What are Lambda Expressions?

Lambda expressions were introduced in Java 8 as a way to write concise, functional-style code. Essentially, they are **anonymous functions**—blocks of code that can be passed around and executed later, without needing to define a full method or class.

Syntax:

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

Example:

```
// A lambda expression to add two numbers
(int a, int b) -> a + b
```

- **Lambda Expressions** are used with functional interfaces (interfaces with a single abstract method).
- They make it easier to implement behaviors and pass them as arguments, promoting functional programming in Java.

Advantages of Lambda Expressions

1. **Conciseness:**
    a. Lambda expressions reduce boilerplate code by eliminating the need for anonymous inner classes. **Example (Before Java 8 - Verbose):**

1.
```
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Task executed!");
    }
};
```

Example (Using Lambda):

```
Runnable task = () -> System.out.println("Task executed!");
```

Improved Readability:

a. Code is easier to understand and less cluttered.
b. Focuses on *what to do* rather than *how to do it*.

2. **Functional Programming:**
    a. Encourages functional programming concepts like immutability, lazy evaluation, and method references.
3. **Parallel Processing with Streams:**
    a. Lambda expressions are often used with Streams API for declarative operations like filtering, mapping, and reducing.
4. **Reusability:**
    a. Promotes modular, reusable code by passing behavior (functions) as arguments.
5. **Enhanced Productivity:**
    a. Reduces development time by simplifying coding patterns.

6. Disadvantages of Lambda Expressions

7. **Readability for Complex Logic:**
    a. Overusing lambdas, especially for complex logic, can lead to hard-to-read and maintain code.
8. **Example - Overly Complex Lambda:**

1. **list.stream().filter(x -> x.length() > 3 && x.startsWith("A") && x.endsWith("Z")).forEach(System.out::println);**

Better to refactor it into named functions.

Limited Debugging:

a. Debugging can be challenging because lambdas don't have meaningful names or stack traces.

2. **Type Inference Limitations:**
   a. In some cases, the compiler may fail to infer types correctly, leading to compile-time errors.
3. **Performance Overhead:**
   a. Lambdas might introduce a slight performance overhead (due to capturing variables or creating objects for closures).
4. **One Abstract Method Restriction:**
   a. Lambda expressions can only implement functional interfaces (interfaces with exactly one abstract method).

5. Code Examples of Lambda Expressions

6. *1. Replacing Anonymous Classes*

7. **Before Java 8:**

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
});
thread.start();
```

After Java 8:

```
Thread thread = new Thread(() -> System.out.println("Thread is running"));
thread.start();
```

2. Using Built-in Functional Interfaces

a) Predicate (Functional Interface):

Used for evaluating a condition (returns true/false).

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

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

        System.out.println(isEven.test(4)); // true
        System.out.println(isEven.test(5)); // false
    }
}
```

b) Function (Functional Interface):

Used for transforming a value.

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

public class LambdaFunction {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = str -> str.length();

        System.out.println(stringLength.apply("Shankar")); // 7
    }
}
```

c) Consumer (Functional Interface):

Used to perform an action without returning a result.

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

public class LambdaConsumer {
    public static void main(String[] args) {
        Consumer<String> greet = name -> System.out.println("Hello, " + name

        greet.accept("Shankar"); // Hello, Shankar
    }
}
```

3. Using Lambda in Streams

The Streams API pairs beautifully with lambda expressions to process data in a declarative style.

a) Filtering and Printing:

```java
import java.util.Arrays;
import java.util.List;

public class LambdaStream {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Shankar", "Alice", "Bob", "Kumar

        names.stream()
             .filter(name -> name.startsWith("S"))
             .forEach(System.out::println); // Shankar
    }
}
```

b) Mapping Data:

```java
import java.util.Arrays;
import java.util.List;

public class LambdaStreamMap {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("shankar", "alice", "bob");

        names.stream()
             .map(String::toUpperCase) // Convert to uppercase
             .forEach(System.out::println); // SHANKAR, ALICE, BOB
    }
}
```

c) Reducing Data:

```java
import java.util.Arrays;
import java.util.List;

public class LambdaStreamReduce {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.stream()
                         .reduce(0, (a, b) -> a + b);

        System.out.println("Sum: " + sum); // 15
    }
}
```

