Java 8 Features

**🔹 Step 1: What is a Functional Interface?**

A **Functional Interface** is an interface that has **exactly one abstract method**.

* It can have any number of **default** and **static** methods.
* It is also called **SAM (Single Abstract Method) interface**.
* Lambda expressions can be used to represent the implementation of that single abstract method.

Example:

@FunctionalInterface

interface MyFunctionalInterface {

void sayMessage(String message); // only one abstract method

}

**🔹 Step 2: Why Functional Interfaces?**

Before Java 8, if we wanted to pass a block of code, we needed to create an **anonymous inner class**.

* This was lengthy and not readable.
* With functional interfaces + lambda expressions, we can pass **behavior as a method argument** in a concise way.

**🔹 Step 3: Real-time Example**

Suppose you want to run some task in a separate thread.

👉 Without Lambda (Before Java 8):

public class Demo {

public static void main(String[] args) {

Runnable r = new Runnable() {

@Override

public void run() {

System.out.println("Task executed without Lambda");

}

};

new Thread(r).start();

}

Output

Task executed with out Lambda

👉 With Lambda (Java 8):

public class Demo {

public static void main(String[] args) {

Runnable r = () -> System.out.println("Task executed with Lambda");

new Thread(r).start();

}

}

Output

Task executed with Lambda

Here, Runnable is a functional interface because it has only one abstract method: run().

**🔹 Step 4: Inbuilt Functional Interfaces in Java 8**

Java 8 introduced many ready-made functional interfaces inside java.util.function package:

* Predicate<T> → returns boolean
* Function<T, R> → takes input T and returns output R
* Consumer<T> → takes input T but returns nothing
* Supplier<T> → returns a value of type T, takes no input

## Predicate<T>

* **Definition**: Takes one input, returns a boolean.
* **Usage**: To test a condition (like filter logic).

📌 Example:

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

Predicate<Integer> isAdult = age -> age >= 18;

System.out.println(isAdult.test(20)); // true

System.out.println(isAdult.test(15)); // false

}

}

Output

true

false

👉 **Real-time use**: Check if a user is eligible for voting, filter employees based on salary, etc.

## 🔹 2. Function<T, R>

* **Definition**: Takes one input of type T, returns output of type R.
* **Usage**: For transformations (like converting one type to another).

📌 Example:

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) {

Function<String, Integer> strLength = str -> str.length();

System.out.println(strLength.apply("Prashanth")); // 9

System.out.println(strLength.apply("Java")); // 4

}

}

Output

9

4

👉 **Real-time use**: Convert Employee object to Employee Name, calculate tax from salary, etc.

## 🔹 3. Consumer<T>

* **Definition**: Takes one input, returns nothing (void).
* **Usage**: To perform some action (like printing, saving, logging).

📌 Example:

import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

Consumer<String> printMessage = msg -> System.out.println("Message: " + msg);

printMessage.accept("Hello Prashanth!");

printMessage.accept("Java 8 is powerful!");

}

}

Output

Message: Hello Prashanth!

Message: Java 8 is powerful!

👉 **Real-time use**: Printing reports, sending notifications, logging messages.

## 🔹 4. Supplier<T>

* **Definition**: Takes no input, returns a value.
* **Usage**: To supply values whenever required.

📌 Example:

import java.util.function.Supplier;

import java.util.Random;

public class SupplierExample {

public static void main(String[] args) {

Supplier<Integer> randomNumber = () -> new Random().nextInt(100);

System.out.println(randomNumber.get()); // random number

System.out.println(randomNumber.get());

}

}

Output

96

56

👉 **Real-time use**: Generating OTP, default values, timestamps, etc.

# Streams in Java 8

### 1. What is a Stream?

* A **Stream** is a sequence of data (like elements in a collection) that can be processed using functional-style operations.
* It does **not store data**, but it **operates on data from a collection, array, or I/O channels**.

📌 Example (basic stream usage):

import java.util.\*;

import java.util.stream.\*;

public class StreamDemo {

public static void main(String[] args) {

List<String> names = Arrays.asList("Prashanth", "Ravi", "Sita", "Anu");

names.stream()

.filter(name -> name.startsWith("P")) // Intermediate

.map(String::toUpperCase) // Intermediate

.forEach(System.out::println); // Terminal

}

}

👉 Output: PRASHANTH

### 2. Stream Pipeline

A **Stream pipeline** has three parts:

1. **Source** → Where the data comes from (Collection, Array, File, etc.)
2. **Intermediate Operations** → Transformations (filter, map, sorted, distinct, etc.)
3. **Terminal Operations** → Final result (collect, forEach, reduce, count, etc.)

