



Functional Interfaces & Functional Programming in Java

Functional Interfaces & Functional Programming in Java

1. What is Functional Programming (FP)?

Functional Programming is a **programming paradigm** where:

- Computation is treated as the **evaluation of mathematical functions**.
- Focus is on **what to do** rather than **how to do it** (declarative vs imperative).
- Avoids **mutable state** and **side effects** (though in Java, it's "hybrid FP" since Java isn't purely functional).
- Functions are **first-class citizens** → they can be stored in variables, passed as arguments, and returned from methods.

👉 It's called *functional* because the **function itself** becomes the **primary unit of computation** instead of classes or objects.

2. Functional Interfaces

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

They are the backbone of FP in Java because **lambdas** and **method references** rely on them.

📌 Examples:

- `Runnable` → `void run()`
- `Callable<T>` → `T call()`
- `Comparator<T>` → `int compare(T o1, T o2)`
- `Supplier<T>`, `Consumer<T>`, `Function<T,R>`, `Predicate<T>`

Custom Example

```

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

public class Demo {
    public static void main(String[] args) {
        // Using Lambda
        Calculator add = (a, b) -> a + b;
        Calculator mul = (a, b) -> a * b;

        System.out.println(add.operate(5, 3)); // 8
        System.out.println(mul.operate(5, 3)); // 15
    }
}

```

3. Key Java 8+ Functional Features

Lambdas

- Anonymous functions, concise replacement for inner classes.

```
list.forEach(item -> System.out.println(item));
```

Method References

- Shortcuts to call methods/lambdas.

```
list.forEach(System.out::println);
```

Streams API

- Declarative, functional-style processing of collections.

```

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

```

```
.sum();
```

✅ Optional

- A container to avoid `null` checks.

```
Optional<String> name = Optional.of("Hardik");  
name.ifPresent(System.out::println);
```

4. 💠 Advantages of Functional Programming in Java

- **Cleaner & Concise Code** → fewer lines than traditional loops/boilerplate.
- **Parallel Processing** → Streams can be parallelized (`parallelStream()`).
- **Immutability** → reduces bugs from unintended state changes.
- **Declarative Style** → focus on *what* rather than *how*.
- **Reusability & Testability** → pure functions are easy to test.

5. 💠 When to Use

✅ Good Use Cases:

- Data processing pipelines (filtering, mapping, reducing).
- Transforming collections/arrays.
- Event handling & callbacks (`Runnable`, `ActionListener`).
- Concurrent programming (`CompletableFuture`, `parallelStream`).
- Lightweight operations where immutability is preferred.

❌ When NOT to Use:

- Performance-critical loops where object creation overhead matters (lambdas may create extra objects).
- Complex recursive algorithms (Java lacks proper tail recursion optimization).
- Situations where imperative style is more readable for the team.
- If the operation inherently requires mutability (e.g., in-place updates to large arrays).

6. 💠 Data Structures with Functional Programming

Functional style fits best with **immutable and collection-based data structures**.

- **Collections Framework** (`List`, `Set`, `Map`) → work seamlessly with **Streams**.
 - **Arrays** → can also be used with `Arrays.stream()`.
 - **Concurrent Collections** (`ConcurrentHashMap`) → work with FP operations safely.
 - **Immutable Collections** (`List.of()`, `Set.of()` introduced in Java 9) → align well with FP.
 - **Optional** → safe wrapper instead of null values.
-

7. ♦ Putting It All Together

Example: Find top 3 highest-paid employees in a department using functional style:

```
List<Employee> employees = ...;

List<Employee> top3 = employees.stream()
    .filter(e -> e.getDepartment().equals("IT"))
    .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
    .limit(3)
    .toList();
```

Declarative, expressive, parallelizable.

8. ♦ Summary

- **Functional Programming** = programming with functions as first-class citizens, avoiding mutation.
 - **Functional Interfaces** = 1 abstract method, basis of lambdas/method references.
 - **Features** = Streams, Lambdas, Method References, Optional.
 - **Advantages** = concise, immutable, parallelizable, testable.
 - **When to Use** = data transformations, pipelines, callbacks.
 - **When NOT to Use** = heavy mutability, performance-sensitive loops, overly complex cases.
 - **Data Structures** = Collections, Arrays, Optionals, Immutable DS.
-

👉 Would you like me to also make this into a **visual Markdown note** (with diagrams/flow) for quick revision, like a cheat sheet?

You said:

give in more detail like method reference different interfaces in functional programming/interfaces different methods like accept, supplier, consumer interface, etc and all their methods with examples

ChatGPT said:

Perfect 👍 You want the **complete detailed breakdown of functional interfaces** in Java — including

`Supplier`, `Consumer`, `Function`, `Predicate`, method references, default methods like `accept()`, `get()`, `apply()`, `test()`, and their variations.

Let's go step by step.

Functional Interfaces & Functional Programming in Java (Detailed Guide)

1. Functional Interfaces Recap

- A **Functional Interface** = an interface with exactly **1 abstract method**.
 - Can have **default** and **static** methods too.
 - Annotated with `@FunctionalInterface` (not mandatory, but helps enforce the rule).
 - Used as the target for **Lambdas** and **Method References**.
-

2. Built-in Functional Interfaces (`java.util.function`)

Java 8 introduced a whole set of functional interfaces under `java.util.function`.

