Java 8 introduced **Lambda Expressions** to enable **functional-style programming**, making code **concise**, **readable**, and **flexible**, especially with **collections and streams**.

**🔷 What is a Lambda Expression?**

A **lambda expression** is a **short block of code** that takes in parameters and returns a value. It can be used to represent the **implementation of a functional interface**.

**🔹 Syntax:**

(parameters) -> expression

Or:

(parameters) -> { statements; }

**✅ Example**

**Before Java 8 (using Anonymous Class)**

Runnable r = new Runnable() {

public void run() {

System.out.println("Running");

}

};

**With Java 8 Lambda:**

Runnable r = () -> System.out.println("Running");

**🧠 Functional Interface**

A **Functional Interface** is an interface with exactly **one abstract method**.

Examples:

* Runnable → void run()
* Callable<V> → V call() throws Exception
* Comparator<T> → int compare(T o1, T o2)
* Function<T, R>, Predicate<T>, Consumer<T>, Supplier<T>

You can annotate with @FunctionalInterface to ensure only one abstract method exists.

**🔹 Lambda with Collections**

**Sorting with Lambda:**

List<String> names = Arrays.asList("Zara", "Bob", "Alex");

Collections.sort(names, (s1, s2) -> s1.compareTo(s2));

**📦 Built-in Functional Interfaces (in java.util.function package)**

| **Interface** | **Signature** | **Description** |
| --- | --- | --- |
| Predicate<T> | boolean test(T t) | For boolean condition testing |
| Function<T,R> | R apply(T t) | Converts T to R |
| Consumer<T> | void accept(T t) | Consumes T (no return) |
| Supplier<T> | T get() | Supplies a value of type T |
| BiFunction<T,U,R> | R apply(T t, U u) | Two inputs, one output |

**🧪 Examples**

**1. Predicate Example**

Predicate<String> isLong = str -> str.length() > 5;

System.out.println(isLong.test("Lambda")); // true

**2. Function Example**

Function<String, Integer> strLength = str -> str.length();

System.out.println(strLength.apply("Java")); // 4

**3. Consumer Example**

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

printer.accept("Alice");

**4. Supplier Example**

Supplier<Double> random = () -> Math.random();

System.out.println(random.get());

**💡 Lambdas in Streams**

List<String> names = Arrays.asList("Zara", "Bob", "Alex", "Eva");

names.stream()

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

.map(String::toUpperCase)

.forEach(System.out::println);

**🔁 With Threads**

new Thread(() -> System.out.println("Thread running")).start();

**🧯 Caveats / Limitations**

* Cannot reference non-final local variables (unless effectively final).
* Debugging stack traces is harder.
* No checked exceptions directly (need to wrap or handle).
* Overuse reduces readability in complex cases.

**🧩 Lambda with Custom Functional Interface**

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

public class LambdaExample {

public static void main(String[] args) {

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

MathOperation mul = (a, b) -> a \* b;

System.out.println(add.operate(5, 3)); // 8

System.out.println(mul.operate(5, 3)); // 15

}

}

**🧠 Conclusion**

| **Benefit** | **Explanation** |
| --- | --- |
| Concise Syntax | Less boilerplate than anonymous classes |
| Enhanced Collections/Streams | Functional-style code pipelines |
| Encourages Functional Design | Leverages higher-order functions |