📌 Diagram (conceptual):

Collection / Array → Stream()

|

|

Intermediate Ops (filter → map → sorted …)

|

|

Terminal Op (forEach / collect / reduce)

### 3. Intermediate Operations

These are **lazy** (executed only when a terminal operation is called).

* filter(Predicate) → filter elements
* map(Function) → transform elements
* sorted() → sort elements
* distinct() → remove duplicates
* limit(n) → take first n elements
* skip(n) → skip first n elements

### 4. Terminal Operations

These **trigger execution** of the stream.

* forEach(Consumer) → perform action
* collect(Collectors.toList()) → gather result
* count() → count elements
* reduce() → reduce elements to a single value
* min(), max(), findFirst(), findAny()

### 5. Real-Time Example

👉 Employee list: filter employees by salary > 50k, get names in uppercase, and print.

import java.util.\*;

import java.util.stream.\*;

class Employee {

String name;

int salary;

Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

}

public class EmployeeStream {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Prashanth", 60000),

new Employee("Ravi", 40000),

new Employee("Raj", 70000),

new Employee("Sita", 30000)

);

employees.stream()

.filter(emp -> emp.salary > 50000) // Predicate

.map(emp -> emp.name.toUpperCase()) // Function

.forEach(System.out::println); // Consumer

}

}

👉 Output:

PRASHANTH

RAJ

### 6. Parallel Streams

* Streams can also be run in **parallel** to take advantage of multi-core CPUs.

List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);

numbers.parallelStream()

.forEach(n -> System.out.println(n + " - " + Thread.currentThread().getName()));

👉 Multiple threads will process data at the same time.

# Method References in Java 8

### 1. What are Method References?

* A **Method Reference** is a short hand way of writing a lambda expression that calls an existing method.
* Instead of writing () -> someMethod(), we can directly write ClassName::methodName.

👉 It improves **readability** and **reduces code size**.

### 2. Types of Method References

There are **four types**:

#### (a) Reference to a static method

Syntax → ClassName::staticMethodName

📌 Example:

import java.util.function.Function;

public class StaticMethodRef {

public static void main(String[] args) {

// Lambda

Function<String, Integer> strLength1 = str -> str.length();

// Method Reference

Function<String, Integer> strLength2 = String::length;

System.out.println(strLength1.apply("Prashanth")); // 9

System.out.println(strLength2.apply("Java")); // 4

}

}

Output

9

4

#### (b) Reference to an instance method of a particular object

Syntax → instance::methodName

📌 Example:

import java.util.function.Consumer;

public class InstanceMethodRef {

public void printMessage(String msg) {

System.out.println("Message: " + msg);

}

public static void main(String[] args) {

InstanceMethodRef obj = new InstanceMethodRef();

// Lambda

Consumer<String> con1 = msg -> obj.printMessage(msg);

// Method Reference

Consumer<String> con2 = obj::printMessage;

con1.accept("Hello with Lambda");

con2.accept("Hello with Method Reference");

}

}

Output

Message: Hello with Lambda

Message: Hello with Method Reference

#### (c) Reference to an instance method of an arbitrary object of a particular type

Syntax → ClassName::methodName

📌 Example:

import java.util.\*;

public class ArbitraryMethodRef {

public static void main(String[] args) {

List<String> names = Arrays.asList("Prashanth", "Ravi", "Anu");

// Lambda

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

// Method Reference

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

}

}

Output

Prashanth

Ravi

Anu

Prashanth

Ravi

Anu

#### (d) Reference to a constructor

Syntax → ClassName::new

📌 Example:

import java.util.function.Supplier;

class Employee {

public Employee() {

System.out.println("Employee object created");

}

}

public class ConstructorRef {

public static void main(String[] args) {

// Lambda

Supplier<Employee> emp1 = () -> new Employee();

// Constructor Reference

Supplier<Employee> emp2 = Employee::new;

emp1.get();

emp2.get();

}

}

Output

Employee object created

Employee object created

### 3. Real-time Example

👉 Using **method reference** in streams.

import java.util.\*;

public class MethodRefStream {

public static void main(String[] args) {

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

// Lambda

fruits.stream().map(f -> f.toUpperCase()).forEach(f -> System.out.println(f));

// Method Reference

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

}

}

✅ Output:

MANGO

BANANA

APPLE

MANGO

BANANA

APPLE

### 4. Key Points Recap

* **Method References** are just a cleaner way to write **lambdas**.
* Types:
  1. Static method → ClassName::staticMethod
  2. Instance method of object → obj::method
  3. Instance method of arbitrary object → ClassName::method
  4. Constructor → ClassName::new

**Step 1: What is Optional?**

* Optional is a **container object** introduced in Java 8.
* It may or may not contain a **non-null value**.
* It is mainly used to **avoid NullPointerException (NPE)**.

