3. 람다 표현식 - 람다 어디서 들어봤는데...

🚀 Java 8 람다 표현식 완벽 가이드

> **람다 (Lambda)**
> 이름 없는 함수, 즉 **익명 함수**를 의미한다.
> 주로 짧고 간단한 함수를 한 줄로 정의할 때 사용된다.

📌 1. 람다란 무엇인가?

"람다 표현식"은 메소드로 전달할 수 있는 익명 함수를 단순화한 것이다. 람다 표현식에는 이름이 없지만 **파라미터 리스트**, **바디**, **반환 형식**, **발생할 수 있는 예외 리스트**를 가질 수 있다.

✨ 람다의 특징

특징	설명
익명	이름이 없어 익명이라 표현한다
함수	특정 클래스에 종속되지 않으므로 함수라고 부른다
전달	람다 표현식을 메소드 인수로 전달하거나 변수로 저장할 수 있다
간결성	익명 클래스처럼 많은 보일러플레이트 코드가 필요 없다

🔄 기존 코드 vs 람다 표현식

```
// 기존 방식 (익명 클래스)
Comparator&lt;Apple&gt; byWeight = new Comparator&lt;Apple&gt;() {
    public int compare(Apple a1, Apple a2) {
        return a1.getWeight().compareTo(a2.getWeight());
    }
};

// 람다 표현식 사용
Comparator&lt;Apple&gt; byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

// 더 간결하게 (타입 추론 활용)
Comparator&lt;Apple&gt; byWeight =
    (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

> 💡 **코드가 훨씬 간결해지고 가독성이 향상된다!**

📖 2. 람다 표현식의 구조

람다 표현식은 세 부분으로 구성된다: **파라미터**, **화살표**, **바디**

```
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
     ↑                  ↑                      ↑
파라미터 리스트        화살표                 람다 바디
```

구성 요소 설명

- **파라미터 리스트**: `(Apple a1, Apple a2)` - Comparator의 compare 메소드 파라미터
- **화살표**: `->` - 람다의 파라미터 리스트와 바디를 구분한다
- **람다 바디**: `a1.getWeight().compareTo(a2.getWeight())` - 람다의 반환값에 해당하는 표현식

🎯 다양한 람다 표현식 예시

```java
// 1) String 형식 파라미터 하나, int 반환
(String s) -> s.length()

// 2) Apple 형식 파라미터 하나, boolean 반환
(Apple a) -> a.getWeight() > 150

// 3) int 형식 파라미터 2개, void 반환 (블록 스타일)
(int x, int y) -> {
    System.out.println("Result: ");
    System.out.println(x + y);
}

// 4) 파라미터 없음, int 42 반환
() -> 42

// 5) 파라미터 타입 추론
(a1, a2) -> a1.getWeight().compareTo(a2.getWeight())
```

📋 3. 람다 작성 규칙

1️⃣ 매개변수 하나인 경우

```java
// 괄호 생략 가능
x -> x + 1

// 명시적 타입 지정 시 괄호 필수
(Integer x) -> x + 1
```

2️⃣ 매개변수 여러 개인 경우

```java
// 표현식 스타일 (return 생략)
(x, y) -> x + y
```

```
// 블록 스타일 (return 필수)
(x, y) -> {
    int sum = x + y;
    return sum;
}
```

3️⃣ 타입 명시 규칙

```
// ✅ 올바른 예시 - 모든 파라미터 타입 명시
(Integer x, Integer y) -> x * y

// ❌ 잘못된 예시 - 일부만 타입 명시
(Integer x, y) -> x * y
```

4️⃣ 변수 캡처 (Variable Capture)

```
int base = 10;  // 사실상 final
Function&lt;Integer, Integer&gt; addBase = x -> x + base;

// ❌ 컴파일 에러 - base가 변경되면 안 됨
// base = 20;
```

> ⚠️ **주의**: 람다가 참조하는 지역 변수는 **final** 또는 **사실상 final**이어야 한다.

🎯 4. 어디에, 어떻게 람다를 사용할까?

람다 표현식은 **함수형 인터페이스**라는 문맥에서 사용할 수 있다.

💡 함수형 인터페이스란?

> **정확히 하나의 추상 메소드를 지정하는 인터페이스**

함수형 인터페이스는 디폴트 메소드와 정적 메소드를 가질 수 있지만, **추상 메소드는 오직 하나**여야 한다.

