Java Lambda Expressions

Lambda expressions in Java are a **feature introduced in Java 8** that allow you to write **anonymous methods** (methods without names) in a **concise way**, especially useful when working with **functional interfaces** and collections like `List`, `Map`, etc.

🔰 1. What is a Lambda Expression?

A **lambda expression** is a **short block of code** which takes in parameters and returns a value.

Syntax:

```
java

(parameter1, parameter2, ...) -> { body }
```

- `->` is called the **arrow token** or lambda operator.
- It is used to **implement a method of a functional interface**.

🔹 2. What is a Functional Interface?

A **functional interface** is an interface with **only one abstract method**.

```
java

@FunctionalInterface
interface MyInterface {
    void sayHello();  // One abstract method
}
```

✅ 3. Basic Example of Lambda

Example: Implementing a functional interface using Lambda

```
java

@FunctionalInterface
interface Greeting {
    void sayHello();
}

public class LambdaExample {
    public static void main(String[] args) {
        // Lambda Expression
        Greeting greet = () -> {
            System.out.println("Hello from Lambda!");
        };

        greet.sayHello();
```

```
    }
}
```

✅ Output:

```
csharp

Hello from Lambda!
```

🟢 Explanation:

- No parameters
- Implements **`sayHello()`** using **`() -> {}`** syntax.

🔹 4. Lambda with Parameters

```
java

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

public class LambdaCalc {
    public static void main(String[] args) {
        Calculator calc = (a, b) -> a + b;

        System.out.println("Sum = " + calc.add(10, 20));
    }
}
```

✅ Output:

```
ini

Sum = 30
```

🟢 Explanation:

- **`(a, b)`** are parameters
- **`a + b`** is the return value
- No need for return keyword if it's a single statement

🔹 5. Lambda with Collections (Java 8+)

```
java

import java.util.*;

public class LambdaList {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Koti", "Eswar", "Ravi");
```

```
        // Using lambda with forEach
        names.forEach(name -> System.out.println("Hello " + name));
    }
}
```

✅ Output:

```
nginx


Hello Koti
Hello Eswar
Hello Ravi
```

🔹 6. Lambda with Threads

Before Java 8:

```
java

Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Running thread");
    }
});
t.start();
```

With Lambda:

```
java

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

✅ Output:

```
cpp

Running thread
```

🔹 7. Lambda with Comparator

```
java

import java.util.*;

public class LambdaSort {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Zebra", "Apple", "Mango");

        // Sort using lambda
        Collections.sort(names, (s1, s2) -> s1.compareTo(s2));

        System.out.println(names); // [Apple, Mango, Zebra]
    }
}
```

🔚 Summary

Feature	Description
`()->{}`	Basic lambda syntax
Used With	Functional interfaces
Benefits	Less code, cleaner, readable
Common Use	**`Runnable`**, **`Comparator`**, Streams, forEach

🧪 Assignment for Practice

1. Create a lambda that multiplies two integers.
2. Use a lambda to print all even numbers in a list.
3. Use lambda to implement a thread that prints numbers from 1 to 5.

Would you like me to guide you through a **mini-project** using lambdas or go deeper into **Streams + Lambda**?

✅ Functional Interface in Java – Beginner Guide with Examples

🔹 What is a Functional Interface?

A **Functional Interface** is an interface that contains **exactly one abstract method**. It can have **any number of default or static methods**, but **only one abstract method**.

It is the foundation of **Lambda Expressions** in Java.

📌 Syntax:

```
java

@FunctionalInterface
interface MyInterface {
    void doSomething();
}
```

📝 The `@FunctionalInterface` annotation is **optional**, but it helps the compiler ensure that the interface has **only one abstract method**.

🔸 Why Functional Interface?

Functional interfaces enable **functional programming** in Java by letting you use **lambda expressions** as method arguments.

🔹 Examples

1. Simple Functional Interface

```
java

@FunctionalInterface
interface Greet {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greet g = () -> System.out.println("Hello, Functional Interface!");
        g.sayHello();
    }
}
```

✅ Output:

```
graphql

Hello, Functional Interface!
```

2. Functional Interface with Parameters

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

public class Calculator {
    public static void main(String[] args) {
        MathOperation add = (a, b) -> a + b;
        MathOperation multiply = (a, b) -> a * b;

        System.out.println("Sum: " + add.operate(10, 5));
        System.out.println("Product: " + multiply.operate(10, 5));
    }
}
```

