In Java, lambda expressions are typically used to provide the implementation of a functional interface (an interface with a single abstract method but can have default/static methods).

These are primarily used to provide clear and concise syntax for writing code that implements functional interfaces.

Commonly used in conjunction with Java’s Streams API for operations like filtering, mapping, and reducing data.

A concise way to create an instance of a functional interface without writing a full implementation of the interface.

A **lambda expression** is basically a **short block of code** that takes in **parameters** and returns a **value**. You can think of it like an **anonymous method** — a method without a name — that you can pass around as data.

A **lambda expression** is essentially shorthand for implementing a method from a functional interface.

It reduces boilerplate code compared to anonymous class implementations.

The basic syntax of a lambda expression looks like this:

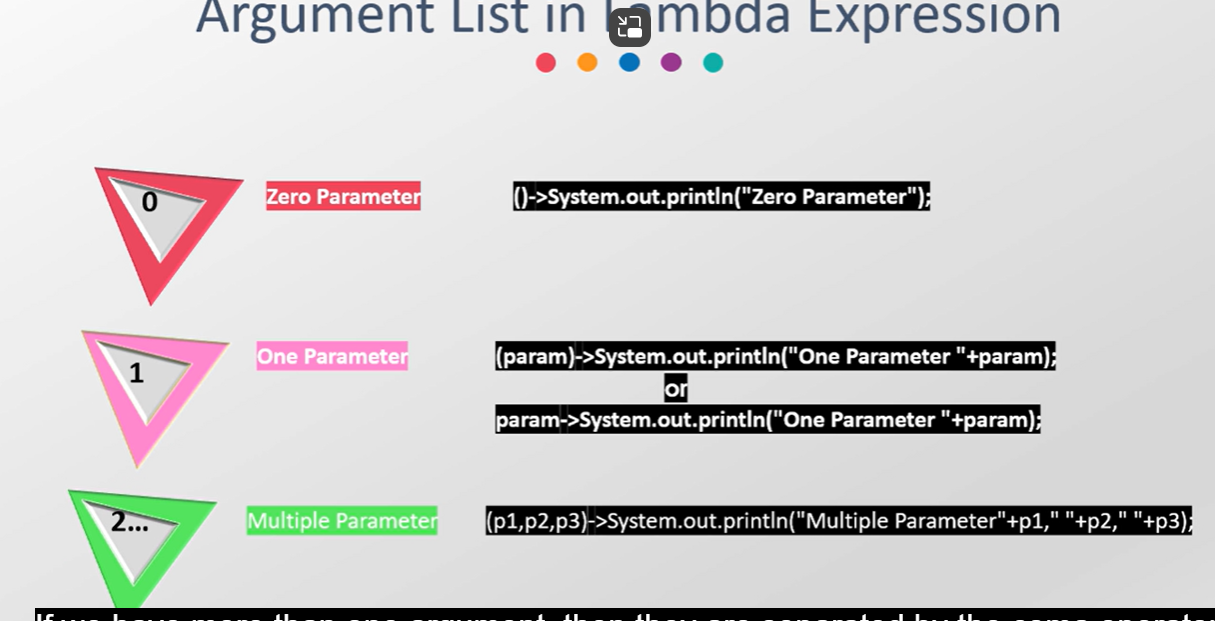
****

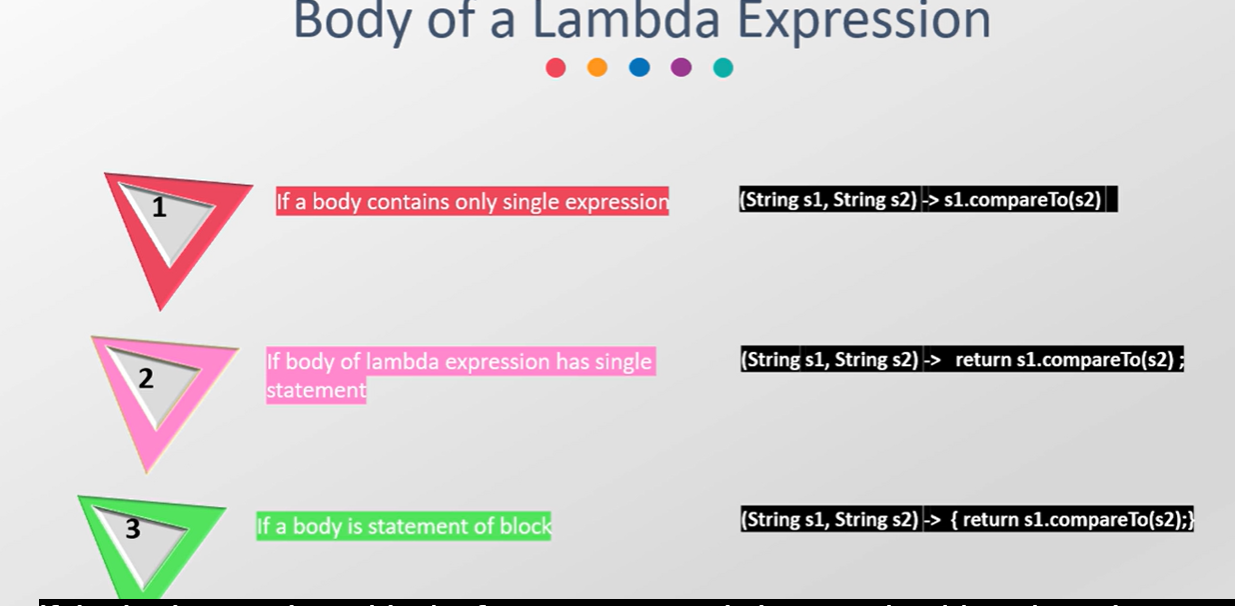
**Parameters**: These are the input values to the lambda expression (like method parameters to methods of interfaces).

A lambda expression can contain one or more arguments.

**Arrow** (->): Separates the parameters from the body of the lambda expression.

**Expression or Block:** The code that is executed when the lambda is invoked which can either be a single expression or a block of code.





Eg:

A computer screen shot of a code

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

Eg:

A screenshot of a computer program

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.



Example -2 :

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Example – 3:



A screenshot of a computer program

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

Where Do We Use Lambda Expressions?

Lambda expressions are commonly used with functional interfaces.

A functional interface is an interface that has only one abstract method (i.e., it can have many default or static methods, but only one abstract method).

Predefined functional interfaces – Function, Consumer, Supplier, Predicate

Common Examples of Functional Interfaces:

Runnable: A task that can be executed in a thread.

Comparator: Used to compare two objects.

Predicate: Used for testing conditions (returns true or false).

Function: Takes one argument and returns a result.

**Function (A Functional Interface)**

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

}

Function represents a function that takes an argument of type **T** and returns a result of type **R.** It’s widely used in Stream operations, like **map().**

**Predicate**

A Predicate is a functional interface that takes one argument and returns a Boolean(true or false).

import java.util.function.Predicate;

public class Main {

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(7)); // false

}

}

Explanation:

The lambda expression n -> n % 2 == 0 checks if the number n is even.

We used the test() method of the Predicate to check whether a number is even or not.

isEven now an **object** (technically an instance of an anonymous class implementing Predicate<Integer>) that holds this lambda logic.

