# 16. 람다와 함수형 프로그래밍

## 람다식 기본 문법

자바의 **람다식(lambda expression)**은 **익명 함수(anonymous function)**를 간결하게 표현하는 문법이다.
Java 8부터 도입되었고, **함수형 인터페이스(Functional Interface)**와 함께 사용된다.

---

### 🟦 1. 기본 문법 구조

```
(매개변수) -> { 실행문 }
```

#### 📌 예시

```
(int x, int y) -> { return x + y; }
(x, y) -> x + y  // 자료형 생략, return 생략 가능
() -> System.out.println("Hello")  // 매개변수 없음
s -> s.length()  // 매개변수 1개일 땐 괄호 생략 가능
```

---

### 🟧 2. 예제: Runnable vs 람다식

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello from thread");
    }
};

Runnable r2 = () -> System.out.println("Hello from lambda");

new Thread(r1).start();
new Thread(r2).start();
```

---

### 🟩 3. 람다식의 필수 조건: 함수형 인터페이스

- 추상 메서드가 **단 하나**인 인터페이스
- 예시: `Runnable`, `Callable`, `Comparator`, `Function`, `Consumer`, `Predicate` 등
- 사용자 정의 시 `@FunctionalInterface` 어노테이션 사용 권장

```
@FunctionalInterface
interface MyPrinter {
    void print(String message);
}
```

람다식 적용:

```
MyPrinter printer = msg -> System.out.println(">> " + msg);
printer.print("Hello");
```

---

## 🟥 4. 다양한 형태 예시

| 형태 | 예시 |
|---|---|
| 매개변수 없음 | `() -> System.out.println("Hi")` |
| 매개변수 1개 | `x -> x * x` |
| 매개변수 2개 | `(x, y) -> x + y` |
| 반환이 있는 문장 | `(x, y) -> { return x + y; }` |
| 타입 명시 | `(int x, int y) -> x * y` |

---

## 🧠 자주 쓰는 내장 함수형 인터페이스 + 람다 예시

| 인터페이스 | 메서드 시그니처 | 예시 |
|---|---|---|
| `Function<T,R>` | `R apply(T t)` | `x -> x + 1` |
| `Consumer<T>` | `void accept(T t)` | `s -> System.out.println(s)` |
| `Supplier<T>` | `T get()` | `() -> "hello"` |
| `Predicate<T>` | `boolean test(T t)` | `x -> x > 0` |

---

## 🔁 Stream과 함께

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

names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(n -> System.out.println(n));
```

---

## 🧩 람다 vs 익명 클래스

| 항목 | 익명 클래스 | 람다 |
|---|---|---|
| 길이 | 길고 중복 많음 | 간결 |
| `this` 참조 | 내부 클래스 자신 | 외부 객체를 참조 |
| 직렬화 | 별도 클래스 | 더 가볍고 빠름 |

# 함수형 인터페이스 (`@FunctionalInterface`)

함수형 인터페이스(Functional Interface)는 **추상 메서드를 하나만 가지는 인터페이스**를 말한다.
자바의 **람다식**은 이 인터페이스와 반드시 함께 써야 한다.
Java 8부터 `@FunctionalInterface`라는 어노테이션도 제공된다.

---

## 🟦 1. 정의 및 예시

```
@FunctionalInterface
interface MyFunction {
    int apply(int x);
}
```

이제 람다식으로 구현 가능:

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

---

## 🟧 2. `@FunctionalInterface` 어노테이션

- 이 어노테이션은 **컴파일러에게 함수형 인터페이스임을 명시적으로 알림**
- 만약 추상 메서드가 2개 이상이면 **컴파일 에러** 발생

```
@FunctionalInterface
interface Broken {
    void doSomething();
    void doAnother();  // ❌ 컴파일 에러
}
```

- 선택적(default, static) 메서드는 허용됨

```
@FunctionalInterface
interface Sayable {
    void say(String msg);

    default void log() {
        System.out.println("default log");
    }

    static void version() {
        System.out.println("v1.0");
    }
}
```

