

JAVA 함수형 프로그래밍

JAVA Interface

- Interface 키워드를 사용하여 정의
- 상수와 추상 메소드로 구성 (Java 8 부터 default 메소드와 static 메소드 사용 가능)
- 모든 상수는 public static final 타입 (생략가능)
- 모든 추상 메소드는 abstract public 타입 (생략가능)
- 클래스에서 인터페이스 구현은 implements 키워드를 사용하여 구현할 인터페이스 지정 후, 추상 메소드를 모두 오버라이드하여 내용을 완성하여야 함.

JAVA Interface 예시

```
interface JavaInterface {  
    String call();  
    boolean test();  
}
```

JAVA Interface 구현

인터페이스 상속 구현

```
class InterfaceImpl implements JavaInterface {  
  
    @Override  
    public String call() {  
        return "called";  
    }  
  
    @Override  
    public boolean test() {  
        return true;  
    }  
}  
  
public class Playground {  
  
    public void test1() {  
        InterfaceImpl impl = new InterfaceImpl();  
        impl.test();  
    }  
}
```

익명 클래스 구현

```
public class Playground {  
  
    public void test2() {  
        JavaInterface javaInterface = new JavaInterface() {  
            @Override  
            public String call() {  
                return "called";  
            }  
  
            @Override  
            public boolean test() {  
                return true;  
            }  
        };  
  
        javaInterface.call();  
    }  
}
```

JAVA Functional Interface

- 1개의 추상 메소드를 갖는 인터페이스
- JAVA 8 부터 인터페이스는 기본 구현체를 포함한 default 메소드를 포함할 수 있음
- 여러개의 default 메소드가 있더라도 추상 메소드가 오직 하나이면 함수형 인터페이스임.
- JAVA의 람다 표현식은 함수형 인터페이스에만 가능

Functional Interface 예시

```
@FunctionalInterface
interface TestFunctionalInterface<T> {
    T call();

    // default method 는 존재해도 상관없음
    default void printDefault() {
        System.out.println("Hello Default");
    }

    // static method 는 존재해도 상관없음
    static void printStatic() {
        System.out.println("Hello Static");
    }
}
```

Functional Interface 구현

인터페이스 상속 구현 가능

익명 클래스 구현 가능

람다식 가능

```
TestFunctionalInterface<String> testFunctionalInterface = () -> {  
    return "called";  
};  
  
testFunctionalInterface.call();
```

주요 자바 Functional Interface - Consumer

- 파라미터를 전달받아 사용하고 아무것도 반환하지 않음 – 이를 소비(Consume)한다고 표현.
- accept 추상 메소드를 가지고 있음.

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}
```


주요 자바 Functional Interface – Consumer 사용 예시

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);  
numbers.stream().forEach(number -> System.out.println(number));
```

```
System.out::println  
(  
:  
)
```

Legacy

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);  
for(Integer number: numbers) {  
    System.out.println(number);  
}
```

주요 자바 Functional Interface – Function

- 파라미터도 있고, 리턴값도 있음.
- 주로 파라미터를 리턴값으로 매핑($X \rightarrow Y$)할 때 사용

```
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
}
```

주요 자바 Functional Interface – Function 사용 예시

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
List<Integer> numbers2 = numbers.stream()  
    .map(number -> number * 2)  
    .toList();
```

Legacy

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
List<Integer> numbers2 = new ArrayList<>();
```

```
for(Integer number: numbers) {  
    numbers2.add(number);  
}
```

주요 자바 Functional Interface – Predicate

- 파라미터가 있고, 리턴값은 boolean임.
- 파라미터를 전달받아 검사하여 true 혹은 false를 반환할 때 사용.
- test 추상 메소드를 가지고 있음.

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     *         otherwise {@code false}
     */
    boolean test(T t);
}
```

주요 자바 Functional Interface – Predicate 사용 예시

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
List<Integer> evenNumbers = numbers.stream()  
    .filter(number -> number % 2 == 0)  
    .toList();
```

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
List<Integer> evenNumbers = new ArrayList<>();
```

```
for(Integer number: numbers) {  
    if( number % 2 == 0) {  
        evenNumbers.add(number);  
    }  
}
```

Integer 배열에서 짝수인 수만 뽑아서 2를 곱한 배열 만들기

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
List<Integer> evenNumbers = numbers.stream()  
    .filter(number -> number % 2 == 0)  
    .map(number -> number * 2)  
    .toList();
```

Legacy

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
List<Integer> evenNumbers = new ArrayList<>();
```

```
for(Integer number: numbers) {  
    if( number % 2 == 0) {  
        evenNumbers.add(number * 2);  
    }  
}
```