**Core Java 8:**

**Concurrent Paterns in Java, Concurrent Collections, Lambda Expressions, Stream API, Introduction to Design Pattern, GitHub, Introduction to JDBC: Connection, Statement, PreparedS tatement, ResultSet.**

1. **Explain Lambda Expression with a syntax and differentiate between Lambda Expression and over Regular Methods. Write JAVA Lambda Expression program using one parameter.**

Java Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

## Syntax:

## The simplest lambda expression contains a single parameter and an expression:

## *parameter* -> *expression*

## To use more than one parameter, wrap them in parentheses:

## *(*parameter1*,* parameter2*)* -> expression

Use a lambda expression in the ArrayList's forEach() method to print every item in the list:

import java.util.ArrayList;

public class p1 {

  public static void main(String[] args) {

    ArrayList<Integer> numbers = new ArrayList<Integer>();

    numbers.add(5);

    numbers.add(9);

    numbers.add(8);

    numbers.add(1);

    numbers.forEach( (n) -> { System.out.println(n); } );

  }

}

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

import java.util.ArrayList;

import java.util.function.Consumer;

public class  p2{

  public static void main(String[] args) {

    ArrayList<Integer> numbers = new ArrayList<Integer>();

    numbers.add(5);

    numbers.add(9);

    numbers.add(8);

    numbers.add(1);

    Consumer<Integer> method = (n) -> { System.out.println(n); };

    numbers.forEach( method );

  }

}

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

interface StringFunction {

    String run(String str);

  }

  public class p3 {

    public static void main(String[] args) {

      StringFunction exclaim = (s) -> s + "!";

      StringFunction ask = (s) -> s + "?";

      printFormatted("Hello", exclaim);

      printFormatted("Hello", ask);

    }

    public static void printFormatted(String str, StringFunction format) {

      String result = format.run(str);

      System.out.println(result);

    }

  }

### **Lambda Expression in Java**

Lambda expressions were introduced in Java 8 to provide a concise way to represent anonymous functions — functions without a name that can be passed around as arguments to methods or stored in variables. They are particularly useful in functional programming paradigms.

#### **Syntax of Lambda Expression**

The syntax of a lambda expression in Java has the following parts:

(parameters) -> expression

or

(parameters) -> { statements; }

Here's a breakdown:

* **Parameters**: These are similar to method parameters but without explicit types. If there's only one parameter, you can omit the parentheses. For example, x -> ... instead of (x) -> ....
* **Arrow (->)**: This separates parameters from the body of the lambda expression.
* **Expression/Body**: This is the code that implements the function. If it's a single expression, you can omit the braces {}. If it's a block of statements, you must use braces {}.

### Differences Between Lambda Expressions and Regular Methods

1. **Syntax**: Lambda expressions are more concise and do not require a method name, return type declaration, and sometimes even parameter types (if they can be inferred).
2. **Purpose**: Lambda expressions are primarily used for implementing functional interfaces (interfaces with a single abstract method), while regular methods can have any access modifier, be static or instance methods, and can belong to any class.
3. **Flexibility**: Lambda expressions are more flexible in terms of where they can be used (like as arguments to methods or stored in variables), while regular methods are more rigidly defined in terms of where and how they can be called.

### Example Program: Lambda Expression with One Parameter

// Functional interface with one abstract method

interface MyFunctionalInterface {

    void sayMessage(String message);

}

public class LambdaExample {

    public static void main(String[] args) {

        // Lambda expression to implement MyFunctionalInterface

        MyFunctionalInterface myLambda = (message) -> System.out.println("Hello, " + message);

        // Calling the lambda expression

        myLambda.sayMessage("Lambda Expression!");

    }

}

### Example Program: Lambda exp With Multiple Parameters

// Functional interface with multiple parameters

interface MathOperation {

    int operate(int a, int b);

}

public class LambdaWithMultipleParameters {

    public static void main(String[] args) {

        // Lambda expression to implement MathOperation interface for addition

        MathOperation addition = (a, b) -> a + b;

        // Lambda expression to implement MathOperation interface for subtraction

        MathOperation subtraction = (a, b) -> a - b;

        // Lambda expression to implement MathOperation interface for multiplication

        MathOperation multiplication = (a, b) -> a \* b;

        // Using the lambda expressions

        System.out.println("Addition: " + addition.operate(10, 5)); // Output: 15

        System.out.println("Subtraction: " + subtraction.operate(10, 5)); // Output: 5

        System.out.println("Multiplication: " + multiplication.operate(10, 5)); // Output: 50

    }

}

#### Explanation of the Example

* **MyFunctionalInterface**: This is a functional interface with a single abstract method sayMessage(String message).
* **Lambda Expression**: (message) -> System.out.println("Hello, " + message) is the lambda expression that implements the sayMessage method of MyFunctionalInterface. Here, message is the parameter passed to the lambda expression.
* **Main Method**: In the main method, we create an instance myLambda of MyFunctionalInterface using the lambda expression. Then, we invoke myLambda.sayMessage("Lambda Expression!"), which prints Hello, Lambda Expression!.