```
@FunctionalInterface
public interface Comparator&lt;T&gt; {
    int compare(T o1, T o2);  // 추상 메소드

    // 디폴트 메소드는 여러 개 가능
    default Comparator&lt;T&gt; reversed() {
        return Collections.reverseOrder(this);
    }

    // 정적 메소드도 가능
    static &lt;T&gt; Comparator&lt;T&gt; naturalOrder() {
        return (Comparator&lt;T&gt;) Comparators.NaturalOrderComparator.INSTANCE;
```

```
    }
}
```

📚 5. 주요 함수형 인터페이스

Java 8은 `java.util.function` 패키지에 다양한 함수형 인터페이스를 제공한다.

1️⃣ Runnable - 인수와 반환값이 없는 동작

```
@FunctionalInterface
public interface Runnable {
    void run();
}

// 사용 예시
Runnable task = () -> System.out.println("Hello Lambda!");
new Thread(task).start();
```

2️⃣ Consumer<T> - T를 소비하고 반환값이 없음

```
@FunctionalInterface
public interface Consumer&lt;T&gt; {
    void accept(T t);
}

// 사용 예시
Consumer&lt;String&gt; printer = s -> System.out.println(s);
printer.accept("Hello World");

// 리스트의 모든 요소 출력
List&lt;Integer&gt; numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(n -> System.out.println(n));
```

3️⃣ Supplier<T> - 인수 없이 T를 공급

```
@FunctionalInterface
public interface Supplier&lt;T&gt; {
    T get();
}

// 사용 예시
Supplier&lt;Double&gt; randomGenerator = () -> Math.random();
System.out.println(randomGenerator.get());

// 지연 평가에 활용
Supplier&lt;List&lt;String&gt;&gt; listSupplier = () -> {
    System.out.println("Creating list...");
```

```
    return Arrays.asList("A", "B", "C");
};
```

4 Function<T, R> - T를 인수로 받아 R을 반환

```
@FunctionalInterface
public interface Function&lt;T, R&gt; {
    R apply(T t);
}

// 사용 예시
Function&lt;String, Integer&gt; strLength = s -> s.length();
System.out.println(strLength.apply("Lambda")); // 6

// 문자열을 정수로 변환
Function&lt;String, Integer&gt; stringToInt = Integer::parseInt;
```

5 Predicate<T> - T를 인수로 받아 boolean 반환

```
@FunctionalInterface
public interface Predicate&lt;T&gt; {
    boolean test(T t);
}

// 사용 예시
Predicate&lt;Integer&gt; isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // true

// 문자열 길이 검사
Predicate&lt;String&gt; isEmpty = String::isEmpty;
```

6 BiFunction<T, U, R> - 두 인수를 받아 결과 반환

```
@FunctionalInterface
public interface BiFunction&lt;T, U, R&gt; {
    R apply(T t, U u);
}

// 사용 예시
BiFunction&lt;Integer, Integer, Integer&gt; add = (a, b) -> a + b;
System.out.println(add.apply(3, 4)); // 7

// 문자열 결합
BiFunction&lt;String, String, String&gt; concat = String::concat;
```

7 BinaryOperator<T> - 같은 타입의 두 인수를 받아 같은 타입 반환

```
@FunctionalInterface
public interface BinaryOperator&lt;T&gt; extends BiFunction&lt;T, T, T&gt; {
    // BiFunction을 상속받아 특화된 인터페이스
}

// 사용 예시
BinaryOperator&lt;Integer&gt; multiply = (a, b) -> a * b;
System.out.println(multiply.apply(3, 4)); // 12

// 두 수 중 큰 값 반환
BinaryOperator&lt;Integer&gt; max = Integer::max;
```

📊 함수형 인터페이스 요약표

함수형 인터페이스	함수 디스크립터	기본형 특화
`Predicate<T>`	`T -> boolean`	`IntPredicate`, `LongPredicate`, `DoublePredicate`
`Consumer<T>`	`T -> void`	`IntConsumer`, `LongConsumer`, `DoubleConsumer`
`Function<T, R>`	`T -> R`	`IntFunction<R>`, `IntToDoubleFunction`, `IntToLongFunction`
`Supplier<T>`	`() -> T`	`BooleanSupplier`, `IntSupplier`, `LongSupplier`
`UnaryOperator<T>`	`T -> T`	`IntUnaryOperator`, `LongUnaryOperator`, `DoubleUnaryOperator`
`BinaryOperator<T>`	`(T, T) -> T`	`IntBinaryOperator`, `LongBinaryOperator`, `DoubleBinaryOperator`
`BiPredicate<L, R>`	`(L, R) -> boolean`	-
`BiConsumer<T, U>`	`(T, U) -> void`	`ObjIntConsumer<T>`, `ObjLongConsumer<T>`
`BiFunction<T, U, R>`	`(T, U) -> R`	`ToIntBiFunction<T, U>`, `ToLongBiFunction<T, U>`

🎯 기본형 특화 인터페이스를 사용하는 이유

Java의 제네릭은 참조형만 사용할 수 있어, 기본형을 사용하려면 박싱(boxing)이 필요하다. 이는 성능 저하를 일으킬 수 있다.

