

# Effective Java

---

Item 42 : 익명 클래스보다는 람다를 사용하라

Item 43 : 람다보다는 메서드 참조를 사용하라

Item 44 : 표준 함수형 인터페이스를 사용하라

# Overview

함수형 인터페이스

---

# Functional Interface

추상 메서드를 딱 하나만 가지고 있는 인터페이스

인터페이스는 public abstract 생략 가능

```
@FunctionalInterface
public interface RunSomething {

    void doIt();

    // void doIt2(); // 컴파일 에러

    static void printName() {
        System.out.println("Seungtaek");
    }

    default void printAge() {
        System.out.println("25");
    }

}
```

static, default 메서드는 상관없다.

# Functional Interface

```
public class RunImpl implements RunSomething {  
    @Override  
    public void doIt() {  
        System.out.println("Hello");  
    }  
}
```

```
RunSomething runSomething = new RunImpl();
```

구현 메소드 생성

```
RunSomething runSomething = new RunSomething() {  
    @Override  
    public void doIt() {  
        System.out.println("Hello");  
    }  
};
```

익명 클래스 이용

```
RunSomething runSomething2 = () -> System.out.println("Hello2")
```

람다 이용

# Item 42

익명 클래스보다는 람다를 사용하라

---

# Lambda Expression

기본형태 : ( 인자 리스트 ) -> { 바디 }

```
RunSomething runSomething = (a, b) -> {  
    System.out.println("a + b = " + (a + b));  
    return a + b;  
};
```

## 인자 리스트

- 인자가 없을 때: ()
- 인자가 한 개: (one) 또는 one
- 인자가 여러 개: (one, two)
- 인자의 타입은 생략 가능

```
Supplier<Integer> ex1 = () -> 10;  
UnaryOperator<String> ex2 = s -> s + " world";  
BinaryOperator<Integer> ex3 = (a, b) -> a + b;
```

# 변수 캡처(Variable Capture)

로컬 변수 캡처: 변수가 final 이거나 effective final 인 경우에만 참조할 수 있다.

```
int baseNumber = 10;
IntConsumer printInt = (i) → {
    System.out.println(i + baseNumber);
};
```

baseNumber += 10;

람다 표현식에 사용되는 변수는 final 또는 유사 final이어야 합니다

'baseNumber'을(를) 실질적 final 임시 변수로 복사

액션 더보기...

## effective final

- 자바 8부터 지원하는 기능
- “사실상” final 인 변수.
- final 키워드를 사용하지 않은 변수를 익명 클래스 구현체 또는 람다에서 참조할 수 있다.

# 변수 캡처(Variable Capture)

익명 클래스 구현체와 달리 shadowing 하지 않는다.

익명 클래스는 새로 스코프를 만들지만, 람다는 람다를 감싸는 스코프와 같다.


+ 람다의 this 키워드는 바깥 인스턴스를 가리킨다.

```
int baseNumber = 10;

IntConsumer printInt1 = new IntConsumer() {


    @Override
    public void accept(int i) {
        System.out.println(i + baseNumber);
    }

};
```



---

```
IntConsumer printInt2 = (i) -> {
    int baseNumber = 12;
    System.out.println(i + baseNumber);
};
```



```
IntConsumer printInt2 = (i) -> {
    int baseNumber = 12;
    System.out.println(i + baseNumber);
};
```

변수 'baseNumber'은(는) 범위에 이미 정의되어 있습니다  
이전에 선언된 변수 'baseNumber'(으)로 이동합니다



# 그 외

---

1. 람다는 이름도 없고 문서화도 못한다.  
세 줄을 넘어가면 람다를 쓰지 말자
2. 람다는 함수형 인터페이스에만 쓰인다.  
예를 들어, 추상 클래스 인스턴스를 만들 때는 람다를 사용할 수 없다.
3. 람다는 자신을 참조할 수 없다.
4. 람다는 직렬화하지 말자