**Consumer (A Functional Interface)**

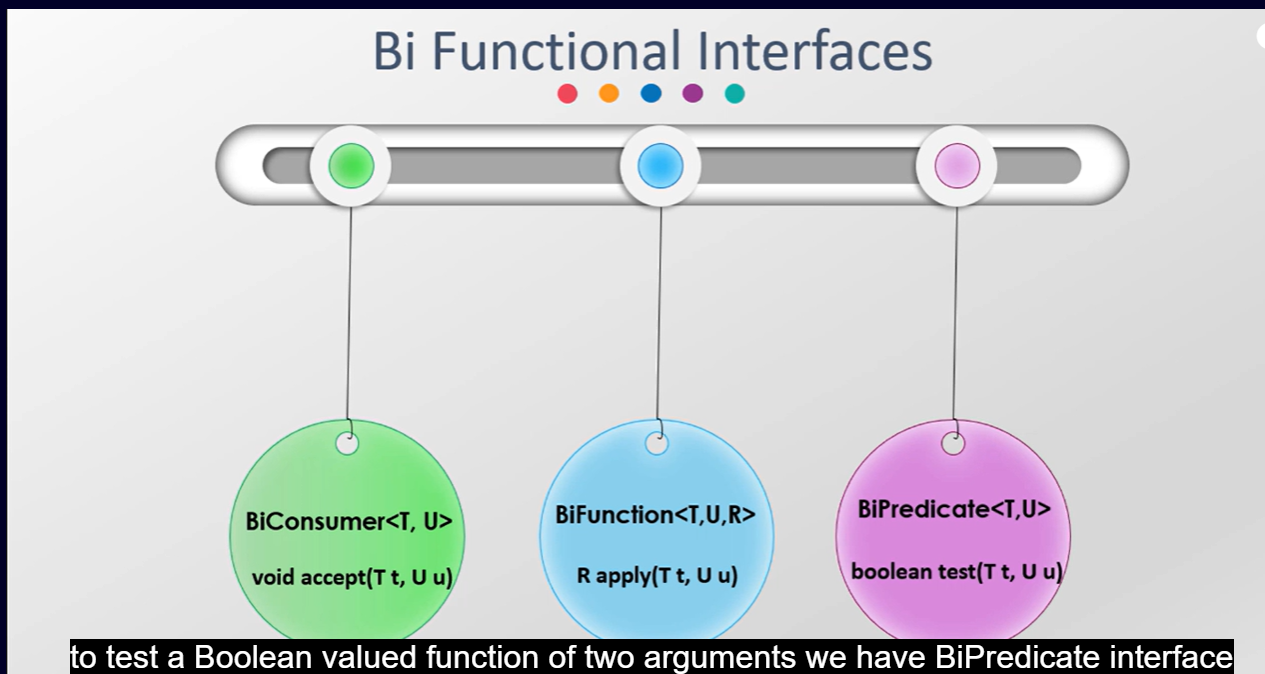
@FunctionalInterface

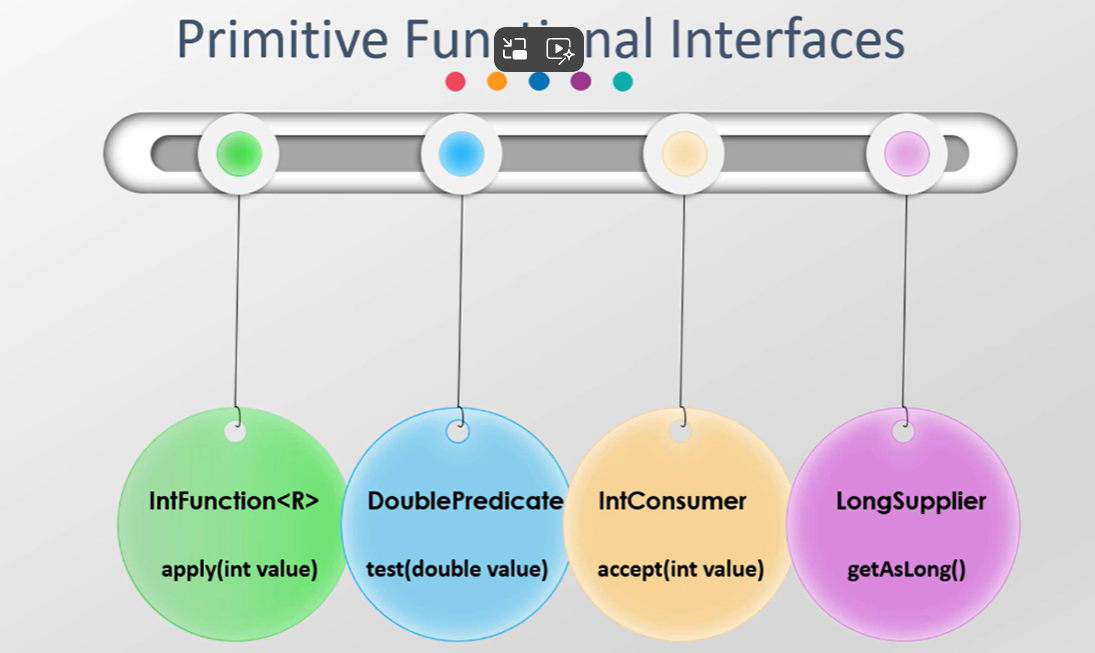
public interface Consumer<T> {

void accept(T t);

}

Consumer represents an operation that takes an argument of type **T** and returns nothing (**void**). It’s often used to consume elements from a collection.





Accessing on final variables inside lambda expression can cause compile time error.

Compiler uses an inner class to instantiate the functional interface.

**The Runnable interface**

It's a functional interface in Java that represents a task that can be executed by a thread. It has exactly one abstract method called run().