Think of it like a **box**:

* If the box has something → you can use it.
* If the box is empty → you can safely check before using it.

**🔹 Step 2: Why Optional?**

Before Java 8, if a method returned null, we had to write multiple null checks:

String name = getName();

if(name != null) {

System.out.println(name.toUpperCase());

}

This is boring and error-prone. If we forget the null check, boom 💥 NullPointerException.

With Optional, we can write safer and cleaner code.

**🔹 Step 3: Creating Optional**

There are three main ways:

Optional<String> opt1 = Optional.of("Prashanth"); // value present

Optional<String> opt2 = Optional.empty(); // no value

Optional<String> opt3 = Optional.ofNullable(null); // null safe

**🔹 Step 4: Using Optional**

👉 Example:

import java.util.Optional;

public class OptionalDemo {

public static void main(String[] args) {

Optional<String> name = Optional.ofNullable("Prashanth");

// Check if value present

if(name.isPresent()) {

System.out.println(name.get().toUpperCase());

}

// Or simpler: ifPresent

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

// Default value if empty

String finalName = name.orElse("Default Name");

System.out.println(finalName);

}

}

Output

PRASHANTH

Hello Prashanth

Prashanth

**🔹 Step 5: Real-time Use Case**

Suppose you are fetching a user from database:

public Optional<User> findUserById(int id) {

User user = database.find(id);

return Optional.ofNullable(user); // return empty if user not found

}

Caller side:

findUserById(101)

.ifPresent(user -> System.out.println("User Found: " + user.getName()));

This way, you don’t need null checks everywhere.

✅ **Summary**

* Optional is used to avoid null pointer issues.
* It provides methods like:
  + isPresent(), ifPresent()
  + orElse(), orElseGet(), orElseThrow()
* Very useful in **database calls, service responses, or APIs** where null may come.

# Default and Static Methods in Interfaces (Java 8)

### 1. Background

Before Java 8:

* Interfaces could **only have abstract methods**.
* Adding a new method to an existing interface would **break all implementing classes**.

Java 8 solved this with **default and static methods**.

### 2. Default Methods

* Declared using the keyword default in the interface.
* Provides a **default implementation**.
* Implementing classes can **override** if needed.

📌 Example:

interface Vehicle {

void start(); // abstract method

default void stop() { // default method

System.out.println("Vehicle stopped");

}

}

class Car implements Vehicle {

public void start() {

System.out.println("Car started");

}

}

public class DefaultMethodDemo {

public static void main(String[] args) {

Vehicle car = new Car();

car.start(); // Car started

car.stop(); // Vehicle stopped (default method)

}

}

Output

Car started

Vehicle stopped

✅ Key point: You don’t have to implement stop() in every class.

### 3. Static Methods

* Declared using the keyword static in the interface.
* Belongs to the **interface**, not the object.
* Cannot be overridden by implementing classes.

📌 Example:

interface Vehicle {

static void info() {

System.out.println("Vehicles are for transportation");

}

}

public class StaticMethodDemo {

public static void main(String[] args) {

Vehicle.info(); // call directly from interface

}

}

Output

Vehicles are for transportation

### 4. Real-time Use Case

* **Default methods** → backward compatible interfaces (e.g., Collections API).
* **Static methods** → utility methods related to the interface (like Comparator.comparing()).

### 5. Key Points Recap

| **Feature** | **Belongs to** | **Can Override?** | **Use Case** |
| --- | --- | --- | --- |
| Default | Object | Yes | Provide default behavior in interface |
| Static | Interface | No | Utility/helper methods |

# Java 8 Date and Time API (java.time package)

### 1. Background

Before Java 8:

* We used java.util.Date and java.util.Calendar.
* Problems: mutable, confusing, thread-unsafe, poor API design.

Java 8 introduced **java.time package** to solve these issues.

