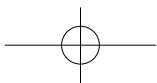
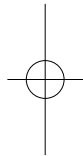
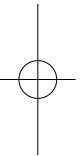
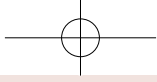


자바의 람다식 기초





A. 자바의 람다식 기초



람다식

Java 8에서 눈에 띄는 가장 큰 변화는 람다식(lambda expression)의 도입이다. 람다는 람다 대수(lambda calculus)에서 유래한다. 람다 대수에서 람다식(lambda expression)은 수학의 함수를 단순하게 표현하는 방법이다. 수학의 람다에 대해 간단히 알아보자.

다음은 x, y 의 합을 계산하는 수학 함수 f 를 보여준다.

```
f(x, y) = x + y // x, y의 합을 구하는 수학의 함수
```

이를 수학의 람다식으로 바꾸면, 다음과 같이 함수 이름을 빼고 간소하게 표현한다.

```
(x, y) -> x + y
```

수학에서 람다식은 이름 없는 함수를 뜻한다. 그리고 다음과 같이 괄호와 함께 x, y 에 대입될 값을 지정하면 람다식의 계산이 이루어진다. 이 람다식의 계산 결과는 5이다.

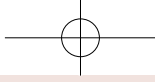
```
((x, y) -> x + y)(2, 3)  
= 2 + 3  
= 5
```

1930년대에 수학(mathematics)에 도입된 람다 대수와 람다식은, 1960년대 중반 프로그래밍 언어에 익명의 함수 형태로 도입되었고, 최근 여러 프로그래밍 언어에서 유행처럼 지원되고 있는데 자바는 Java 8부터 지원한다.

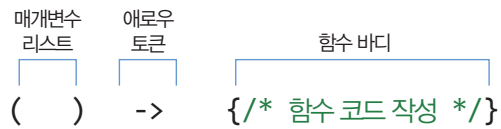
자바의 람다식

이제 프로그래밍 세계에서 람다식에 대해 알아보자. 람다는 이름 없는 함수, 람다식(lambda expression) 혹은 람다 함수(lambda function)로 불린다. 자바에서는 함수를 메소드라고 부르며 반드시 클래스의 멤버로만 존재할 수 있다. 이 글에서는 메소드와 함수를 같은 의미로 사용하기로 한다.

자바에서 람다식은 **[그림 1]**과 같이 매개변수 리스트, 애로우 토큰, 함수 바디의 3부



분으로 작성되는데 각 요소들에 대해 알아보자.



[그림 1] 자바의 람다식 구조

● 매개변수 리스트

매개변수 리스트에는 함수에 전달되는 매개변수들이 나열되며, 매개변수를 생략하면 컴파일러가 추론 기능을 이용하여 알아서 처리한다. 매개변수가 하나인 경우 괄호를 생략할 수 있다.

● 애로우 토큰(->)

애로우 토큰은 매개변수 리스트와 함수 코드를 분리시키는 역할이다. 독자들은 -> 기호를, '매개변수들을 전달하여 함수 바디 { }에 작성된 코드를 실행시킨다.'라는 뜻으로 이해하는 것이 좋겠다.

● 함수 바디

함수 바디는 함수의 코드이다. 중괄호({ })로 둘러싸는 일반적이지만, 한 문장인 경우 중괄호({ })를 생략해도 된다. 한 문장이더라도 return 문이 있으면 반드시 중괄호로 둘러싸야 한다.

람다식의 사례

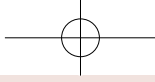
람다식을 만든 몇 가지 간단한 사례를 보자. 람다식을 만든다고 해서 바로 람다식이 실행되는 것은 아니다. 람다식을 호출하고 실행하는 과정은 다음 절에서 소개한다. 일단, 람다식을 만들어보자.

● 매개변수로 나이를 받아 출력하는 람다식

매개변수로 정수 값 나이를 받아 출력하는 람다식은 다음과 같이 작성할 수 있다.

```
(int age) -> { System.out.println("나이는 " + age); }
```

이 람다식은 매개변수의 타입을 생략하여 다음과 같이 작성해도 된다.



```
(age) -> { System.out.println("나이는 " + age); } // int 타입 생략
```

매개변수가 한 개인 경우 괄호를, 함수 바디가 한 문장인 경우 중괄호를 생략할 수 있는데, 앞의 람다식은 다음과 같이 생략하여 쓸 수 있다.

```
age -> System.out.println("나이는 " + age); // ()와 {} 생략
```

● 매개변수로 두 정수를 받아 합을 출력하는 람다식

x, y 두 값을 받아 합을 출력하는 람다식은 다음과 같이 2가지 방법으로 만들 수 있다.


```
(int x, int y) -> { System.out.println(x + y); }  
(x, y) -> System.out.println(x + y);
```

● return 문을 가진 람다식

람다 함수는 return 문을 사용하여 값을 리턴할 수 있다. 다음은 매개변수 x, y의 값을 합하여 리턴하는 람다식을 작성한 사례이다.

```
(x, y) -> { return x + y; } // 합을 리턴하는 람다식
```

return 문을 가진 함수에 중괄호({ })를 생략할 수 없다. 그러므로 다음은 오류이다.

