함수형 인터페이스

소유자	종수 김
태그	

- 함수형 인터페이스도 인터페이스, 때문에 제네릭 도입 가능.

제네릭이 필요한 이유

- 타입만 다르고 같은 형태인데, 변수 타입마다 모든 인터페이스를 만드는건 불편함.

```
package lambda.lambda3;

public class GenericMain1 {
    public static void main(String[] args) {
        StringFunction upperCase = (s -> s.toUpperCase());
        System.out.println(upperCase.apply("hello"));

        NumberFunction square = s -> s * s;
        System.out.println(square.apply(3));
    }

    interface StringFunction {
        String apply(String s);
    }

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

- Object로 사용할 순 있으나, 타입 변환 에러의 가능성을 품고감.

```
package lambda.lambda3;

public class GenericMain2 {
    public static void main(String[] args) {
```

```
        ObjectFunction upperCase = (s -> ((String) s).toUpperCase());
        System.out.println(upperCase.apply("hello"));

        ObjectFunction square = (s -> ((Integer) s) * ((Integer) s));
        System.out.println(square.apply(3));
    }

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

- 위 두 방법은, 코드 재사용이 떨어지거나, 타입 안정성이 떨어지거나 두 가지를 한번에 만족할 순 없음.

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

제네릭이 필요한 이유 2

- 제네릭 도입

```
package lambda.lambda3;

public class GenericMain3 {
    public static void main(String[] args) {
        GenericFunction<String, String> upperCase = str -> str.toUpperCase();
        GenericFunction<Integer, Integer> square = n -> n * n;
```

```
        System.out.println(upperCase.apply("Hello"));
        System.out.println(square.apply(3));
    }

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

- 재사용성
  - GenericFunction은 매개변수 1개이고, 반환 값이 있는 모든 람다에 사용 가능.
- 타입 안정성
  - 제네릭을 통해 타입을 고정시켰으므로 컴파일 시점에 타입 에러를 방지 가능.

```
package lambda.lambda3;

public class GenericMain3 {
    public static void main(String[] args) {
        GenericFunction<String, String> upperCase = str -> str.toUpperCase();
        GenericFunction<Integer, Integer> square = n -> n * n;

        System.out.println(upperCase.apply("Hello"));
        System.out.println(square.apply(3));


        GenericFunction<Integer, Boolean> isEven = n -> n % 2 == 0;
        GenericFunction<Integer, String> str = n -> "str " + n;

        System.out.println(isEven.apply(5));
        System.out.println(str.apply(151515));
    }

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

```
    }
}
```

람다와 타겟 타입

GenericFunction을 통해 코드 중복을 줄이고, 타입 안정성을 챙겼지만 문제가 있음.

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 만들 어서 사용해야 한다. 그리고 비슷한 모양의 `GenericFunction` 이 많이 만들어질 것이다.

2. A가 만든 함수형 인터페이스와, B가 만든 함수형 인터페이스는 서로 호환되지 않는다.

```
package lambda.lambda3;

public class TargetType1 {
    public static void main(String[] args) {
        // 람다 직접 대입 : 문제 없음
        FunctionA functionA = i -> "value : " + i;
        System.out.println(functionA.apply(10));
            // 시그니쳐가 같으므로 문제 없음.
        FunctionB functionB = i -> "value : " + i;
        System.out.println(functionB.apply(20));

        // 이미 만들어진 FunctionA 인스턴스를 FunctionB에 대입
        FunctionB targetB = functionA; // 컴파일 에러 - 타입이 다르니까.
    }
    @FunctionalInterface
    interface FunctionA {
        String apply(Integer i);
    }

    @FunctionalInterface
    interface FunctionB {
```

```
        String apply(Integer i);
    }
}
```

람다와 타겟 타입

람다는 그 자체만으로는 구체적인 타입이 정해져 있지 않고, 타겟 타입(Target Type)이라고 불리는 맥락(대입되는 참조형)에 의해 타입이 결정.

```
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.

- 타입이 결정 되고 나면 이후에는 다른 타입에 대입하는 것이 불가능.
- 메서드의 시그니쳐가 같더라도 FunctionA와 FunctionB는 다르므로 대입이 불가능.

> **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

위 처럼 대입이 안되는 메서드 시그니쳐가 같더라도 함수형 인터페이스끼리는 대입이 안되므로, 서로 비슷한 유형의 인터페이스들이 매우 많아질 수 있음.

- 비슷한 함수형 인터페이스를 불필요하게 만드는 문제

```
package lambda.lambda3;

import java.util.function.Function;

// 자바가 기본으로 제공하는 Function 대입
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);

        Function<Integer, Integer> square = x -> x * x;
        Integer result2 = square.apply(5);
        System.out.println("result2 = " + result2);
    }
}
```

- 함수형 인터페이스의 호환성 문제

  Ex) 필터를 위해 Predicate를 만들었지만, 이 필터를 다른 함수형 인터페이스에서는 사용할 수 없음.
  Stream.filter

```
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 : 문제없음
        Function<Integer, String> fun = x -> "value: " + x;
        System.out.println(fun.apply(10));
```

```
        Function<Integer, String> fun1 = fun;
        System.out.println(fun1.apply(20));
    }
}
```

기본 함수형 인터페이스

자바가 기본으로 제공하는 가장 대표적인 함수형 인터페이스

1. Function
    a. 입력 O / 반환 O
2. Consumer
    a. 입력 O / 반환 X
3. Supplier
    a. 입력 X / 반환 O
4. Runnable
    a. 입력 X / 반환 X

Function

