**Java Functional Programming**

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Java has been incorporating functional programming features since Java 8, and Java 17 continues to support and enhance these capabilities. This chapter will explore various aspects of functional programming in Java 17.

**1. Lambda Expressions**

Lambda expressions are a concise way to represent anonymous functions. They are the cornerstone of functional programming in Java.

import java.util.Arrays;

import java.util.List;

public class LambdaExample {

public static void main(String[] args) {

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

// Using lambda expression to define behavior

numbers.forEach(n -> System.out.println(n \* 2));

}

}

Output:

2

4

6

8

10

**2. Functional Interfaces**

Functional interfaces are interfaces with a single abstract method. They can be used as the assignment target for lambda expressions or method references.

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

MathOperation addition = (a, b) -> a + b;

MathOperation subtraction = (a, b) -> a - b;

System.out.println("10 + 5 = " + operate(10, 5, addition));

System.out.println("10 - 5 = " + operate(10, 5, subtraction));

}

private static int operate(int a, int b, MathOperation operation) {

return operation.operate(a, b);

}

}

Output:

10 + 5 = 15

10 - 5 = 5

**3. Method References**

Method references provide a way to refer to methods or constructors without invoking them. They can often be used as a more readable alternative to lambda expressions.

import java.util.Arrays;

import java.util.List;

public class MethodReferenceExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Arun", "Chandni", "Basu");

// Using method reference to print each name

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

// Using method reference to sort strings

names.sort(String::compareToIgnoreCase);

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

}

}

Output:

Arun

Chandni

Basu

Sorted names: [Arun, Basu, Chandni]

**4. Streams API**

The Streams API allows for functional-style operations on streams of elements. It supports both intermediate and terminal operations that can be chained together.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamExample {

public static void main(String[] args) {

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

// Using streams to filter, map, and collect

List<Integer> evenSquares = numbers.stream()

.filter(n -> n % 2 == 0)

.map(n -> n \* n)

.collect(Collectors.toList());

System.out.println("Squares of even numbers: " + evenSquares);

// Using streams to find the sum of all numbers

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

System.out.println("Sum of all numbers: " + sum);

}

}

Output:

Squares of even numbers: [4, 16, 36, 64, 100]

Sum of all numbers: 55

**5. Optional Class**

The Optional class is a container object that may or may not contain a non-null value. It's used to represent nullable values and avoid null pointer exceptions.

import java.util.Optional;

public class OptionalExample {

public static void main(String[] args) {

String name = "Anil";

String nullName = null;

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

Optional<String> optNullName = Optional.ofNullable(nullName);

System.out.println("Non-empty Optional: " + optName.orElse("Unknown"));

System.out.println("Empty Optional: " + optNullName.orElse("Unknown"));

optName.ifPresent(n -> System.out.println("Hello, " + n + "!"));

optNullName.ifPresent(n -> System.out.println("Hello, " + n + "!")); // This won't be printed

}

}

Output:

Non-empty Optional: Anil

Empty Optional: Unknown

Hello, Anil!

**6. Functional Programming with Collections**

Java's Collection framework has been enhanced with methods that support functional programming styles.

import java.util.HashMap;

import java.util.Map;

public class FunctionalCollectionsExample {

public static void main(String[] args) {

Map<String, Integer> salaries = new HashMap<>();

salaries.put("Janani", 40000);

salaries.put("Arun", 50000);

salaries.put("Basu", 60000);

// Using computeIfPresent to give everyone a 10% raise

salaries.replaceAll((name, salary) -> (int)(salary \* 1.1));

System.out.println("Updated salaries: " + salaries);

// Using merge to add a new employee or update existing salary

salaries.merge("Charu", 55000, (oldSalary, newSalary) -> Math.max(oldSalary, newSalary));

salaries.merge("Arun", 70000, (oldSalary, newSalary) -> Math.max(oldSalary, newSalary));

System.out.println("Salaries after merge: " + salaries);

}

}

Output:

Updated salaries: {Basu=66000, Janani=44000, Arun=55000}

Salaries after merge: {Bob=66000, Janani=44000, Charu=55000, Arun=70000}

**7. Immutability and Pure Functions**

Functional programming emphasizes immutability and pure functions. Here's an example of creating an immutable class and using it with pure functions:

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class ImmutableExample {

public static final class ImmutablePerson {

private final String name;

private final int age;

private final List<String> hobbies;

public ImmutablePerson(String name, int age, List<String> hobbies) {

this.name = name;

this.age = age;

this.hobbies = Collections.unmodifiableList(new ArrayList<>(hobbies));

}

public String getName() { return name; }

public int getAge() { return age; }

public List<String> getHobbies() { return hobbies; }

public ImmutablePerson withName(String newName) {

return new ImmutablePerson(newName, this.age, this.hobbies);

}

}

public static void main(String[] args) {

ImmutablePerson jai = new ImmutablePerson("Jai", 30, List.of("Reading", "Hiking"));

System.out.println("Original: " + jai.getName() + ", " + jai.getAge() + ", " + jai.getHobbies());

ImmutablePerson jaiShankar = jai.withName("Jai Shankar");

System.out.println("Updated: " + jaiShankar.getName() + ", " + jaiShankar.getAge() + ", " + jaiShankar.getHobbies());

System.out.println("Original unchanged: " + jai.getName());

}

}

Output:

Original: Jai, 30, [Reading, Hiking]

Updated: Jai Shankar, 30, [Reading, Hiking]

Original unchanged: Jai

**Conclusion**

Functional programming in Java 17 offers powerful tools for writing more concise, maintainable, and potentially more parallel code. By leveraging lambda expressions, streams, and other functional programming concepts, developers can write more expressive and efficient Java code. As you become more comfortable with these concepts, you'll find many opportunities to apply functional programming principles in your Java projects.