4. Custom Functional Interfaces

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

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

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

    System.out.println("Addition: " + addition.operate(10, 5)); // 15
    System.out.println("Multiplication: " + multiplication.operate(10, 5)); // 50
  }
}
```

5. Capturing Variables in Lambda

Lambdas can capture local variables declared in the surrounding scope (these variables must be effectively final).

Example:

```java
public class LambdaCapture {
    public static void main(String[] args) {
        int multiplier = 2;

        Function<Integer, Integer> multiply = n -> n * multiplier;

        System.out.println(multiply.apply(5)); // 10
    }
}
```

Summary

- **What is Lambda?** A shorthand way to write anonymous functions in Java, introduced in Java 8.
- **Advantages:**
  - Improves readability and reduces boilerplate code.
  - Enables functional programming with Stream API.
  - Promotes modular and reusable code.
- **Disadvantages:**
  - Debugging and readability can suffer for complex lambdas.

- Works only with functional interfaces.

```java
public class LambdaExample {

    Run | Debug
    public static void main(String[] args) {
        LambdaExample examples = new LambdaExample();

        System.out.println(x:"Function Example: Getting String Length");
        examples.functionExample();

        System.out.println(x:"Predicate Example: Checking Even Numbers");
        examples.predicateExample();

        System.out.println(x:"Consumer Example: Printing Greetings");
        examples.consumerExample();

        System.out.println(x:"Supplier Example: Generating Messages");
        examples.supplierExample();

        System.out.println(x:"Custom Functional Interface Example");
        examples.customFunctionalInterfaceExample();

        System.out.println(x:"Filtering Names Using Stream API");
        examples.streamFilterExample();

        System.out.println(x:"Mapping Names to Uppercase");
        examples.streamMapExample();

        System.out.println(x:"Reducing Numbers Using Stream API");
        examples.streamReduceExample();

        System.out.println(x:"Method References Example");
        examples.methodReferenceExample();

        System.out.println(x:"Handling Nulls Using Optional");
        examples.optionalExample();

        System.out.println(x:"Sorting Data Using Lambda");
        examples.sortingWithLambda();

        System.out.println(x:"Grouping Data Using Streams");
        examples.groupingWithStreams();

        System.out.println(x:"Multithreading Using Lambda");
        examples.multithreadingExample();
```

```
    public void sortingWithLambda() {
        List<String> cities = Arrays.asList(...a:"Mumbai", "Kolkata", "Bengaluru", "Delhi");
        cities.sort(String::compareTo);
        System.out.println("Sorted cities: " + cities);
    }

    public void groupingWithStreams() {
        List<String> names = Arrays.asList(...a:"Raj", "Ramesh", "Sita", "Sanjay", "Arjun", "Asha");
        Map<Character, List<String>> groupedNames = names.stream().collect(Collectors.groupingBy(name -> name.charAt(index:0)));
        System.out.println("Grouped names by first letter: " + groupedNames);
    }

    public void multithreadingExample() {
        ExecutorService executor = Executors.newFixedThreadPool(nThreads:2);
        executor.submit(() -> System.out.println("Task 1 executed by " + Thread.currentThread().getName()));
        executor.submit(() -> System.out.println("Task 2 executed by " + Thread.currentThread().getName()));
        executor.shutdown();
    }

    public void parallelStreamExample() {
        List<String> names = Arrays.asList(...a:"Ram", "Krishna", "Arjun", "Manoj", "Sita", "Meera");
        names.parallelStream().forEach(name -> System.out.println(Thread.currentThread().getName() + " processed: " + name));
    }

    public void lambdaWithExceptionHandling() {
        Consumer<Integer> safePrint = i -> {
            try {
                if (i == 0) throw new ArithmeticException(s:"Division by zero!");
                System.out.println(10 / i);
            } catch (ArithmeticException e) {
                System.out.println("Error: " + e.getMessage());
            }
        };

        safePrint.accept(t:2);
        safePrint.accept(t:0);
    }
```

```
        safePrint.accept(t:2);
        safePrint.accept(t:0);
    }

    public void fileOperationsExample() {
        try {
            Path path = Files.createTempFile(prefix:"lambda", suffix:".txt");
            Files.write(path, Arrays.asList(...a:"Hello from Java Lambda!"));
            Files.lines(path).forEach(System.out::println);
            Files.delete(path);
        } catch (IOException e) {
            System.out.println("File Error: " + e.getMessage());
        }
    }

