람다

소유자	종수 김
태그	

- 변하는 부분과 변하지 않는 부분을 분리.
- 변하는 부분은 그대로 유지하고 변하지 않는 부분은 어떻게 처리할 것인가
- **람다를 사용하면 코드 조각을 매우 편리하게 전달 할**

람다가 필요한 이유

리팩토링 전

```
public class Ex1Main {
    public static void helloDice() {
        long startNs = System.nanoTime();
        //코드 조각 시작
        int randomValue = new Random().nextInt(6) + 1; System.out.println("주사
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns"); }
    public static void helloSum() {
        long startNs = System.nanoTime();
        //코드 조각 시작
        for (int i = 1; i <= 3; i++) {
            System.out.println("i = " + i);
        }
        //코드 조각 종료
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns"); }
    public static void main(String[] args) {
        helloDice();
        helloSum();
    }
}
```

리팩토링 후

```
public class Ex1Ref {
```

```
    static class Dice implements Procedure {
        @Override
        public void run() {
            int randomValue = new Random().nextInt(6) + 1; System.out.println("주
        }
    }

    static class Sum implements Procedure {
        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
        }
    }

    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();

        procedure.run();

        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }
    public static void main(String[] args) {
        Dice dice = new Dice();
        Sum sum = new Sum();

        hello(dice);
        hello(sum);
    }
}
```

코드 조각(= 메서드)을 매개변수로 넘김으로써 재사용성을 높임.
변하지 않는 '코드 조각'을 매개변수화. <<
코드 조각을 외부에서 전달 받은것.

어떻게 외부에서 코드 조각을 전달할 수 있을까 ?

= 코드 조각은 보통 메서드에 정의. 따라서 코드 조각을 전달하기 위해서는 메서드가 필요. 근데, 메서드를 전달할 수 있는 방법이 없음. 인스턴스를 전달하고 인스턴스의 메서드를 호출.

(인터페이스를 정의하고, 다형성을 갖는 구현클래스를 생성)

동작 매개변수화(Behavior Parameterization)

- ***값 매개변수화(Value Parameterization)***
  문자값(**
  Value*), 숫자값(****Value***)처럼 구체적인 값을 메서드(함수) 안에 두는 것이 아니라, ****매개변수***(파라미터)를
  통해 외부에서 전달 받도록 해서, 메서드의 동작을 달리하고, 재사용성을 높이는 방법을
  **
  값 매개변수화*라 한다.
  값 매개변수화, 값 파라미터화 등으로 부른다.
- ***동작 매개변수화(Behavior Parameterization)***
  코드 조각(코드의 동작 방법, 로직, **
  Behavior*)을 메서드(함수) 안에 두는 것이 아니라, ****매개변수***(파라미터)를 통
  해서 외부에서 전달 받도록 해서, 메서드의 동작을 달리하고, 재사용성을 높이는 방법을
  동작 매개변수화라 한다.
  동작 매개변수화, 동작 파라미터화, 행동 매개변수화(파라미터화), 행위 파라미터화 등
  으로 부른다.
- ***정리하면 다음과 같다.***
  **
  값 매개변수화*: 값(숫자, 문자열 등)을 바꿔가며 메서드(함수)의 동작을 달리 함
  **
  동작 매개변수화*: 어떤 동작(로직)을 수행할지를 메서드(함수)에 전달(인스턴스 참조,
  람다 등)

자바에서 동작 매개변수화를 하려면 클래스를 정의하고 해당 클래스를 인스턴스로 만들어서 전달해야 한다.
자바8에서 등장한
람다를 사용하면 코드 조각을 매우 편리하게 전달 할 수 있는데, 람다를 알아보기 전에 기존에 자바로
할 수 있는 다양한 방법들을 먼저 알아보자.

익명 클래스

```
public class Ex1RefV2 {
    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();

        procedure.run();

        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }
    public static void main(String[] args) {
        Procedure dice = new Procedure() {
            @Override
            public void run() {
                int randomValue = new Random().nextInt(6) + 1; System.out.println("
            }
        };
        Procedure sum = new Procedure() {
            @Override
            public void run() {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("i = " + i);
                }
            }
        };

        hello(dice);
        hello(sum);
    }
}
```

→ 매개변수로 직접 넘김

```
public class Ex1RefV3 {
    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();

        procedure.run();
```

```
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }
    public static void main(String[] args) {


        hello(new Procedure() {
            @Override
            public void run() {
                int randomValue = new Random().nextInt(6) + 1; System.out.println("
            }
        });
        hello(new Procedure() {
            @Override
            public void run() {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("i = " + i);
                }
            }
        });
    }
}
```

람다

자바에서 메서드의 매개변수에 인수로 전달할 수 있는 것.

1. int, double과 같은 기본형 타입
2. Procedure, Member와 같은 참조형 타입.

