



# 함수형 인터페이스

📅 날짜	@2022년 10월 30일
🏷 태그	

## 함수형 인터페이스

1. 객체지향 프로그래밍과 함수형 프로그래밍
  2. 익명 내부 클래스
  3. 람다식
  4. 함수형 인터페이스
  5. 함수형 인터페이스 특징
    - 5-1. `@FunctionalInterface`
    - 5-2. `static` 메서드, `default` 메서드
    - 5-3. 함수형 인터페이스 사용
  6. 함수형 인터페이스 패키지
- 결론

참조

## 함수형 인터페이스

함수형 인터페이스는 함수형 프로그래밍에 연관이 있습니다.

Java는 객체지향 프로그래밍인데 왜 Java8에서 이런 함수형 프로그래밍을 넣게 됐을까요 ?

함수형 인터페이스에 대해서 알아보기 전에,

객체지향 프로그래밍과 함수형 프로그래밍의 차이를 간단하게 보겠습니다.

### 1. 객체지향 프로그래밍과 함수형 프로그래밍

사람들은 간단한 계산을 하고 있습니다.

객체지향 프로그래밍은 객체를 중심으로 프로그래밍을 하고 있고

함수형 프로그래밍은 함수를 중심으로 프로그래밍을 하고 있습니다.

사용자 1 은 더하는 작업을 부탁해야 합니다.

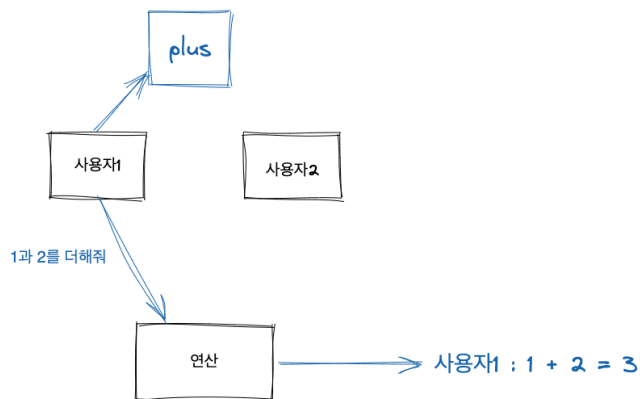
사용자 2 는 빼는 작업을 부탁해야 합니다.

사용자 1 , 사용자 2 는 둘 다 외부 참조의 변수를 보고 연산 작업을 부탁합니다.

사용자 1 의 작업이 끝나면 외부 참조가 "minus"로 변합니다.

사용자 2 의 작업이 끝나면 외부 참조가 "plus"로 변합니다.

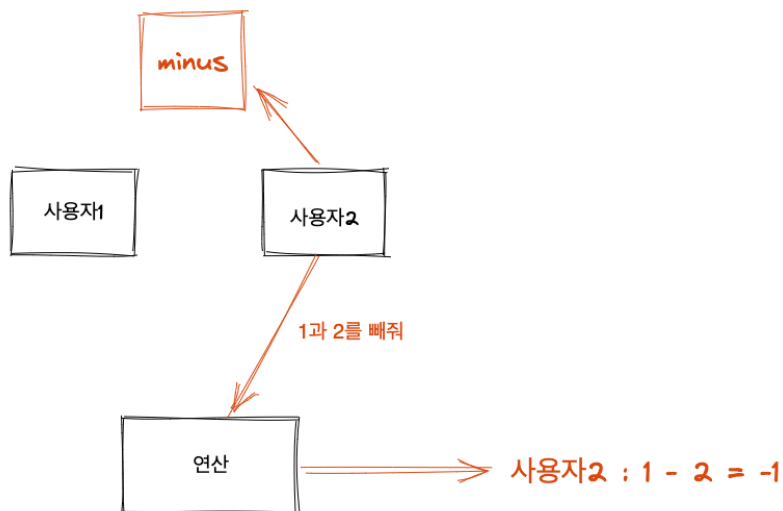
## Object Oriented Programming



사용자 1 은

외부 변수를 참조하여  
연산을 부탁합니다.

