Chapter 17: Functional Programming in Java

Introduction

With the introduction of **Java 8**, functional programming became an integral part of Java through features like **lambda expressions**, **functional interfaces**, **Streams API**, and **method references**. This paradigm shift allowed Java developers to write more concise, expressive, and parallelizable code. Functional programming treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

In this chapter, we'll explore functional programming in Java, covering key concepts, syntax, and practical applications.

Table of Contents

---

1. What is Functional Programming?

Functional Programming (FP) is a declarative programming paradigm where functions are treated as first-class citizens.

✅ Key Principles:

- **Immutability**: Data cannot be changed after it's created.
- **First-class Functions**: Functions can be passed as arguments and returned as values.
- **No Side Effects**: Functions produce the same output for the same input without modifying any external state.
- **Pure Functions**: A function that has no side effects and returns the same output for the same input.

---

2. Functional Interfaces

A **functional interface** is an interface with only one abstract method. It can have multiple default or static methods.

🔷 Syntax:

```
javaCopy code@FunctionalInterface
interface MyFunctionalInterface {
    void execute();
}
```

You can use this interface with lambda expressions or method references.

🔷 Examples of Predefined Functional Interfaces:

- `Runnable`
- `Callable`
- `Comparator`
- `ActionListener`

---

3. Lambda Expressions

Lambda expressions provide a clear and concise way to represent a functional interface.

🔷 Syntax:

```
javaCopy code(parameters) -> expression
```

◈ Example:

```
javaCopy codeMyFunctionalInterface mfi = () -> System.out.println("Hello Func
tional World!");
mfi.execute();
```

◈ With Parameters:

```
javaCopy codeBinaryOperator<Integer> adder = (a, b) -> a + b;
System.out.println(adder.apply(10, 20));  // Output: 30
```

---

4. Method References

Method references provide a shorthand for calling methods using the `::` operator.

◈ Types of Method References:

- Static method: `ClassName::staticMethod`
- Instance method: `object::instanceMethod`
- Constructor: `ClassName::new`

◈ Example:

```
javaCopy codeList<String> list = Arrays.asList("Java", "Python", "C++");
list.forEach(System.out::println);
```

---

5. Built-in Functional Interfaces in `java.util.function`

Java 8 provides several ready-to-use functional interfaces in the `java.util.function` package.

Interface	Description	Example
`Predicate<T>`	boolean-valued function of one argument	`(x) -> x > 10`
`Function<T, R>`	function from T to R	`(s) -> s.length()`
`Consumer<T>`	performs an operation on a single input	`System.out::println`
`Supplier<T>`	supplies a result of type T	`() -> "Hello"`
`UnaryOperator<T>`	unary operation on a type T	`x -> x * 2`
`BinaryOperator<T>`	binary operation on two values of type T	`(a, b) -> a + b`

---

6. Stream API and Functional Operations

The **Stream API** allows processing collections in a functional style.

◈ Stream Creation:

```
javaCopy codeList<String> names = Arrays.asList("John", "Jane", "Jack");
Stream<String> stream = names.stream();
```

◈ Common Functional Operations:

- `map()`: transform data
- `filter()`: filter data based on a condition
- `forEach()`: apply action to each element
- `reduce()`: reduce to a single result
- `collect()`: collect elements into a container (e.g., List)

◈ Example:

```
javaCopy codeList<String> names = Arrays.asList("John", "Jane", "Jack");
names.stream()
     .filter(name -> name.startsWith("J"))
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

---

7. Optional Class

`Optional<T>` is a container object which may or may not contain a non-null value. It helps avoid `NullPointerException`.

◈ Example:

```
javaCopy codeOptional<String> name = Optional.ofNullable(getName());
name.ifPresent(System.out::println);
```

---

8. Functional Programming vs Object-Oriented Programming

Aspect	Functional Programming	OOP
Data Mutability	Immutable	Mutable
Function Type	First-class citizen	Bound to objects
State	Stateless (pure)	Stateful

Aspect	Functional Programming	OOP
Reusability	High via functions	High via classes
Parallelism	Easier with stateless	More complex

---

9. Best Practices

- Use **lambda expressions** for short, inline implementations.
- Prefer **method references** when lambda just calls a method.
- Use `Optional` to handle nullable return types.
- Leverage **Streams** for processing collections.
- Avoid side-effects in functional operations.
- Keep functions **pure and stateless** whenever possible.

---

10. Use Cases and Real-World Applications

- **Data transformation pipelines** with Streams.
- **Event handling** in GUI applications using lambda expressions.
- **Filtering and sorting** large datasets.
- **Reactive programming** models in web applications.
- **Concurrency-friendly code** due to immutability.

✅ Summary

Functional Programming in Java enhances code readability, maintainability, and parallel execution. With features like lambda expressions, method references, Streams, and built-in functional interfaces, Java enables a more expressive and concise style of programming. Understanding and applying functional principles effectively helps developers write cleaner and more efficient code in modern Java applications.
