Chapter 22: Lambda Expressions and Functional Interfaces

Introduction

In modern Java development (Java 8 onwards), **lambda expressions** and **functional interfaces** are at the core of the language's support for **functional programming**. These features make code more concise, readable, and expressive, especially when working with APIs like **Streams**, **Collections**, and **multithreading**.

This chapter provides an in-depth understanding of lambda expressions, functional interfaces, their syntax, usage patterns, and how they fit into the broader landscape of Java programming.

22.1 Functional Programming in Java

Java has traditionally been an object-oriented language. However, from Java 8 onwards, functional programming paradigms have been integrated into Java to allow:

- **Passing behavior as a parameter**
- **Reducing boilerplate code**
- **Creating more flexible and reusable APIs**

Functional programming emphasizes **pure functions**, **immutability**, and **statelessness**. Lambda expressions are the foundation of this functional capability.

22.2 What is a Lambda Expression?

A **lambda expression** is an **anonymous function**—a block of code that can be passed around and executed. It can be used to provide the implementation of a method defined by a functional interface.

Syntax:

```
(parameters) -> expression
```

Or

```
(parameters) -> { statements }
```

Examples:

```
(int a, int b) -> a + b
```

```
() -> System.out.println("Hello World")

(String s) -> { System.out.println(s); }
```

22.3 Key Characteristics of Lambda Expressions

1. **No need to define a method explicitly.**
2. **Can be assigned to variables or passed as parameters.**
3. **No need to use an anonymous inner class.**
4. **Infers types from context (type inference).**

22.4 Functional Interfaces

A **functional interface** is an interface that has **exactly one abstract method**. Lambda expressions can be used to instantiate these interfaces.

Example:

```
@FunctionalInterface
interface MyFunction {
    int operation(int a, int b);
}

MyFunction add = (a, b) -> a + b;
System.out.println(add.operation(5, 3));  // Output: 8
```

@FunctionalInterface Annotation

Although not mandatory, using the `@FunctionalInterface` annotation is a good practice. It ensures the interface conforms to the single abstract method (SAM) rule.

22.5 Built-in Functional Interfaces (java.util.function)

Java provides a set of pre-defined functional interfaces in the `java.util.function` package.

Interface	Description	Example Lambda Expression
`Predicate<T>`	Returns boolean value	`x -> x > 10`
`Function<T,R>`	Takes T, returns R	`s -> s.length()`
`Consumer<T>`	Takes T, returns void	`s -> System.out.println(s)`
`Supplier<T>`	Returns T, takes nothing	`() -> new Random().nextInt()`
`UnaryOperator<T>`	Takes T and returns T	`x -> x * x`
`BinaryOperator<T>`	Takes (T, T) and returns T	`(x, y) -> x + y`

22.6 Type Inference and Target Typing

Java can often infer parameter types of lambda expressions based on the **context** (i.e., target type of the functional interface).

```
Comparator<String> comp = (s1, s2) -> s1.compareToIgnoreCase(s2);
```

In the above example, Java infers that `s1` and `s2` are of type `String`.

22.7 Lambda vs Anonymous Class

Feature	Lambda Expression	Anonymous Class
Syntax	Concise	Verbose
`this` Keyword	Refers to enclosing class	Refers to anonymous class itself
Overriding Methods	Only one (functional interface)	Can override multiple
Object Overhead	Less (no additional class)	More (creates a separate class)

22.8 Scope and Access

- Lambda expressions can access **effectively final variables**.

  ```
  int x = 10;
  Runnable r = () -> System.out.println(x); // x must be effectively
  final
  ```

- You **cannot modify** local variables inside a lambda unless they are final or effectively final.

22.9 Lambda Expressions in Collections API

Using forEach with Lambda:

```
List<String> list = Arrays.asList("Java", "Python", "C++");
list.forEach(item -> System.out.println(item));
```

Using removeIf:

```
list.removeIf(s -> s.startsWith("J"));
```

Using sort with Comparator:

```
list.sort((s1, s2) -> s1.compareTo(s2));
```

22.10 Lambda Expressions in Multithreading

Lambda simplifies thread creation:

```
new Thread(() -> {
    System.out.println("Running in a separate thread");
}).start();
```

22.11 Method References and Constructor References

A **method reference** is a shorthand notation of a lambda expression calling a method.

Syntax:

```
ClassName::methodName
```

Examples:

```
Consumer<String> printer = System.out::println;

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

Supplier<List<String>> listSupplier = ArrayList::new;
```

22.12 Stream API and Lambda Expressions

Lambda expressions are often used with **Stream API** to process data in a functional way:

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

int sum = numbers.stream()
                 .filter(n -> n % 2 == 0)
                 .mapToInt(n -> n)
                 .sum();
```

22.13 Custom Functional Interface Example

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

public class LambdaExample {
    public static void main(String[] args) {
        MathOperation addition = (a, b) -> a + b;
        MathOperation multiply = (a, b) -> a * b;
```

```
        System.out.println("Addition: " + addition.operate(5, 3));
        System.out.println("Multiplication: " + multiply.operate(5, 3));
    }
}
```

22.14 Best Practices

- Prefer **built-in functional interfaces**.
- Keep lambda expressions **short and clear**.
- Avoid complex business logic in lambdas.
- Use **method references** for cleaner syntax.
- Use **@FunctionalInterface** for clarity and compiler safety.

22.15 Limitations of Lambda Expressions

- Can't throw checked exceptions directly (must handle or wrap).
- Not suitable for all scenarios—sometimes a **named class** or **anonymous class** is clearer.
- Debugging stack traces from lambdas can be harder.

Summary

Lambda expressions and functional interfaces introduce a powerful functional style of programming in Java. They enable more expressive, concise, and readable code, especially when used with the Collections and Streams API. While lambda expressions bring great benefits, they must be used wisely, keeping readability and maintainability in mind.

Understanding this chapter equips developers with modern tools and idioms critical to writing clean and efficient Java applications.