## Object Oriented Programming



사용자 2 도

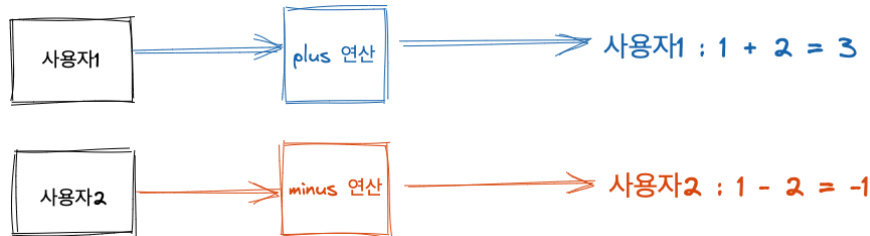
마찬가지로

외부 변수를 참조하여  
연산을 부탁합니다.

사용자 1 은 두 정수를 더하는 연산에게 넘깁니다.

사용자 2 는 두 정수를 빼는 연산을 넘깁니다.

## Functional Programming



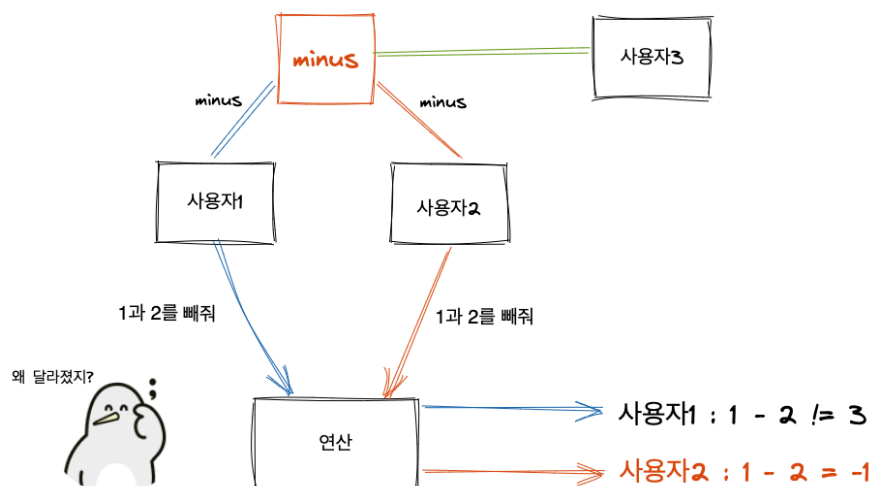
사용자 1 은 두 정수를 plus 연산 함수로 넘깁니다.

사용자 2 는 두 정수를 minus 연산 함수로 넘깁니다.

갑자기 사용자 3 이 등장해서 연산을 진행하여 외부 변수가 변했습니다.

후에 사용자 1 , 사용자 2 가 다시 연산을 진행하니 다음과 같이 값이 다르게 나오기 시작했습니다.

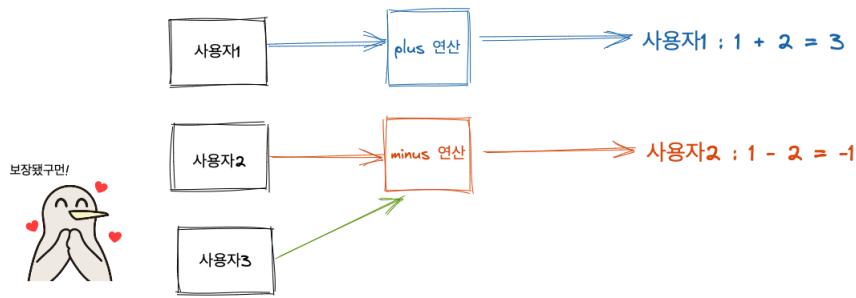
## Object Oriented Programming



외부 변수 참조를 동시에 하던 도중 사용자 1 은 잘못된 변수를 참조하게 됩니다.