 (x, y) -> return x + y; // 오류. {} 생략 안 됨

하지만, 이 람다식은 return을 생략하고 다음과 같이 쓸 수 있다.

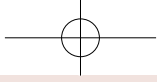
```
(x, y) -> x + y; // return 생략 가능. x+y의 합을 리턴하는 람다식
```

람다식 만들고 호출하기: 람다식은 함수형 인터페이스를 구현한 객체이다.

자바에서 람다식은 함수형 인터페이스에 선언된 추상 메소드를 구현하는 방식으로 작성된다. 지금부터 자바에서 람다식을 작성하는 과정을 하나씩 알아보자.

1. 함수형 인터페이스 작성

함수형 인터페이스(functional interface)란 추상 메소드 하나만 있는 인터페이스를



말한다. 다음은 추상 메소드 `calc()` 한 개만을 가진 함수형 인터페이스 `MyFunction`을 작성한 사례이다.

```
interface MyFunction { // 함수형 인터페이스
    int calc(int x, int y); // 추상 메소드
}
```

2. 함수형 인터페이스의 추상 메소드를 람다식으로 구현

람다식의 작성은 함수형 인터페이스의 추상 메소드에 코드를 작성하는 것과 같다. 인터페이스를 상속받은 클래스를 명시적으로 작성하지 않고 익명의 클래스가 만들어지는 방식으로 이루어진다. 그러므로 자바에서 람다식은 함수형 인터페이스를 구현한 객체(익명의 클래스)로 다루어져서 다음과 같이 람다식을 인터페이스 타입의 변수에 치환할 수 있다.

함수형인터페이스 변수 = 람다식;

이제, `MyFunction` 인터페이스의 `calc()` 메소드를 구현한 람다식을 만들어 보자. `calc()` 메소드는 2개의 `int` 타입 매개변수를 가지고 `int` 타입의 값을 리턴하므로, 람다식은 다음과 같이 만들 수 있다.

```
(x, y) -> { return x + y; }
```

람다식을 사용하기 위해서는 람다식을 다음과 같이 변수 `f`에 치환한다.

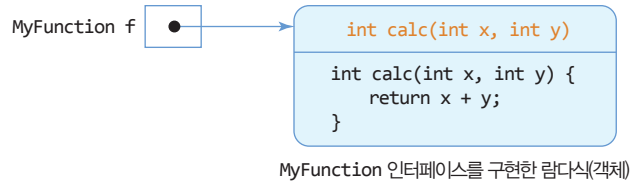
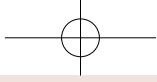
```
MyFunction f = (x, y) -> { return x + y; }
```

컴파일러는 이 문장을 다음과 같이 처리한다. `MyFunction` 인터페이스의 `calc()` 메소드를 구현한 람다식을 가진 익명의 클래스 객체를 생성하고, 이 객체를 레퍼런스 변수 `f`가 가리키게 한다(이런 이유로 람다식을 객체라고 말하기도 한다).

결국, 생성된 객체 안에는 `calc(x, y)` 메소드가 다음과 같이 구현된 것으로 보인 된다.

```
int calc(int x, int y) { return x + y; }
```

[그림 2]는 이것을 그림으로 보여준다.



[그림 2] 람다식과 함수형 인터페이스의 calc() 메소드를 구현한 객체

3. 람다식 호출

레퍼런스 변수 f를 이용하면 람다식을 호출할 수 있다. 다음은 정수 1, 2를 매개변수로 주어 람다식을 호출하고 리턴한 값 3을 출력한 사례이다.

```
System.out.println(f.calc(1, 2));
```

예제 1은 완성된 코드를 보여준다.



예제 1

매개변수 x, y의 합과 차를 출력하는 2개의 람다식 만들기

이 예제에서 람다식이 한 문장이므로 중괄호({})를 생략하였다. 예제를 실행하면 람다식은 x + y의 결과 값을 리턴한다.