코드 조각을 넘기기 위해서는 클래스를 정의하고 인스턴스를 생성하고, 넘겨야 하는데 이 작업이 번거로움.

람다를 통해서는 코드 블럭을 전달할 수 있음.

```
public class Ex1RefV4 {
    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();
```

```
        procedure.run();

        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }
    public static void main(String[] args) {
        hello(() -> {
                int randomValue = new Random().nextInt(6) + 1;
                System.out.println("주사위 = " + randomValue);
            });
        hello(() -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
        });
    }
}
```

람다란 ?

- 결국 람다는, 변하지 않는 부분과 변하는 부분 중 '변하는 부분'을 매개변수화 하기 위한 방법 중 하나.
- '변하는 부분'을 매개변수화 하는 이유는 재사용성을 높이기 위해.
- '변하는 부분'은 코드 블럭 (= 메서드에 정의된 로직)일 수도 있는데. 이를 매개변수로 넘기기 위해서는, 인터페이스를 만들고, 구현체를 만들고, 인스턴스를 정의하고, 메서드를 작성하는 등 많은 작업이 필요.
- 이를 최대한으로 압축해서 사용할 수 있는 방식이 바로 람다.

함수 vs 메서드

둘 다 어떤 작업(로직)을 수행하는 코드의 묶음.

함수 : 독립적으로 존재하며, 클래스(객체)와 직접적인 연관이 없음.

- 객체지향 언어라 하더라도 python, js처럼 클래스 밖에서도 정의할 수 있는 함수 개념을 지원하는 경우 이를 그냥 함수라고 부름.

메서드 : 클래스(객체)에 속해 있는 함수.

- 객체의 상태(필드, 프로퍼티 등)에 직접 접근하거나, 객체가 제공해야할 기능을 구현
- 클래스 내부에 정의된 함수는 보통 메서드 라고 함.

호출 방식과 스코프

- ***함수(Function)***

호출 시에 객체 인스턴스가 필요 없다.
보통 `이름(매개변수)` 형태로 호출된다.
지역 변수, 전역 변수 등과 함께 동작하며, 클래스나 객체 특유의 속성(인스턴스 변수 등)은 다루지 못한다.

- ***메서드(Method)***
  보통 `객체(인스턴스).메서드이름(매개변수)` 형태로 호출한다.
  호출될 때, 해당 객체의 필드(속성)나 다른 메서드에 접근 가능하며, 이를 이용해 로직을 수행한다.
  인스턴스 메서드, 클래스(정적) 메서드, 추상 메서드 등 다양한 형태가 있을 수 있다.
- ***정리***
  **
  메서드는 기본적으로 클래스(객체) 내부의 함수*를 가리키며, 객체의 상태와 밀접한 관련이 있다.
  **
  함수는 클래스(객체)와 상관없이, 독립적으로 호출 가능한 로직의 단위*이다.
  메서드는 객체지향에서 클래스 안에 정의하는 특별한 함수라고 생각하면 된다.

따라서 ****함수****와 ****메서드****는 수행하는 역할 자체는 같지만, 소속(클래스 or 독립)과 호출 방식에서 차이

람다 시작

매개 변수가 없는 익명클래스와 람다

```
package lambda.lambda1;

import lambda.Procedure;

public class ProcedureMain1 {
    public static void main(String[] args) {
        Procedure procedure = new Procedure() {
```

```
            @Override
            public void run() {
                System.out.println("hello! lambda!");
            }
        };

        procedure.run();
    }
}
```

```
package lambda.lambda1;

import lambda.Procedure;

public class ProcedureMain2 {
    public static void main(String[] args) {
        Procedure procedure = () -> {
            System.out.println("hello2 lambda2");
        };

        procedure.run();
    }
}
```

매개 변수가 있는 익명 클래스와 람다

```
package lambda.lambda1;

import lambda.MyFunction;

public class MyFunction1 {
    public static void main(String[] args) {
        MyFunction myFunction = new MyFunction() {
            @Override
            public int apply(int a, int b) {
                return a + b;
```

```
        }
    };

    int result = myFunction.apply(1, 2);
    System.out.println("result = " + result);
  }
}
```

```
package lambda.lambda1;

import lambda.MyFunction;

public class MyFunction2 {
  public static void main(String[] args) {
    MyFunction myFunction = (a, b) -> {
      return a + b;
    };

    int result = myFunction.apply(1, 2);
    System.out.println("result = " + result);
  }
}
```

- 람다는 () -> {}와 같이 표현.
  - () : 매개변수
  - {} : 본문(코드조각)

익명 클래스를 만들기 위한 모든 부분을 생략하고, 매개 변수와 코드 조각만 존재.

람다 정의

자바에서 함수형 프로그래밍을 지원하기 위한 핵심기능.