---

## 🟩 3. 대표적인 자바 내장 함수형 인터페이스

| 인터페이스 | 추상 메서드 | 설명 |
|---|---|---|
| `Runnable` | `void run()` | 인자 없고 반환값 없음 |
| `Callable<T>` | `T call()` | 인자 없고 반환값 있음 (예외 가능) |
| `Function<T, R>` | `R apply(T t)` | 입력 T → 출력 R |
| `Consumer<T>` | `void accept(T t)` | 입력만 있고 반환 없음 |
| `Supplier<T>` | `T get()` | 입력 없음, 값 반환 |
| `Predicate<T>` | `boolean test(T t)` | 조건 검사용 boolean 반환 |
| `BiFunction<T,U,R>` | `R apply(T, U)` | 두 개 입력, 하나 출력 |
| `BiConsumer<T,U>` | `void accept(T,U)` | 두 개 입력, 반환 없음 |
| `UnaryOperator<T>` | `T apply(T t)` | 동일 타입 입력 → 출력 |
| `BinaryOperator<T>` | `T apply(T t1, T t2)` | 동일 타입 두 개 입력 → 출력 |

## 🟥 4. 실전 예시

```
Function<String, Integer> strLength = s -> s.length();
System.out.println(strLength.apply("Hello")); // 5

Predicate<Integer> isPositive = x -> x > 0;
System.out.println(isPositive.test(-3)); // false

Consumer<String> printer = msg -> System.out.println(">> " + msg);
printer.accept("Hi there!");

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

## 🧠 주의 사항

- 람다는 함수형 인터페이스에만 사용할 수 있음
- 하나의 추상 메서드만 있어야 진짜 함수형
- `@FunctionalInterface`는 생략 가능하지만, 실수를 막기 위해 **적극 권장**

# 주요 함수형 인터페이스 (`Function`, `Consumer`, `Supplier`, `Predicate`)

## 🟦 1. `Function<T, R>`

### 🧩 구조

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

### 📌 설명

- 하나의 인자를 받아서 **변환 결과를 반환**
- 매핑, 계산, 변형 등에 사용

### 🧪 예제

```
Function<String, Integer> strLen = s -> s.length();
System.out.println(strLen.apply("hello")); // 5
```

---

## 🟩 2. `Consumer<T>`

### 🧩 구조

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

### 📌 설명

- 하나의 인자를 받고 **결과를 반환하지 않음**
- 주로 **출력, 저장, 처리 등 부수효과(side effect)**를 낼 때 사용

### 🧪 예제

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

---

## 🟧 3. `Supplier<T>`

### 🧩 구조

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

### 📌 설명

- 인자를 받지 않고 값을 **생성하거나 제공**
- **지연된 연산(lazy evaluation)**, **무작위값** 제공 등에 자주 사용

### 🧪 예제

```
Supplier<Double> random = () -> Math.random();
System.out.println(random.get()); // 0.123456789 (예시)
```

---

## 🟥 4. `Predicate<T>`

### 🧩 구조

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

### 📌 설명

- 하나의 인자를 받아서 **조건에 따라 true/false 반환**
- 필터링, 검증, 조건 분기 등에 자주 사용

### 🧪 예제

```
Predicate<Integer> isEven = x -> x % 2 == 0;
System.out.println(isEven.test(4)); // true
System.out.println(isEven.test(7)); // false
```

---

## 🔁 함수형 인터페이스 조합 메서드

각 인터페이스는 연쇄적으로 결합할 수 있는 디폴트 메서드를 지원함:

| 인터페이스 | 결합 메서드 |
| --- | --- |
| `Function` | `andThen()`, `compose()` |

| 인터페이스 | 결합 메서드 |
| --- | --- |
| `Predicate` | `and()`, `or()`, `negate()` |
| `Consumer` | `andThen()` |

