3. 함수형 인터페이스

#1.인강/0.자바/7.자바-고급3편

함수형 인터페이스와 제네릭1

이번 시간부터는 함수형 인터페이스에 대해 더 자세히 알아보자.

함수형 인터페이스도 인터페이스이기 때문에 제네릭을 도입할 수 있다.
먼저 함수형 인터페이스에 제네릭이 필요한 이유를 알아보자.

각각 다른 타입 사용

다음 코드는 문자 타입, 숫자 타입을 각각 처리하는 두 개의 함수형 인터페이스를 사용한다.

```
package lambda.lambda3;

public class GenericMain1 {

    public static void main(String[] args) {
        StringFunction upperCase = s -> s.toUpperCase();
        String result1 = upperCase.apply("hello");
        System.out.println("result1 = " + result1);

        NumberFunction square = n -> n * n;
        Integer result2 = square.apply(3);
        System.out.println("result2 = " + result2);
    }

    @FunctionalInterface
```

```
    interface StringFunction {
        String apply(String s);
    }

    @FunctionalInterface
    interface NumberFunction {
        Integer apply(Integer s);
    }
}
```

실행 결과

```
result1 = HELLO
result2 = 9
```

`StringFunction`이 제공하는 `apply` 메서드와 `NumberFunction`이 제공하는 `apply` 메서드는 둘다 하나의 인자를 입력받고, 결과를 반환한다. 다만 입력받는 타입과 반환 타입이 다를 뿐이다. 이렇게 매개변수나 반환 타입이 다를 때마다 계속 함수형 인터페이스를 만들어야 할까?

Object 타입으로 합치기

`Object`는 모든 타입의 부모이다. 따라서 다형성(다형적 참조)를 사용해서 이 문제를 간단히 해결할 수 있을 것 같다.

```
package lambda.lambda3;

public class GenericMain2 {

    public static void main(String[] args) {
        ObjectFunction upperCase = s -> ((String)s).toUpperCase();
        String result1 = (String) upperCase.apply("hello");
        System.out.println("result1 = " + result1);

        ObjectFunction square = n -> (Integer) n * (Integer) n;
        Integer result2 = (Integer) square.apply(3);
        System.out.println("result2 = " + result2);
    }
```

```
    @FunctionalInterface
    interface ObjectFunction {
        Object apply(Object s);
    }

}
```

- 메서드가 `Object`를 매개변수로 사용하고, `Object`를 반환하면 모든 타입을 입력 받고, 또 모든 타입을 반환할 수 있다. 따라서 이전과 같이 타입에 따라 각각 다른 함수형 인터페이스를 만들지 않아도 된다. 따라서 앞서 각각 만든 함수형 인터페이스 2개를 1개로 합칠 수 있다.
- 물론 `Object`를 사용하기 때문에 복잡하고 안전하지 않은 캐스팅 과정이 필요하다.
- 실행 결과는 기존과 같다.

코드를 이해하기 쉽게 익명 클래스로 변경해보자.

```
package lambda.lambda3;

public class GenericMain3 {

    public static void main(String[] args) {
        ObjectFunction upperCase = new ObjectFunction() {
            @Override
            public Object apply(Object s) {
                return ((String)s).toUpperCase();
            }
        };
        String result1 = (String) upperCase.apply("hello");
        System.out.println("result1 = " + result1);

        ObjectFunction square = new ObjectFunction() {
            @Override
            public Object apply(Object n) {
                return (Integer) n * (Integer) n;
            }
        };
        Integer result2 = (Integer) square.apply(3);
        System.out.println("result2 = " + result2);
    }

    @FunctionalInterface
    interface ObjectFunction {
        Object apply(Object s);
```

```
    }

}
```

실행 결과는 기존과 같다.

정리

`Object`와 다형성을 활용한 덕분에 코드의 중복을 제거하고, 재사용성을 늘리게 되었다. 하지만 `Object`를 사용하므로 다운 캐스팅을 해야 하고, 결과적으로 타입 안전성 문제가 발생한다.

지금까지 개발한 프로그램은 코드 재사용과 타입 안전성이라는 2마리 토끼를 한번에 잡을 수 없다. 코드 재사용을 늘리기 위해 `Object`와 다형성을 사용하면 타입 안전성이 떨어지는 문제가 발생한다.

- `StringFunction`, `NumberFunction` 각각의 타입별로 함수형 인터페이스를 모두 정의
  - 코드 재사용X
  - 타입 안전성O
- `ObjectFunction`를 사용해서 Object의 다형성을 활용해서 하나의 함수형 인터페이스만 정의
  - 코드 재사용O
  - 타입 안전성X

함수형 인터페이스와 제네릭2

제네릭 도입

이제 함수형 인터페이스에 제네릭을 도입해서 코드 재사용도 늘리고, 타입 안전성까지 높여보자.

참고로 제네릭에 대한 내용은 중급2편에서 충분히 다루었기 때문에 이해가 어렵다면 중급2편을 다시 복습하자.

```
package lambda.lambda3;

public class GenericMain4 {

    public static void main(String[] args) {
        GenericFunction<String, String> upperCase = new GenericFunction<>() {
            @Override
            public String apply(String s) {
                return s.toUpperCase();
            }
        };
```

```
        String result1 = upperCase.apply("hello");
        System.out.println("result1 = " + result1);

        GenericFunction<Integer, Integer> square = new GenericFunction<>() {
            @Override
            public Integer apply(Integer n) {
                return n * n;
            }
        };
        Integer result2 = square.apply(3);
        System.out.println("result2 = " + result2);
    }

    @FunctionalInterface
    interface GenericFunction<T, R> {
        R apply(T s);
    }

}
```

- 실행 결과는 기존과 같다.

`ObjectFunction` ➜ `GenericFunction`으로 변경했다.

```
@FunctionalInterface
interface GenericFunction<T, R> {
    R apply(T s);
}
```

- `T`: 매개변수의 타입
- `R`: 반환 타입

함수형 인터페이스에 제네릭을 도입한 덕분에 메서드 `apply()`의 매개변수와 반환 타입을 유연하게 변경할 수 있다.

```
GenericFunction<String, String> upperCase = new GenericFunction<>() {
    @Override
    public String apply(String s) {
        return s.toUpperCase();
    }
};
```

- `upperCase`는 `String`을 입력받아서 `String`을 반환한다.
- 따라서 `GenericFunction<String, String>`과 같이 사용하면 된다.

```
GenericFunction<Integer, Integer> square = new GenericFunction<>() {
    @Override
    public Integer apply(Integer n) {
        return n * n;
    }
};
```

- `square`는 `Integer`을 입력받아서 `Integer`을 반환한다.
- 따라서 `GenericFunction<Integer, Integer>`과 같이 사용하면 된다.