✅ Output:

```makefile
Sum: 15
Product: 50
```

3. With Default and Static Methods

```java
@FunctionalInterface
interface Display {
    void show(String msg);

    // default method
    default void greet() {
        System.out.println("Welcome!");
    }

    // static method
    static void info() {
        System.out.println("This is a functional interface.");
    }
}

public class Example {
    public static void main(String[] args) {
        Display d = (msg) -> System.out.println("Message: " + msg);
        d.show("Hello");
        d.greet();          // default method
        Display.info();     // static method
    }
}
```

✅ Output:

```vbnet
Message: Hello
Welcome!
This is a functional interface.
```

🔹 Built-in Functional Interfaces in `java.util.function`

Java provides many ready-made functional interfaces:

Interface	Abstract Method	Description
`Predicate<T>`	**`test(T t)`**	Returns **`boolean`**, used for conditions
`Function<T,R>`	**`apply(T t)`**	Converts T to R
`Consumer<T>`	**`accept(T t)`**	Takes input, returns nothing
`Supplier<T>`	**`get()`**	Takes no input, returns value

✅ Example using `Predicate`

```
java

import java.util.function.Predicate;

public class TestPredicate {
    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(7));   // false
    }
}
```

🧠 Key Points

- Only **one abstract method**
- Can have **default** and **static methods**
- Supports **lambda expressions**
- Java has **many built-in** functional interfaces

🔪 Practice Assignments

1. Create a functional interface **`Printer`** that prints a custom message.
2. Write a lambda expression to find the square of a number using a functional interface.
3. Use **`Consumer<String>`** to print a welcome message.
4. Use **`Predicate<String>`** to check if a string is empty.

Would you like a **cheat sheet of all functional interfaces in `java.util.function`** or a **mini-project using functional interfaces and lambdas**?

✅ Java Streams API – Beginner Guide with Examples

The **Streams API**, introduced in **Java 8**, allows you to **process collections (like Lists, Sets)** in a **functional and declarative style** — making your code more **concise, readable, and powerful**.

🔹 What is a Stream?

A **Stream** is not a data structure. It's a **pipeline of data** from a source (like a `List`) that supports **aggregate operations** like `filter()`, `map()`, `sorted()`, `collect()` etc.

🔹 Why Use Streams?

Before Java 8:

```
java

List<String> list = Arrays.asList("A", "B", "C");
for (String s : list) {
    System.out.println(s);
}
```

With Streams:

```
java

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

🟢 **Less code**, more **readability** and **functional power**.

🔹 Basic Stream Workflow

```
scss

collection.stream()
          .filter(...)     // Intermediate operation
          .map(...)        // Intermediate operation
          .collect(...)    // Terminal operation
```

🔸 1. Creating a Stream

```
java

List<String> list = Arrays.asList("Java", "Python", "C++");

Stream<String> stream = list.stream();  // From List
```

🔸 2. `forEach()` – Terminal Operation

```
java

list.stream().forEach(lang -> System.out.println(lang));
```

✅ Output:

```
mathematica


Java
Python
C++
```

🔸 3. `filter()` – Intermediate Operation

```
java

List<String> names = Arrays.asList("Koti", "Ravi", "Eswar");

names.stream()
     .filter(name -> name.startsWith("E"))
     .forEach(System.out::println);
```

✅ Output:

```
nginx


Eswar
```

🔸 4. `map()` – Transform Each Element

```
java

List<String> names = Arrays.asList("koti", "ravi", "eswar");

names.stream()
     .map(name -> name.toUpperCase())
     .forEach(System.out::println);
```

✅ Output:

```
nginx


KOTI
RAVI
ESWAR
```

🔸 5. `collect()` – Gather Results

```
java

List<String> names = Arrays.asList("Java", "Python", "C");

List<String> filtered =
    names.stream()
         .filter(name -> name.length() > 3)
         .collect(Collectors.toList());
```

```
System.out.println(filtered);  // [Java, Python]
```

🔸 6. `sorted()` – Sort Stream

```
java

List<Integer> nums = Arrays.asList(4, 2, 9, 1);

nums.stream()
    .sorted()
    .forEach(System.out::println);
```

✅ Output:

```
1
2
4
9
```

🔸 7. `count()`, `min()`, `max()`

```
java

List<String> list = Arrays.asList("Apple", "Mango", "Banana");

long count = list.stream().count();              // Count
Optional<String> max = list.stream().max(String::compareTo);  // Max

System.out.println("Count: " + count);
System.out.println("Max: " + max.get());
```

🔸 8. Stream with `reduce()` – Aggregate Values

