

람다식(Lambda Expressions)

1. 람다식(Lambda Expressions) 이전 프로그램 구현 방식

람다식이 나오기 전과 후로 프로그램 구현 방식을 구분하자면 람다식이 소개되기 이전에 구현하던 방식을 '명령형 스타일', 람다식으로 구현하는 방식을 '함수형 스타일'이라고 한다. 명령형 스타일은 프로그램 구현 시 모든 작업 내용을 자바 코드로 자세하게 작성하는 방식이다. 그래서 구현해야 할 코드양이 많다. 그런데 함수형 스타일은 개발자가 핵심 내용만 구현하고 나머지는 자바 언어에서 자동으로 처리하는 방식이어서 코드가 간결하다.

(1) 명령형 스타일

다음은 명령형 스타일로 구현한 간단한 코드이다.

```
package exam_interface;
import java.util.Arrays;
import java.util.List;

public class Example01 {
    public static void main(String[] args) {
        // 명령형 스타일
        List<String> list = Arrays.asList("서울", "북경", "뉴욕", "방콕");
        boolean result = false;
        for(String city : list) {
            if(city.equals("서울")) {
                result = true;
                break;
            }
        }
        System.out.println(result);
    }
}
```

(2) 서술형 스타일

다음은 명령형 스타일의 코드를 서술형 스타일로 변경한 코드이다.

```
package exam_interface;

import java.util.Arrays;
import java.util.List;

public class Example01 {
    public static void main(String[] args) {
        //서술형 스타일
```

```
List<String> list = Arrays.asList("서울", "북경", "뉴욕", "방콕");
System.out.println(list.contains("서울"));

}

}
```

서술형 스타일은 명령형 스타일처럼 변수와 로직을 구현해서 실행하는 것이 아니라, 처리 로직이 구현된 메서드를 호출해서 처리한다.

(3) 함수형 스타일

함수형 스타일은 서술형 스타일에 객체의 개념과 명령문을 추가하여 처리하는 방식이다. 이 방식을 지원하는 핵심 기능이 바로 랴다식이다.

2. 인터페이스 구현 방법

구현하고자 하는 인터페이스는 다음과 같다.

```
interface MyInterface {
    public void print();
}
```

2.1. 방법1: implements 키워드로 클래스 선언

인터페이스를 구현하는 가장 일반적인 방법은 클래스를 선언할 때 implements 인터페이스명을 선언한 후 인터페이스에 선언된 추상 메서드를 오버라이딩하는 것이다.

```
class MyClass1 implements _____ {
    @Override
    _____ {
        System.out.println("MyClass1");
    }
}
```

인터페이스에 선언된 메서드를 실행할 때는 인터페이스를 구현한 클래스의 인스턴스를 생성한 후 참조 변수를 이용해 호출한다.

```
_____ mc1 = new _____ ;
mc1.print();
```

방법2: 익명 클래스 사용

인터페이스를 구현하는 방법 중 하나는 익명 클래스를 사용하는 것이다. 익명 클래스로 인터페이스를 구현할 때는 클래스를 별도로 선언할 필요없이 인터페이스 구현과 동시에 인스턴스를 생성할 수 있다.

```
MyInterface mi = _____
    @Override
    _____ {
        System.out.println("익명 클래스로 구현");
    }
```

```
};

mi.print(); // 익명 클래스로 구현
```

2.3. 방법3: 선언, 생성, 호출을 한번에 처리

인터페이스 구현과 동시에 객체가 생성되고 메서드를 호출이 실행되기 때문에 참조변수가 필요하지 않다.

```
(
    _____
    @Override
    _____ {
        System.out.println("선언, 생성, 호출을 한번에 처리");
    }
____)._____ ; // 선언, 생성, 호출을 한번에 처리
```

2.4. 매개변수

인터페이스 타입으로 매개변수가 선언된 경우를 살펴보면 다음의 test() 메서드는 MyInterface 타입으로 매개변수가 선언되어 있다.

```
public static void test(MyInterface mi) {
    mi.print();
}
```

test() 메서드 호출 시 인터페이스를 구현한 클래스의 참조변수를 인자로 전달할 수 있다.

```
test(mc1); // MyClass1
test(mi); // 익명 클래스로 구현
```

2.5. 리턴타입

인터페이스 타입으로 반환값이 선언된 메서드를 살펴보면 test2()메서드는 MyInterface 타입을 반환한다.

```
public static MyInterface test2() {
    MyInterface mi = new MyInterface() {
        @Override
        public void print() {
            System.out.println("test2()메서드에서 반환된 MyInterface");
        }
    };
    return mi;
}
```

다음은 인터페이스를 반환하는 메서드를 호출하는 메서드 예제 코드이다.

```
MyInterface mi2 = test2();
mi2.print(); // test2() 메서드에서 반환된 MyInterface
```

3. 람다식 사용하기

3.1. 람다식 기본

람다식을 사용해 인터페이스를 구현하고 생성한 후 사용하는 함수형 스타일로 코드를 작성하고자 한다.