* Immutable classes
* Thread-safe
* Easy-to-use

### 2. Main Classes

| **Class** | **Description** |
| --- | --- |
| LocalDate | Date without time (yyyy-MM-dd) |
| LocalTime | Time without date (HH:mm:ss) |
| LocalDateTime | Date + Time (yyyy-MM-ddTHH:mm:ss) |
| ZonedDateTime | Date + Time + Timezone |
| Instant | Timestamp (UTC) |
| Duration | Time-based amount (seconds, minutes) |
| Period | Date-based amount (days, months, years) |
| DateTimeFormatter | Format/parse date and time |

### 3. Examples

#### (a) LocalDate

import java.time.LocalDate;

public class LocalDateDemo {

public static void main(String[] args) {

LocalDate today = LocalDate.now();

System.out.println("Today: " + today); // e.g., 2025-09-21

LocalDate dob = LocalDate.of(2003, 1, 15);

System.out.println("Date of Birth: " + dob);

LocalDate nextWeek = today.plusWeeks(1);

System.out.println("Next Week: " + nextWeek);

}

}

Output

Today: 2025-09-22

Date of Birth: 2003-01-15

Next Week: 2025-09-29

#### (b) LocalTime

import java.time.LocalTime;

public class LocalTimeDemo {

public static void main(String[] args) {

LocalTime now = LocalTime.now();

System.out.println("Current Time: " + now);

LocalTime meeting = LocalTime.of(15, 30);

System.out.println("Meeting Time: " + meeting);

}

}

Output

Current Time: 08:49:47.180508264

Meeting Time: 15:30

#### (c) LocalDateTime

import java.time.LocalDateTime;

public class LocalDateTimeDemo {

public static void main(String[] args) {

LocalDateTime current = LocalDateTime.now();

System.out.println("Current DateTime: " + current);

LocalDateTime event = LocalDateTime.of(2025, 12, 25, 10, 0);

System.out.println("Event DateTime: " + event);

}

}

Output

Current DateTime: 2025-09-22T08:50:16.622414607

Event DateTime: 2025-12-25T10:00

#### (d) Formatting Date and Time

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

public class DateTimeFormatDemo {

public static void main(String[] args) {

LocalDateTime now = LocalDateTime.now();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");

String formatted = now.format(formatter);

System.out.println("Formatted DateTime: " + formatted);

}

}

Output

Formatted DateTime: 22-09-2025 08:50

### 4. Real-Time Use Case

* Logging timestamp in proper format
* Scheduling events
* Calculating age or time difference

# CompletableFuture in Java 8

### 1. What is CompletableFuture?

* Introduced in **Java 8** inside java.util.concurrent package.
* Allows **asynchronous, non-blocking execution**.
* Supports **chaining multiple tasks**, **handling results**, and **handling exceptions**.

Think of it like:

“I will start this task in the background, and when it’s done, I will continue with the next task.”

### 2. Creating CompletableFuture

import java.util.concurrent.CompletableFuture;

public class CompletableFutureDemo {

public static void main(String[] args) {

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

return "Hello from async task!";

});

future.thenAccept(result -> System.out.println(result));

}

}

Output:

Hello from async task!

### 3. Key Methods

| **Method** | **Purpose** |
| --- | --- |
| supplyAsync() | Runs a task asynchronously and returns a result |
| runAsync() | Runs a task asynchronously but returns void |
| thenApply() | Transform the result of a future (like map) |
| thenAccept() | Consume the result (like forEach) |
| thenRun() | Run another task after previous completes (no input) |
| exceptionally() | Handle exceptions |
| thenCombine() | Combine results of two futures |
| thenCompose() | Chain dependent futures |

### 4. Real-Time Example: Chain Tasks

import java.util.concurrent.CompletableFuture;

public class ChainExample {

public static void main(String[] args) {

CompletableFuture.supplyAsync(() -> {

System.out.println("Step 1: Fetching data...");

return 20;

}).thenApply(data -> {

System.out.println("Step 2: Processing data...");

return data \* 2;

}).thenAccept(result -> {

System.out.println("Step 3: Final result: " + result);

}).exceptionally(ex -> {

System.out.println("Error: " + ex.getMessage());

return null;

});

try { Thread.sleep(2000); } catch(Exception e) {}

}

}

✅ Output:

Step 1: Fetching data...

Step 2: Processing data...