# Item 44

표준 함수형 인터페이스를 사용하라

---

# 자바에서 제공하는 함수형 인터페이스

인터페이스	함수 시그니처	설명
Function<T, R>	R apply(T t)	T 타입을 받아서 R 타입을 리턴
BiFunction<T, U, R>	R apply(T t, U u)	두 개의 값을 받아서 R 타입을 리턴
Consumer<T>	void accept(T t)	T 타입을 받아서 아무것도 리턴하지 않음
Supplier<T>	T get()	T 타입의 값을 반환
Predicate<T>	boolean test(T t)	T 타입을 받아서 boolean을 리턴
UnaryOperator<T>	T apply(T t)	Function<T, R>의 특수 형태
BinaryOperator<T>	T apply(T t1, T t2)	BiFunction<T, U, R>의 특수 형태

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

## 1. 익명 클래스, 람다 모두 사용 가능하다.

```
Function<Integer, String> function1 = new Function<Integer, String>() {  
    @Override  
    public String apply(Integer i) {  
        return "number" + i;  
    }  
};  
  
Function<Integer, String> function2 = (i) → "number" + i;
```

## 2. 모두 @FunctionalInterface 이다.

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T var1);  
  
    //.. 생략  
}
```

# 함수형 인터페이스

## Function<T, R>

T 타입을 받아서 R 타입을 리턴: R apply(T t)

```
Function<Integer, String> function2 = (i) -> "number" + i;  
String str = function2.apply(10);
```

함수 조합용 메서드

```
Function<Integer, String> function1 = (i) -> "number" + i;  
Function<String, Boolean> function2 = (i) -> i.startsWith("number");  
  
Function<Integer, Boolean> andThen = function1.andThen(function2);  
Function<Integer, Boolean> compose = function2.compose(function1);
```

First Second

# 함수형 인터페이스

## BiFunction<T, U, R>

두 개의 값(T, U)을 받아서 R 타입을 리턴: R apply(T t, U u)

```
BiFunction<Integer, String, Long> biFunction = (a, b) -> a + Long.parseLong(b);  
Long apply = biFunction.apply(10, "20");
```

---

## Consumer<T>

T 타입을 받아서 아무것도 리턴하지 않음: void accept(T t)

```
Consumer<Integer> printT = (i) -> System.out.println(i);  
printT.accept(10);
```

# 함수형 인터페이스

## Supplier<T>

T 타입의 값을 반환: T get()

```
Supplier<Integer> supplier = () -> 10;  
Integer i = supplier.get();
```

---

## Predicate<T>

T 타입을 받아서 boolean을 리턴하는 함수 인터페이스: boolean test(T t)

```
Predicate<Integer> predicate = i -> i > 0;  
predicate.test(10);
```

```
Predicate<Integer> and = predicate.and(i -> i < 100);  
Predicate<Integer> or = predicate.or(i -> i < 0);  
Predicate<Integer> negate = predicate.negate();
```

함수 조합용 메서드

# 함수형 인터페이스

## UnaryOperator<T>

Function<T, R> 의 특수한 형태. T 타입을 입력받아 T 타입을 리턴: T apply(T t)

```
UnaryOperator<Integer> unaryOperator = (i) -> i + 10;  
Integer apply = unaryOperator.apply(10);
```

---

## BinaryOperator<T>

BiFunction<T, U, R> 의 특수한 형태. T 타입 2개를 입력받아 T 타입을 리턴: T apply(T t1, T t2)

```
BinaryOperator<Integer> binaryOperator = (a, b) -> a + b;  
Integer apply = binaryOperator.apply(10, 20);
```



# Item 43

람다보다는 메서드 참조를 사용하라

---

# 메소드 레퍼런스

```
public class Greeting {  
  
    private String tmp;  
  
    public Greeting() {  
    }  
  
    public Greeting(String tmp) {  
        this.tmp = tmp;  
    }  
  
    public String hello(String name) {  
        return "Hello, " + name;  
    }  
  
    public static String hi(String name) {  
        return "hi, " + name;  
    }  
}
```

```
// 스테틱 메소드 참조  
UnaryOperator<String> hi = Greeting::hi;  
  
// 인스턴스 메소드 참조  
Greeting greeting = new Greeting();  
UnaryOperator<String> hello1 = greeting::hello;  
BiFunction<Greeting, String, String> hello2 = Greeting::hello;  
  
// 생성자 참조  
Supplier<Greeting> newGreeting = Greeting::new;  
Function<String, Greeting> newGreeting2 = Greeting::new;
```