**Design and implement a simple JDBC application program**:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class JDBCTest {

    // JDBC URL, username, and password of MySQL server

    private static final String JDBC\_URL = "jdbc:mysql://localhost:3306/testdb";

    private static final String JDBC\_USER = "your\_username";

    private static final String JDBC\_PASSWORD = "your\_password";

    // JDBC variables for opening and managing connection

    private static Connection connection;

    private static PreparedStatement preparedStatement;

    private static ResultSet resultSet;

    public static void main(String[] args) {

        try {

            // Connect to the MySQL database

            connection = DriverManager.getConnection(JDBC\_URL, JDBC\_USER, JDBC\_PASSWORD);

            // Insert a new record into the employees table

            insertEmployee("John Doe", 30, "IT");

            // Print all records from the employees table

            printEmployees();

        } catch (SQLException e) {

            e.printStackTrace();

        } finally {

            // Close JDBC objects in finally block to ensure they're closed even if an exception is thrown

            try {

                if (resultSet != null) {

                    resultSet.close();

                }

                if (preparedStatement != null) {

                    preparedStatement.close();

                }

                if (connection != null) {

                    connection.close();

                }

            } catch (SQLException e) {

                e.printStackTrace();

            }

        }

    }

    private static void insertEmployee(String name, int age, String department) throws SQLException {

        // SQL query to insert a new record into the employees table

        String sql = "INSERT INTO employees (name, age, department) VALUES (?, ?, ?)";

        preparedStatement = connection.prepareStatement(sql);

        // Set parameters for the preparedStatement

        preparedStatement.setString(1, name);

        preparedStatement.setInt(2, age);

        preparedStatement.setString(3, department);

        // Execute the query

        int rowsInserted = preparedStatement.executeUpdate();

        if (rowsInserted > 0) {

            System.out.println("A new employee was inserted successfully!");

        }

    }

    private static void printEmployees() throws SQLException {

        // SQL query to select all records from the employees table

        String sql = "SELECT id, name, age, department FROM employees";

        preparedStatement = connection.prepareStatement(sql);

        // Execute the query and get the ResultSet

        resultSet = preparedStatement.executeQuery();

        // Iterate through the ResultSet and print each employee's information

        while (resultSet.next()) {

            int id = resultSet.getInt("id");

            String name = resultSet.getString("name");

            int age = resultSet.getInt("age");

            String department = resultSet.getString("department");

            System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Department: " + department);

        }

    }

}

**Explain the following with a program example: i. Statement ii. PreparedStatement**

### 1. Statement Interface

The Statement interface in JDBC is used to execute a static SQL statement that is sent to the database. It doesn't allow parameters to be passed dynamically into the SQL query. Here’s how you typically use Statement:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class StatementExample {

    // JDBC URL, username, and password of MySQL server

    private static final String JDBC\_URL = "jdbc:mysql://localhost:3306/testdb";

    private static final String JDBC\_USER = "your\_username";

    private static final String JDBC\_PASSWORD = "your\_password";

    public static void main(String[] args) {

        Connection connection = null;

        Statement statement = null;

        ResultSet resultSet = null;

        try {

            // Connect to the MySQL database

            connection = DriverManager.getConnection(JDBC\_URL, JDBC\_USER, JDBC\_PASSWORD);

            // Create a statement

            statement = connection.createStatement();

            // Execute a query

            String sql = "SELECT \* FROM employees";

            resultSet = statement.executeQuery(sql);

            // Process the ResultSet

            while (resultSet.next()) {

                int id = resultSet.getInt("id");

                String name = resultSet.getString("name");

                int age = resultSet.getInt("age");

                String department = resultSet.getString("department");

                System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Department: " + department);

            }

        } catch (SQLException e) {

            e.printStackTrace();}}}

### 2. PreparedStatement Interface

The PreparedStatement interface extends Statement and provides more flexibility and performance improvements, especially when executing the same SQL statement multiple times with different parameters. It allows you to dynamically pass parameters into the SQL query. Here’s how you typically use PreparedStatement:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class PreparedStatementExample {