    public void databaseSimulationExample() {
        List<String> database = Arrays.asList(...a:"Ramesh", "Sita", "Rahul", "Amit");
        String searchQuery = "Sita";
        Optional<String> result = database.stream().filter(name -> name.equalsIgnoreCase(searchQuery)).findFirst();
        System.out.println("Database Query Result: " + result.orElse(other:"Not Found"));
    }

    public void httpClientExample() {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder().uri(URI.create(str:"https://jsonplaceholder.typicode.com/posts/1")).GET().build();
        client.sendAsync(request, HttpResponse.BodyHandlers.ofString()).thenApply(HttpResponse::body).thenAccept(System.out::println).join();
    }
}
```

```
        System.out.println(x:"Multithreading Using Lambda");
        examples.multithreadingExample();

        System.out.println(x:"Parallel Processing Using Streams");
        examples.parallelStreamExample();

        System.out.println(x:"Exception Handling Inside Lambda");
        examples.lambdaWithExceptionHandling();

        System.out.println(x:"File Operations Using Lambda");
        examples.fileOperationsExample();

        System.out.println(x:"Simulating Database Queries Using Lambda");
        examples.databaseSimulationExample();

        System.out.println(x:"Making HTTP Requests Using Lambda");
        examples.httpClientExample();
    }

    public void functionExample() {
        Function<String, Integer> stringLength = str -> str.length();
        System.out.println("Length of 'Ramesh': " + stringLength.apply(t:"Ramesh"));
    }

    public void predicateExample() {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println("Is 8 even? " + isEven.test(t:8));
        System.out.println("Is 11 even? " + isEven.test(t:11));
    }

    public void consumerExample() {
        Consumer<String> greet = name -> System.out.println("Namaste, " + name);
        greet.accept(t:"Vikas");
    }

    public void supplierExample() {
        Supplier<String> supplyGreeting = () -> "Welcome to India!";
        System.out.println(supplyGreeting.get());
    }
```

This class showcases **lambda expressions** in **functional interfaces, Streams API, exception handling, file operations, database queries, multithreading, and web requests**.

Functional Interfaces

1 Function Example (Getting String Length)

```java
Function<String, Integer> stringLength = str -> str.length();
System.out.println("Length of 'Ramesh': " + stringLength.apply("Ramesh"));
```

- **Concept:** `Function<T, R>` takes an input (`String`) and returns an output (`Integer`).
- **Why it's useful:** It allows processing strings dynamically without creating a separate method.
- **How it works:** `apply("Ramesh")` calls the function, executing `str.length()`.

Expected Output:

```
Length of 'Ramesh': 6
```

2 Predicate Example (Checking Even Number)

```java
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println("Is 8 even? " + isEven.test(8));
System.out.println("Is 11 even? " + isEven.test(11));
```

- **Concept:** `Predicate<T>` returns a **boolean** based on a condition.
- **Why it's useful:** It helps filter collections and validate inputs.
- **How it works:** `.test(value)` checks if the number is divisible by `2`.

Expected Output:

```
Is 8 even? true
Is 11 even? false
```

---

3 Consumer Example (Printing Greetings)

Java Copy

```
Consumer<String> greet = name -> System.out.println("Namaste, " + name);
greet.accept("Vikas");
```

- **Concept:** `Consumer<T>` performs an action but does not return anything.
- **Why it's useful:** It simplifies logging and event handling.
- **How it works:** `accept("Vikas")` prints the greeting message.

Expected Output:

Copy

```
Namaste, Vikas
```

4 Supplier Example (Generating Messages)

Java Copy

```
Supplier<String> supplyGreeting = () -> "Welcome to India!";
System.out.println(supplyGreeting.get());
```

- **Concept:** `Supplier<T>` generates values dynamically without taking input.
- **Why it's useful:** It helps in lazy-loading data and retrieving default values.
- **How it works:** `.get()` returns `"Welcome to India!"`.

Expected Output:

Copy

```
Welcome to India!
```

Custom Functional Interface Example

5 Custom Interface for Arithmetic Operations

Java Copy

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

Java Copy

