Lambda Expression

1. What is the lambda expression of Java 8

Ans1. In Java 8, **lambda expressions** provide a clear and concise way to represent **anonymous functions** (functional interfaces). They enable you to pass behavior as parameters, making your code more readable and reducing boilerplate.

Syntax of Lambda Expressions

A lambda expression consists of three parts:

```
(parameters) -> { body }
```

- **Parameters**: The input arguments (can be zero, one, or multiple).
- **Arrow (`->`)**: Separates parameters from the function body.
- **Body**: The logic to be executed (can be a single statement or a block of statements).

Examples of Lambda Expressions

1. Basic Lambda with One Parameter

```
// Using a lambda expression to define a simple function

InterfaceExample example = (name) -> System.out.println("Hello, " + name);
```

```
example.sayHello("Java 8"); // Output: Hello, Java 8
```

2. **Lambda with Multiple Parameters**

```
// Functional interface with two parameters

interface MathOperation {

    int operate(int a, int b);

}

// Using a lambda expression

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

```
System.out.println(addition.operate(5, 3)); // Output: 8
```

3. Lambda without Parameters

```
Runnable task = () -> System.out.println("Running a thread using Lambda");

new Thread(task).start();
```

4. Lambda with a Block of Statements

```
MathOperation multiply = (a, b) -> {

    int result = a * b;

    return result;

};
```

```
System.out.println(multiply.operate(4, 5)); // Output: 20
```

Using Lambda with Java 8 Functional Interfaces

Java 8 introduced functional interfaces (interfaces with a single abstract method), such as:

- `Runnable`
- `Callable`
- `Comparator<T>`
- `Function<T, R>`
- `Predicate<T>`
- `Consumer<T>`

Example using `Predicate<T>`:

```
import java.util.function.Predicate;

public class LambdaExample {

    public static void main(String[] args) {

        Predicate<Integer> isEven = (n) -> n % 2 == 0;
        System.out.println(isEven.test(10)); // Output: true

        System.out.println(isEven.test(11)); // Output: false

    }

}
```

Benefits of Lambda Expressions

- **Reduces Boilerplate Code: No need for anonymous classes.**
- **Improves Readability: More concise and expressive.**
- **Enhances Functional Programming: Works well with Java 8 Stream API.**
- **Enables Parallel Processing: Especially useful with the `Stream` API.**

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

Ans2. Yes! You can pass lambda expressions to a method when the method expects a functional interface as a parameter. Since lambda expressions are essentially implementations of functional interfaces (interfaces with a single abstract method), they can be used wherever such interfaces are required.

Example: Passing Lambda Expression to a Method

1. Passing a Lambda to a Custom Method

Let's define a functional interface and a method that accepts it as a parameter.

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

public class LambdaExample {
    // Method that accepts a lambda expression (functional interface)
    static int executeOperation(int a, int b, MathOperation operation) {
        return operation.operate(a, b);
    }

    public static void main(String[] args) {
        // Passing lambda expressions as arguments
        MathOperation addition = (x, y) -> x + y;
        MathOperation multiplication = (x, y) -> x * y;

        System.out.println(executeOperation(5, 3, addition));       // Output: 8
        System.out.println(executeOperation(5, 3, multiplication)); // Output: 15
    }
```

```
}
```

Explanation:

- **`executeOperation` expects a `MathOperation` functional interface.**
- **We pass different lambda expressions (`addition` and `multiplication`) to change the behavior dynamically.**

---

2. Passing a Lambda to Java 8 Built-in Functional Interfaces

Java 8 introduced common functional interfaces in `java.util.function`. We can pass lambda expressions as arguments to methods that accept these interfaces.

Example: Using `Predicate<T>`

```
import java.util.function.Predicate;

public class LambdaPredicateExample {

    static void filterNumber(int num, Predicate<Integer> condition) {

        if (condition.test(num)) {
```

```
        System.out.println(num + " meets the condition");

    } else {

        System.out.println(num + " does not meet the condition");

    }

}



public static void main(String[] args) {

    Predicate<Integer> isEven = (n) -> n % 2 == 0;



    filterNumber(10, isEven); // Output: 10 meets the condition
```

```
        filterNumber(11, isEven); // Output: 11 does not meet the condition

    }

}
```

3. Passing a Lambda to a Stream API Method

Lambda expressions are frequently used in the Java Stream API.

```
import java.util.Arrays;

import java.util.List;



public class StreamLambdaExample {

    public static void main(String[] args) {

        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
        // Using lambda with filter (Predicate) in a stream
        names.stream()
            .filter(name -> name.startsWith("A"))
            .forEach(System.out::println); // Output: Alice
    }
}
```

3. What is the functional interface in Java 8?

Ans3. **Functional Interface in Java 8**

A **functional interface** in Java 8 is an interface that contains **exactly one abstract method**. It can have multiple **default** and **static** methods, but only **one abstract method**.