```
Function<String, String> trim = s -> s.trim();
Function<String, String> upper = s -> s.toUpperCase();
Function<String, String> combined = trim.andThen(upper);

System.out.println(combined.apply("   java ")); // "JAVA"
```

```
Predicate<Integer> isPositive = x -> x > 0;
Predicate<Integer> isEven = x -> x % 2 == 0;
Predicate<Integer> positiveEven = isPositive.and(isEven);

System.out.println(positiveEven.test(4)); // true
System.out.println(positiveEven.test(-4)); // false
```

---

## 💡 정리 요약

| 인터페이스 | 메서드 | 입력 | 반환 | 주요 용도 |
| --- | --- | --- | --- | --- |
| `Function<T, R>` | `R apply(T)` | 있음 | 있음 | 변환, 매핑 |
| `Consumer<T>` | `void accept(T)` | 있음 | 없음 | 출력, 처리 |
| `Supplier<T>` | `T get()` | 없음 | 있음 | 값 제공, 생성 |
| `Predicate<T>` | `boolean test(T)` | 있음 | boolean | 조건 검사 |

# 메서드 참조 (`::`)

자바의 **메서드 참조(Method Reference)**는 람다식을 더 간결하게 표현하는 문법이다.
람다에서 **단순히 메서드만 호출하는 경우**, `::` 문법으로 대체할 수 있다.
`클래스이름::메서드이름`, `참조변수::메서드이름` 형식이 대표적이다.

## 🟦 1. 기본 문법 종류

| 유형 | 예시 | 의미 |
| --- | --- | --- |
| 정적 메서드 참조 | `ClassName::staticMethod` | 정적 메서드를 참조 |
| 특정 객체의 인스턴스 메서드 | `instance::methodName` | 이미 존재하는 객체의 메서드 참조 |
| 임의 객체의 인스턴스 메서드 | `ClassName::methodName` | 첫 번째 매개변수를 객체로 간주 |
| 생성자 참조 | `ClassName::new` | 생성자 참조 |

---

## 🟧 2. 예시: 정적 메서드 참조

```
Function<String, Integer> parser = Integer::parseInt;
System.out.println(parser.apply("123")); // 123
```

람다식으로 쓰면:

```
Function<String, Integer> parser = s -> Integer.parseInt(s);
```

---

## 🟩 3. 예시: 인스턴스 메서드 참조

```
Consumer<String> printer = System.out::println;
printer.accept("Hello"); // Hello
```

이건 다음과 같아:

```
Consumer<String> printer = s -> System.out.println(s);
```

---

## 🟥 4. 임의 객체의 인스턴스 메서드 참조

```
List<String> list = List.of("b", "a", "c");
list.sort(String::compareTo); // compareTo가 내부에서 쓰임
System.out.println(list); // [a, b, c]
```

비교 대상 객체를 첫 번째 인자로 보는 방식이야.

---

## 💔 5. 생성자 참조

```
Supplier<ArrayList<String>> listMaker = ArrayList::new;
ArrayList<String> newList = listMaker.get();
```

이건 이렇게도 가능해:

```
Supplier<ArrayList<String>> listMaker = () -> new ArrayList<>();
```

---

## 🧠 메서드 참조와 람다의 관계

| 람다식 | 메서드 참조 |
| --- | --- |
| `s -> s.toUpperCase()` | `String::toUpperCase` |
| `x -> Math.abs(x)` | `Math::abs` |

| 람다식 | 메서드 참조 |
| --- | --- |
| `() -> new HashMap<>()` | `HashMap::new` |
| `(s1, s2) -> s1.compareTo(s2)` | `String::compareTo` |

---

## 🔁 실전 예: Stream과 함께

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

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

// 메서드 참조
names.forEach(System.out::println);
```

---

## 💡 Tip: 어느 경우에 쓰는 게 좋은가?

- 람다식이 너무 짧고 메서드 하나만 호출 → `::`로 치환
- 코드가 읽기 쉬워짐
- 하지만 로직이 복잡한 람다는 람다식 그대로 쓰는 게 낫다

# Stream API와 함께 사용

Stream API는 Java 8에서 도입된 **데이터 처리용 파이프라인 구조**이고,
**람다식**과 **함수형 인터페이스 (`Function`, `Predicate`, `Consumer`, `Supplier`)**를
결합해 **선언적 방식**으로 데이터를 다룰 수 있게 해준다.