```java
MathOperation addition = (a, b) -> a + b;
MathOperation multiplication = (a, b) -> a * b;
System.out.println("Addition: " + addition.operate(10, 5));
System.out.println("Multiplication: " + multiplication.operate(10, 5));
```

- **Concept:** Custom interface with a **single abstract method**, allowing flexibility in implementation.
- **Why it's useful:** Helps define behavior dynamically without modifying existing code.
- **How it works:** Different operations (addition, multiplication) are implemented using lambda expressions.

Expected Output:

Copy

```
Addition: 15
Multiplication: 50
```

Streams API

6 Filtering Names Starting with 'A'

Java Copy

```
List<String> names = Arrays.asList("Amit", "Rahul", "Sita", "Krishna", "Arju
List<String> filteredNames = names.stream().filter(name -> name.startsWith('
System.out.println("Filtered names: " + filteredNames);
```

- **Concept:** `.filter(predicate)` keeps elements that match the condition.
- **Why it's useful:** Helps filter collections efficiently.
- **How it works:** The stream processes each name and selects only those starting with `"A"`.

Expected Output:

Copy

```
Filtered names: [Amit, Arjun]
```

7 Mapping Names to Uppercase

Java Copy

```
List<String> names = Arrays.asList("amit", "rahul", "sita");
List<String> upperNames = names.stream().map(String::toUpperCase).collect(Cc
System.out.println("Uppercase names: " + upperNames);
```

- **Concept:** `.map(function)` transforms each element.
- **Why it's useful:** Allows **bulk transformation**.
- **How it works:** Each name is converted to uppercase.

Expected Output:

Copy

```
Uppercase names: [AMIT, RAHUL, SITA]
```

8 Computing Sum Using Reduce

Java Copy

```java
List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 9);
int sum = numbers.stream().reduce(0, Integer::sum);
System.out.println("Sum of numbers: " + sum);
```

- **Concept:** `.reduce(initialValue, operation)` aggregates values.
- **Why it's useful:** Simplifies accumulation logic.
- **How it works:** Adds each number iteratively.

Expected Output:

Copy

```
Sum of numbers: 25
```

Handling Optional Values

9 Preventing NullPointerException

Java Copy

```
Optional<String> city = Optional.ofNullable(null);
System.out.println("City (or default): " + city.orElse("Delhi"));
```

- **Concept:** `Optional<T>` avoids null checks.
- **Why it's useful:** Ensures safe handling of missing values.
- **How it works:** `.orElse(defaultValue)` returns `"Delhi"` if `city` is `null`.

Expected Output:

Copy

```
City (or default): Delhi
```

Multithreading and Parallel Processing

10 Running Tasks in Threads

Java Copy

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.submit(() -> System.out.println("Task 1 executed by " + Thread.curr
executor.submit(() -> System.out.println("Task 2 executed by " + Thread.curr
executor.shutdown();
```

- **Concept:** `ExecutorService` executes multiple tasks in parallel.
- **Why it's useful:** Optimizes performance in concurrent applications.
- **How it works:** Submits tasks for execution by separate threads.

11 Processing Names in Parallel

Java Copy

```
List<String> names = Arrays.asList("Ram", "Krishna", "Arjun", "Manoj", "Sita
names.parallelStream().forEach(name -> System.out.println(Thread.currentThre
```

- **Concept:** `.parallelStream()` enables **parallel computation**.
- **Why it's useful:** Improves efficiency for large datasets.
- **How it works:** Data is processed across multiple CPU cores.

1️⃣2️⃣ Handling Division by Zero

Java Copy

```
Consumer<Integer> safePrint = i -> {
    try {
        if (i == 0) throw new ArithmeticException("Division by zero!");
        System.out.println(10 / i);
    } catch (ArithmeticException e) {
        System.out.println("Error: " + e.getMessage());
    }
};
safePrint.accept(2);
safePrint.accept(0);
```

- **Concept:** `try-catch` inside a lambda.
- **Why it's useful:** Prevents program crashes due to arithmetic errors.

Making HTTP Requests

1️⃣3️⃣ Fetching Data Using Java 11 HttpClient

Java Copy

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder().uri(URI.create("https://jsonp
client.sendAsync(request, HttpResponse.BodyHandlers.ofString()).thenApply(Ht
```

- **Concept:** `HttpClient` makes asynchronous web requests.
- **Why it's useful:** Allows fetching live data.
- **How it works:** `.sendAsync()` executes the request in the background.