외부의 변화에 무관하여 여전히

## Functional Programming



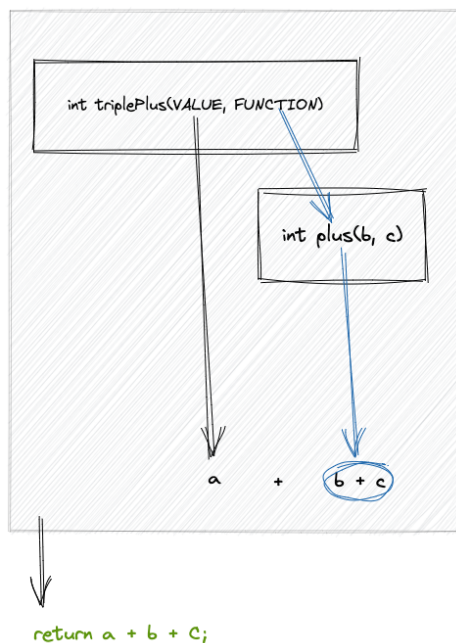
사용자 1, 사용자 2 는  
주어진 입력에 대한  
정확한 결과를 반환합니다.

### 이처럼

객체지향 프로그래밍은 외부 변수에 영향을 받아 언제든지 값이 변동될 위험이 있지만

함수형 프로그래밍은 외부 변수에 영향을 받지 않고 동일한 input에 동일한 output을 보장합니다.

또한 함수형 프로그래밍은 함수의 인자를 함수로 전달할 수 있습니다.



인자에 plus 함수를 넣은  
triplePlus(...)를 작성할 수 있습니다.

이러한  
함수의 인자로 함수를 전달,  
함수의 리턴값으로 함수로 사용하는 것은

함수 자체도 재사용 할 수 있게 되는 용이함이 있습니다.



## 2. 익명 내부 클래스

Java에서도 충분히 가능합니다.

바로 익명 내부 클래스를 이용하면 됩니다.

`words`를 정렬하기 위해 `Comparator` 인터페이스를 내부 익명 클래스로 작성하면 됩니다.

```
Collections.sort(words, new Comparator<String>() {  
    @Override  
    public int compare(String o1, String o2) {  
        return o1.compareTo(o2);  
    }  
});
```

하지만 함수형 프로그래밍을 하기에는 너무 복잡합니다.

## 3. 랴다식

Java 8에서 랴다식이 등장하면서

이름이 없는 클래스(익명 클래스) 말고

이름이 없는 메서드(람다식)를 사용하면서 더 간결한 표현이 가능해졌습니다.

```
Collections.sort(words, (o1, o2) -> o1.compareTo(o2));
```

이런 랴다식 사용은 컴파일러가 추론할 수 있기 때문에 사용 가능합니다.

## 4. 함수형 인터페이스

람다식을 이용하는데

해당 인터페이스에 추상 메서드가 여러 개라면 ?  
컴파일러가 추론하지 못하여 람다식을 이용할 수 없습니다.

```
interface Operator {  
    int plus(int leftOperator, int rightOperator);  
    int minus(int leftOperator, int rightOperator);  
    int multiply(int leftOperator, int rightOperator);  
    int divide(int leftOperator, int rightOperator);  
}
```

도대체 뭐야



```
Operator operator = (leftOperand, rightOperand) -> leftOperand + rightOperand;
```

람다식은 컴파일러가 추론할 수 있어야 합니다.

추상 메서드의 매개변수 타입, 반환 타입으로 추론할 수 있습니다.


```
interface Operator {  
    int calculate(int leftOperator, int rightOperator);  
}
```



저 녀석인가..

```
Operator operator = (leftOperand, rightOperand) -> leftOperand + rightOperand;
```

때문에 이렇게 추상 메서드가 오직 하나만 있는 인터페이스를 “함수형 인터페이스”라고 합니다.

 함수형 인터페이스 : 단 하나의 추상 메서드를 가지는 인터페이스

## 5. 함수형 인터페이스 특징

- 단 하나의 추상 메서드를 갖는다.
- static 메서드, default 메서드를 가질 수 있다.
- @FunctionalInterface

함수형 인터페이스 코드를 한번 작성해봤습니다.

```
@FunctionalInterface
interface Operator {
    int calculate(int leftOperator, int rightOperator);
}
```

### 5-1. @FunctionalInterface

이 `@FunctionalInterface` 애노테이션 추상 메서드가 하나임을 알려줍니다.

