
K G 아 이 티 뱅 크
JAVA
J A V A

LAMBDA

LAMBDA

❖ **자바의 장황한 코드를 줄여주기 위하여 등장한 코드 작성법**

- ➢ 내부 동작은 익명객체의 생성방식과 유사하게 작동함
- ➢ 익명객체의 복잡한 작성방식을 최대한 간결하게 줄인 구조

❖ **함수명 인터페이스는 단 하나의 추상 메서드를 가지고 있음**

- ➢ 추상 클래스에 적용할 수 없으며 인터페이스에만 가능
- ➢ 함수형 인터페이스가 가지는 필드는 람다식에 관여하지 않음

❖ **람다식은 함수형 인터페이스에만 적용하기에 특징을 가짐**

① 익명성 : 한번 쓰고 버리거나, 저장해서 여러번 사용함

② 함수형 : 다른 언어의 함수(Function)에 대응이 됨

③ 간결성 : 절대로 어렵지 않고, 더 쉽게 만들어주는 문법

❖ 람다식 작성시의 조건 및 주의사항 : 오버라이딩을 하는 과정

① 함수형 인터페이스여야만 가능하고 그 외에는 불가능

② 대상 메서드의 매개변수/반환형을 정확하게 준수해야 함

```
// 함수형 인터페이스 : 필드는 별 관계없음
public interface Sample {
    void method(int value);
}
```

```
// 작성방식1 : (args) -> expression;
Sample lam1 = (int value) -> System.out.println(value);

// 작성방식2 : (args) -> { statements; };
Sample lam2 = (int count) -> {
    while ( count > 0 ) {
        System.out.printf("%3d", count);
        count -= 1;
    }
};
```

LAMBDA

❖ **람다식을 잘 이용하면 융통성이 있고 유연한 코드가 완성됨**

① 코드 작성과정에서 문법적으로 좀 더 간결하게 가능함

② 더 나아가 함수형 프로그래밍 패턴을 도입할 수 있음

❖ **단, 람다식도 제한사항이 존재하고 이에 대해 주의해야 함**

① 함수형 인터페이스에만 가능하며, 익명 객체와 유사함

② 람다식을 **남발하면** 되려 지나치게 복잡한 코드가 됨

```
// 이런 것은 익명 객체보다 람다가 유리함
Runnable thread = new Runnable() {
    public void run() { System.out.println("Thread!"); }
};

// 그렇다고 이렇게 남발하면 지나치게 복잡함
new Thread(()->System.out.println("T1")).start();
new Thread(()->System.out.println("T2")).start();
new Thread(()->System.out.println("T3")).start();
new Thread(()->System.out.println("T4")).start();
.....
```

Function

Function

❖ **람다식을 더 편하게 사용하라고 제공하는 기본 인터페이스**

- java.util.Function 패키지에 포함되어 있는 인터페이스
- 메서드를 재활용할 수 있도록 하는 것이 주 목적인 패키지

❖ **4가지 주요 인터페이스들을 이용해 빠르게 함수를 준비함**

- Predicate<E> : 하나의 E객체를 받아 boolean값을 반환함
- Consumer<E> : 하나의 E객체를 받아 소비만 수행함
- Function<E1, E2> : E1객체를 받아 E2객체로 반환함
- Supplier<E> : 매개변수없이 E객체를 반환함

❖ **여기에 고유 메서드를 활용하여 함수를 결합/합성이 가능**

① compose() : 주어진 함수를 먼저 실행하고 이어서 수행

② andThen() : 주어진 함수를 나중에 실행

③ and() : 논리연산자 &&의 동작을 수행함

④ or() : 논리연산자 ||의 동작을 수행함

❖ 4가지 인터페이스는 제네릭이며 람다식과 조합하여 사용함

① 제네릭이기에 자료형의 지정없이 변수명만 설정함

② 람다식이니 해당 인터페이스의 구조를 외워서 기억해야 함

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

// Predicate<E>
Predicate<Integer> greaterThan5 = (num)->num > 5;
greaterThan5.test(100);
// Consumer<E>
Consumer<Integer> show = (num)->System.out.print(num);
show.accept(100);
// Supplier<E>
Supplier<Integer> value = ()->100;
value.get();
// Function<E1, E2>
Function<Integer, Interger> add5 = (n1)->n1 + 5;
add5.apply(10);
```

❖ 결합/합성의 경우 자료형이 호환/일치해야 하니 주의해야 함

① Function에 대한 합성(compose/andThen)을 수행함

② Predicate에 대한 결합(and/or)을 수행함

```
Function<Integer, Double> f1 = (value)->value+0.5;
Function<Double, Integer> f2 = (value)->(int)value;
// Function 합성 : andThen – f1을 실행하고 f2로 이어짐
Function<Integer, Integer> f12 = f1.andThen(f2);
// Function 합성 : compose – f2를 실행하고 f1로 이어짐
Function<Double,Double> f21 = f1.compose(f2);

Predicate<Integer> gThan5 = (value)->value>5;
Predicate<Integer> sThan10 = (value)->value<10;
// Predicate 결합 : and – 5보다 크고 10보다 작음
Predicate<Integer> bet5and10 = gThan5.and(sThan10);
// Predicate 결합 : or – 5보다 크거나 10보다 작음
Predicate<Integer> bet5and10 = gThan5.or(sThan10);
```