# Java 8

## Lambda Expression

### □ What is a Lambda Expression?

Lambda Expression is a new feature introduced in **Java 8** that allows us to implement methods of functional interfaces in a **clear and concise way** using an **anonymous function**.

It's primarily used to implement the abstract method of a **Functional Interface** (interface with a single abstract method).

---

### □ Why use Lambda Expression?

- To write **cleaner and concise code**
- Avoid boilerplate code while using **anonymous classes**
- Makes code **more readable**
- Enables **functional programming** style

---

### □ Syntax of Lambda Expression

(parameters) -> { body }

### □ Examples:

1. **Without parameters**:

   () -> System.out.println("Hello from Lambda!");

2. **With one parameter (no type needed)**:

   name -> System.out.println("Hello, " + name);

3. **With multiple parameters and return**:

   (a, b) -> {
   return a + b;

```
    }
```

---

**Traditional vs Lambda**

**Before Java 8 (Anonymous Class):**

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

**Using Lambda (Java 8+):**

```
Runnable r = () -> System.out.println("Thread is running...");
```

---

 **Functional Interface Example with Lambda**

```
@FunctionalInterface
interface MyFunctional {
   void sayHello();
}

public class LambdaDemo {
   public static void main(String[] args) {
      MyFunctional greeting = () -> System.out.println("Hello from Lambda
Expression!");
      greeting.sayHello();
   }
}
```

---

 **Use Cases of Lambda**

- Iterating collections using forEach
- Event handling in GUIs
- Functional-style programming
- Working with **Streams API**
- Multithreading (Runnable and Callable)

## Functional Interface

### □ What is a Functional Interface?

A **Functional Interface** is an interface that contains **only one abstract method**. It may contain **multiple default or static methods**, but only **one abstract method**.

Functional interfaces are the foundation of **lambda expressions** in Java.

---

### □ Key Features

- Marked with @FunctionalInterface annotation (optional, but recommended)
- Can be used as the **target type** for lambda expressions and method references
- Enhances support for **functional programming** in Java

---

### □ Syntax

```
@FunctionalInterface
interface MyInterface {
   void display(); // only one abstract method
}
```

---

### □ Example 1: Custom Functional Interface with Lambda

```
@FunctionalInterface
interface Greetable {
   void greet(String name);
}

public class FunctionalDemo {
   public static void main(String[] args) {
      Greetable g = (name) -> System.out.println("Hello, " + name + "!");
      g.greet("Prathamesh");
   }
}
```

---

**Built-in Functional Interfaces in java.util.function**

Java 8 provides several pre-defined functional interfaces in the java.util.function package:

| Interface | Description | Abstract Method |
|---|---|---|
| Predicate<T> | Returns a boolean | test(T t) |
| Function<T,R> | Takes input and returns result | apply(T t) |
| Consumer<T> | Takes input and returns nothing | accept(T t) |
| Supplier<T> | Takes nothing, returns a result | get() |
| BiFunction<T,U,R> | Takes two inputs, returns a result | apply(T t, U u) |

---

 **Example 2: Using Predicate Functional Interface**

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

public class PredicateDemo {
    public static void main(String[] args) {
        Predicate<String> isLong = str -> str.length() > 5;
        System.out.println(isLong.test("Prathamesh")); // true
    }
}
```

---

 **Example 3: Using Function**

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

public class FunctionDemo {
    public static void main(String[] args) {
        Function<Integer, Integer> square = n -> n * n;
        System.out.println(square.apply(4)); // 16
    }
}
```

---

### □ Real-world Use Cases

- Passing logic as data (functional programming)
- Writing cleaner **event-handling** code
- Working with **Streams API**
- Callback mechanisms
- Enhancing **testability** and **maintainability**

---

### □ Functional Interface vs Normal Interface

| Feature | Functional Interface | Normal Interface |
|---|---|---|
| Abstract Methods | Exactly one | One or more |
| Use in Lambda | Supported | Not suitable |
| Java 8 Annotation | @FunctionalInterface | Not applicable |

---

### □ Notes

- Using @FunctionalInterface gives a **compile-time error** if more than one abstract method is added.
- Functional interfaces can have:
  - **1 abstract method**
  - **Any number of default/static methods**

**Default Method in Interface**

### □ What is a Default Method in Java 8?

A **default method** is a method defined inside an interface with the keyword default, and it **has a method body**.

