Assignment No: 4

Que1 . Answer :

```
// Defining an interface
interface Vehicle {
    void start();
    void stop();
}
// Implementing the interface
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started");
    }
    @Override
    public void stop() {
        System.out.println("Car stopped");
    }
}
class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike started");
    }
    @Override
    public void stop() {
        System.out.println("Bike stopped");
    }
}
// Main class
public class InterfaceExample {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start();
        car.stop();
        Vehicle bike = new Bike();
        bike.start();
        bike.stop();
    }
}
```

Que2.Answer :

```
abstract class AbstractClass {
    // Concrete method
    public void concreteMethod1() {
        System.out.println("This is concrete method 1.");
    }
    // Concrete method
    public void concreteMethod2() {
        System.out.println("This is concrete method 2.");
    }
    // Abstract method
    public abstract void abstractMethod1();
    // Abstract method
    public abstract void abstractMethod2();
}
// Concrete subclass
class ConcreteClass extends AbstractClass {
    @Override
    public void abstractMethod1() {
        System.out.println("This is implementation of abstract method 1.");
    }
    @Override
    public void abstractMethod2() {
        System.out.println("This is implementation of abstract method 2.");
    }
}
// Main class
public class AbstractExample {
    public static void main(String[] args) {
        ConcreteClass obj = new ConcreteClass();
        obj.concreteMethod1();
        obj.concreteMethod2();
        obj.abstractMethod1();
        obj.abstractMethod2();
    }
}
```

Que3.Answer :

```
interface MyFunction {
   void performAction();
}

// Main class
public class FunctionalInterfaceExample {
   public static void main(String[] args) {
      // Creating an instance of the functional interface using a lambda expression
      MyFunction myFunction = () -> {
         System.out.println("Performing an action...");
      };

      // Calling the method defined in the functional interface
      myFunction.performAction();
   }
}
```

Que4.Answer :

Declaration: *An interface is declared the usage of the* ***"interface"*** *keyword, accompanied by the interface name and a fixed of method signatures.*

Method signatures*: Interfaces contain only technique declarations with out technique our bodies. These techniques are implicitly public and abstract. Starting from Java 8, interfaces can also have default and static methods with approach bodies.*

Implementation*: A magnificence implements an interface with the aid of imparting an implementation for all the strategies declared within the interface.*
*The "****implements****" keyword is used to set up the relationship between the elegance and the interface it implements.*

Multiple inheritance*: Java permits a category to put in force more than one interfaces. This allows for a class to inherit and provide implementations for methods defined in more than one interfaces.*

Contract: *An interface defines a settlement, specifying what methods a category enforcing the interface have to offer. Any magnificence that implements the interface have to adhere to this contract and offer the specified techniques.*

Que5.Answer :

In simple use of interfaces in Java

1. ***Defining a contract**: An interface allows you to define a set of methods that a class must implement. It acts as a contract or agreement that specifies what methods a class should have.*
2. ***Achieving polymorphism**: Interfaces enable polymorphic behaviour, where objects of different classes that implement the same interface can be treated interchangeably. This means you can write code that works with any object that adheres to the interface, providing flexibility and reusability.*
3. ***API design and modularity**: Interfaces are often used in designing APIs (Application Programming Interfaces) to establish a clear set of operations that classes should support. This promotes modularity, allowing different parts of a program to interact through well-defined interfaces without worrying about implementation details.*
4. ***Supporting multiple inheritances**: Unlike classes, which can only inherit from a single superclass, a class can implement multiple interfaces. This allows you to inherit behaviour and contracts from multiple sources, enabling greater flexibility in structuring and reusing code.*
5. ***Future compatibility and extensibility**: Interfaces provide a stable contract between components. This means that even if you change or add new implementations of an interface, the existing code that relies on the interface will still work. It allows for easier .*

Que6.Answer :

Lambda expression consists of:

1. ***Syntax**: A lambda expression starts with a set of parameters (if any) enclosed in parentheses, followed by the arrow symbol **->**, and then the body of the function.*
2. ***Parameters**: The parameters represent the inputs to the function. They can be optional if the function doesn't require any inputs. If there is only one parameter, you can omit the parentheses around it. For multiple parameters, separate them with commas.*
3. ***Arrow Symbol**: The arrow symbol **->** separates the parameters from the body of the function. It indicates the start of the lambda expression.*
4. ***Body**: The body of the lambda expression contains the code that is executed when the lambda function is called. It can be a single expression or a block of code enclosed in curly braces. If it's a block, you may need to include a return statement if a value is expected.*

Que7.Answer :

Yes, you can pass lambda expressions as arguments to methods in Java. This allows you to provide custom conduct or capability to a method with out explicitly enforcing a separate class or interface.
Lambda expressions are usually utilized in conditions in which strategies accept practical interfaces as parameters. A functional interface is an interface that incorporates best one summary method.
you can pass lambda expressions to methods:

***Callbacks**: If a method expects a callback characteristic to be finished at a positive factor, you can bypass a lambda expression that represents that callback. The method can then invoke the lambda expression at the perfect time, executing the preferred behaviour.*
***Sorting and Filtering**: Methods that perform sorting or filtering operations on collections often be given lambda expressions to outline custom evaluation common sense or filtering situations. The lambda expression specifies the standards for sorting or filtering, allowing for bendy and custom designed operations.*
***Iteration and Mapping**: Methods that iterate over collections or perform mapping operations may additionally be given lambda expressions to define the behavior for each element. The lambda expression specifies the action to be finished on every detail, offering a manner to customize the processing.*
***Asynchronous Programming**: When operating with asynchronous programming, strategies can take lambda expressions to outline the behavior to be performed in a separate thread or as a callback while an asynchronous operation completes.*

Que8.Answer :
In simple terms, a practical interface in Java 8 is an interface that includes precisely one abstract method. It is designed to be used with lambda expressions or approach references, permitting you to express behavior as a single, functional unit.

Here are some key factors approximately purposeful interfaces in Java 8

***Single Abstract Method:** A functional interface should have most effective one summary technique. It also can include default techniques and static techniques delivered in Java 8, however it must have exactly one summary approach.*

***Lambda Expressions and Method References**: Functional interfaces are regularly used with lambda expressions or approach references. Lambda expressions let you provide a concise implementation for the unmarried summary approach, while technique references let you talk over with current techniques as implementations.*

Que9.Answer :

In simple terms, lambda expressions in Java 8 provide numerous blessings that make your code extra concise, expressive, and flexible:

***Conciseness**: Lambda expressions allow you to express functionality as a compact and concise piece of code. They dispose of the need for writing boilerplate code whilst enforcing purposeful interfaces, reducing the general verbosity of your code.*

***Readability**: Lambda expressions decorate code clarity through focusing on the center conduct or functionality. By getting rid of the need for outlining separate classes or techniques for simple operations, lambda expressions make it less difficult to recognize the reason of the code.*

***Code Reusability**: Lambda expressions promote code reusability. Instead of writing more than one implementations of a functional interface, you could bypass lambda expressions as arguments to techniques, making your code greater modular and reusable.*

***Flexibility**: Lambda expressions allow you to write down code this is extra bendy and adaptable. By presenting behavior as a controversy, you can without problems customise the behavior of techniques, making them extra versatile and able to dealing with one-of-a-kind use instances.*

***Functional Programming**: Lambda expressions facilitate useful programming paradigms in Java. They allow you to treat functions as fine residents, enabling functional composition, higher-order capabilities, and other practical programming strategies.*

Que10.Answer :

No, it is not mandatory for a lambda expression to have parameters. Lambda expressions can be parameter less if the corresponding functional interface does not require any input.
In simpler terms, a lambda expression can be written without any parameters when the desired behavior or functionality does not depend on any specific inputs. This is especially useful when working with functional interfaces that define methods without parameters.
*For example, consider a functional interface called **Runnable** that represents a task to be executed without any inputs or outputs. Here's an example of a lambda expression without parameters that implements the **Runnable** interface.*