```
// 박싱이 발생하는 경우
Function&lt;Integer, Integer&gt; square = x -> x * x;  // Integer 객체 생성

// 기본형 특화 - 박싱 없음
IntUnaryOperator squarePrimitive = x -> x * x;  // int 그대로 사용
```

🔍 @FunctionalInterface 어노테이션

```java
@FunctionalInterface
public interface MyFunctionalInterface {
    void doSomething();

    // 디폴트 메소드는 여러 개 가능
    default void doSomethingElse() {
        System.out.println("Default implementation");
    }

    // 정적 메소드도 가능
    static void utility() {
        System.out.println("Utility method");
    }
}
```

@FunctionalInterface 사용 이유:

1. **컴파일 타임 검증** - 추상 메소드가 정확히 하나인지 검증
2. **문서화** - 이 인터페이스가 함수형 인터페이스임을 명시
3. **실수 방지** - 실수로 추상 메소드를 추가하는 것을 방지

🔄 6. 람다 활용: 실행 어라운드 패턴

자원 처리에 사용하는 순환 패턴은 자원을 열고, 처리한 다음, 자원을 닫는 순서로 이루어진다. 설정과 정리 과정은 대부분 비슷하다.

```
┌──────────────────────┐
│   초기화/준비 코드     │
├──────────────────────┤
│       작업 수행       │  ← 변하는 부분
├──────────────────────┤
│   정리/마무리 코드     │
└──────────────────────┘
```

🎯 실행 어라운드 패턴 구현

1단계: 동작 파라미터화를 추가

```java
// 한 줄만 읽기
String result = processFile((BufferedReader br) -> br.readLine());

// 두 줄 읽기
String result = processFile((BufferedReader br) -> br.readLine() +
br.readLine());
```

2단계: 함수형 인터페이스를 이용해서 동작 전달

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

3단계: 동작 실행

```
public String processFile(BufferedReaderProcessor p) throws IOException {
    try (BufferedReader br =
              new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);  // 동작 실행
    }
}
```

4단계: 람다 전달

```
// 한 줄 읽기
String oneLine = processFile((BufferedReader br) -> br.readLine());

// 두 줄 읽기
String twoLines = processFile((BufferedReader br) -> br.readLine() +
br.readLine());

// 가장 긴 줄 찾기
String longest = processFile((BufferedReader br) -> {
    String longest = "";
    String line;
    while ((line = br.readLine()) != null) {
        if (line.length() > longest.length()) {
            longest = line;
        }
    }
    return longest;
});
```

🔍 7. 형식 검사와 형식 추론

📋 형식 검사 과정

컴파일러는 람다가 사용되는 콘텍스트를 이용해서 람다의 형식을 추론한다.

```
List&lt;Apple&gt; heavierThan150g =
    filter(inventory, (Apple apple) -> apple.getWeight() > 150);
```

형식 검사 과정:

1. `filter` 메소드 선언 확인
2. `filter` 메소드는 두 번째 파라미터로 `Predicate<Apple>` 형식 기대
3. `Predicate<Apple>`은 `test`라는 한 개의 추상 메소드 정의
4. `test` 메소드는 `Apple`을 받아 `boolean`을 반환
5. 람다의 시그니처와 일치하는지 확인

🎯 같은 람다, 다른 함수형 인터페이스

하나의 람다 표현식을 다양한 함수형 인터페이스에 사용할 수 있다.

```
// Comparator&lt;Apple&gt;
Comparator&lt;Apple&gt; c1 = (Apple a1, Apple a2) ->
a1.getWeight().compareTo(a2.getWeight());

// BiFunction&lt;Apple, Apple, Integer&gt;
BiFunction&lt;Apple, Apple, Integer&gt; c2 = (Apple a1, Apple a2) ->
a1.getWeight().compareTo(a2.getWeight());

// 람다 바디가 호환되는 경우
Callable&lt;Integer&gt; c = () -> 42;
Supplier&lt;Integer&gt; s = () -> 42;
```

📊 형식 추론

컴파일러는 람다 표현식의 파라미터 형식에 접근할 수 있으므로 람다 문법에서 이를 생략할 수 있다.