Introduced in **Java 8**, default methods allow developers to **add new functionalities to interfaces** without breaking the classes that already implement them.

---

### □ Why Default Methods?

Before Java 8, interfaces could not have method implementations. If a new method was added to an interface, all implementing classes had to override it — this broke backward compatibility.

□ Default methods solve this by providing a **default implementation** directly in the interface.

---

### □ Syntax

```
interface MyInterface {
   void show();  // abstract method

   default void display() {
      System.out.println("Default display method");
   }
}
```

---

### □ Example: Default Method Usage

```
interface Vehicle {
   void start();

   default void fuelType() {
      System.out.println("Default fuel type: Petrol");
   }
}

class Car implements Vehicle {
   public void start() {
      System.out.println("Car is starting...");
   }
}

public class DefaultMethodDemo {
   public static void main(String[] args) {
      Car myCar = new Car();
```

```
        myCar.start();
        myCar.fuelType();  // uses default method
    }
}
```

---

### □ Overriding Default Method

Default methods can be **overridden** by the implementing class:

```
class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting...");
    }

    public void fuelType() {
        System.out.println("Car uses Diesel");
    }
}
```

---

### □ Diamond Problem in Interfaces

If a class implements **multiple interfaces** with the **same default method**, the compiler will throw an error unless the class **overrides the method**.

**Example:**

```
interface A {
    default void show() {
        System.out.println("A's show");
    }
}

interface B {
    default void show() {
        System.out.println("B's show");
    }
}

class Test implements A, B {
```

```
    public void show() {
        System.out.println("Test's own show"); // required
    }
}
```

---

### □ Use Cases of Default Methods

- Add **backward-compatible enhancements** to interfaces
- Define **common behavior** shared by all implementers
- Useful in **API evolution** without breaking existing code

---

### □ Key Points

- Defined using the default keyword in an interface
- Must have a **method body**
- Can be **overridden** by the implementing class
- Helps solve **interface evolution** problems in Java 8
- Can cause ambiguity in **multiple inheritance**, which must be resolved

## Predicate Functional Interface

### □ What is Predicate in Java 8?

Predicate<T> is a **functional interface** introduced in java.util.function package. It represents a **boolean-valued function** of one argument.

It is commonly used for **filtering** or **evaluating conditions** on collections and streams.

---

### □ Key Purpose

- To **evaluate a condition** (true/false) based on input
- Used in **filtering, validation, matching**, etc.
- Returns true or false for the given input

---

### □ Method Signature

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

### □ Single Abstract Method:

```
boolean test(T t);
```

---

### □ Syntax Example

```
Predicate<String> isLong = str -> str.length() > 5;
```

---

### □ Example 1: Predicate with Strings

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

public class PredicateDemo {
   public static void main(String[] args) {
      Predicate<String> isLong = str -> str.length() > 5;

      System.out.println(isLong.test("Java"));       // false
      System.out.println(isLong.test("Prathamesh")); // true
   }
}
```

---

### □ Example 2: Predicate with Integer

```
Predicate<Integer> isEven = num -> num % 2 == 0;

System.out.println(isEven.test(10)); // true
System.out.println(isEven.test(7));  // false
```

---

### □ Predicate Chaining Methods

| Method | Description |
|---|---|
| and() | Combines two predicates using logical AND |
| or() | Combines two predicates using logical OR |
| negate() | Reverses the result of the predicate |

---

### □ Example 3: Chaining with and() and or()