Step 3: Final result: 40

### 5. Combining Multiple Futures

CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> 10);

CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> 20);

future1.thenCombine(future2, (a, b) -> a + b)

.thenAccept(sum -> System.out.println("Sum: " + sum));

Output:

Sum: 30

### 6. Key Advantages

* Non-blocking & asynchronous execution
* Chaining tasks easily
* Exception handling built-in
* Combines multiple async tasks

# Parallel Streams in Java 8

### 1. What is a Parallel Stream?

* A **Parallel Stream** divides the data into multiple chunks and processes them **in parallel** using **multiple threads**.
* It takes advantage of **multi-core CPUs** to **improve performance**.
* Easy to switch from sequential streams by using .parallelStream().

### 2. Sequential vs Parallel Stream

import java.util.\*;

import java.util.stream.\*;

public class StreamComparison {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

System.out.println("Sequential Stream:");

numbers.stream()

.map(n -> n \* 2)

.forEach(System.out::println);

System.out.println("\nParallel Stream:");

numbers.parallelStream()

.map(n -> n \* 2)

.forEach(System.out::println);

}

}

Output:

* Sequential stream → processed in order
* 2
* 4
* 6
* 8
* 10
* 12
* 14
* 16
* 18
* 20
* Parallel stream → may **process out of order** (multiple threads)
* 12
* 14
* 18
* 20
* 16
* 2
* 4
* 8
* 6
* 10

### 3. Real-Time Use Case

Suppose you want to **calculate sum of a large list** using multiple threads:

List<Integer> numbers = IntStream.rangeClosed(1, 1000000).boxed().toList();

// Sequential sum

long start = System.currentTimeMillis();

long sumSeq = numbers.stream().mapToLong(Integer::longValue).sum();

System.out.println("Sequential Sum: " + sumSeq + " Time: " + (System.currentTimeMillis() - start));

// Parallel sum

start = System.currentTimeMillis();

long sumPar = numbers.parallelStream().mapToLong(Integer::longValue).sum();

System.out.println("Parallel Sum: " + sumPar + " Time: " + (System.currentTimeMillis() - start));

Output

Sequential Sum: 500000500000 Time: X

Parallel Sum: 500000500000 Time: Y

✔ Parallel stream often **reduces execution time** for large datasets.

# Project: Employee Management System (Java 8)

### Scenario

We have a list of **employees**. Each employee has:

* id
* name
* department
* salary

**Tasks:**

1. Filter employees with salary > 50k
2. Group them by department
3. Calculate average salary per department
4. Get a list of employee names in uppercase
5. Handle missing employee safely using Optional
6. Perform parallel processing for large datasets

### 1. Employee Class

import java.util.\*;

class Employee {

int id;

String name;

String department;

int salary;

Employee(int id, String name, String dept, int salary) {

this.id = id;

this.name = name;

this.department = dept;

this.salary = salary;

}

@Override

public String toString() {

return name + " (" + department + ") - " + salary;

}

}

### 2. Main Class with Java 8 Features

import java.util.\*;

import java.util.stream.\*;

import java.util.concurrent.\*;

public class EmployeeManagement {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee(1, "Prashanth", "IT", 60000),

new Employee(2, "Raj", "HR", 40000),

new Employee(3, "Lakshmi", "IT", 70000),

new Employee(4, "Sita", "Finance", 50000),

new Employee(5, "Sai", "HR", 55000)

);

// 1. Filter salary > 50k

System.out.println("Employees with salary > 50k:");

employees.stream()

.filter(emp -> emp.salary > 50000)

.forEach(System.out::println);

// 2. Group by department

Map<String, List<Employee>> groupByDept = employees.stream()

.collect(Collectors.groupingBy(emp -> emp.department));

System.out.println("\nEmployees grouped by department:");

groupByDept.forEach((dept, emps) -> System.out.println(dept + ": " + emps));

// 3. Average salary per department

Map<String, Double> avgSalary = employees.stream()

.collect(Collectors.groupingBy(

emp -> emp.department,

Collectors.averagingInt(emp -> emp.salary)

));

System.out.println("\nAverage salary per department:");

avgSalary.forEach((dept, avg) -> System.out.println(dept + ": " + avg));

// 4. Employee names in uppercase

List<String> namesUpper = employees.stream()

.map(emp -> emp.name.toUpperCase())

.collect(Collectors.toList());

System.out.println("\nEmployee names in uppercase: " + namesUpper);

// 5. Handle missing employee safely with Optional

Optional<Employee> empOpt = employees.stream()

.filter(emp -> emp.id == 10)

.findFirst();