제네릭과 람다

앞서 만든 익명 클래스를 이제 람다로 변경해보자.

```
package lambda.lambda3;

public class GenericMain5 {

    public static void main(String[] args) {
        GenericFunction<String, String> upperCase = s -> s.toUpperCase();
        String result1 = upperCase.apply("hello");
        System.out.println("result1 = " + result1);

        GenericFunction<Integer, Integer> square = n -> n * n;
        Integer result2 = square.apply(3);
        System.out.println("result2 = " + result2);
    }

    @FunctionalInterface
    interface GenericFunction<T, R> {
        R apply(T s);
    }

}
```

- 실행 결과는 기존과 같다.
- 익명 클래스를 람다로 변경했다.

`GenericFunction`은 **매개변수가 1개이고, 반환값이 있는 모든 람다에 사용**할 수 있다.
매개변수의 타입과 반환값은 사용시점에 제네릭을 활용해서 얼마든지 변경할 수 있기 때문이다.
제네릭이 도입된 함수형 인터페이스는 재사용성이 매우 높다.

제네릭이 도입된 함수형 인터페이스의 활용

```
package lambda.lambda3;

public class GenericMain6 {

    public static void main(String[] args) {
        GenericFunction<String, String> toUpperCase = str ->
str.toUpperCase();
        GenericFunction<String, Integer> stringLength = str -> str.length();
        GenericFunction<Integer, Integer> square = x -> x * x;
        GenericFunction<Integer, Boolean> isEven = num -> num % 2 == 0;

        System.out.println(toUpperCase.apply("hello"));
        System.out.println(stringLength.apply("hello"));
        System.out.println(square.apply(3));
        System.out.println(isEven.apply(3));
    }

    @FunctionalInterface
    interface GenericFunction<T, R> {
        R apply(T s);
    }

}
```

실행 결과

```
HELLO
5
9
false
```

정리

- 제네릭을 사용하면 동일한 구조의 함수형 인터페이스를 다양한 타입에 재사용할 수 있다.
- 예제에서는 문자열을 대문자로 변환하기, 문자열의 길이 구하기, 숫자의 제곱 구하기, 짝수 여부 확인하기 등 서로 다른 기능들을 하나의 함수형 인터페이스로 구현했다.
- `T`는 입력 타입을, `R`은 반환 타입을 나타내며, 실제 사용할 때 구체적인 타입을 지정하면 된다.
- 이렇게 제네릭을 활용하면 타입 안정성을 보장하면서도 유연한 코드를 작성할 수 있다.
- 컴파일 시점에 타입 체크가 이루어지므로 런타임 에러를 방지할 수 있다.
- 제네릭을 사용하지 않았다면 각각의 경우에 대해 별도의 함수형 인터페이스를 만들어야 했을 것이다.
- 이는 코드의 중복을 줄이고 유지보수성을 높이는데 큰 도움이 된다.

람다와 타겟 타입

남은 문제

우리가 만든 `GenericFunction`은 코드 중복을 줄이고 유지보수성을 높여주지만 2가지 문제가 있다.

문제 1. 모든 개발자들이 비슷한 함수형 인터페이스를 개발해야 한다.

우리가 만든 `GenericFunction`은, 매개변수가 1개이고, 반환값이 있는 모든 람다에 사용할 수 있다. 그런데 람다를 사용하려면 함수형 인터페이스가 필수이기 때문에 전 세계 개발자들 모두 비슷하게 `GenericFunction`을 각각 만들어서 사용해야 한다. 그리고 비슷한 모양의 `GenericFunction`이 많이 만들어질 것이다.

문제 2. 개발자A가 만든 함수형 인터페이스와 개발자B가 만든 함수형 인터페이스는 서로 호환되지 않는다.

이 문제는 다음 코드를 통해 확인해보자.

```
package lambda.lambda3;

public class TargetType1 {
    public static void main(String[] args) {
        // 람다 직접 대입: 문제 없음
        FunctionA functionA = i -> "value = " + i;
        FunctionB functionB = i -> "value = " + i;

        // 이미 만들어진 FunctionA 인스턴스를 FunctionB에 대입: 불가능
        //FunctionB targetB = functionA;  // 컴파일 에러!
    }
```

```
    @FunctionalInterface
    interface FunctionA {
        String apply(Integer i);
    }

    @FunctionalInterface
    interface FunctionB {
        String apply(Integer i);
    }
}
```

람다를 함수형 인터페이스에 대입할 때는 `FunctionA`, `FunctionB` 모두 메서드 시그니처가 맞으므로 문제없이 잘 대입된다.

`FunctionB targetB = functionA` 부분은 컴파일 오류가 발생한다.
두 인터페이스 모두 `Integer`를 받아 `String`을 리턴하는 동일한 `apply(...)` 메서드를 갖고 있지만, **자바 타입 시스템상 전혀 다른 인터페이스**이므로 서로 호환되지 않는다.

이 부분을 좀 더 자세히 알아보자.

람다와 타겟 타입

람다는 그 자체만으로는 구체적인 타입이 정해져 있지 않고, **타겟 타입(target type)**이라고 불리는 맥락(대입되는 참조형)에 의해 타입이 결정된다.

```
FunctionA functionA = i -> "value = " + i;
```

- 이 코드에서 `i -> "value = " + i`라는 람다는 `FunctionA`라는 타겟 타입을 만나서 비로소 `FunctionA` 타입으로 결정된다.

```
FunctionB functionB = i -> "value = " + i;
```

- 동일한 람다라도 이런 코드가 있었다면, 똑같은 람다가 이번에는 `FunctionB` 타입으로 타겟팅되어 유효하게 컴파일된다.

정리하면 람다는 그 자체만으로는 구체적인 타입이 정해져 있지 않고, 대입되는 함수형 인터페이스(타겟 타입)에 의해

비로소 타입이 결정된다.