```
Predicate<Integer> greaterThan5 = x -> x > 5;
Predicate<Integer> evenNumber = x -> x % 2 == 0;

Predicate<Integer> both = greaterThan5.and(evenNumber);
Predicate<Integer> either = greaterThan5.or(evenNumber);

System.out.println(both.test(8));   // true
System.out.println(either.test(3)); // false
```

---

### □ Example 4: Negate

```
Predicate<String> isEmpty = str -> str.isEmpty();

Predicate<String> isNotEmpty = isEmpty.negate();

System.out.println(isNotEmpty.test("Hello")); // true
System.out.println(isNotEmpty.test(""));      // false
```

---

### □ Use Cases of Predicate

- Filtering data using **Stream API**
- Input **validation**
- Conditional **execution**
- Creating **dynamic queries**

---

### □ Example: Predicate with Stream filter()

```
import java.util.*;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class StreamFilterPredicate {
   public static void main(String[] args) {
      List<String> names = Arrays.asList("Ram", "Shyam", "Ravi", "Sita");

      Predicate<String> startsWithR = name -> name.startsWith("R");

      List<String> filteredNames = names.stream()
                                        .filter(startsWithR)
                                        .collect(Collectors.toList());

      System.out.println(filteredNames); // [Ram, Ravi]
   }
}
```

---

**□ Summary Points**

- Predicate is a **functional interface** with boolean test(T t)
- Mainly used for **conditional logic**
- Works great with **Streams and Collections**
- Supports logical chaining with and(), or(), negate()
- Part of java.util.function package

**Function Functional Interface**

**□ What is Function in Java 8?**

Function<T, R> is a **functional interface** provided in the java.util.function package. It takes **one input** of type T and returns a **result** of type R.

---

**□ Purpose of Function Interface**

- Represents a **function**: input → output
- Useful for **data transformation**, **mapping**, or **computations**
- Frequently used in **Stream map() operations**

---

**□ Method Signature**

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);
}
```

**□ Single Abstract Method:**

```
R apply(T t);
```

---

**□ Syntax Example**

```
Function<Integer, String> convertToString = i -> "Number: " + i;
```

---

**□ Example 1: Function with Integer Input and String Output**

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

public class FunctionDemo {
   public static void main(String[] args) {
      Function<Integer, String> numberToString = num -> "Number is: " + num;

      System.out.println(numberToString.apply(10));
      // Output: Number is: 10
   }
}
```

---

**□ Example 2: Square of a Number**

```
Function<Integer, Integer> square = x -> x * x;

System.out.println(square.apply(5)); // Output: 25
```

---

**□ Function Chaining Methods**

| Method | Description |
|---|---|
| andThen(Function after) | First apply current function, then after |
| compose(Function before) | First apply before, then current |

---

### □ Example 3: Using andThen()

```
Function<Integer, Integer> multiplyBy2 = x -> x * 2;
Function<Integer, Integer> add3 = x -> x + 3;

Function<Integer, Integer> combined = multiplyBy2.andThen(add3);

System.out.println(combined.apply(5)); // (5 * 2) + 3 = 13
```

---

### □ Example 4: Using compose()

```
Function<Integer, Integer> multiplyBy2 = x -> x * 2;
Function<Integer, Integer> add3 = x -> x + 3;

Function<Integer, Integer> combined = multiplyBy2.compose(add3);

System.out.println(combined.apply(5)); // (5 + 3) * 2 = 16
```

---

### □ Use Cases of Function

- Used in **Stream's map()** for transforming data
- Data **conversion**, **formatting**, and **processing**
- Custom **logic pipelines** using chaining
- UI transformations or form validations

---

### □ Example: Function with Stream map()

```
import java.util.*;
import java.util.function.Function;
import java.util.stream.Collectors;
```

