Java 8 Features with Complete Example

# Lambda Expressions

A **lambda expression** in Java is a concise way to represent an **anonymous function**—a function without a name that can be passed around as a value.

**✅ Example**

*List names = Arrays.asList("Alice", "Bob", "Charlie");*

*names.forEach(name -> System.out.println(name));*

name -> System.out.println(name) is a lambda expression.

It takes one parameter (name) and prints it.

It's passed to the forEach method to apply the action to each element.

**Why Use Lambda Expressions?**

Lambda expressions are used primarily to:

Simplify code

Make it more readable

Enable functional programming features in Java

Replace anonymous inner classes

Lambda expressions enable you to treat functionality as a method argument or pass code as data.

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class LambdaExample {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.asList(10, 25, 30, 45, 60);  
 List<Integer> filtered = numbers.stream()  
 .filter(n -> n > 30)  
 .collect(Collectors.toList());  
 filtered.forEach(n -> System.out.println("Filtered: " + n));  
 }  
}

# 2. Functional Interface

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

@FunctionalInterface  
interface Calculator {  
 int operation(int a, int b);  
}  
  
public class FunctionalInterfaceExample {  
 public static void main(String[] args) {  
 Calculator add = (a, b) -> a + b;  
 System.out.println("Sum: " + add.operation(5, 3));  
 }  
}

*@FunctionalInterface*

*interface MyFunction {*

*void apply(String input);*

*}*

*MyFunction func = (input) -> System.out.println("Hello " + input);*

*func.apply("World");  // Output: Hello World*

**✅ Why Is It Called "Functional"?**

Because it can be used as the **target for a lambda expression** or a method reference—just like functions in functional programming.

The @FunctionalInterface annotation is optional but recommended.

It ensures the interface has only **one abstract method**.

You can now use a lambda expression to implement it:

🧩 Common Functional Interfaces in Java (from java.util.function)

| Interface | Abstract Method | Description |
| --- | --- | --- |
| Predicate<T> | test(T t) | Returns true or false |
| Function<T,R> | apply(T t) | Converts T to R |
| Consumer<T> | accept(T t) | Performs an action on T |
| Supplier<T> | get() | Supplies a value of type T |
| UnaryOperator<T> | apply(T t) | Operates on a single operand |
| BinaryOperator<T> | apply(T t1, T t2) | Operates on two operands |

# Java Functional Interfaces types:

## Predicate<T>

Method: test(T t)  
Returns true or false

Example:

Predicate<Integer> isPositive = n -> n > 0;  
System.out.println(isPositive.test(5)); // true

## Function<T,R>

Method: apply(T t)  
Converts T to R

Example:

Function<String, Integer> stringLength = s -> s.length();  
System.out.println(stringLength.apply("Hello")); // 5

## Consumer<T>

Method: accept(T t)  
Performs an action on T

Example:

Consumer<String> greeter = name -> System.out.println("Hello " + name);  
greeter.accept("Alice"); // Hello Alice

## Supplier<T>

Method: get()  
Supplies a value of type T

Example:

Supplier<Double> randomValue = () -> Math.random();  
System.out.println(randomValue.get()); // e.g., 0.7234

## UnaryOperator<T>

Method: apply(T t)  
Operates on a single operand

Example:

UnaryOperator<Integer> square = n -> n \* n;  
System.out.println(square.apply(4)); // 16

## BinaryOperator<T>

Method: apply(T t1, T t2)  
Operates on two operands

Example:

BinaryOperator<Integer> add = (a, b) -> a + b;  
System.out.println(add.apply(3, 7)); // 10

# 3. Stream API

## Introduction

The Java Stream API, introduced in Java 8, provides a powerful way to process collections of data in a functional style. It allows for operations such as filtering, mapping, and reducing data using a pipeline of operations.

## Stream Creation Methods

Streams can be created from various data sources such as collections, arrays, or generated dynamically. Common methods include:

# Java Stream Creation Methods with Examples

## 1. Collection.stream()

Creates a sequential stream from a collection.

Example:  
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");  
names.stream()  
 .filter(name -> name.startsWith("A"))  
 .forEach(System.out::println);

## 2. Collection.parallelStream()

Creates a parallel stream from a collection.

Example:  
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);  
long count = numbers.parallelStream()  
 .filter(n -> n % 2 == 0)  
 .count();  
System.out.println("Even numbers count: " + count);

## 3. Stream.of(...)

Creates a stream from specified values.

Example:  
Stream<String> stream = Stream.of("Java", "Python", "C++");  
stream.forEach(System.out::println);

## 4. Arrays.stream(array)

Creates a stream from an array.