Employee emp = empOpt.orElseGet(() -> new Employee(0, "Default Employee", "None", 0));

System.out.println("\nEmployee found or default: " + emp);

// 6. Parallel Stream: sum of all salaries

int totalSalary = employees.parallelStream()

.mapToInt(emp2 -> emp2.salary)

.sum();

System.out.println("\nTotal Salary (Parallel Stream): " + totalSalary);

}

}

### Features Used

| **Feature** | **How Used** |
| --- | --- |
| **Streams** | filter, map, groupingBy, averagingInt |
| **Lambda** | emp -> emp.salary > 50000 |
| **Method Reference** | System.out::println |
| **Optional** | findFirst().orElseGet() |
| **Collectors** | groupingBy, averagingInt, toList |
| **Parallel Stream** | Sum of salaries |

### ✅ Output (Example)

Employees with salary > 50k:

Prashanth (IT) - 60000

Lakshmi (IT) - 70000

Sai (HR) - 55000

Employees grouped by department:

Finance: [Sita (Finance) - 50000]

HR: [Raj (HR) - 40000, Sai (HR) - 55000]

IT: [Prashanth (IT) - 60000, Lakshmi (IT) - 70000]

Average salary per department:

Finance: 50000.0

HR: 47500.0

IT: 65000.0

Employee names in uppercase: [PRASHANTH, RAJ, LAKSHMI, SITA, SAI]

Employee found or default: Default Employee (None) - 0

Total Salary (Parallel Stream): 275000

### Explanation of Flow

1. **Data Source** → Any collection, array, or database list.
2. **Stream API** → .stream() (sequential) or .parallelStream() (multi-threaded).
3. **Intermediate Operations** → Transform/filter data (filter, map, distinct, groupingBy).
4. **Lambda Expressions** → Define the logic concisely (emp -> emp.salary > 50000).
5. **Method References** → Shortcut for simple operations (System.out::println).
6. **Optional** → Handle potential null safely (findFirst().orElseGet()).
7. **Terminal Operation** → Executes the pipeline (forEach, collect, reduce).
8. **Result/Output** → Final processed data (List, Map, sum, average, etc.).

# Java 8 Cheat Sheet (Quick Reference)

| **Feature** | **Syntax / Method** | **Usage / Purpose** | **Real-Time Example** |
| --- | --- | --- | --- |
| **Lambda Expressions** | (parameters) -> expression | Implement functional interfaces concisely | emp -> emp.salary > 50000 |
| **Functional Interface** | @FunctionalInterface interface MyFunc {} | Interface with one abstract method for lambda | Predicate<Employee> p = emp -> emp.salary > 50000 |
| **Predicate<T>** | test(T t) | Boolean test on object | emp -> emp.salary > 50k |
| **Function<T,R>** | apply(T t) | Transform object from T → R | emp -> emp.name.toUpperCase() |
| **Consumer<T>** | accept(T t) | Perform action without returning value | System.out::println |
| **Supplier<T>** | get() | Provide value | () -> new Employee(0, "Default", "None", 0) |
| **Streams** | list.stream() / parallelStream() | Process collections in functional style | Filter, map, collect, reduce |
| **Intermediate Ops** | filter(), map(), distinct(), sorted() | Transform or filter data | filter(emp -> emp.salary>50k) |
| **Terminal Ops** | forEach(), collect(), reduce() | Produce final result | collect(Collectors.toList()) |
| **Method References** | ClassName::methodName | Shortcut for lambda calling a method | System.out::println |
| **Optional<T>** | Optional.of(), Optional.ofNullable() | Avoid NullPointerException | findFirst().orElseGet() |
| **Default Methods** | default methodName() {} | Provide default implementation in interface | default void stop() {} |
| **Static Methods** | static methodName() {} | Utility methods in interface | static void info() |
| **Date & Time API** | LocalDate, LocalTime, LocalDateTime | Immutable, thread-safe date-time handling | LocalDate.now() |
| **DateTimeFormatter** | DateTimeFormatter.ofPattern("dd-MM-yyyy") | Format / parse dates | date.format(formatter) |
| **CompletableFuture** | CompletableFuture.supplyAsync() | Async / non-blocking execution | thenApply(), thenAccept() |
| **Parallel Streams** | list.parallelStream() | Multi-core parallel processing | Sum large list faster |
| **Collectors** | Collectors.toList(), groupingBy(), averagingInt() | Collect or aggregate stream results | Group employees by dept |