**Java Functional Programming & Streams — Full Day Training Material**

**🔹 1. Introduction to Functional Programming**

**Theory Notes:**

* Introduced in Java 8 to simplify code and encourage immutability & declarative style.
* Focuses on **what to do**, not **how to do it**.
* Uses **functions as first-class citizens** (you can pass them as parameters, return them, store them in variables).
* Reduces boilerplate code, especially with collections and concurrency.

**Key Concepts:**

* **Pure functions:** No side effects, same output for same input.
* **Immutability:** Don’t modify data, create new versions.
* **Higher-order functions:** Functions that take or return other functions.

**Example: Traditional vs Functional**

// Imperative style

for (String name : names) {

if (name.startsWith("A")) {

System.out.println(name);

}

}

// Functional style

names.stream()

.filter(n -> n.startsWith("A"))

.forEach(System.out::println);

**🔹 2. Lambda Expressions**

**Theory Notes:**

* Introduced in Java 8 for writing short, concise functions.
* Syntax:  
  (parameters) -> expression  
  or  
  (parameters) -> { statements }

**Rules:**

* Type inference → compiler deduces parameter types.
* Used with functional interfaces.

**Example:**

@FunctionalInterface

interface Greet {

void sayHello(String name);

}

public class LambdaDemo {

public static void main(String[] args) {

Greet greet = (name) -> System.out.println("Hello " + name + "!");

greet.sayHello("Aashish");

}

}

**Mini Exercise:**  
Write a lambda to add two integers and print the result.

**🔹 3. Functional Interfaces**

**Theory Notes:**

* A **Functional Interface** contains exactly **one abstract method**.
* Annotated with @FunctionalInterface.
* Used as target for lambda expressions or method references.

**Common Functional Interfaces:**

| **Interface** | **Method** | **Purpose** |
| --- | --- | --- |
| Predicate<T> | test(T t) | Returns boolean |
| Consumer<T> | accept(T t) | Performs an action |
| Function<T,R> | apply(T t) | Returns a result |
| Supplier<T> | get() | Supplies a value |

**Code Example:**

import java.util.function.\*;

public class FunctionalInterfaceDemo {

public static void main(String[] args) {

Predicate<Integer> isEven = n -> n % 2 == 0;

Consumer<String> print = s -> System.out.println("Hello " + s);

Function<Integer, String> toString = n -> "Number: " + n;

System.out.println(isEven.test(4)); // true

print.accept("Java");

System.out.println(toString.apply(10));

}

}

**🔹 4. Method References**

**Theory Notes:**

* Shorthand for calling existing methods using :: operator.
* Used when lambda only calls an existing method.

**Types:**

1. **Static method:** ClassName::methodName
2. **Instance method:** instance::methodName
3. **Constructor reference:** ClassName::new

**Example:**

import java.util.\*;

public class MethodRefDemo {

public static void printMsg(String msg) {

System.out.println(msg);

}

public static void main(String[] args) {

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

// Lambda way

list.forEach(s -> System.out.println(s));

// Method reference way

list.forEach(MethodRefDemo::printMsg);

}

}

**🔹 5. Optional Class**

**Theory Notes:**

* Designed to avoid **NullPointerException**.
* Optional<T> represents a container which may or may not hold a value.
* Useful for cleaner null checks.

**Key Methods:**

| **Method** | **Description** |
| --- | --- |
| isPresent() | Returns true if value exists |
| ifPresent(Consumer) | Executes action if value present |
| orElse(value) | Returns value if empty |
| orElseGet(Supplier) | Returns value from supplier if empty |
| orElseThrow() | Throws exception if empty |

**Code Example:**

import java.util.Optional;

public class OptionalDemo {

public static void main(String[] args) {

Optional<String> name = Optional.ofNullable(null);

System.out.println("Is Present: " + name.isPresent());

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

System.out.println("Result: " + result);

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

Optional<String> city = Optional.of("Mumbai");

System.out.println(city.orElseGet(() -> "Unknown City"));

System.out.println(city.orElseThrow(() -> new RuntimeException("No city found")));

}

}

**Exercise:**  
Wrap an email field with Optional. Print email if present, else print a default string.

**🔹 6. Streams API**

**Theory Notes:**

* Introduced in Java 8 to process collections in a functional style.
* Uses pipelines:  
  **source → intermediate operations → terminal operation**
* Streams **don’t modify** the original data.

**Common Operations:**

| **Type** | **Example** |
| --- | --- |
| Intermediate | filter(), map(), sorted(), distinct() |
| Terminal | forEach(), collect(), count(), reduce() |

**Code Example:**

import java.util.\*;

import java.util.stream.\*;

class Employee {

int id;

String name;

double salary;

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

this.id = id; this.name = name; this.salary = salary;

}

}

public class StreamDemo {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee(1, "Ravi", 50000),

new Employee(2, "Neha", 70000),

new Employee(3, "Amit", 40000),

new Employee(4, "Kiran", 90000)

);

// 1. Filter employees with salary > 50k

employees.stream()

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

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

// 2. Map names

List<String> names = employees.stream()

.map(e -> e.name)

.collect(Collectors.toList());

System.out.println("Names: " + names);

// 3. Find highest salary

Optional<Employee> max = employees.stream()

.max(Comparator.comparingDouble(e -> e.salary));

max.ifPresent(e -> System.out.println("Highest: " + e.name + " - " + e.salary));

}

}

**Practice Tasks:**

1. Filter students with marks > 60.
2. Count how many start with letter "A".
3. Sort by name and collect to a new list.
4. Use map() to get a list of uppercase names.

**🔹 7. Date & Time API (java.time)**

**Theory Notes:**

* Replaces old Date and Calendar (which were mutable & confusing).
* New package: java.time.
* Classes are **immutable** and **thread-safe**.

**Common Classes:**

| **Class** | **Description** |
| --- | --- |
| LocalDate | Date (year, month, day) |
| LocalTime | Time (hour, minute, second) |
| LocalDateTime | Date and time |
| Period | Time between two dates |
| Duration | Time between two times |
| DateTimeFormatter | Format and parse date/time |

**Code Example:**

import java.time.\*;

import java.time.format.DateTimeFormatter;

public class DateTimeDemo {

public static void main(String[] args) {

LocalDate today = LocalDate.now();

LocalTime time = LocalTime.now();

LocalDateTime dateTime = LocalDateTime.now();

System.out.println("Today: " + today);

System.out.println("Time: " + time);

System.out.println("DateTime: " + dateTime);

LocalDate nextWeek = today.plusWeeks(1);

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

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

System.out.println("Formatted: " + dateTime.format(formatter));

}

}

**🧩 End-of-Day Exercises**

1. Implement a Predicate to check if an employee’s salary is above 50k.
2. Use Function to convert employee names to uppercase.
3. Use Consumer to print employee details.
4. Use Optional to safely handle null employee names.
5. Use Streams to find:
   * Average salary
   * Employee with minimum salary
   * List of employees whose name starts with “N”.