Functional interfaces are the foundation of **lambda expressions** in Java 8 because lambda expressions provide an implementation for the single abstract method.

Declaring a Functional Interface

A functional interface is typically annotated with `@FunctionalInterface`, though this annotation is optional. It helps ensure that the interface follows the **single abstract method rule**.

Example of a Functional Interface

```
@FunctionalInterface

interface MyFunctionalInterface {

    void doSomething(); // Single abstract method

}
```

Now, we can use a **lambda expression** to provide an implementation:

```
public class LambdaExample {

    public static void main(String[] args) {
        MyFunctionalInterface func = () -> System.out.println("Hello from Lambda!");

        func.doSomething(); // Output: Hello from Lambda!

    }

}
```

Examples of Java 8 Functional Interfaces

1. `Predicate<T>` Example

A `Predicate` is used for **filtering** values.

```
import java.util.function.Predicate;



public class PredicateExample {

    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;

        System.out.println(isEven.test(10)); // true

        System.out.println(isEven.test(15)); // false

    }

}
```

2. `Function<T, R>` Example

A `Function` transforms an input into an output.

```
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {

        Function<String, Integer> stringLength = str -> str.length();

        System.out.println(stringLength.apply("Java 8")); // Output: 6

    }

}
```

3. `Consumer<T>` Example

A `Consumer` performs an action but **does not return** anything.

```
import java.util.function.Consumer;

public class ConsumerExample {
    public static void main(String[] args) {

        Consumer<String> printMessage = message -> System.out.println("Message: " + message);

        printMessage.accept("Hello, Java!"); // Output: Message: Hello, Java!

    }

}
```

4. `Supplier<T>` Example

A `Supplier` provides a value without taking any input.

```
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {

        Supplier<Double> randomNumber = () -> Math.random();

        System.out.println(randomNumber.get()); // Output: Random number

    }

}
```

4. Why do we use lambda expressions in Java

Ans4. Lambda expressions in Java bring several significant advantages that improve both **code readability** and **maintainability**. They are primarily used to enable functional programming in Java, allowing you to treat functions as **first-class citizens**.

Here are some of the key reasons **why we use lambda expressions** in Java:

1. Concise Code

Lambda expressions allow you to write **more compact and expressive code**. Instead of writing verbose anonymous classes, lambdas provide a cleaner and more succinct way to define behavior.

This is particularly useful in cases where you need to pass small code snippets, such as event listeners or callback functions.

Example: Without Lambda (Anonymous Class)

```
// Without lambda

Runnable task = new Runnable() {

    @Override

    public void run() {

        System.out.println("Running a task");

    }

};

new Thread(task).start();
```

With Lambda Expression

```
// With lambda

Runnable task = () -> System.out.println("Running a task");

new Thread(task).start();
```

2. Improves Readability

Lambda expressions make the code more **readable** by removing boilerplate code and making the intent clear. For example, when using lambdas in the **Stream API**, operations like filtering, mapping, and reducing are expressed in a declarative way.

Example: Using Lambda with Streams

```
List<String> names = Arrays.asList("John", "Jane", "James", "Jack");

// Using lambda in Stream API for filtering

names.stream()
```

```
    .filter(name -> name.startsWith("J"))

    .forEach(System.out::println);
```

3. Enables Functional Programming

Java 8 introduced **functional programming features**, and lambda expressions are a key part of that. Lambdas allow Java developers to pass behavior as arguments (functions as first-class citizens), which is a core idea in functional programming.

You can use lambdas with functional interfaces like `Predicate`, `Function`, `Consumer`, and `Supplier` to write code in a **functional style**, enabling more declarative and concise operations on collections and streams.