만약 아래와 같이 추상 메서드가 두 개라면 컴파일 시에 오류를 발생 시켜줍니다.

```
@FunctionalInterface
interface Operator {
    int calculate(int leftOperator, int rightOperator);
    int returnZero();
}
```

### 5-2. static 메서드, default 메서드

static 메서드, default 메서드는 추상 메서드 개수와 관계 없습니다.

```
@FunctionalInterface
public interface Operator {
    int calculation(int leftOperand, int rightOperand);

    default int returnZero() {
        return 0;
    }

    static int returnOne() {
        return 1;
    }
}
```

### 5-3. 함수형 인터페이스 사용

위에서 만든 함수형 인터페이스를 다음과 같이 사용할 수 있습니다.

```

public static void main(String[] args) {
    Operator plus = (leftOperand, rightOperand) -> leftOperand + rightOperand;
    Operator minus = (leftOperand, rightOperand) -> leftOperand - rightOperand;
    Operator multiply = (leftOperand, rightOperand) -> leftOperand * rightOperand;
    Operator divide = (leftOperand, rightOperand) -> leftOperand / rightOperand;

    plus.calculate(1, 2);
    minus.calculate(1, 2);
    multiply.calculate(1, 2);
    divide.calculate(1, 2);
}

@FunctionalInterface
interface Operator {
    int calculate(int leftOperand, int rightOperand);
}

```

함수형 인터페이스 `Operator` 를 보면

오직 하나의 추상 메서드를 가지고 있습니다.

```
int calculate(int leftOperand, int rightOperand)
```

그리고 이 추상 메서드 `calculate` 를 람다식으로

함수 단위로 재사용도 가능한 것을 확인할 수 있습니다.

```

(leftOperand, rightOperand) -> leftOperand + rightOperand
(leftOperand, rightOperand) -> leftOperand - rightOperand
(leftOperand, rightOperand) -> leftOperand * rightOperand
(leftOperand, rightOperand) -> leftOperand / rightOperand

```

## 6. 함수형 인터페이스 패키지



Java에는 미리 제공하는 함수형 인터페이스 패키지가 있습니다.

`java.util.function`을 보면 40여개의 함수형 인터페이스 종류가 있습니다.

Interface	Description
<b><code>BiConsumer</code></b> <T,U>	Represents an operation that accepts two input arguments and returns no result.
<b><code>BiFunction</code></b> <T,U,R>	Represents a function that accepts two arguments and produces a result.



Interface	Description
<b><u>BinaryOperator</u></b> <T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<b><u>BiPredicate</u></b> <T,U>	Represents a predicate (boolean-valued function) of two arguments.
<b><u>BooleanSupplier</u></b>	Represents a supplier of <code>boolean</code> -valued results.
<b><u>Consumer</u></b> <T>	Represents an operation that accepts a single input argument and returns no result.
<b><u>DoubleBinaryOperator</u></b>	Represents an operation upon two <code>double</code> -valued operands and producing a <code>double</code> -valued result.
....	...

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

표 안에 있는 함수형 인터페이스 중에

`BinaryOperator<T>` 를 보면 제네릭 타입이 있음을 알 수 있습니다.



#### “제네릭 함수형 인터페이스”

함수형 인터페이스에 있는 추상메서드에 제네릭 타입을 사용하여 모든 타입에 대해 허용하거나, 타입에 대해 제약이 존재하는 인터페이스를 “제네릭 함수형 인터페이스” 라고 합니다.

이러한 제네릭 함수형 인터페이스는 인자를 받아 어떤 값을 반환할 지 제네릭 타입을 통해 명시할 수 있습니다.

다시 돌아와서

Java 에서 제공하는 함수형 인터페이스를 가지고

이전에 구현한 사칙연산을 동일하게 구현해보겠습니다.