이렇게 타입이 결정되고 나면 이후에는 다른 타입에 대입하는 것이 불가능하다. 이후 함수형 인터페이스를 다른 함수형 인터페이스에 대입하는 것은 타입이 서로 다르기 때문에, 메서드의 시그니처가 같아도 대입이 되지 않는다.

```
FunctionB targetB = functionA;  // 컴파일 에러!
```

위 코드를 보면, `functionA`는 분명 `FunctionA` 타입의 변수가 이미 된 상태이다. 즉, `FunctionA`라는 "명시적인 인터페이스 타입"을 가진 객체가 되어 있다. 그런데 이 객체를 `FunctionB` 타입에 대입하려고 할 때, 자바 컴파일러는 `FunctionA`와 `FunctionB`가 서로 다른 타입임을 명확히 인식한다. 쉽게 이야기해서 `FunctionA`와 `FunctionB`는 서로 타입이 다르다. 따라서 대입이 불가능하다. 이것은 마치 `Integer`에 `String`을 대입하는 것과 같다.

두 인터페이스가 시그니처가 같고 똑같은 모양의 함수형 인터페이스라도, 타입 자체는 별개이므로 상호 대입은 허용되지 않는다.

정리

- **람다**는 익명 함수로서 특정 타입을 가지지 않고, 대입되는 참조 변수가 어떤 함수형 인터페이스를 가리키느냐에 따라 타입이 결정된다.
- 한편 **이미 대입된 변수**(`functionA`)는 엄연히 `FunctionA` 타입의 객체가 되었으므로, 이를 `FunctionB` 참조 변수에 그대로 대입할 수는 없다. 두 인터페이스 이름이 다르기 때문에 자바 컴파일러는 다른 타입으로 간주한다.
- 따라서 시그니처가 똑같은 함수형 인터페이스라도, 타입이 다르면 상호 대입이 되지 않는 것이 자바의 타입 시스템 규칙이다.

자바가 기본으로 제공하는 함수형 인터페이스

자바는 이런 문제들을 해결하기 위해 필요한 함수형 인터페이스 대부분을 기본으로 제공한다.
자바가 제공하는 함수형 인터페이스를 사용하면, 비슷한 함수형 인터페이스를 불필요하게 만드는 문제는 물론이고, 함수형 인터페이스의 호환성 문제까지 해결할 수 있다.

Function - 자바 기본 제공

```
package java.util.function;

@FunctionalInterface
```

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

- 자바는 `java.util.function` 패키지에 다양한 기본 함수형 인터페이스들을 제공한다.

```
package lambda.lambda3;

import java.util.function.Function;

// 자바가 기본으로 제공하는 Function 사용
public class TargetType2 {

    public static void main(String[] args) {
        Function<String, String> upperCase = s -> s.toUpperCase();
        String result1 = upperCase.apply("hello");
        System.out.println("result1 = " + result1);

        Function<Integer, Integer> square = n -> n * n;
        Integer result2 = square.apply(3);
        System.out.println("result2 = " + result2);
    }

}
```

- 실행 결과는 기존과 같다.

같은 타입을 사용하므로 대입도 문제 없다.

```
package lambda.lambda3;

import java.util.function.Function;

// 자바가 기본으로 제공하는 Function 대입
public class TargetType3 {

    public static void main(String[] args) {
        // 람다 직접 대입: 문제 없음
        Function<Integer, String> functionA = i -> "value = " + i;
        System.out.println(functionA.apply(10));
```

```
        Function<Integer, String> functionB = functionA;
        System.out.println(functionB.apply(20));
    }
}
```

실행 결과

```
value = 10
value = 20
```

따라서 자바가 기본으로 제공하는 함수형 인터페이스를 사용하자

그럼 본격적으로 자바가 제공하는 다양한 함수형 인터페이스들을 알아보자

기본 함수형 인터페이스

자바가 기본으로 제공하는 가장 대표적인 함수형 인터페이스는 다음과 같다.

자바가 제공하는 대표적인 함수형 인터페이스

- `Function`: 입력O, 반환O
- `Consumer`: 입력O, 반환X
- `Supplier`: 입력X, 반환O
- `Runnable`: 입력X, 반환X

함수형 인터페이스들은 대부분 제네릭을 활용하므로 종류가 많을 필요는 없다.
함수형 인터페이스는 대부분 `java.util.function` 패키지에 위치한다. (`Runnable`은 `java.lang` 패키지에 위치)

하나씩 자세히 알아보자.

Function

핵심 코드만 적어두었다. 앞으로 자바가 제공하는 함수형 인터페이스를 소개할 때는 간략하게 핵심 코드만 적어두겠다.

```java
package java.util.function;

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

- 하나의 매개변수를 받고, 결과를 반환하는 함수형 인터페이스이다. (둘 이상의 매개변수를 받는 함수형 인터페이스는 뒤에서 설명한다.)
- 입력값(`T`)을 받아서 다른 타입의 출력값(`R`)을 반환하는 연산을 표현할 때 사용한다. 물론 같은 타입의 출력 값도 가능하다.
- 일반적인 함수(Function)의 개념에 가장 가깝다.
- 예: 문자열을 받아서 정수로 변환, 객체를 받아서 특정 필드 추출 등

용어 설명

- Function은 수학적인 "함수" 개념을 그대로 반영한 이름이다.
- apply는 "적용하다"라는 의미로, 입력값에 함수를 적용해서 결과를 얻는다는 수학적 개념을 표현한다.
- 예: f(x)처럼 입력 x에 함수 f를 적용(apply)하여 결과를 얻는다.

```java
package lambda.lambda4;

import java.util.function.Function;

public class FunctionMain {

    public static void main(String[] args) {
        // 익명 클래스
        Function<String, Integer> function1 = new Function<>() {
            @Override
            public Integer apply(String s) {
                return s.length();
            }
        };
        System.out.println("function1 = " + function1.apply("hello"));

        // 람다
        Function<String, Integer> function2 = s -> s.length();
        System.out.println("function2 = " + function2.apply("hello"));
    }
```

```
}
```

실행 결과

```
function1 = 5
function2 = 5
```

Consumer

```
package java.util.function;

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

- 입력 값(T)만 받고, 결과를 반환하지 않는(`void`) 연산을 수행하는 함수형 인터페이스이다.
- 입력값(T)을 받아서 처리하지만 결과를 반환하지 않는 연산을 표현할 때 사용한다.
- 입력 받은 데이터를 기반으로 내부적으로 처리만 하는 경우에 유용하다.
  - 예) 컬렉션에 값 추가, 콘솔 출력, 로그 작성, DB 저장 등