(1) 일반 인터페이스 구현

```
public static MyInterface test3() {  
    return new MyInterface() {  
        @Override  
        public void print() {  
            System.out.println("Hello");  
        }  
    };  
}
```

test3() 메서드에서 반환한 익명 클래스를 사용하려면 다음처럼 test3() 메서드를 호출하고, 반환되는 값을 저장한다. 이때 반환하는 익명 클래스가 구현한 MyInterface 타입으로 참조변수를 선언한 후 참조변수를 이용해 print() 메서드를 호출한다.

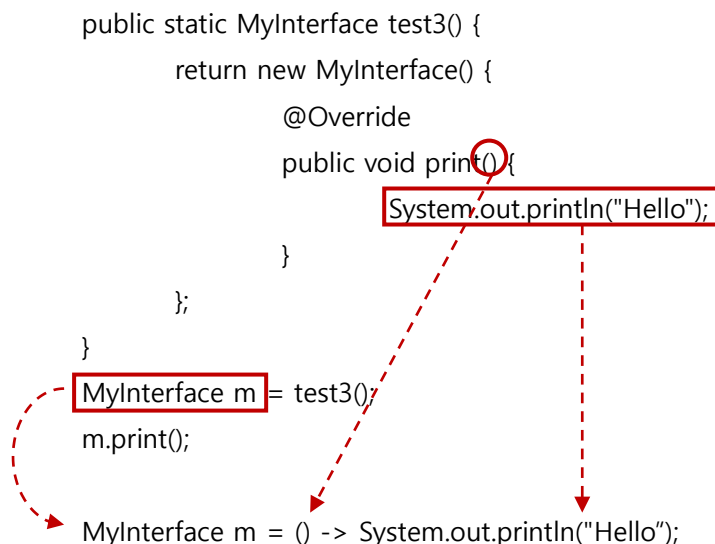
```
MyInterface m = test3();  
m.print(); // test3() 메서드에서 반환된 MyInterface
```

(2) 람다식 구현

바로 앞에서 구현한 코드를 람다식으로 구현하면 다음과 같다.

```
MyInterface m = () -> System.out.println("Hello");
```

다음은 명령형 스타일과 함수형 스타일 코드를 비교한 것이다.



메서드의 매개변수들이 선언되는 괄호()는 그대로 표현하고, 인터페이스에 선언된 추상 메서드의 본문을 구현할 때는 화살표(->) 기호 다음에 작성한다. 본문을 구현할 때 명령문이 한 줄이면 위와 같이 작성하고, 여러 줄일 때는 중괄호 {}로 감싸준다. 이때 인터페이스에 선언된 추상 메서드가 여러 개일 때는 어

떤 메서드의 본문인지 구별할 방법이 없다. 따라서 람다식으로 구현하려는 인터페이스는 반드시 하나의 메서드만 선언되어야 한다. 그리고 **하나의 메서드만 선언된 인터페이스를 “함수형 인터페이스”라고 한다.** 정리하면 람다식으로 구현할 수 있는 인터페이스는 함수형 인터페이스만 가능하다.

(3) 람다식 문법

람다식의 문법은 다음과 같다.

```

() -> 명령문; ← 함수형 인터페이스의 추상 메서드 구현 시 명령문이 한 개일 때
() -> {      ← 함수형 인터페이스의 추상 메서드 구현 시 명령문이 여러 개일 때
    명령문1;
    명령문n;
};
    
```

LambdaExample01.java

```

package exam_interface;

interface Multiply {
    double getValue();
}

public class LambdaExample01 {
    public static void main(String[] args) {
        Multiply m;
        m = new Multiply() {
            @Override
            public double getValue() {
                return 3 + 6;
            }
        };
        System.out.println(m.getValue());

        m = () -> 5.0 * 2;
        System.out.println(m.getValue());

        m = () -> 10 / 3;
        System.out.println(m.getValue());
    }
}
    
```

(4) @FunctionalInterface 어노테이션

람다식으로 구현할 수 있는 인터페이스는 하나의 추상 메서드를 포함하는 함수형 인터페이스만 가능하다. 따라서 함수형 인터페이스를 작성할 때 두 개 이상의 추상 메서드가 선언되는 오류를 예방하고자

@FunctionalInterface 어노테이션을 사용할 수 있다.

```
MyInterface.java
1 package exam_interface;
2
3 @FunctionalInterface
4 interface MyInterface {
5     public void put();
6     public void get();
7 }
```

@FunctionalInterface 어노테이션을 지정하면 하나 이상의 추상 메서드가 선언될 때 위와 같이 컴파일 오류가 발생하도록 명시해 준다.