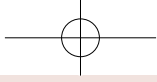
```
interface MyFunction { // 함수형 인터페이스
    int calc(int x, int y); // 람다식으로 구현할 추상 메소드
}

public class LambdaEx1 {
    public static void main(String[] args) {
        MyFunction add = (x, y) -> { return x + y; }; // 람다식
        MyFunction minus = (x, y) -> x - y; // 람다식. {}와 return 생략

        System.out.println(add.calc(1, 2)); // 합 구하기
        System.out.println(minus.calc(1, 2)); // 차 구하기
    }
}
```

실행 결과 

```
3
-1
```



예제 2

매개변수 x의 제곱을 출력하는 람다식 만들기

매개변수가 1개 있는 람다식을 만들어보자. 매개변수가 1개 있는 함수형 인터페이스를 먼저 작성하고 람다식을 작성한다. 매개변수가 1개이므로 람다식에서 매개변수 리스트의 괄호를 생략하였다.

```
interface MyFunction { // 함수형 인터페이스
    int calc(int x); // 람다식으로 구현할 추상 메소드
}

public class LambdaEx2 {
    public static void main(String[] args) {
        MyFunction square = x -> x * x;
        System.out.println(square.calc(2));
    }
}
```

실행 결과



4



예제 3

매개변수 없는 람다식 만들기

매개변수 없는 람다식을 만들어보자. 매개변수가 없는 함수형 인터페이스를 먼저 작성하고 람다식을 작성해야 한다.

```
interface MyFunction { // 함수형 인터페이스
    void print(); // 람다식으로 구현할 추상 메소드
}

public class LambdaEx3 {
    public static void main(String[] args) {
        MyFunction f = () -> { // 람다식 작성
            System.out.println("Hello");
        };

        f.print(); // 람다식 호출

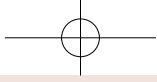
        f = () -> System.out.println("안녕"); // 람다식 작성

        f.print(); // 람다식 호출
    }
}
```

실행 결과



Hello
안녕



메소드의 매개변수로 람다식 전달

람다식 그 자체를 메소드의 매개변수로 사용할 수 있다. 예제 4를 통해 알아보자.



예제 4

람다식을 매개변수로 전달하기

이 예제에서는 곱을 리턴하는 람다식을 만들어, `printMultiply()` 메소드의 매개변수로 전달하는 사례를 보인다.

```
interface MyFunction {
    int calc(int x, int y);
}

public final class LambdaEx4 {
    public static void main(String[] args) {
        printMultiply(3, 4, (x,y)->x*y); // 람다식((x,y)->x*y)을 매개변수로 전달
    }

    static void printMultiply(int x, int y, MyFunction f) { // f로 (x,y)->x*y
                                                            람다식 전달받음
        System.out.println(f.calc(x, y));
    }
}
```

실행 결과



12

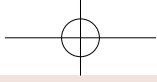


잠깐!

함수형 인터페이스 앞에 `@FunctionalInterface` 주석문(annotation)을 사용하여, 컴파일러에게 함수형 인터페이스를 작성을 알릴 수 있다.

```
@FunctionalInterface
interface MyFunction {
    int calc(int x, int y);
}
```

`@FunctionalInterface`의 사용은 컴파일러에게 인터페이스가 추상 메소드가 1개만 있는 함수형 인터페이스인지 확인하도록 하여, 처음부터 잘못된 인터페이스 작성을 막는 장점이 있다.



앞의 코드가 다음과 같이 작성되었다면, **MyFunction** 인터페이스는 2개의 추상 메소드를 가지고 있어 함수형 인터페이스가 아니므로 컴파일 오류가 발생한다.

```
@FunctionalInterface
오류 interface MyFunction { // 이 라인에 컴파일 오류 발생
    int calc(int x, int y);
    void print();
}
```

제네릭 함수형 인터페이스

함수형 인터페이스를 제네릭으로 만들면 다양한 타입을 처리할 수 있는 람다식을 만들 수 있다. 다음 예제를 통해 알아보자. 개념만 이해할 수 있도록 간단한 사례를 만들었다.



예제 5

매개변수로 주어진 객체를 문자열로 출력하는 람다식 만들기

함수형 인터페이스를 제네릭 타입으로 만들고, 람다식에 **String** 타입과 **Integer** 타입 객체를 각각 매개변수로 넘겨주는 사례이다.

```
@FunctionalInterface
interface MyFunction<T> { // 제네릭 타입 T를 가진 함수형 인터페이스
    void print(T x); // 람다식으로 구현할 추상 메소드
}

public class LambdaEx5 {
    public static void main(String[] args) {
        MyFunction<String> f1 = (x) -> System.out.println(x.toString());
        f1.print("ABC"); // String 객체를 람다식에 넘겨준다.

        MyFunction<Integer> f2 = (x) -> System.out.println(x.toString());
        f2.print(Integer.valueOf(100)); // Integer 객체를 람다식에 넘겨준다.
    }
}
```

실행 결과



```
ABC
100
```