용어 설명

- Consumer는 "소비자"라는 의미로, 데이터를 받아서 소비(사용)만 하고 아무것도 돌려주지 않는다는 개념을 표현한다.
- accept는 "받아들이다"라는 의미로, 입력값을 받아들여서 처리한다는 동작을 설명한다.
- 예: 로그를 출력하는 consumer는 데이터를 받아서 출력만 하고 끝난다.
- 쉽게 이야기해서 입력 값을 받아서(accept) 소비(consume)해 버린다고 생각하면 된다.

```
package lambda.lambda4;

import java.util.function.Consumer;

public class ConsumerMain {

    public static void main(String[] args) {
```

```
        // 익명 클래스
        Consumer<String> consumer1 = new Consumer<>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        };
        consumer1.accept("hello consumer");

        // 람다
        Consumer<String> consumer2 = (s) -> System.out.println(s);
        consumer2.accept("hello consumer");
    }
}
```

실행 결과

```
hello consumer
hello consumer
```

Supplier

```
package java.util.function;

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

- 입력을 받지 않고(`()`) 어떤 데이터를 공급(supply)해주는 함수형 인터페이스이다.
- 객체나 값 생성, 지연 초기화 등에 주로 사용된다. (지연 초기화는 뒤에서 설명)

용어 설명

- Supplier는 "공급자"라는 의미로, 요청할 때마다 값을 공급해주는 역할을 한다.
- get은 "얻다"라는 의미로, supplier로부터 값을 얻어온다는 개념을 표현한다.
- 예: 랜덤 값을 제공하는 supplier는 호출할 때마다 새로운 랜덤 값을 공급한다.

```java
package lambda.lambda4;

import java.util.Random;
import java.util.function.Supplier;

public class SupplierMain {

    public static void main(String[] args) {
        // 익명 클래스
        Supplier<Integer> supplier1 = new Supplier<Integer>() {
            @Override
            public Integer get() {
                return new Random().nextInt(10);
            }
        };
        System.out.println("supplier1.get() = " + supplier1.get());

        // 람다
        Supplier<Integer> supplier2 = () -> new Random().nextInt(10);
        System.out.println("supplier2.get() = " + supplier2.get());
    }
}
```

실행 결과

```
supplier1.get() = 9
supplier2.get() = 2
```

- 결과가 랜덤으로 나타난다.

Runnable

```java
package java.lang;

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

- 입력값도 없고 반환값도 없는 함수형 인터페이스이다. 자바에서는 원래부터 스레드 실행을 위한 인터페이스로 쓰였지만, 자바 8 이후에는 람다식으로도 많이 표현된다. 자바8로 업데이트 되면서 `@FunctionalInterface` 애노테이션도 붙었다.
- `java.lang` 패키지에 있다. 자바의 경우 원래부터 있던 인터페이스는 하위 호환을 위해 그대로 유지한다.
- 주로 멀티스레딩에서 스레드의 작업을 정의할 때 사용한다.
- 입력값도 없고, 반환값도 없는 함수형 인터페이스가 필요할 때 사용한다.

```
package lambda.lambda4;

public class RunnableMain {
    public static void main(String[] args) {

        Runnable runnable1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello Runnable");
            }
        };
        runnable1.run();

        Runnable runnable2 = () -> System.out.println("Hello Runnable");
        runnable2.run();
    }
}
```

실행 결과

```
Hello Runnable
Hello Runnable
```

요약

인터페이스	메서드 시그니처	입력	출력	대표 사용 예시
Function<T, R>	R apply(T t)	1개 (T)	1개 (R)	데이터 변환, 필드 추출 등

Consumer<T>	void accept(T t)	1개 (T)	없음	로그 출력, DB 저장 등
Supplier<T>	T get()	없음	1개 (T)	객체 생성, 값 반환 등
Runnable	void run()	없음	없음	스레드 실행(멀티스레드)

특화 함수형 인터페이스

특화 함수형 인터페이스는 의도를 명확하게 만든 조금 특별한 함수형 인터페이스다.

- `Predicate`: 입력O, 반환 `boolean`
  - 조건 검사, 필터링 용도
- `Operator` (`UnaryOperator`, `BinaryOperator`): 입력O, 반환O
  - 동일한 타입의 연산 수행, 입력과 같은 타입을 반환하는 연산 용도

Predicate

```
package java.util.function;

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

- 입력 값(T)을 받아서 `true` 또는 `false`로 구분(판단)하는 함수형 인터페이스이다.
- 조건 검사, 필터링 등의 용도로 많이 사용된다(뒤에서 설명할 스트림 API에서 필터 조건을 지정할 때 자주 등장한다)

용어 설명

- Predicate는 수학/논리학에서 "술어"를 의미하며, 참/거짓을 판별하는 명제를 표현한다.
  - 술어: 어떤 대상의 성질이나 관계를 설명하면서, 그 설명이 참인지 거짓인지를 판단할 수 있게 해주는 표현
- test는 "시험하다"라는 의미로, 주어진 입력값이 조건을 만족하는지 테스트한다는 의미이다. 그래서 반환값이 `boolean`이다.
- 예: 숫자가 짝수인지 테스트하는 predicate는 조건 충족 여부를 판단한다.

```java
package lambda.lambda4;

import java.util.function.Predicate;

public class PredicateMain {

    public static void main(String[] args) {
        Predicate<Integer> predicate1 = new Predicate<>() {
            @Override
            public boolean test(Integer value) {
                return value % 2 == 0;
            }
        };
        System.out.println("predicate1.test(10) = " + predicate1.test(10));

        Predicate<Integer> predicate2 = value -> value % 2 == 0;
        System.out.println("predicate2.test(10) = " + predicate2.test(10));
    }
}
```

실행 결과

```
predicate1.test(10) = true
predicate2.test(10) = true
```

Predicate가 꼭 필요할까?

`Predicate`는 입력이 `T`, 반환이 `boolean`이기 때문에 결과적으로 `Function<T, Boolean>`으로 대체할 수 있다. 그럼에도 불구하고 `Predicate`를 별도로 만든 이유는 다음과 같다.