```
// 형식을 명시적으로 지정
Comparator&lt;Apple&gt; c =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

// 형식 추론 활용
Comparator&lt;Apple&gt; c =
    (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());

// 더 간결하게
List&lt;Apple&gt; greenApples =
    filter(inventory, apple -> GREEN.equals(apple.getColor()));
```

🔒 지역변수 사용과 제약

람다는 자유 변수(파라미터로 넘겨진 변수가 아닌 외부에서 정의된 변수)를 활용할 수 있다. 이를 **람다 캡처링**이라고 한다.

```
int portNumber = 1337;  // 사실상 final
Runnable r = () -> System.out.println(portNumber);
```

```
// ❌ 컴파일 에러 - 람다에서 참조하는 변수는 final이어야 함
// portNumber = 31337;
```

제약 사항의 이유:

- 인스턴스 변수는 **힙**에 저장된다
- 지역 변수는 **스택**에 위치한다
- 람다에서 지역 변수에 바로 접근할 수 있다면 람다가 스레드에서 실행될 때 변수를 할당한 스레드가 사라져서 변수 할당이 해제되었을 수도 있다
- 따라서 자바는 자유 지역 변수의 복사본을 제공한다
- 복사본의 값이 바뀌지 않아야 하므로 지역 변수에는 한 번만 값을 할당해야 한다

🔗 8. 메소드 참조

메소드 참조를 이용하면 기존의 메소드 정의를 재활용해서 람다처럼 전달할 수 있다.

🔄 람다 vs 메소드 참조

```
// 람다 표현식
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));

// 메소드 참조
inventory.sort(comparing(Apple::getWeight));

// 더 많은 예시
map.merge(key, 1, (count, incr) -> count + incr);  // 람다
map.merge(key, 1, Integer::sum);                    // 메소드 참조
```

📋 메소드 참조 유형

1️⃣ 정적 메소드 참조

```
// 람다
Function&lt;String, Integer&gt; stringToInteger =
    (String s) -> Integer.parseInt(s);

// 메소드 참조
Function&lt;String, Integer&gt; stringToInteger = Integer::parseInt;
```

2️⃣ 다양한 형식의 인스턴스 메소드 참조

```
// 람다
BiPredicate&lt;List&lt;String&gt;, String&gt; contains =
    (list, element) -> list.contains(element);
```

```
// 메소드 참조
BiPredicate&lt;List&lt;String&gt;, String&gt; contains = List::contains;
```

3️⃣ 기존 객체의 인스턴스 메소드 참조

```
// 람다
Predicate&lt;String&gt; startsWithNumber =
    (String string) -> this.startsWithNumber(string);

// 메소드 참조
Predicate&lt;String&gt; startsWithNumber = this::startsWithNumber;
```

🏗️ 생성자 참조

```
// 기본 생성자
Supplier&lt;Apple&gt; c1 = Apple::new;
Apple a1 = c1.get();  // new Apple() 호출

// 인수가 있는 생성자
Function&lt;Integer, Apple&gt; c2 = Apple::new;  // Apple(Integer weight)
Apple a2 = c2.apply(110);

// 두 인수를 갖는 생성자
BiFunction&lt;Color, Integer, Apple&gt; c3 = Apple::new;  // Apple(Color,
Integer)
Apple a3 = c3.apply(GREEN, 110);

// 임의의 인수를 갖는 생성자
// 직접 함수형 인터페이스를 만들어야 함
public interface TriFunction&lt;T, U, V, R&gt; {
    R apply(T t, U u, V v);
}

TriFunction&lt;Integer, Integer, Integer, Color&gt; colorFactory = Color::new;
```

⚡ 9. 람다 표현식 조합하기

Java 8의 함수형 인터페이스는 다양한 유틸리티 메소드를 포함한다. 이를 이용해 람다 표현식을 조합할 수 있다.

🔀 Comparator 조합

```
// 역정렬
inventory.sort(comparing(Apple::getWeight).reversed());
```

```
// Comparator 연결
inventory.sort(comparing(Apple::getWeight)
    .reversed()  // 무게를 내림차순으로 정렬
    .thenComparing(Apple::getCountry));  // 무게가 같으면 국가별로 정렬
```

🎯 Predicate 조합

```
// 기본 프레디케이트
Predicate&lt;Apple&gt; redApple = apple -> RED.equals(apple.getColor());