Example:  
int[] numbers = {1, 2, 3, 4, 5};  
int sum = Arrays.stream(numbers)  
 .sum();  
System.out.println("Sum: " + sum);

## Intermediate Operations

Intermediate operations return a new stream and are used to transform or filter elements. Common intermediate operations include:

### filter

Filters elements based on a predicate.

Example:

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

numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);

### map

Transforms each element using a function.

Example:

List<String> names = Arrays.asList("Alice", "Bob");

names.stream().map(String::toUpperCase).forEach(System.out::println);

### sorted

Sorts the elements of the stream.

Example:

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

numbers.stream().sorted().forEach(System.out::println);

## Terminal Operations

Terminal operations produce a result or a side-effect and mark the end of the stream pipeline. Common terminal operations include:

### forEach

Performs an action for each element.

Example:

List<String> names = Arrays.asList("Alice", "Bob");

names.stream().forEach(System.out::println);

### collect

Collects the elements into a collection or other data structure.

Example:

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

### reduce

Combines elements to produce a single result.

Example:

int sum = numbers.stream().reduce(0, Integer::sum);

Stream API enables functional-style operations on collections.

import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIExample {  
 public static void main(String[] args) {  
 List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");  
 names.stream()  
 .filter(name -> name.startsWith("C"))  
 .forEach(System.out::println);  
 }  
}

# 4. Default and Static Methods in Interfaces

Interfaces can have default and static methods with implementations.

interface Vehicle {  
 default void start() {  
 System.out.println("Vehicle is starting...");  
 }  
 static void stop() {  
 System.out.println("Vehicle is stopping...");  
 }  
}  
  
class Car implements Vehicle {}  
  
public class InterfaceDefaultStaticExample {  
 public static void main(String[] args) {  
 Car car = new Car();  
 car.start();  
 Vehicle.stop();  
 }  
}

# 5. Method References

# Method References in Java

Method references in Java provide a shorthand syntax for calling methods directly using a reference. They are a more concise way to write lambda expressions when the method being called already exists.

## Syntax

ClassName::methodName

## Types of Method References

### 1. Static Method Reference

Syntax: ClassName::staticMethodName  
Example:  
Function<String, Integer> parseInt = Integer::parseInt;  
System.out.println(parseInt.apply("123")); // Output: 123

### 2. Instance Method Reference of a Particular Object

Syntax: instance::instanceMethodName  
Example:  
String str = "hello";  
Supplier<String> supplier = str::toUpperCase;  
System.out.println(supplier.get()); // Output: HELLO

### 3. Instance Method Reference of an Arbitrary Object of a Particular Type

Syntax: ClassName::instanceMethodName  
Example:  
List<String> names = Arrays.asList("alice", "bob", "charlie");  
names.forEach(String::toUpperCase);

### 4. Constructor Reference

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

Method references are a shorthand for calling a method via a lambda expression.

import java.util.Arrays;  
import java.util.List;  
  
public class MethodReferenceExample {  
 public static void main(String[] args) {  
 List<String> names = Arrays.asList("Java", "Python", "C++");  
 names.forEach(System.out::println);  
 }  
}

# 6. Optional

# Java Optional Class

The Optional class in Java is a container object which may or may not contain a non-null value. It is used to avoid null checks and NullPointerExceptions by providing a more expressive way to handle optional values.

## Purpose

Optional is primarily used to represent the presence or absence of a value. It helps in writing cleaner code by avoiding explicit null checks and encourages functional-style programming.

## Common Methods

### 1. isPresent()

Returns true if the value is present, otherwise false.

Example:  
Optional<String> name = Optional.of("Alice");  
if (name.isPresent()) {  
 System.out.println("Name is present");  
}

### 2. get()

Returns the value if present, otherwise throws NoSuchElementException.

Example:  
Optional<String> name = Optional.of("Alice");  
String value = name.get();  
System.out.println(value);

### 3. orElse()

Returns the value if present, otherwise returns the specified default value.

Example:  
Optional<String> name = Optional.empty();  
String value = name.orElse("Default Name");  
System.out.println(value);

### 4. ifPresent()

Performs the given action if the value is present.