```java
Function<Integer, Boolean> f1 = value -> value % 2 == 0;
Predicate<Integer> f1 = value -> value % 2 == 0;
```

`Predicate<T>`는 "입력 값을 받아 `true/false`로 결과를 판단한다"라는 **의도를 명시적으로 드러내기 위해 정의**

된 함수형 인터페이스이다.
물론 "`boolean`을 반환하는 함수"라는 측면에서 보면 `Function<T, Boolean>`으로도 충분히 구현할 수 있다. 하지만 `Predicate<T>`를 별도로 둠으로써 다음과 같은 이점들을 얻을 수 있다.

1. **의미의 명확성**
    - `Predicate<T>`를 사용하면 "이 함수는 조건을 검사하거나 필터링 용도로 쓰인다"라는 **의도가 더 분명**해진다.
    - `Function<T, Boolean>`을 쓰면 "이 함수는 무언가를 계산해 `Boolean`을 반환한다"라고 볼 수도 있지만, "조건 검사"라는 목적이 분명히 드러나지 않을 수 있다.
2. **가독성 및 유지보수성**
    - 여러 사람과 협업하는 프로젝트에서, "조건을 판단하는 함수"는 `Predicate<T>`라는 패턴을 사용함으로써 의미 전달이 명확해진다.
    - `boolean` 판단 로직이 들어가는 부분에서 `Predicate<T>`를 사용하면 코드 가독성과 유지보수성이 향상된다.
        - 이름도 명시적이고, 제네릭에 `<Boolean>`을 적지 않아도 된다.

정리하면 `Function<T, Boolean>`로도 같은 기능을 구현할 수는 있지만, **목적(조건 검사)과 용도(필터링 등)에 대해 더 분명히 표현하고, 가독성과 유지보수를 위해** `Predicate<T>`라는 별도의 함수형 인터페이스가 마련되었다.

의도가 가장 중요한 핵심

자바가 제공하는 다양한 함수형 인터페이스들을 선택할 때는 단순히 입력값, 반환값만 보고 선택하는게 아니라 해당 함수형 인터페이스가 제공하는 **의도**가 중요하다. 예를 들어서 조건 검사, 필터링 등을 사용한다면 `Function`이 아니라 `Predicate`를 선택해야 한다. 그래야 다른 개발자가 "아~ 이 코드는 조건 검사 등에 사용할 의도가 있구나" 하고 코드를 더욱 쉽게 이해할 수 있다.

Operator

Operator는 `UnaryOperator`, `BinaryOperator` 2가지 종류가 제공된다.

용어 설명

- Operator라는 이름은 수학적인 연산자(Operator)의 개념에서 왔다.
- 수학에서 연산자는 보통 **같은 타입의 값들을 받아서 동일한 타입의 결과를 반환**한다.
    - 덧셈 연산자(+): `숫자 + 숫자` → `숫자`
    - 곱셈 연산자(*): `숫자 * 숫자` → `숫자`
    - 논리 연산자(AND): `boolean AND boolean` → `boolean`

- 자바에서는 수학처럼 숫자의 연산에만 사용된다기 보다는 입력과 반환이 동일한 타입의 연산에 사용할 수 있다. 예를 들어 문자를 입력해서 대문자로 바꾸어 반환하는 작업도 될 수 있다.

UnaryOperator(단항 연산)

```
package java.util.function;

@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    T apply(T t); // 실제 코드가 있지는 않음
}
```

- 단항 연산은 **하나의 피연산자(operand)**에 대해 연산을 수행하는 것을 말한다.
  - 예) 숫자의 부호 연산(`-x`), 논리 부정 연산(`!x`) 등
- 입력(피연산자)과 결과(연산 결과)가 **동일한 타입**인 연산을 수행할 때 사용한다.
  - 예) 숫자 5를 입력하고 그 수를 제곱한 결과를 반환한다.
  - 예) `String`을 입력받아 다시 `String`을 반환하면서, 내부적으로 문자열을 대문자로 바꾼다든지, 앞뒤에 추가 문자열을 붙이는 작업을 할 수 있다.
- `Function<T, T>`를 상속받는데, 입력과 반환을 모두 같은 `T`로 고정한다. 따라서 `UnaryOperator`는 입력과 반환 타입이 반드시 같아야 한다.

BinaryOperator(이항 연산)

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {
    T apply(T t1, T t2); // 실제 코드가 있지는 않음
}
```

- 이항 연산은 **두 개의 피연산자(operand)**에 대해 연산을 수행하는 것을 말한다.
  - 예: 두 수의 덧셈(`x + y`), 곱셈(`x * y`) 등
- **같은 타입**의 두 입력을 받아, **같은 타입**의 결과를 반환할 때 사용된다.
  - 예) `Integer` 두 개를 받아서 더한 값을 반환
  - 예) `Integer` 두 개를 받아서 둘 중에 더 큰 값을 반환
- `BiFunction<T,T,T>`를 상속받는 방식으로 구현되어 있는데, 입력값 2개와 반환을 모두 같은 T로 고정한다. 따라서 `BinaryOperator`는 모든 입력값과 반환 타입이 반드시 같아야 한다.
  - `BiFunction`은 입력 매개변수가 2개인 `Function`이다. 뒤에서 설명한다.

```java
package lambda.lambda4;

import java.util.function.BiFunction;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.UnaryOperator;

public class OperatorMain {

    public static void main(String[] args) {
        // UnaryOperator
        Function<Integer, Integer> square1 = x -> x * x;
        UnaryOperator<Integer> square2 = x -> x * x;
        System.out.println("square1: " + square1.apply(5));
        System.out.println("square2: " + square1.apply(5));

        // BinaryOperator
        BiFunction<Integer, Integer, Integer> addition1 = (a, b) -> a + b;
        BinaryOperator<Integer> addition2 = (a, b) -> a + b;

        System.out.println("addition1: " + addition1.apply(1, 2));
        System.out.println("addition2: " + addition2.apply(1, 2));

    }
}
```

- `BiFunction`은 입력 매개변수가 2개인 `Function`이다.