Here are the **main categories**:

(A) Consumer Family – takes input, returns nothing

- `Consumer<T>` → `void accept(T t)`

```
Consumer<String> printer = s -> System.out.println("Hello " + s);
printer.accept("World"); // Hello World
```

- `BiConsumer<T,U>` → `void accept(T t, U u)`

```
BiConsumer<String, Integer> display = (name, age) ->
    System.out.println(name + " is " + age + " years old");
display.accept("Hardik", 25);
```

- `IntConsumer`, `LongConsumer`, `DoubleConsumer` → primitives.
-

◆ (B) Supplier Family – provides value, no input

- **Supplier<T>** → `T get()`

```
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

- Useful for lazy loading, caching, generating values.
-

◆ (C) Function Family – takes input, returns output

- **Function<T,R>** → `R apply(T t)`

```
Function<String, Integer> length = s -> s.length();
System.out.println(length.apply("Java")); // 4
```

- **BiFunction<T,U,R>** → two inputs, one output

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
System.out.println(add.apply(10, 20)); // 30
```

- **UnaryOperator<T>** → special case of Function: $T \rightarrow T$

```
UnaryOperator<Integer> square = x -> x * x;
System.out.println(square.apply(5)); // 25
```

- **BinaryOperator<T>** → special case of BiFunction: $(T, T) \rightarrow T$

```
BinaryOperator<Integer> max = Integer::max;
System.out.println(max.apply(10, 20)); // 20
```

◆ (D) Predicate Family – tests condition, returns boolean

- `Predicate<T>` → `boolean test(T t)`

```
Predicate<Integer> isEven = n -> n % 2 == 0;  
System.out.println(isEven.test(10)); // true
```

- `BiPredicate<T,U>` → `boolean test(T t, U u)`

```
BiPredicate<String, Integer> longerThan = (s, n) -> s.length() > n;  
System.out.println(longerThan.test("Hardik", 4)); // true
```

3. ◆ Method References

A **method reference** is a shorthand for a lambda that just calls a method.

Types:

1. Static Method Reference

`ClassName::staticMethod`

```
Function<String, Integer> parser = Integer::parseInt;  
System.out.println(parser.apply("100")); // 100
```

1. Instance Method Reference of a Particular Object

`instance::instanceMethod`

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

```
printer.accept("Hello!");
```

1. Instance Method Reference of an Arbitrary Object of a Type

`ClassName::instanceMethod`

```
Function<String, String> upper = String::toUpperCase;  
System.out.println(upper.apply("java")); // JAVA
```

1. Constructor Reference

`ClassName::new`

```
Supplier<List<String>> listSupplier = ArrayList::new;  
List<String> list = listSupplier.get();
```

4. ♦ Streams + Functional Interfaces

Streams rely heavily on these interfaces:

- `filter(Predicate<T>)`
- `map(Function<T,R>)`
- `forEach(Consumer<T>)`
- `collect(Collector<T,A,R>)`
- `reduce(BinaryOperator<T>)`

Example:

```
List<String> names = Arrays.asList("John", "Maya", "Hardik");  
  
names.stream()  
    .filter(s -> s.startsWith("H")) // Predicate  
    .map(String::toUpperCase)       // Function  
    .forEach(System.out::println);  // Consumer
```

5. ♦ Optional (also functional style)

```
Optional<String> name = Optional.ofNullable("Hardik");
name.ifPresent(System.out::println);
```

- Uses **Consumer** (`ifPresent`).
 - `orElseGet(Supplier<T>)`.
 - `map(Function<T,R>)`.
-

6. ♦ Advantages of Using Functional Interfaces

- ✓ Write less boilerplate (inner classes → lambdas/method refs).
 - ✓ Encourages immutability & functional style.
 - ✓ Enables **parallelism** with Streams.
 - ✓ Improves readability (declarative → “what” not “how”).
 - ✓ Easy integration with **existing APIs** like Collections, Concurrency (`CompletableFuture`).
-

7. ♦ When and When NOT to Use

Use Functional Interfaces when:

- Defining single-method contracts (e.g., callbacks, event handling).
- Writing code that is transformation-heavy (like pipelines).
- You want testable, composable logic.

Avoid Functional Style when:

- You need complex state mutation (better with OOP).
 - Performance-critical hot-loops where lambda allocation overhead matters.
 - Recursive-heavy logic (Java doesn't optimize recursion).
-

8. ♦ Data Structures with Functional Interfaces

- **Collections** (List, Set, Map) → best with Streams & Lambdas.
- **ConcurrentHashMap** → supports `forEach`, `computeIfAbsent`, `merge` using functional interfaces.
- **Immutable Collections** (Java 9+) → (`List.of()`, `Map.of()`) → align well with FP.

- **Arrays** → work with `Arrays.stream()` and functional APIs.
 - **Optional** → designed for functional chaining.
-

9. Complete Example

```
import java.util.*;import java.util.function.*;
public class FunctionalDemo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Maya", "Hardik", "Meera");

        // Predicate
        Predicate<String> startsWithM = s -> s.startsWith("M");

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

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

        // Supplier
        Supplier<Date> now = Date::new;

        names.stream()
            .filter(startsWithM)    // Predicate
            .map(length)            // Function
            .forEach(System.out::println); // Consumer

        System.out.println("Current Date: " + now.get());
    }
}
```

Output:

```
4
5
Current Date: Fri Sep 12 11:50:00 IST 2025
```

✅ Now you have **all functional interfaces explained with methods + examples + when/when not to use**.
Would you like me to create a **cheat sheet Markdown table** listing **all main functional interfaces** (Predicate,

Function, Supplier, Consumer, BiX versions, Operators) with their **abstract method name, signature, and example** for quick revision?