(5) 람다식 매개변수

【 매개변수가 있는 람다식 구현 】

(변수명) -> 명령문: ← 매개변수가 한 개일 때
(변수명1, ...변수명n) -> 명령문; ← 매개변수가 여러 개일 때

지정해 줄 때 변수명 앞에 타입을 지정하지 않는다는 것이다. 이처럼 변수의 타입을 지정하지 않으면 자바 언어에서는 자동으로 인터페이스에 선언된 추상 메서드의 매개변수 타입으로 처리해주므로 생략해도 상관없다. 이처럼 함수형 스타일은 개발자가 핵심적인 내용만 구현하고 부가적인 처리는 자바 언어에 맡기는 특징이 있다.

물론 괄호 ()안에 타입을 지정하고 싶다면 다음처럼 할 수 있다.

(타입 변수명) -> 명령문;

- 매개변수를 한 개 사용하는 람다식

LambdaExample02.java

```
package exam_lambda;

interface Verify {
    boolean check(int n);
}

public class LambdaExample02 {
    public static void main(String[] args) {
        Verify isEven = new Verify() {
            @Override
            public boolean check(int n) {
                return (n % 2) == 0;
            }
        };
        System.out.println("결과값: "+isEven.check(3));
    }
}
```

```

        isEven = (n) -> (n % 2) == 0;
        System.out.println("결과값: "+isEven.check(10));

        Verify isOdd = (n) -> (n % 2) == 1;
        System.out.println("결과값: "+isOdd.check(9));
    }
}

```

- 매개변수를 여러 개 사용하는 람다식

LambdaExample03.java
<pre> package exam_lambda; interface Verify2 { boolean check(int n, int d); } public class LambdaExample03 { public static void main(String[] args) { Verify2 vf = (n, d) -> (n % d) == 0; System.out.println("결과값: " + vf.check(24, 3)); } } </pre>

(6) 람다식 블록

람다식 문법에서 () -> 기호 다음 부분은 람다식 본문이다. 람다식 본문의 명령문이 한 줄이면 다음처럼 표현하며 이때는 return 문을 생략해도 명령문에서 처리된 값이 자동으로 반환된다. 람다식 본문에 명령문이 여러 줄로 구현하려면 다음처럼 블록{}으로 감싸야 한다.

[람다식 본문이 여러 줄일 때]
<pre> () -> { 명령문1; 명령문n; return 값; } </pre>

람다식에서 return문은 명령문이 한 줄 일 때는 생략해도 되지만, 블록 {}을 사용하면 생략할 수 없다.

LambdaExample04.java
<pre> package exam_lambda; interface NumberFunction { int sum(int n); } </pre>

```

}

public class LambdaExample04 {
    public static void main(String[] args) {
        NumberFunction number = (n) -> {
            int result = 0;
            for(int i=1; i<=n; i++) {
                result += i;
            }
            return result;
        };

        System.out.println("1부터 10까지의 합: " + number.sum(10));
        System.out.println("1부터 100까지의 합: " + number.sum(100));
    }
}

```

(7) 제네릭 함수형 인터페이스

```

interface StringFunction {
    String modify(String str);
}

```

```

interface IntegerFunction {
    Integer modify(Integer in);
}

```

그래서 두 개의 인터페이스를 한번에 처리할 수 있는 제네릭 함수형 인터페이스(MyFunction)를 정의하면 된다.

```

interface _____ {
    _____
}

```

[제네릭 인터페이스에 타입 인자 전달]

인터페이스<타입인자> 변수명 = () -> 명령문

LambdaExample05.java

```

package exam_lambda;

interface MyFunction<T> {
    T modify(T data);
}

```



```
}

public class LambdaExample05 {
    public static void main(String[] args) {
        MyFunction<String> mf1 = (str) -> str.toUpperCase() + ":" + str.length();
        System.out.println("결과값: " + mf1.modify("java"));
        System.out.println("결과값: " + mf1.modify("java programming"));

        MyFunction<Integer> mf2 = (n) -> n * 2;
        System.out.println("결과값: " + mf2.modify(22));
        System.out.println("결과값: " + mf2.modify(43));
    }
}
```