Example: Passing Behavior via Lambda

```
public class FunctionalExample {

    static void executeOperation(int a, int b, MathOperation operation) {

        System.out.println("Result: " + operation.operate(a, b));

    }
```

```
   public static void main(String[] args) {

       executeOperation(5, 3, (x, y) -> x + y); // Lambda for addition

       executeOperation(5, 3, (x, y) -> x * y); // Lambda for multiplication

   }

}
```

4. Enables Parallel Processing

Lambda expressions are commonly used with the **Stream API** in Java, which allows you to process collections in a **parallel** and **concurrent** manner easily. Since lambdas can be executed in parallel, they enhance the ability to process large data sets or perform operations asynchronously.

Example: Parallel Stream with Lambda

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Using parallel stream to process data in parallel
```

```
numbers.parallelStream()
        .map(n -> n * n) // Squaring each number
        .forEach(System.out::println);
```

5. Reduces Boilerplate Code

Without lambda expressions, you'd need to write **anonymous inner classes** for many operations. This increases code size and complexity. Lambdas allow you to replace anonymous classes with more concise, easier-to-read expressions.

Example: Without Lambda (Event Listener)

```
// Without lambda
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});
```

With Lambda

```
// With lambda
button.addActionListener(e -> System.out.println("Button clicked"));
```

6. Enables Higher-Order Functions

Lambda expressions allow Java to use **higher-order functions**, which means you can pass functions as arguments to other functions or return them. This leads to more **flexible** and **dynamic** code.

Example: Returning a Function from a Method

```
public class HigherOrderExample {
    static Function<Integer, Integer> getMultiplier(int factor) {
        return x -> x * factor;
    }

    public static void main(String[] args) {

        Function<Integer, Integer> multiplyBy2 = getMultiplier(2);

        System.out.println(multiplyBy2.apply(5)); // Output: 10

    }

}
```

7. Better Integration with APIs

Many Java 8 APIs, such as **Stream**, **Optional**, and **java.time**, make extensive use of lambda expressions. Using lambdas allows you to leverage these APIs efficiently and express complex operations with a small, declarative style.

Example: Using Lambda with Optional

```
Optional<String> name = Optional.of("Java");
```

```
// Using lambda to transform the value inside Optional

name.ifPresent(n -> System.out.println("Name is: " + n));
```

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

Ans5. No, it is **not mandatory** for a lambda expression to have parameters. Lambda expressions in Java can **have zero or more parameters**, depending on the requirements of the functional interface you are working with.

Here’s how it works:

1. Lambda Expression with No Parameters

If the functional interface method doesn’t require any parameters, you can define a lambda expression without parameters.

Example: Lambda Expression with No Parameters

```
// Functional interface with no parameters

@FunctionalInterface

interface NoParamFunction {
    void execute();
}

public class LambdaExample {
    public static void main(String[] args) {
        // Lambda with no parameters
        NoParamFunction greet = () -> System.out.println("Hello, World!");
        greet.execute(); // Output: Hello, World!
    }
}
```

2. Lambda Expression with One Parameter

If the functional interface method takes one parameter, the lambda expression must provide one parameter.

Example: Lambda Expression with One Parameter

```
// Functional interface with one parameter

@FunctionalInterface

interface SingleParamFunction {

    void printMessage(String message);

}



public class LambdaExample {

    public static void main(String[] args) {
        // Lambda with one parameter

        SingleParamFunction print = (message) -> System.out.println(message);

        print.printMessage("Hello, Lambda!"); // Output: Hello, Lambda!

    }

}
```

3. Lambda Expression with Multiple Parameters

If the functional interface method takes multiple parameters, the lambda expression must provide multiple parameters.

Example: Lambda Expression with Multiple Parameters

```
// Functional interface with two parameters
```

```
@FunctionalInterface
interface AddNumbers {
    int add(int a, int b);
}

public class LambdaExample {
    public static void main(String[] args) {
        // Lambda with two parameters
        AddNumbers sum = (a, b) -> a + b;
        System.out.println(sum.add(5, 3)); // Output: 8
    }

}
```

Example of All Variants:

```
public class LambdaExample {

    public static void main(String[] args) {

        // Lambda with no parameters

        Runnable task = () -> System.out.println("Task running...");

        task.run(); // Output: Task running...
        // Lambda with one parameter

        Consumer<String> printMessage = message -> System.out.println(message);

        printMessage.accept("Hello from Lambda!"); // Output: Hello from Lambda!



        // Lambda with two parameters

        BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;

        System.out.println(sum.apply(10, 20)); // Output: 30

    }

}
```