// negate - 특정 프레디케이트를 반전
Predicate&lt;Apple&gt; notRedApple = redApple.negate();

// and - 두 프레디케이트를 연결해서 AND 조건
Predicate&lt;Apple&gt; redAndHeavyApple =
    redApple.and(apple -> apple.getWeight() > 150);

// or - OR 조건
Predicate&lt;Apple&gt; redAndHeavyAppleOrGreen =
    redApple.and(apple -> apple.getWeight() > 150)
            .or(apple -> GREEN.equals(apple.getColor()));
```

> 💡 **우선순위**: `a.or(b).and(c)`는 `(a || b) && c`와 같다

🔄 Function 조합

```
Function&lt;Integer, Integer&gt; f = x -> x + 1;
Function&lt;Integer, Integer&gt; g = x -> x * 2;

// andThen - f를 먼저 적용하고 그 결과에 g를 적용
Function&lt;Integer, Integer&gt; h = f.andThen(g);  // g(f(x))
int result = h.apply(1);  // 4 (2 * 2)

// compose - g를 먼저 적용하고 그 결과에 f를 적용
Function&lt;Integer, Integer&gt; h = f.compose(g);  // f(g(x))
int result = h.apply(1);  // 3 (2 + 1)
```

📧 실용적인 예제: 편지 작성

```
public class Letter {
    public static String addHeader(String text) {
        return "From Raoul, Mario and Alan: " + text;
    }

    public static String addFooter(String text) {
        return text + " Kind regards";
    }
```

```
    public static String checkSpelling(String text) {
        return text.replaceAll("labda", "lambda");
    }
}

// Function 조합으로 변환 파이프라인 생성
Function&lt;String, String&gt; addHeader = Letter::addHeader;
Function&lt;String, String&gt; transformationPipeline =
    addHeader.andThen(Letter::checkSpelling)
             .andThen(Letter::addFooter);

String result = transformationPipeline.apply("labda is awesome!");
// "From Raoul, Mario and Alan: lambda is awesome! Kind regards"
```

🎯 10. 실전 예제: 정렬 코드 개선

코드 전달에서 람다 표현식까지의 진화 과정을 살펴보자.

1단계: 코드 전달

```
public class AppleComparator implements Comparator&lt;Apple&gt; {
    public int compare(Apple a1, Apple a2) {
        return a1.getWeight().compareTo(a2.getWeight());
    }
}

inventory.sort(new AppleComparator());
```

2단계: 익명 클래스 사용

```
inventory.sort(new Comparator&lt;Apple&gt;() {
    public int compare(Apple a1, Apple a2) {
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

3단계: 람다 표현식 사용

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));

// 또는
inventory.sort(comparing(apple -> apple.getWeight()));
```

4단계: 메소드 참조 사용

```
inventory.sort(comparing(Apple::getWeight));
```

🚨 11. 예외 처리

함수형 인터페이스는 확인된 예외를 던지는 동작을 허용하지 않는다. 예외를 던지는 람다 표현식을 만들려면 다음과 같은 방법을 사용한다.

방법 1: 확인된 예외를 선언하는 함수형 인터페이스 정의

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

방법 2: 람다를 try-catch 블록으로 감싸기

```
Function&lt;BufferedReader, String&gt; f = (BufferedReader b) -> {
    try {
        return b.readLine();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
};
```

🎯 12. 마무리

✅ 핵심 정리

- **람다 표현식**은 익명 함수의 일종으로 간결한 코드 구현이 가능하다
- **함수형 인터페이스**는 하나의 추상 메소드만 정의하는 인터페이스이다
- 람다 표현식 전체가 **함수형 인터페이스의 인스턴스**로 취급된다
- **메소드 참조**를 이용하면 기존 메소드 구현을 재사용하고 직접 전달할 수 있다
- `Comparator`, `Predicate`, `Function` 등의 함수형 인터페이스는 람다 표현식을 조합할 수 있는 **디폴트 메소드**를 제공한다

🚀 람다가 유용한 상황

- 컬렉션 정렬
- 필터링 조건 전달
- 매핑/변환 작업
- 행동 파라미터화

> 💡 **람다 표현식을 마스터하면 더 함수형 프로그래밍에 가까운 자바 코드를 작성할 수 있다!**

📚 참고 자료

- **Modern Java In Action**
- 옮긴이: 우정은
- 펴낸이: 전태호
- 지은이: 라울-게이브리얼 우르마, 마리오 푸스코, 앨런 마이크로프트

🌊 다음 포스팅에서는 스트림 API에 대해 알아보자!