```
public class StreamFunctionExample {
   public static void main(String[] args) {
      List<String> names = Arrays.asList("ram", "shyam", "sita");

      Function<String, String> toUpper = s -> s.toUpperCase();

      List<String> upperNames = names.stream()
                              .map(toUpper)
                              .collect(Collectors.toList());

      System.out.println(upperNames); // [RAM, SHYAM, SITA]
   }
}
```

---

**Summary Points**

- Function<T, R> takes input T and returns output R
- Main method: apply(T t)
- Can chain functions using compose() and andThen()
- Great for **transforming**, **mapping**, and **computing** values
- Frequently used in **Streams and data pipelines**

**Stream API**

**What is Stream API?**

**Stream API** in Java 8 is used to **process collections of objects** in a functional style. It allows performing **filtering, mapping, sorting, and reducing** operations in a clean, declarative manner without modifying the original data source.

---

**Key Features**

- Supports **sequential** and **parallel** operations
- Doesn't change the original collection (non-destructive)
- Offers **lazy evaluation** and **pipelining**
- Works well with **lambda expressions**

---

## □ Stream vs Collection

| Feature | Collection | Stream |
|---|---|---|
| Data structure | Stores data | Processes data |
| Consumption | Can be reused | Can be used once |
| Evaluation | Eager | Lazy |
| Operations | External iteration | Internal iteration |
| Modifies data | Yes | No (immutable) |

---

## □ How to Create Streams?

### 1. From Collection:

```
List<Integer> list = Arrays.asList(1, 2, 3);
Stream<Integer> stream = list.stream();
```

### 2. From Arrays:

```
int[] arr = {1, 2, 3};
IntStream intStream = Arrays.stream(arr);
```

### 3. Using Stream.of():

```
Stream<String> stream = Stream.of("A", "B", "C");
```

---

## □ Core Stream Operations

| Type | Method Examples | Description |
|---|---|---|
| Intermediate | filter(), map(), sorted() | Return a new stream |
| Terminal | collect(), forEach(), count() | Trigger processing and close stream |

---

## □ Example: Stream Pipeline

```
List<String> names = Arrays.asList("Ram", "Shyam", "Ravi", "Sita");
```

```
names.stream()
    .filter(name -> name.startsWith("R"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

**Output:**

```
RAM
RAVI
```

---

**Common Stream Methods**

 **filter(Predicate)**

Filters elements based on condition.

```
list.stream().filter(x -> x > 10);
```

 **map(Function)**

Transforms each element.

```
list.stream().map(x -> x * x);
```

 **sorted()**

Sorts elements.

```
list.stream().sorted();
```

 **forEach(Consumer)**

Performs action on each element.

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

 **collect(Collectors.toList())**

Collects stream into a list.

```
List<Integer> even = list.stream()
                .filter(x -> x % 2 == 0)
                .collect(Collectors.toList());
```

---

**Terminal Methods – Reduce, Count, etc.**

 **reduce()**

Reduces elements to a single value.

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

 **count()**

Counts number of elements.

```
long count = list.stream().filter(x -> x > 10).count();
```

---

 **Parallel Stream**

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

Used for **multithreaded parallel processing**, improves performance on large datasets.

---

 **Use Cases of Stream API**

- Filtering and transformation
- Aggregation (sum, max, min)
- Data analytics and processing
- Working with big data (with parallel streams)
- Clean pipelines instead of nested loops

## 1. Method References

### □ What is it?

Method reference is a shorthand notation for a lambda expression that only calls an existing method.

### □ Syntax:

```
ClassName::staticMethod
object::instanceMethod
ClassName::new  // constructor reference
```

### □ Example 1: Static Method Reference

```
class MyUtil {
    public static void greet() {
        System.out.println("Hello from static method!");
    }
}

Runnable r = MyUtil::greet;
r.run();
```

### □ Example 2: Instance Method Reference

```
String str = "hello";
Supplier<Integer> supplier = str::length;
System.out.println(supplier.get()); // 5
```

### □ Example 3: Constructor Reference

```
Supplier<List<String>> listSupplier = ArrayList::new;
List<String> list = listSupplier.get();
```

---

## 2. Optional Class

### □ What is it?

Optional is a container object used to contain not-null or possibly-null values.