실행 결과

```
square1: 25
square2: 25
addition1: 3
addition2: 3
```

Operator를 제공하는 이유

`Function<T, R>`와 `BiFunction<T, U, R>`만으로도 사실상 거의 모든 함수형 연산을 구현할 수 있지만,

`UnaryOperator<T>`와 `BinaryOperator<T>`를 별도로 제공하는 이유는 다음과 같다.

```
Function<Integer, Integer> f1 = x -> x * x;
UnaryOperator<Integer> f2 = x -> x * x;

Function<Integer, Integer, Integer> f1 = (a, b) -> a + b;
BinaryOperator<Integer> f2 = (a, b) -> a + b;
```

1. **의도(목적)의 명시성**
    - `UnaryOperator<T>`는 입력과 출력 타입이 **동일**한 "단항 연산"을 수행한다는 것을 한눈에 보여준다.
        - 예: `x`의 제곱 결과를 반환
        - 예: `String`을 받아 `String`을 돌려주는 변환 작업
    - `BinaryOperator<T>`는 같은 타입을 **두 개** 입력받아 같은 타입을 결과로 반환하는 "이항 연산"을 수행한다는 것을 명확히 드러낸다.
        - 예: `Integer`와 `Integer`를 받아서 `Integer` 결과를 만드는 연산(최댓값 구하기, 덧셈 등)
    - 만약 모두 `Function<T, R>`나 `BiFunction<T, U, R>`만으로 처리한다면, "타입이 같은 연산"임을 코드만 보고 즉시 파악하기 조금 힘들다.
2. **가독성과 유지보수성**
    - 코드에 `UnaryOperator<T>`가 등장하면, "이건 단항 연산이구나"를 바로 알 수 있다.
    - `BinaryOperator<T>`의 경우도, "같은 타입 두 개를 받아 같은 타입으로 결과를 내는 연산"이라는 사실이 명확하게 전달된다.
    - 제네릭을 적는 코드의 양도 하나로 줄일 수 있다.
    - 여러 사람이 협업하는 프로젝트에서는 이런 명시성이 **코드 가독성**과 **유지보수성**에 큰 도움이 된다.

정리하면

- "단항 연산(입력 하나)"이고 **타입이 동일**하다면 `UnaryOperator<T>`를,
- "이항 연산(입력 두 개)"이고 **타입이 동일**하다면 `BinaryOperator<T>`를 쓰는 것이 개발자의 **의도**와 **로직**을 더 명확히 표현하고, **가독성**을 높일 수 있는 장점이 있다.

기타 함수형 인터페이스

입력 값이 2개 이상

매개변수가 2개 이상 필요한 경우에는 `BiXxx` 시리즈를 사용하면 된다. Bi는 Binary(이항, 둘)의 줄임말이다.

- 예) `BiFunction`, `BiConsumer`, `BiPredicate`

```
package lambda.lambda4;

import java.util.function.BiFunction;
import java.util.function.BiConsumer;
import java.util.function.BiPredicate;

public class BiMain {

    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
        System.out.println("Sum: " + add.apply(5, 10));

        BiConsumer<String, Integer> repeat = (c, n) -> {
            for (int i = 0; i < n; i++) {
                System.out.print(c);
            }
            System.out.println();
        };
        repeat.accept("*", 5);

        BiPredicate<Integer, Integer> isGreater = (a, b) -> a > b;
        System.out.println("isGreater: " + isGreater.test(10, 5));
    }
}
```

- 예제의 경우 타입이 같기 때문에 `BiFunction` 대신에 `BinaryOperator`를 사용하는 것이 더 나은 선택이다.
- `Supplier`는 매개변수가 없으므로 `BiSupplier`는 존재하지 않는다.

입력값이 3개라면?

입력값이 3개라면 `TriXxx`가 있으면 좋겠지만, 이런 함수형 인터페이스는 기본으로 제공하지 않는다. 보통 함수형 인터페이스를 사용할 때 3개 이상의 매개변수는 잘 사용하지 않기 때문이다.
만약 입력값이 3개일 경우라면 다음과 같이 직접 만들어서 사용하면 된다.

```
package lambda.lambda4;
```

```
public class TriMain {

    public static void main(String[] args) {
        TriFunction<Integer, Integer, Integer, Integer> triFunction =
                                                (a, b, c) -> a + b + c;
        System.out.println(triFunction.apply(1, 2, 3));
    }

    @FunctionalInterface
    interface TriFunction<A, B, C, R> {
        R apply(A a, B b, C c);
    }
}
```

기본형 지원 함수형 인터페이스

다음과 같이 기본형(primitive type)을 지원하는 함수형 인터페이스도 있다.

```
package java.util.function;

@FunctionalInterface
public interface IntFunction<R> {
    R apply(int value);
}
```

기본형 지원 함수형 인터페이스가 존재하는 이유

- 오토박싱/언박싱(auto-boxing/unboxing)으로 인한 성능 비용을 줄이기 위해
- 자바 제네릭의 한계(제네릭은 primitive 타입을 직접 다룰 수 없음)를 극복하기 위해
  - 자바의 제네릭은 기본형(primitive) 타입을 직접 다룰 수 없어서, `Function<int, R>` 같은 식으로는 선언할 수 없다.

```
package lambda.lambda4;

import java.util.function.*;
```

```java
public class PrimitiveFunction {

    public static void main(String[] args) {
        // 기본형 매개변수, IntFunction, LongFunction, DoubleFunction
        IntFunction<String> function = x -> "숫자: " + x;
        System.out.println("function.apply(100) = " + function.apply(100));

        // 기본형 반환, ToIntFunction, ToLongFunction, ToDoubleFunction
        ToIntFunction<String> toIntFunction = s -> s.length();
        System.out.println("toIntFunction = " +
toIntFunction.applyAsInt("hello"));

        // 기본형 매개변수, 기본형 반환
        IntToLongFunction intToLongFunction = x -> x * 100L;
        System.out.println("intToLongFunction = " +
intToLongFunction.applyAsLong(10));

        // IntUnaryOperator: int -> int
        IntUnaryOperator intUnaryOperator = x -> x * 100;
        System.out.println("intUnaryOperator = " +
intUnaryOperator.applyAsInt(10));

        // 기타, IntConsumer, IntSupplier, IntPredicate
    }
}
```

Function

- `IntFunction`은 매개변수가 기본형 `int`이다.
- `ToIntFunction`은 반환 타입이 기본형 `int`이다.
- `IntToLongFunction`은 매개변수가 `int`, 반환 타입이 `long`이다.
  - `IntToIntFunction`은 없는데, `IntOperator`를 사용하면 된다.

