Functional Programming and lambda expression

What is Functional Programming in Java?

- **Functional Programming (FP)** focuses on **"what to do" rather than "how to do it"**.
- Java 8+ supports FP via:
    - **Lambda expressions** → concise way to write functions
    - **Streams API** → process collections in a declarative, pipeline style
    - **Method references** → shortcut for lambdas

2️⃣ Lambda Expressions

A **lambda expression** is an **anonymous function** that can be passed around.

Syntax:

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

Examples:

```
// Runnable using lambda
Runnable r = () -> System.out.println("Hello, Lambda!");
r.run();

// Function taking one argument
Function<Integer, Integer> square = x -> x * x;
System.out.println(square.apply(5)); // 25
```

3️⃣ Streams API

A **Stream** represents a sequence of elements supporting **functional-style operations**.

- **Source:** collection, array, or generator
- **Operations:** intermediate (map, filter, sorted) and terminal (collect, forEach, reduce)

3a) Filtering and Mapping

```
List<String> names = List.of("Alice", "Bob", "Charlie", "David",
"Bob");

// Filter names with length <= 3
List<String> shortNames = names.stream()
                               .filter(name -> name.length() <= 3)
                               .toList();

// Convert all names to uppercase
List<String> upperNames = names.stream()
                               .map(String::toUpperCase)
                               .toList();

System.out.println(shortNames); // [Bob, Bob]
System.out.println(upperNames); // [ALICE, BOB, CHARLIE, DAVID, BOB]
```

3b) Sorting

```
List<String> sortedNames = names.stream()
                                .sorted()
                                .toList();
System.out.println(sortedNames); // [Alice, Bob, Bob, Charlie, David]
```

3c) Aggregation (reduce)

```
int sum = List.of(1, 2, 3, 4, 5).stream()
```

```
                                .reduce(0, Integer::sum);
System.out.println(sum); // 15
```

3d) Grouping and Counting

```
Map<String, Long> frequency = names.stream()
                                .collect(Collectors.groupingBy(n ->
n, Collectors.counting()));
System.out.println(frequency); // {Alice=1, Bob=2, Charlie=1, David=1}
```

3e) Combining Operations

```
List<String> processed = names.stream()
                            .filter(n -> n.startsWith("A") ||
n.startsWith("B"))
                            .map(String::toUpperCase)
                            .sorted()
                            .toList();
System.out.println(processed); // [ALICE, BOB, BOB]
```

- This is **functional, declarative, and chainable** — no loops needed.

4️⃣ Parallel Streams

```
int sumLarge = IntStream.rangeClosed(1, 1_000_000)
                        .parallel()
                        .reduce(0, Integer::sum);
System.out.println(sumLarge);
```

- **Parallel streams** automatically divide work across cores.
- Combine with FP style to scale processing without manual threads.

5️⃣ Method References

Shortcut for lambdas:

```
List<String> names = List.of("Alice", "Bob", "Charlie");

// Lambda
names.forEach(name -> System.out.println(name));

// Method reference
names.forEach(System.out::println);
```

- Can reference **static methods, instance methods, and constructors**.

6️⃣ Key Benefits of FP with Streams & Lambdas

1. **Less boilerplate** — no explicit loops or temporary collections.
2. **Declarative style** — focus on “what” instead of “how.”
3. **Easier parallelization** — `parallelStream()` for multi-core processing.
4. **Composability** — intermediate operations chain nicely.
5. **Immutability friendly** — encourages avoiding side effects.