 **Common Methods:**

- Optional.of(value)
- Optional.ofNullable(value)
- isPresent()
- ifPresent(Consumer)
- get()
- orElse(defaultValue)

 **Example:**

```
Optional<String> name = Optional.of("Prathamesh");
name.ifPresent(System.out::println); // prints "Prathamesh"
```

 **Handling null safely:**

```
String value = null;
String result = Optional.ofNullable(value).orElse("Default");
System.out.println(result); // "Default"
```

---

### 3. Static Methods in Interfaces

 **What is it?**

Java 8 allows interfaces to define **static methods** with implementation.

 **Example:**

```
interface MathUtil {
   static int square(int x) {
      return x * x;
   }
}

int result = MathUtil.square(4); // 16
```

---

### 4. Extended Built-in Functional Interfaces

| Interface | Description | Method |
|---|---|---|
| BiPredicate<T,U> | Returns boolean for two inputs | test(T, U) |
| BiFunction<T,U,R> | Takes two inputs, returns a result | apply(T,U) |
| BiConsumer<T,U> | Takes two inputs, returns nothing | accept(T,U) |
| UnaryOperator<T> | Same input and output type | apply(T) |
| BinaryOperator<T> | Takes two T inputs, returns T | apply(T,T) |

 **Example: BiPredicate**

```
BiPredicate<String, String> equals = String::equalsIgnoreCase;
System.out.println(equals.test("Java", "java")); // true
```

---

### 5. Collectors Class

 **What is it?**

Collectors is a utility class to accumulate elements from streams into collections.

 **Common Methods:**

- toList(), toSet(), toMap()
- joining()
- groupingBy()
- partitioningBy()
- counting()
- summarizingInt()

 **Example: Joining**

```
List<String> names = Arrays.asList("Ram", "Sita", "Shyam");
String result = names.stream().collect(Collectors.joining(", "));
System.out.println(result); // Ram, Sita, Shyam
```

 **Example: GroupingBy**

```
Map<Integer, List<String>> grouped = names.stream()
```

```
    .collect(Collectors.groupingBy(String::length));
```

---

## 6. Date and Time API (java.time)

### □ What is it?

A modern, immutable, thread-safe date/time API introduced in Java 8.

### □ Key Classes:

- LocalDate
- LocalTime
- LocalDateTime
- Period, Duration
- DateTimeFormatter

### □ Example: Current Date and Time

```
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
LocalDateTime dateTime = LocalDateTime.now();
```

### □ Example: Formatting

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
String formatted = date.format(formatter);
```

---

## 7. Stream Advanced Operations

### □ Advanced Methods:

- flatMap() – flatten nested collections
- distinct() – removes duplicates
- limit(n) – limits to first n elements
- skip(n) – skips first n elements
- peek() – debugging intermediate stage
- anyMatch(), allMatch(), noneMatch() – match conditions

---

### □ Example: flatMap()

```
List<List<String>> data = Arrays.asList(
   Arrays.asList("a", "b"),
   Arrays.asList("c", "d")
);

List<String> flatList = data.stream()
   .flatMap(List::stream)
   .collect(Collectors.toList());

// Output: [a, b, c, d]
```

---

### □ Example: Matching

```
List<Integer> list = Arrays.asList(2, 4, 6, 8);

boolean allEven = list.stream().allMatch(x -> x % 2 == 0); // true
boolean anyGreaterThanFive = list.stream().anyMatch(x -> x > 5); // true
boolean noneOdd = list.stream().noneMatch(x -> x % 2 != 0); // true
```