기타

- `IntOperator`: Operator는 매개변수와 반환 타입이 같다. 따라서 이 경우 `int` 입력, `int` 반환이다.
- `IntConsumer`: 매개변수만 존재한다. `int` 입력
- `IntSupplier`: 반환값만 존재한다. `int` 반환
- `IntPredicate`: 반환값은 `boolean`으로 고정이다. `int` 입력, `boolean` 반환

정리 - 함수형 인터페이스의 종류

기본 함수형 인터페이스

인터페이스	메서드 시그니처	입력	출력	대표 사용 예시
Function<T, R>	R apply(T t)	1개 (T)	1개 (R)	데이터 변환, 필드 추출 등
Consumer<T>	void accept(T t)	1개 (T)	없음	로그 출력, DB 저장 등
Supplier<T>	T get()	없음	1개 (T)	객체 생성, 값 반환 등
Runnable	void run()	없음	없음	스레드 실행(멀티스레드)

특화 함수형 인터페이스

인터페이스	메서드 시그니처	입력	출력	대표 사용 예시
Predicate<T>	boolean test(T t)	1개 (T)	boolean	조건 검사, 필터링
UnaryOperator<T>	T apply(T t)	1개 (T)	1개 (T; 입력과 동일)	단항 연산 (예: 문자열 변환, 단항 계산)
BinaryOperator<T>	T apply(T t1, T t2)	2개 (T, T)	1개 (T; 입력과 동일)	이항 연산 (예: 두 수의 합, 최댓값 반환)

- 자바가 기본으로 지원하지 않는다면 직접 만들어서 사용하자. 예(매개변수가 3개 이상)
- 기본형(primitive type)을 지원해야 한다면 `IntFunction` 등을 사용하면 된다.

문제와 풀이

- 앞서 람다에서 만든 문제를 자바가 제공하는 함수형 인터페이스로 대체하자.
- 다음 문제에 나오는 각 코드를 `lambda.ex3` 패키지로 복사해서 풀어보자.
- 만약 문제를 푸는 중에 필요하다면 함수형 인터페이스를 호출하는 메서드의 이름은 변경해도 된다.

문제1 - FilterExampleEx2

```
package lambda.ex2;

import java.util.ArrayList;
import java.util.List;
```

```
public class FilterExampleEx2 {

    // 고차 함수, 함수를 인자로 받아서 조건에 맞는 요소만 뽑아내는 filter
    public static List<Integer> filter(List<Integer> list, MyPredicate
predicate) {
        List<Integer> result = new ArrayList<>();
        for (int val : list) {
            if (predicate.test(val)) {
                result.add(val);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> numbers = List.of(-3, -2, -1, 1, 2, 3, 5);
        System.out.println("원본 리스트: " + numbers);

        // 1. 음수(negative)만 뽑아내기
        List<Integer> negatives = filter(numbers, value -> value < 0);
        System.out.println("음수만: " + negatives);

        // 2. 짝수(even)만 뽑아내기
        List<Integer> evens = filter(numbers, value -> value % 2 == 0);
        System.out.println("짝수만: " + evens);
    }
}
```

- `MyPredicate`는 직접 만든 함수형 인터페이스이다. 이것을 자바가 제공하는 함수형 인터페이스로 변경해보자.

정답

```
package lambda.ex3;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterExampleEx2 {
```

```
    // 고차 함수, 함수를 인자로 받아서 조건에 맞는 요소만 뽑아내는 filter
    public static List<Integer> filter(List<Integer> list, Predicate<Integer>
predicate) {
        List<Integer> result = new ArrayList<>();
        for (int val : list) {
            if (predicate.test(val)) {
                result.add(val);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> numbers = List.of(-3, -2, -1, 1, 2, 3, 5);
        System.out.println("원본 리스트: " + numbers);

        // 1. 음수(negative)만 뽑아내기
        List<Integer> negatives = filter(numbers, value -> value < 0);
        System.out.println("음수만: " + negatives);

        // 2. 짝수(even)만 뽑아내기
        List<Integer> evens = filter(numbers, (value) -> value % 2 == 0);
        System.out.println("짝수만: " + evens);
    }
}
```

- 정답의 패키지 위치는 `ex3`에 있다.

다음 두 함수형 인터페이스 모두 사용할 수 있다. 둘다 정답이다.

- `Predicate<Integer>`
- `IntPredicate`: 기본형 지원 함수형 인터페이스

`Function<Integer, Boolean>`으로도 풀 수 있지만, `Predicate`가 보다 조건을 검사한다는 **의도를 명확히** 드러내므로 `Predicate`가 더 나은 선택이다.

문제2 - MapExample

```
package lambda.ex2;
```

```java
import lambda.ex2.StringFunction;

import java.util.ArrayList;
import java.util.List;

public class MapExample {

    // 고차 함수, 함수를 인자로 받아, 리스트의 각 요소를 변환
    public static List<String> map(List<String> list, StringFunction func) {
        List<String> result = new ArrayList<>();
        for (String str : list) {
            result.add(func.apply(str));
        }
        return result;
    }

    public static void main(String[] args) {
        List<String> words = List.of("hello", "java", "lambda");
        System.out.println("원본 리스트: " + words);

        // 1. 대문자 변환
        List<String> upperList = map(words, s -> s.toUpperCase());
        System.out.println("대문자 변환 결과: " + upperList);

        // 2. 앞뒤에 *** 붙이기 (람다로 작성)
        List<String> decoratedList = map(words, s -> "***" + s + "***");
        System.out.println("특수문자 데코 결과: " + decoratedList);
    }
}
```

- `StringFunction`을 자바가 제공하는 함수형 인터페이스로 변경하자.

정답

```java
package lambda.ex3;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.UnaryOperator;
```

```
public class MapExample {

    // 고차 함수, 함수를 인자로 받아, 리스트의 각 요소를 변환
    public static List<String> map(List<String> list, UnaryOperator<String>
func) {
        List<String> result = new ArrayList<>();
        for (String str : list) {
            result.add(func.apply(str));
        }
        return result;
    }

    public static void main(String[] args) {
        List<String> words = List.of("hello", "java", "lambda");
        System.out.println("원본 리스트: " + words);

        // 1. 대문자 변환
        List<String> upperList = map(words, s -> s.toUpperCase());
        System.out.println("대문자 변환 결과: " + upperList);

        // 2. 앞뒤에 *** 붙이기 (람다로 작성)
        List<String> decoratedList = map(words, s -> "***" + s + "***");
        System.out.println("특수문자 데코 결과: " + decoratedList);
    }
}
```

다음 두 함수형 인터페이스 모두 정답이 될 수 있다.

- `UnaryOperator<String>`
- `Function<String, String>`

의도 관점

- **입력과 출력 타입이 동일한 연산**(예: `String` ➜ `String`)을 표현하려는 의도가 분명하다면, `UnaryOperator<String>`을 사용하는 편이 더 의도를 명확히 드러낼 수 있다.