@FunctionalInterface

public interface Runnable {

void run();

}

So, when you write a lambda expression like this:

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

This matches the run() method of the Runnable interface, which has no parameters and a void return type. That’s why only statements with no parameters are allowed in this context.

Can We Use Other Lambda Expressions with Runnable?

Yes, you can use other types of lambda expressions in the Runnable context, but they need to conform to the method signature of run(). The run() method in the Runnable interface has no parameters and returns nothing (void).

Here are a few examples of other things you can do with Runnable lambda expressions:

1. More Complex Statements in Runnable

Instead of just printing "hi", you can include more complex logic within the lambda body. For example:

Runnable task = () -> {

int a = 5;

int b = 10;

int sum = a + b;

System.out.println("Sum of a and b: " + sum);

};

Here, we perform some calculations, store the result in sum, and then print the result. The logic inside the lambda body can be as complex as you want, as long as it doesn't change the method signature of the run() method (i.e., no parameters and void return type).

2. Passing Parameters to a Runnable

While the run() method in Runnable doesn’t accept parameters, you can pass parameters to a Runnable indirectly via a final or effectively final variable. This is a feature of Java’s lambda expressions: they can capture values from the enclosing scope.

Example:

public class Main {

public static void main(String[] args) {

int x = 5; // this variable must be final or effectively final

Runnable task = () -> System.out.println("Value of x: " + x);

Thread thread = new Thread(task);

thread.start();

}

}

Here, x is captured by the lambda and used inside the run() method. This works because x is effectively final (its value doesn’t change after it’s assigned).

**STREAMS**

A Collection is a data structure that holds elements, while a Stream is an abstraction that allows you to perform aggregate operations on the data.

**intermediate operations** - filter(), map(), sorted(), distinct() etc

Intermediate operations are those that return a new stream and can be chained. Examples include filter(), map(), sorted(), etc.

**terminal operations -** collect(), forEach(), reduce(), count().

Terminal operations trigger the processing of the stream and produce a result or a side-effect.

.count() is a terminal operation that returns how many elements are in the stream (long type).

.forEach() – returns nothing but print statements

.collect() – returns the collection which should be stored in a collection variable.

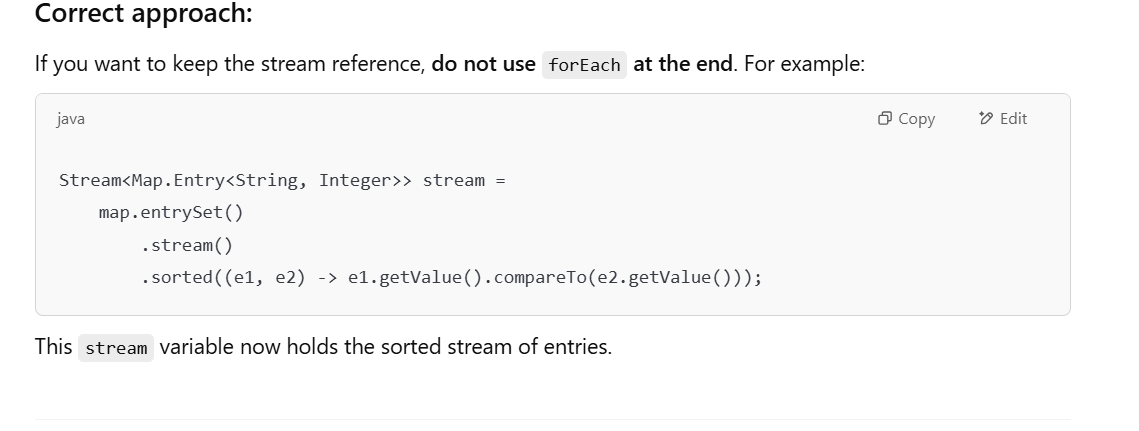
A screenshot of a computer

AI-generated content may be incorrect.

Below is Wrong way of doing :

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Would you like to try:

1. 🔢 **Map sorting** by keys or values?
2. 📊 **Summing or averaging** numbers with streams?
3. 📂 Grouping data (Collectors.groupingBy)?
4. 🤹‍♂️ Custom object streams (e.g., List<Employee>)?

**Ready for some advanced topics next?**

For example:

* Partitioning (Collectors.partitioningBy)
* Reducing or summarizing data (sum, max, min)
* Parallel streams for performance