Modern Java Language Features

Functional Interface

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

- It can have **any number of default or static methods**, but only **one abstract method**.
- Because of this **one method rule**, Java can directly use a **lambda expression** to represent it.

```java
// MyRunnable.java
public interface MyRunnable {
    public abstract void run();
}

public class TaskRunner implements MyRunnable {
    public void run() {
        System.out.println("Task running...");
    }
}
public class Main {
    public static void main(String[] args) {
        MyRunnable r = new TaskRunner(); // normal object creation
        r.run();
    }
}
```

Why Do We Need Functional Interfaces

To Enable "Passing Behaviour" Instead of Just Data or class

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
// in main class
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Task running...");
    }
};          //anonymous inner class
```

What is @FunctionalInterface? //annotation

@FunctionalInterface is a marker annotation in Java (introduced in Java 8) that tells the compiler:

"This interface must have exactly one abstract method.“

If you violate this rule (by adding more than one abstract method), the compiler will show an **error**.

Anonymous Inner Class – Definition

A **class without a name** that is **declared and instantiated at the same time**, usually to **provide an immediate implementation** of an interface or to extend a class.

Lambda Expression //new feature

In **Java**, a **lambda expression** is basically a **shorter way to write an anonymous function** (a function without a name).
It is used to implement **functional interfaces** (interfaces that have exactly **one abstract method**).

General Syntax:

(parameters) -> { body }

- (parameters) → like function parameters
- -> → arrow token (points from parameters to body)
- { body } → code that will run

Lambda Expression Example

```
// Without Lambda (Old way)
Runnable r1 = new Runnable() {
    public void run() {
        System.out.println("Hello from Runnable");
    }
};

// With Lambda
Runnable r2 = () -> System.out.println("Hello from Runnable");
```

Why is it used?

Before Java 8, if you wanted to pass a block of code as a method argument, you had to:

- Create a class
- Or use an **anonymous inner class** → which is long and repetitive

Lambdas help by:

- **Reducing boilerplate** (less code to write)
- Making code **more readable**
- Enabling **functional programming style** in Java
- Allowing **behaviour to be passed as data** (like in JavaScript functions)

What Are Annotations in Java?

In Java, **annotations** are special **metadata tags** that you can add to code (classes, methods, variables, etc.) to give **extra information** to:

- The compiler
- The Java Virtual Machine (JVM)
- Frameworks & tools (like Spring, JUnit)

They **do not** change how the code works directly — but they **can influence how tools or the compiler treat your code**.

Basic Syntax

@AnnotationName

or with parameters:

@AnnotationName(key = "value")

Why Do We Use Annotations?

- Give instructions to compiler (e.g., @Override checks method overriding correctness)
- Reduce boilerplate code (e.g., Lombok’s @Getter, @Setter)
- Provide metadata to frameworks (e.g., @RestController in Spring Boot)
- Runtime processing (e.g., @Entity in Hibernate to map class to a DB table)

Common Built-in Annotations

Annotation	**Purpose**
@Override	Ensures method is correctly overriding a superclass method
@Deprecated	Marks a method/class as outdated and shows a warning
@SuppressWarnings	Hides compiler warnings for specific code
@FunctionalInterface	Ensures interface has exactly one abstract method
@SafeVarargs	Suppresses unsafe operation warnings for varargs

Example

```
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    @Override // compiler checks if overriding is correct
    void display() {
        System.out.println("Child display");
    }
}
```

Types of Annotations

- Marker Annotations – no values (e.g., @Override, @FunctionalInterface)
- Single-Value Annotations – only one value (e.g., @SuppressWarnings("unchecked"))
- Multi-Value Annotations – multiple key-value pairs (e.g., @MyAnnotation(key1="value1", key2="value2"))