문제3 - ReduceExample

```
package lambda.ex2;

import lambda.ex2.MyReducer;
```

```
import java.util.List;

public class ReduceExample {

    // 함수를 인자로 받아, 리스트 요소를 하나로 축약(reduce)하는 고차 함수
    public static int reduce(List<Integer> list, int initial, MyReducer
reducer) {
        int result = initial;
        for (int val : list) {
            result = reducer.reduce(result, val);
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4);
        System.out.println("리스트: " + numbers);

        // 1. 합 구하기 (초깃값 0, 덧셈 로직)
        int sum = reduce(numbers, 0, (a, b) -> a + b);
        System.out.println("합(누적 +): " + sum);

        // 2. 곱 구하기 (초깃값 1, 곱셈 로직, 람다로 작성)
        int product = reduce(numbers, 1, (a, b) -> a * b);
        System.out.println("곱(누적 *): " + product);
    }
}
```

정답

```
package lambda.ex3;

import java.util.List;
import java.util.function.BinaryOperator;

public class ReduceExample {

    // 함수를 인자로 받아, 리스트 요소를 하나로 축약(reduce)하는 고차 함수
    public static int reduce(List<Integer> list, int initial,
BinaryOperator<Integer> reducer) {
```

```
        int result = initial;
        for (int val : list) {
            result = reducer.apply(result, val); // 수정
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4);
        System.out.println("리스트: " + numbers);

        // 1. 합 구하기 (초깃값 0, 덧셈 로직)
        int sum = reduce(numbers, 0, (a, b) -> a + b);
        System.out.println("합(누적 +): " + sum);

        // 2. 곱 구하기 (초깃값 1, 곱셈 로직, 람다로 작성)
        int product = reduce(numbers, 1, (a, b) -> a * b);
        System.out.println("곱(누적 *): " + product);
    }
}
```

다음 두 함수형 인터페이스 모두 정답이 될 수 있다. 호출 메서드 이름은 변경했다.

- `BinaryOperator<Integer>`
- `IntBinaryOperator`

`BiFunction`으로도 풀 수 있지만, `Operator`가 입력과 출력이 같다는 **의도를 명확히** 드러내므로 `Operator`가 더 나은 선택이다. 특히 `reduce`라는 기능은 보통 같은 타입의 연산을 누적하고, 같은 타입의 결과를 낸다.

- `BiFunction<Integer, Integer, Integer>`

정리

지금까지 살펴본 내용을 간단히 정리해보자.

1. **함수형 인터페이스와 제네릭**
    - 함수형 인터페이스에 제네릭을 도입하면 **코드 재사용성**과 **타입 안전성**을 모두 확보할 수 있다.
    - 직접 `Object` 타입을 사용하는 방식(`ObjectFunction`)은 다양한 타입을 다룰 수 있지만, 다운캐스팅 과정이 필요하고 타입 안정성이 떨어진다.
    - 제네릭 타입을 사용하면 컴파일 시점에 타입 체크가 이루어지므로, **런타임 에러를 방지**할 수 있고 **유연한 코**

드를 작성할 수 있다.

2. **람다와 타겟 타입**
   - 람다는 그 자체로 타입이 정해져 있지 않고, 어떤 함수형 인터페이스에 대입되느냐(타겟 타입)에 따라 타입이 결정된다.
   - 같은 람다라도 `FunctionA`에 대입하면 `FunctionA` 타입이 되고, `FunctionB`에 대입하면 `FunctionB` 타입이 된다.
   - 이미 한 번 특정 함수형 인터페이스 타입으로 대입된 람다는, 시그니처가 같더라도 다른 함수형 인터페이스 타입으로 대입할 수 없다.

3. **자바가 제공하는 기본 함수형 인터페이스**
   - `Function<T, R>`: 하나의 매개변수를 받아 결과를 반환한다.
   - `Consumer<T>`: 하나의 매개변수를 받아서 소비(처리)만 하고, 반환값은 없다.
   - `Supplier<T>`: 매개변수가 없고, 값을 공급(생성)하여 반환한다.
   - `Runnable`: 매개변수와 반환값이 모두 없는 실행 작업을 나타낸다(주로 스레드 실행).

4. **특화 함수형 인터페이스**
   - `Predicate<T>`: 입력값을 받아 조건을 검사(필터링)하고 `boolean`을 반환한다.
   - `UnaryOperator<T>`: **하나**의 같은 타입 입력을 받아 같은 타입을 반환(단항 연산)한다.
   - `BinaryOperator<T>`: **두 개**의 같은 타입 입력을 받아 같은 타입을 반환(이항 연산)한다.
   - 이외에도 매개변수가 2개 이상인 경우 `BiFunction<T, U, R>`, `BiConsumer<T, U>`, `BiPredicate<T, U>` 등이 있으며, 기본형(primitive) 전용 함수형 인터페이스(`IntFunction`, `IntPredicate` 등)도 별도로 제공한다.

5. **문제 예시와 풀이**
   - `filter`, `map`, `reduce` 예제를 통해 직접 만든 함수형 인터페이스를 자바가 제공하는 `Predicate`, `UnaryOperator`, `BinaryOperator` 등으로 대체하는 과정을 살펴보았다.
   - 일반적으로 **입력과 반환의 구조**와 **의도**를 가장 명확히 드러내는 함수형 인터페이스를 선택하면 된다.
     - 예: 조건 검사 시 `Predicate`, 단항 연산 시 `UnaryOperator`, 이항 연산 시 `BinaryOperator` 등.

정리하자면, **람다**와 **함수형 인터페이스**를 제대로 활용하면 코드가 간결해지고 가독성이 높아지며, **제네릭**을 도입하면 재사용성과 타입 안전성까지 모두 확보할 수 있다. 또한 자바가 기본적으로 제공하는 다양한 함수형 인터페이스를 적극 활용하면, 불필요하게 유사한 인터페이스를 여러 개 만들 필요가 없고 호환성 문제도 해결된다.

무엇보다 **"의도를 명확하게 드러내는"** 함수형 인터페이스를 적절히 선택하는 것이 중요하다. 조건 검사는 `Predicate`, 입력과 반환 타입이 같은 단항 연산은 `UnaryOperator`, 매개변수가 2개이면서 입력과 반환 타입이

같은 연산은 `BinaryOperator`처럼 상황에 맞는 인터페이스를 사용하면 코드의 목적이 분명해지고 유지보수성이 향상된다.