```
java

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

int sum = nums.stream()
              .reduce(0, (a, b) -> a + b);  // a is accumulator

System.out.println("Sum: " + sum);  // Sum: 10
```

🔸 9. Parallel Streams

```
java

List<String> data = Arrays.asList("A", "B", "C", "D");
```

```
data.parallelStream()
    .forEach(System.out::println);  // Parallel execution
```

📦 Built-in Collectors

Method	Description
`Collectors.toList()`	Collect into List
`Collectors.toSet()`	Collect into Set
`Collectors.joining(",")`	Join strings with comma
`Collectors.groupingBy()`	Group by key
`Collectors.counting()`	Count elements

🧠 Summary

Feature	Purpose
`stream()`	Start a stream from a collection
`filter()`	Filter elements by condition
`map()`	Transform each element
`collect()`	Final result into List, Set, etc.
`forEach()`	Loop through stream elements
`sorted()`	Sort the elements
`reduce()`	Aggregate into a single result
`parallelStream()`	Process in parallel

🧪 Mini Assignment

1. Create a list of numbers. Filter out even numbers, square them, and collect into a new list.
2. Convert all names in a list to uppercase using `map()`.
3. Count how many names start with "A".
4. Find the maximum number using `reduce()`.

Would you like me to guide you through a **Stream + Lambda mini project** (like employee filtering, data summarization, etc.)?

✅ Method Reference in Java – For Beginners (with Examples)

🔹 What is Method Reference?

Method reference is a shorthand **syntax for calling methods via lambda expressions.**
It helps make code **cleaner and more readable**.

🔸 It uses the `::` operator (double colon).

🔹 When to Use It?

When a **lambda expression just calls a method**, you can replace it with a method reference.

🔸 Syntax:

```
java

ClassName::methodName
```

🔹 Types of Method References

Type	Syntax Example	Used When...
1. Static method	`ClassName::staticMethod`	Calling a static method
2. Instance method of object	`object::instanceMethod`	Calling method on an object
3. Instance method of class (any object)	`ClassName::instanceMethod`	Used when lambda's first param is the caller
4. Constructor reference	`ClassName::new`	Used to create objects inside lambda

🔸 1. Static Method Reference

```
java

class Utils {
    public static void sayHello() {
        System.out.println("Hello!");
    }
}

public class Demo {
    public static void main(String[] args) {
        Runnable r = Utils::sayHello;  // method reference
        r.run();
    }
}
```

✅ Output:

```
Hello!
```

🔸 2. Instance Method Reference (specific object)

```java
class Printer {
    public void print(String msg) {
        System.out.println(msg);
    }
}

public class Demo {
    public static void main(String[] args) {
        Printer printer = new Printer();

        Consumer<String> c = printer::print;
        c.accept("Using method reference");
    }
}
```

✅ Output:

```sql
Using method reference
```

🔸 3. Instance Method Reference (class::instanceMethod)

```java
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Koti", "Ravi", "Eswar");

        // Using lambda:
        // names.forEach(name -> System.out.println(name));

        // Using method reference:
        names.forEach(System.out::println);
    }
}
```

✅ Output:

```nginx
Koti
Ravi
Eswar
```

🔸 4. Constructor Reference

```java
class Employee {
    Employee() {
        System.out.println("Employee created!");
    }
}

interface EmpFactory {
    Employee create();
}

public class Demo {
    public static void main(String[] args) {
        EmpFactory factory = Employee::new;  // Constructor reference
        Employee emp = factory.create();
    }
}
```

✅ Output:

```nginx
Employee created!
```

🔹 Real-World Example with Streams

```java
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("java", "python", "aws");

        // Convert all words to uppercase using method reference
        words.stream()
             .map(String::toUpperCase)
             .forEach(System.out::println);
    }
}
```

✅ Output:

```nginx
JAVA
PYTHON
AWS
```

✅ Summary

Type	Syntax	Example
Static method	**`Class::staticMethod`**	**`Math::abs`**
Instance (specific object)	**`object::method`**	**`printer::print`**
Instance (any object of class)	**`Class::method`**	**`String::toLowerCase`**

Type	Syntax	Example
Constructor	`Class::new`	`Employee::new`

🔪 Practice Tasks

1. Use `String::length` to get lengths of names.
2. Replace `list.forEach(name -> System.out.println(name))` with method reference.
3. Create a constructor reference to create a `Book` object.

Would you like me to give you a **mini project using all 4 method reference types**?
”