Example:  
Optional<String> name = Optional.of("Alice");  
name.ifPresent(n -> System.out.println("Name: " + n))

Optional is used to avoid null checks and prevent NullPointerExceptions.

import java.util.Optional;  
  
public class OptionalExample {  
 public static void main(String[] args) {  
 Optional<String> name = Optional.ofNullable("John");  
 name.ifPresent(n -> System.out.println("Name: " + n));  
  
 Optional<String> empty = Optional.empty();  
 System.out.println("Is present: " + empty.isPresent());  
 }  
}

# 7. New Date and Time API

Java 8 introduced a new, immutable, and thread-safe date and time API.

import java.time.LocalDate;  
import java.time.LocalTime;  
import java.time.LocalDateTime;  
  
public class DateTimeExample {  
 public static void main(String[] args) {  
 LocalDate date = LocalDate.now();  
 LocalTime time = LocalTime.now();  
 LocalDateTime dateTime = LocalDateTime.now();  
  
 System.out.println("Date: " + date);  
 System.out.println("Time: " + time);  
 System.out.println("DateTime: " + dateTime);  
 }  
}

# 8. Collectors

Following is the generic syntax to use a stream

<<collection-instance>>.stream().<<non-terminal-operation()>>.<<non-terminal-operation()>>.<<terminal-operation()>>

Collectors are used to gather the results of stream operations into collections.

import java.util.Arrays;  
import java.util.List;  
import java.util.Map;  
import java.util.stream.Collectors;  
  
class Employee {  
 String name;  
 String department;  
  
 Employee(String name, String department) {  
 this.name = name;  
 this.department = department;  
 }  
  
 String getDepartment() {  
 return department;  
 }  
  
 String getName() {  
 return name;  
 }  
}  
  
public class CollectorsExample {  
 public static void main(String[] args) {  
 List<Employee> employees = Arrays.asList(  
 new Employee("Alice", "HR"),  
 new Employee("Bob", "IT"),  
 new Employee("Charlie", "HR")  
 );  
  
 Map<String, List<Employee>> grouped = employees.stream()  
 .collect(Collectors.groupingBy(Employee::getDepartment));  
  
 grouped.forEach((dept, emps) -> {  
 System.out.println(dept + ": " + emps.stream().map(Employee::getName).collect(Collectors.joining(", ")));  
 });  
 }  
}

**employees.stream()**Converts the employees list into a Stream so we can perform functional operations on it.

**Collectors.groupingBy(Employee::getDepartment)**This is a collector that groups the employees by their department.

**Employee::getDepartment** is a method reference that tells Java to use the getDepartment() method of each Employee object to determine the grouping key.

The result is a Map<String, List<Employee>> where:

The key is the department name (String)

The value is a list of employees (List<Employee>) in that department.

grouped.forEach((dept, emps) -> { ... })  
Iterates over each entry in the grouped map:

dept is the department name (key)

emps is the list of employees in that department (value)

**emps.stream()**Converts the list of employees into a stream.

**.map(Employee::getName)**  
Transforms each Employee object into just their name (String).

**.collect(Collectors.joining(", "))**  
Joins all the employee names into a single string, separated by commas.

# 9. Nashorn JavaScript Engine

Nashorn allows execution of JavaScript code within Java (deprecated in Java 11 and removed in Java 15).

import javax.script.ScriptEngine;  
import javax.script.ScriptEngineManager;  
import javax.script.ScriptException;  
  
public class NashornExample {  
 public static void main(String[] args) throws ScriptException {  
 ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");  
 engine.eval("print('Hello from JavaScript')");  
 }  
}

# 10. Parallel Streams

⚡ What Is a Parallel Stream?

A **parallel stream** is a stream that **splits the data into multiple chunks**, processes them **in parallel using multiple threads**, and then **combines the results**.

This is useful for improving performance on large datasets, especially on multi-core processors.

How It Works Internally

**Splitting the Data**:

The stream is divided into **substreams** using a Spliterator.Each substream can be processed independently.

**Processing in Parallel**:

Java uses the **Fork/Join framework** (introduced in Java 7) to process substreams in parallel.

Each substream is handled by a different thread from the **common ForkJoinPool**.

**Combining the Results**:

After processing, the results from all threads are **merged** back together.

Example

The list is split into chunks like [10, 20], [30, 40], [50, 60].

Each chunk is processed in a separate thread.

The filter n > 25 is applied in parallel.

The count is aggregated from all threads.

🧩 Key Differences: stream() vs parallelStream()

| Feature | stream() (Sequential) | parallelStream() (Parallel) |
| --- | --- | --- |
| Execution | Single-threaded | Multi-threaded |
| Performance | Slower on large data | Faster on large data (if CPU-bound) |
| Thread Safety | Not an issue | Must ensure thread safety |
| Order Preservation | Preserved | Not guaranteed |