# Functional programming explained

what is functional programming ? explain stream programming with examples

Great question 👍 Let's break this down step by step.

---

## 🔹 What is Functional Programming?

Functional Programming (FP) is a **programming paradigm** where we treat **functions as the main building blocks**. Instead of focusing on *how* to do something (step-by-step instructions, like in imperative programming), FP focuses on *what* needs to be done.

Key ideas in FP:

1. **Functions as First-Class Citizens** – functions can be passed as arguments, returned from other functions, and stored in variables.
2. **Immutability** – data should not be modified, instead new data should be created.
3. **Pure Functions** – output depends only on input, with no side effects.
4. **Declarative Style** – describe *what* you want, not *how* to do it.

👉 Java is not purely functional, but with **lambdas** and **Streams API**, it supports functional-style programming.

---

## 🔹 Stream Programming in Java

The **Streams API** (introduced in Java 8) is a **functional programming tool** that lets you process collections (like `List`, `Set`, `Array`) in a **declarative** way.

Instead of writing loops manually, you can chain functional operations such as:

- `map` → transform each element
- `filter` → keep only elements that match a condition
- `sorted` → sort elements
- `forEach` → terminal operation to consume the data

---

### ✅ Example 1: Without Streams (imperative)

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

        // Find square of odd numbers and print
        for(Integer n : numbers) {
            int square = n * n;
            if(square % 2 != 0) {
                System.out.println(square);
            }
        }
    }
}
```

This is long and step-by-step.

---

## ✅ Example 2: With Streams (functional)

```
java

import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

        numbers.stream()                      // create stream
               .map(n -> n * n)               // square each number
               .filter(n -> n % 2 != 0)       // keep only odd squares
               .sorted()                      // sort them
               .forEach(System.out::println); // print
    }
}
```

Output:

```

1
9
25
49
81
```

👉 Notice:

- No explicit loops.
- Each step describes *what* should happen, not *how*.
- Functions (`map`, `filter`) are chained together like a pipeline.

---

## ✅ Example 3: Stream with Strings

```
java

import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Uday", "Ankit", "Sunbeam", "Java");

        names.stream()
```

```
                .filter(s -> s.length() > 4)    // keep names longer than 4 chars
                .map(String::toUpperCase)       // convert to uppercase
                .forEach(System.out::println);  // print
    }
}
```

Output:

```
nginx

SUNBEAM
JAVA
```

---

## ✅ Example 4: Collect Results

```
java

import java.util.*;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

        List<String> result = words.stream()
                                   .filter(w -> w.contains("a"))
                                   .map(String::toUpperCase)
                                   .collect(Collectors.toList());

        System.out.println(result);
    }
}
```

Output:

```
csharp

[APPLE, BANANA, DATE]
```

---

# 🔹 Summary

- **Functional Programming** = thinking in terms of *functions* (no side effects, immutability, declarative style).
- **Streams in Java** = a way to apply functional programming concepts to collections.
- Benefits:
  - Cleaner and shorter code.
  - Easier to understand (pipeline style).
  - Encourages immutability.

---

👉 Do you want me to also explain **stateful vs stateless operations in streams** (like why `sorted` behaves differently than `map`/`filter`)?

> Functional Programming (FP) is a programming paradigm where we treat functions as the main building blocks. Instead of focusing on how to do something (step-by-step instructions, like in imperative programming), FP focuses on what needs to be done.
>
> can you explain it on deeper level like how functions

Perfect 👍 you've got the surface definition right. Let's go deeper into **how functions work in Functional Programming (FP)** and why it's different from imperative / OOP styles.

---

## 🔹 1. Functions are *First-Class Citizens*

In FP, **functions are values** just like numbers or objects.
That means you can:

- **Store** them in variables
- **Pass** them as arguments to other functions
- **Return** them from functions

👉 Example in Java:

```java
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        // Function as a value
        Function<Integer, Integer> square = n -> n * n;

        // Passing function to another method
        int result = applyFunction(5, square);
        System.out.println(result); // 25
    }

    static int applyFunction(int x, Function<Integer, Integer> f) {
        return f.apply(x); // executes the function
    }
}
```

Here `square` is not just a method – it's a **value** that can be passed around.

---

## 🔹 2. Pure Functions

A **pure function** means:

- Output depends only on input
- No side effects (does not change global state, files, database, etc.)

👉 Example:

```java
int square(int n) {
    return n * n; // pure: depends only on input
}
```

❌ Impure function:

```java
int counter = 0;

int squareAndCount(int n) {
    counter++;           // modifies global state (side effect)
    return n * n;
}
```

Why purity matters? → If a function is pure, you can:

- Replace a function call with its result (referential transparency).
- Predict behavior easily.
- Safely run it in parallel.

---

## 🔹 3. Immutability

In FP, we don't modify existing data. We create **new values**.

👉 Example (OOP / imperative style):

```java
List<String> names = new ArrayList<>();
names.add("Uday");  // modifies the list
```

👉 Functional style:

```java
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Uday", "Sunbeam");

        // Instead of modifying, create a new list
        List<String> upper = names.stream()
                                  .map(String::toUpperCase)
                                  .collect(Collectors.toList());

        System.out.println(upper); // [UDAY, SUNBEAM]
    }
}
```

---

## 🔹 4. Higher-Order Functions (HOF)

Functions that **take functions as input or return functions**.

👉 Example:

```
java

import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<Integer, Integer> doubler = n -> n * 2;
        Function<Integer, Integer> tripler = n -> n * 3;

        // compose functions: double then triple
        Function<Integer, Integer> doubleThenTriple = doubler.andThen(tripler);

        System.out.println(doubleThenTriple.apply(5)); // (5*2)=10 → (10*3)=30
    }
}
```

---

## 🔹 5. Declarative vs Imperative

- **Imperative**: *How* to do something (step by step).
- **Declarative (FP)**: *What* should happen.

👉 Example: Find sum of squares of odd numbers from 1–10

**Imperative style:**

```
java

int sum = 0;
for(int i=1; i<=10; i++) {
    int square = i*i;
    if(square % 2 != 0) {
        sum += square;
    }
}
System.out.println(sum);
```

**Functional style (Streams):**

```
java

int sum = java.util.stream.IntStream.rangeClosed(1, 10)
                .map(n -> n * n)           // square
                .filter(n -> n % 2 != 0)   // odd only
                .sum();

System.out.println(sum);
```

---

## 🔹 6. Why Functions in FP Are Powerful

- They allow **abstraction** (you don't care about loops, indexes, etc.).
- They make code **predictable** (pure + immutable = fewer bugs).
- They enable **parallelism** (since no shared mutable state).

- They make code **composable** (combine small functions to build big behavior).

---

✨ In short:
Functional programming is about **building software by combining pure, reusable functions** instead of changing state with commands. Functions are not just helpers — they are the *core building blocks*.

---

Would you like me to also show you **how OOP-style Java code (with classes, setters, loops)** can be refactored into a **pure functional style with Streams and functions**, so you can see the contrast clearly?