`BinaryOperator<T>` : <T> 인자 2개를 받고, <T>를 반환하는 인터페이스 입니다.

```
public static void main(String[] args) {
    BinaryOperator<Integer> plus = (leftOperand, rightOperand) -> leftOperand + rightOperand;
    BinaryOperator<Integer> minus = (leftOperand, rightOperand) -> leftOperand - rightOperand;
    BinaryOperator<Integer> multiply = (leftOperand, rightOperand) -> leftOperand * rightOperand;
    BinaryOperator<Integer> divide = (leftOperand, rightOperand) -> leftOperand / rightOperand;

    plus.apply(1, 2);
    minus.apply(1, 2);
    multiply.apply(1, 2);
    divide.apply(1, 2);
}
```

이미 Java에서 만들어진 함수형 인터페이스를 이용하니 조금 더 코드가 줄 수 있었습니다.

뿐만 아니라 불필요한 박싱으로 인한 객체 생성을 하지 않도록 도와주는 `IntConsumer` 같은 함수형 인터페이스도 존재합니다.

## 결론

함수형 프로그래밍의 장점인 side effect가 없다는 것을 보장해줍니다.

또한 함수형 인터페이스를 사용하면 코드가 매우 간결하여 가독성이 높아집니다.

## 참조

### Java 8 Functional Interfaces | DigitalOcean

Technical tutorials, Q&A, events - This is an inclusive place where developers can find or lend support and discover new ways to contribute to the community.

<https://www.digitalocean.com/community/tutorials/java-8-functional-interfaces>



<https://www.baeldung.com/java-functional-programming>

### Functional Interfaces in Java - GeeksforGeeks

Java has forever remained an Object-Oriented Programming language. By object-oriented programming language, we can declare that everything present in the Java programming language rotates throughout the Objects, except for some of the primitive data types and primitive methods for

<https://www.geeksforgeeks.org/functional-interfaces-java/>



### 람다식(feat. 익명 구현 클래스 vs 람다식)

선장님과 함께하는 마지막 자바 스터디입니다. (ㄹ) 자바 스터디 Github [github.com/whiteship/live-study](https://github.com/whiteship/live-study) 나의 Github [github.com/cmg1411/whiteShip\\_live\\_study](https://github.com/cmg1411/whiteShip_live_study) 람다식 글인데 갑자기 함수형 프로그래밍부터 시작한다.

뜬금없다고 느낄수도 있겠지만, 이것 알고 람다식을 보는 것과 모르고 보는 것은 정말로 다르다. 함수형 프로그

❗ <https://alkhwa-113.tistory.com/entry/%EB%9E%8C%EB%8B%A4%EC%8B%9Dfeat-%EC%9D%B5%EB%AA%85-%EA%B5%AC%ED%98%84-%ED%81%B4%EB%9E%98%EC%8A%A4-vs-%EB%9E%8C%EB%8B%A4%EC%8B%9D>



### Functional Interface란


Java8부터 함수형 프로그래밍을 지원한다. 함수를 일급객체처럼 다룰 수 있게 제공하는 Functional Interface 에 대해 알아볼 것이다. 단 하나의 추상 메서드를 가지는 인터페이스. - Java Language Specification 무슨 말 인지 코드로 보여주겠다. 이렇게 메서드를 하나만 가지는 인터페이스를 Functional interface 라고 한다. (단,

❗ <https://tecoble.techcourse.co.kr/post/2020-07-17-Functional-Interface/>



#### java.util.function (Java Platform SE 8 )

Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameter and return types are matched or adapted.

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

#### [Java]제네릭 함수형 인터페이스(Generic Functional Interface)

제네릭 함수형 인터페이스는 함수형 인터페이스의 추상 메서드에 제네릭 타입을 사용하여 모든 타입을 허용하거나 타입에 대해 제약이 존재하는 인터페이스입니다. 랴다식(Lambda expression)에 매개변수의 타입과 반환 타입을 설정할 수 없습니다. 다음 예제를 통해 알아보시다. 다음은 Integer 타입의 두 매개변수를 가지며,

❗ <https://developer-talk.tistory.com/461>

