exclude parameters in a lambda expression is determined by the number and types of

parameters specified in the functional interface's abstract method that the lambda expression is implementing.