## 🟦 Stream API 구조 요약

```
collection.stream()
          .중간연산1
          .중간연산2
          ...
          .최종연산;
```

- **중간 연산 (intermediate)**: `filter`, `map`, `sorted`, `distinct` 등 → *Stream 반환*
- **최종 연산 (terminal)**: `forEach`, `collect`, `reduce`, `count`, `anyMatch` 등 → *값 반환*

---

## 🟧 주요 연산별 함수형 인터페이스와의 관계

| 연산 | 설명 | 함수형 인터페이스 |
| --- | --- | --- |
| `filter(Predicate<T>)` | 조건에 맞는 요소만 통과 | `Predicate<T>` |

| 연산 | 설명 | 함수형 인터페이스 |
| --- | --- | --- |
| `map(Function<T, R>)` | 값을 변환하여 새로운 Stream 생성 | `Function<T, R>` |
| `forEach(Consumer<T>)` | 각 요소에 대해 연산 수행 | `Consumer<T>` |
| `collect(...)` | Stream을 결과로 모음 | `Collector<T, A, R>` |
| `reduce(...)` | 요소들을 누적 계산 | `BinaryOperator<T>` |
| `sorted(...)` | 요소 정렬 | `Comparator<T>` |

---

## 🟩 예제 1: filter + map + forEach

```
List<String> names = List.of("Kim", "Lee", "Park", "Kang");

names.stream()
     .filter(name -> name.startsWith("K"))                // Predicate<String>
     .map(String::toUpperCase)                            // Function<String, String>
     .forEach(System.out::println);                       // Consumer<String>
```

📌 출력:

```
KIM
KANG
```

---

## 🟧 예제 2: collect + Comparator

```
List<Integer> numbers = List.of(3, 6, 1, 9, 2);

List<Integer> sorted = numbers.stream()
                              .filter(n -> n % 2 == 0)           // Predicate
                              .sorted(Comparator.reverseOrder()) // Comparator
                              .collect(Collectors.toList());     // Collector

System.out.println(sorted); // [6, 2]
```

---

## 🟥 예제 3: reduce로 합계 구하기

```
List<Integer> nums = List.of(1, 2, 3, 4, 5);

int sum = nums.stream()
              .reduce(0, (a, b) -> a + b);  // BinaryOperator<Integer>

System.out.println(sum); // 15
```

---

## 🟦 예제 4: Stream + 람다 + 메서드 참조

```
List<String> langs = List.of("Java", "Python", "Go", "Rust");

langs.stream()
     .filter(s -> s.length() <= 4)
     .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);
```

📌 출력:

```
GO
JAVA
```

---

## 🧠 고급: `groupingBy`, `partitioningBy`, `flatMap` 등도 활용 가능

```
Map<Boolean, List<Integer>> evenOdd = List.of(1,2,3,4,5,6)
    .stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));

System.out.println(evenOdd);
// {false=[1, 3, 5], true=[2, 4, 6]}
```

---

## 🧩 정리 요약표

| 연산 | 람다식 사용 예 | 메서드 참조 예 |
|---|---|---|
| `filter` | `x -> x > 10` | `Objects::nonNull` |
| `map` | `x -> x.toUpperCase()` | `String::toUpperCase` |
| `forEach` | `x -> System.out.println(x)` | `System.out::println` |
| `sorted` | `(a, b) -> b - a` | `Comparator.reverseOrder()` |
| `collect` | `Collectors.toList()` | — |