    // JDBC URL, username, and password of MySQL server

    private static final String JDBC\_URL = "jdbc:mysql://localhost:3306/testdb";

    private static final String JDBC\_USER = "your\_username";

    private static final String JDBC\_PASSWORD = "your\_password";

    public static void main(String[] args) {

        Connection connection = null;

        PreparedStatement preparedStatement = null;

        ResultSet resultSet = null;

        try {

            // Connect to the MySQL database

            connection = DriverManager.getConnection(JDBC\_URL, JDBC\_USER, JDBC\_PASSWORD);

            // SQL query with parameters

            String sql = "SELECT \* FROM employees WHERE department = ?";

            // Create a PreparedStatement with parameterized query

            preparedStatement = connection.prepareStatement(sql);

            // Set parameters dynamically

            preparedStatement.setString(1, "IT");

            // Execute the query

            resultSet = preparedStatement.executeQuery();

            // Process the ResultSet

            while (resultSet.next()) {

                int id = resultSet.getInt("id");

                String name = resultSet.getString("name");

                int age = resultSet.getInt("age");

                String department = resultSet.getString("department");

                System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Department: " + department);

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

    }

}

**Define Streams. List out the difference between stream and collections with examples.**

### **Definition of Streams:**

In Java, streams represent a sequence of elements from a source that supports functional-style operations. Streams allow you to process collections of data in a declarative and functional way, without explicitly writing loops and iterations. They operate on the source data structure (like collections, arrays, or I/O channels) to produce a result.

### **Key Characteristics of Streams:**

1. **Sequence of Elements**: Streams represent a sequence of elements that can be processed sequentially.
2. **Functional Operations**: Streams support functional-style operations such as filter, map, reduce, collect, etc., which can be chained together to form a pipeline of operations.
3. **Laziness**: Intermediate operations in a stream are lazily evaluated, meaning they are executed only when necessary (when a terminal operation is invoked).
4. **Internal Iteration**: Streams use internal iteration, where the iteration logic is handled by the Stream API itself, as opposed to external iteration (e.g., using for loops).

### **Differences Between Streams and Collections:**

#### 1. **Nature of Data Handling:**

* **Collections**: Store and manage elements directly. They allow for direct access, insertion, deletion, and modification of elements.
* **Streams**: Do not store elements themselves; they operate on a source (like a collection or array) and process elements from that source.

#### **2. Mutability:**

* **Collections**: Mutable data structures where elements can be added, removed, or modified directly.
* **Streams**: Typically immutable once created; operations on streams do not modify the original source but produce a new stream or result.

#### **3. Execution:**

* **Collections**: Operations are executed eagerly. For example, when iterating through a collection with a loop, each element is processed immediately.
* **Streams**: Operations are executed lazily. Intermediate operations (like filter or map) are deferred until a terminal operation (like forEach or collect) is invoked. This lazy evaluation allows for optimizations like short-circuiting.

#### **4. Processing Paradigm:**

* **Collections**: Emphasize external iteration, where you explicitly control the iteration process (e.g., using for or while loops).
* **Streams**: Emphasize internal iteration, where the Stream API handles the iteration process and you define operations on the elements.

**Example:**

**Example 1: Using Collections**

import java.util.ArrayList;

import java.util.List;

public class CollectionsExampleforDiffinstrdcol{

    public static void main(String[] args) {

        // Creating a list of names using ArrayList

        List<String> names = new ArrayList<>();

        names.add("Alice");

        names.add("Bob");

        names.add("Charlie");

        // Direct access and modification

        System.out.println("Accessing element directly: " + names.get(1)); // Bob

        // Iterating through the collection

        System.out.println("Iterating through the collection:");

        for (String name : names) {

            System.out.println(name);

        }

        // Adding a new element

        names.add("David");

        System.out.println("After adding a new element:");

        for (String name : names) {

            System.out.println(name);

        }

    }

}

**Example Using Streams:**

import java.util.Arrays;

import java.util.List;

public class StreamsExample {

    public static void main(String[] args) {

        // Creating a list of names using Arrays.asList

        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using streams to process elements

        System.out.println("Using streams to process elements:");

        names.stream()

                .filter(name -> name.startsWith("A")) // Intermediate operation (filtering)

                .forEach(System.out::println); // Terminal operation (printing)

        // Attempting to modify the stream (not possible, streams are immutable)

        names.stream().map(String::toUpperCase).forEach(System.out::println); // Uncommenting this line will cause an exception

        // Adding a new element (not possible directly with streams)

        names.add("David"); // Uncommenting this line will cause an UnsupportedOperationException

    }

}