```
package java.util.function;
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

- 하나의 매개변수를 받고, 결과를 반환하는 함수형 인터페이스.
- 입력 값 T를 다른 타입의 출력 값(R)을 반환하는 연산을 표현
- 가장 일반적인 함수

Consumer (소비자)

```
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 인터페이스
- 입력 값 (T)를 받아서 처리하지만, 결과를 반환하지 않는 연산.
- 입력 받은 데이터를 기반으로 내부적으로 처리만 하는 경우
    - 컬렉션에 값 추가, 콘솔 출력, 로그 작성, DB 저장 등

Supplier (공급자)

```
package java.util.function;
@FunctionalInterface
public interface Supplier<T> {
T get(); }
```

- 입력을 받지 않고 어떤 데이터를 공급해주는 함수형 인터페이스
- 객체나 값 생성, 지연 초기화 등에 주로 사용

Runnable

```
package java.lang;
@FunctionalInterface
public interface Runnable {
void run(); }
```

- 입력값도 없고, 반환 값도 없는 함수형 인터페이스
- 원래 쓰레드 실행을 위한 인터페이스로 쓰였고, 자바 8 이후 람다식으로도 표현.
- 주로 멀티쓰레딩에서 쓰레드의 작업을 정의할 때 사용
- 입력 값도 없고, 반환 값도 없는 함수형 인터페이스가 필요할 때 사용

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

의도를 명확하게 만든 조금 특별한 함수형 인터페이스

1. Predicate
    a. 입력O, 반환 boolean
2. Operator(UnaryOperator, BinaryOperator)
    a. 입력O, 반환O
    b. 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 false로 구분(판단)하는 함수형 인터페이스.
- 조건 검사, 필터링 등의 용도로 많이 사용
- 입력 값을 받아 true / false로 결과를 판단한다는 의도를 명시적으로 드러냄.

Operator

```
UnaryOperator(단항 연산)
  package java.util.function;
  @FunctionalInterface
  public interface UnaryOperator<T> extends Function<T, T> {
T apply(T t); // 실제 코드가 있지는 않음 }

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

- 같은 타입의 값들을 받아서 동일한 타입의 결과를 반환.
  - Ex) 덧셈 : 숫자 + 숫자 = 숫자 / AND : boolean AND boolean = boolean
- 단항 연산 : 하나의 피연산자(Operand)에 대해 연산을 수행하는 것.
  - 숫자의 부호 연산 -x / 논리 부정 연산 !x
  - Function<T, T>를 상속받으며, 입력과 반환을 모두 같은 T로 고정. 입력 타입 = 반환 타입
- 이항 연산 : 두 개의 피연산자에 대해 연산을 수행하는 것.
  - 두 수의 덧셈, 곱셈 등
  - 같은 타입의 두 입력을 받아, 같은 타입의 결과를 반환
  - BiFunction<T,T,T>를 상속받는 방식으로 구현되어있음. 입력값 2개와 반환을 모두 같은 T로 고정.
- Function<T, R> / BiFuntion<T, U, R> 만으로도 Operator가 대체되지만, Predicate와 마찬가지로 목적의 명확성을 위해 구분.

기타 함수형 인터페이스

입력 값이 2개 이상

매개 변수가 2개 이상 필요한 경우에는 BiXxx 시리즈.
Bi = Binary(이항, 둘)의 줄임말.

```
package lambda.lambda4;
```

```
import java.util.function.BiConsumer;
import java.util.function.BiFunction;
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(add.apply(1, 5));

        BiConsumer<String, Integer> repeat = (string, integer) -> {
            for(int i = 0; i < integer; i++) {
                System.out.print(string);
            }
            System.out.println();
        };

        repeat.accept("hello", 5);
        BiPredicate<Integer, Integer> is = (a, b) -> a > b;
        System.out.println(is.test(5, 7));

    }
}
```

만약 입력값이 3개라면 ?

직접 만들어야 함. 기본적으로 제공하지 않기 때문.

```
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

기본형(Primitive Type)을 지원하는 함수형 인터페이스

```
package java.util.function;
 @FunctionalInterface
 public interface IntFunction<R> {
    R apply(int value);
 }
```

존재 이유

- 오토박싱/언박싱으로 인한 성능 비용을 줄이기 위해
    - 이 정도 까지 하는 경우는 거의 없음
- 자바 제네릭의 한계(제네릭은 primitive 타입을 직접 다룰 수 없음.)
    - 자바의 제네릭은 기본형 타입을 직접 다룰 수 없어서, Function<int, R> 같은 식으로는 선언할 수 없음.

```
package lambda.lambda4;

import java.util.function.IntFunction;
import java.util.function.IntToLongFunction;
import java.util.function.IntUnaryOperator;
import java.util.function.ToIntFunction;

public class PrimitiveFunction {
    public static void main(String[] args) {
        // 기본형 매개변수, IntFunction, LongFunction, DoubleFuntion
        IntFunction<String> function = x -> "숫자 + " + x;
        System.out.println(function.apply(10));

        // 기본형 반환, ToIntFunction
        ToIntFunction<String> function1 = x -> x.length();
```

```
        System.out.println(function1.applyAsInt("HELOLOLOLOL"));

        // 기본형 매개변수, 기본형 반환
        IntToLongFunction intToLongFunction = x -> x * 100L;
        System.out.println(intToLongFunction.applyAsLong(50));

        // IntUnaryOperator : int -> int
        IntUnaryOperator intUnaryOperator = x -> x * 2;
        System.out.println(intUnaryOperator.applyAsInt(10));

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

정리

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.