람다는 익명 함수. 따라서, 이름 없이 함수를 표현

표현 방법

```
일반적인 메서드
반환타입 이름(변수...){
   본문
}

람다
(변수) -> {
 본문
}
```

- 람다는 표현이 간결
- 람다는 변수처럼 다룰 수 있음

```
Procedure procedure = () -> { // 람다를 변수에 담음
  System.out.println("hello! lambda");
};

procedure.run(); // 변수를 통해 람다를 실행
```

- 람다도 익명 클래스처럼 클래스가 만들어지고, 인스턴스가 생성.

```
package lambda.lambda1;

import lambda.Procedure;

public class InstanceMain1 {
    public static void main(String[] args) {
        Procedure procedure1 = new Procedure() {
            @Override
            public void run() {
                System.out.println("Hello");
            }
        };

        System.out.println("procedure1.getClass() = " + procedure1.getClass(
        System.out.println("class.instance" + procedure1);
```

```
        Procedure procedure2 = () -> {
            System.out.println("Hello");
        };
        System.out.println("procedure1.getClass() = " + procedure2.getClass
        System.out.println("class.instance" + procedure2);
    }
}

---
procedure1.getClass() = class lambda.lambda1.InstanceMain1$1
class.instancelambda.lambda1.InstanceMain1$1@36baf30c
procedure1.getClass() = class lambda.lambda1.InstanceMain1$$Lambda$1
class.instancelambda.lambda1.InstanceMain1$$Lambda$14/0x000000700
```

> **용어 - 람다 vs 람다식(Lambda Expression)****
> **
> **람다****: 익명 함수를 지칭하는 일반적인 용어다. 쉽게 이야기해서 개념이다.
> **
> **람다식****: (매개변수) { 본문 } 형태로 람다를 구현하는 구체적인 문법 표현을 지칭한다.
>
> 쉽게 이야기해서 람다는 개념을, 람다식은 자바에서 그 개념을 구현하는 구체적인 문법을 의미한다. 람다가 넓은
> 의미이고, 또 실무에서 두 용어를 구분해서 사용하지는 않기 때문에 여기서는 대부분 간결하게 람다라고 하겠다.

정리

- 람다를 사용하면 익명 클래스 사용의 보일러플레이트 코드를 크게 줄이고, 간결한 코드로 생산성과 가독성을 높임
- 대부분의 익명 클래스는 람다로 대체 가능

    - 물론 완전히 대체하는 것은 불가능 차이가 있음.
- 람다를 사용할 때 new 키워드를 사용하진 않지만, 람다도 익명 클래스처럼 인스턴스가 생성

함수형 인터페이스

정확히 하나의 추상 메서드를 가지는 인터페이스.

- 람다는 추상 메서드가 하나인 함수형 인터페이스에만 할당
- 단일 추상 메서드를 줄여서 SAM(Single Abstract Method)
- 클래스, 추상 클래스에는 할당 불가능

인터페이스에는 abstract가 생략되어있음.

```
package lambda.lambda1;

public class SamMain {
    public static void main(String[] args) {
        SamInterface samInterface = () -> {
            System.out.println("가능");
        };

        NotSamInterface notSamInterface = () -> {
            System.out.println("불가능");
        };
    }
}
```

- 어느 메서드에 할당해야하는지 알 수 없음. 그래서 사용 불가능

Functional Interface

▼ 함수형 인터페이스임을 보장하는 방법

@FunctionalInterface

잠깐 자바 기본으로 돌아가보자.

```
public class Car {
    public void move() {
        System.out.println("차를 이동합니다.");
    }
}
```

```
public class ElectricCar extends Car {
    @Override
    public void movee() {
        System.out.println("전기차를 빠르게 이동합니다.");
    }
}
```

메서드를 재정의할 때 실수로 재정의할 메서드 이름을 다르게 적으면 재정의가 되지 않는다. 이 예제에서 부모는 `move()` 인데 자식은 `movee()` 라고 `e` 를 하나 더 잘못 적었다. 이런 문제를 컴파일 단계에서 원천적으로 막기 위해 `@Override` 애노테이션을 사용한다. 이 애노테이션 덕분에 개발자가 할 수 있는 실수를 컴파일 단계에서 막을 수 있고, 또 개발자는 이 메서드가 재정의 메서드인지 명확하게 인지할 수 있다.

함수형 인터페이스는 단 하나의 추상 메서드(SAM: Single Abstract Method)만을 포함하는 인터페이스이다.
그리고 람다는 함수형 인터페이스에만 할당할 수 있다.
그런데 단 하나의 추상 메서드만을 포함한다는 것을 어떻게 보장할 수 있을까?
`@FunctionalInterface` 애노테이션을 붙여주면 된다. 이 애노테이션이 있으면 단 하나의 추상 메서드가 아니면 컴파일 단계에서 오류가 발생한다. 따라서 함수형 인터페이스임을 보장할 수 있다.

```
package lambda.lambda1;

@FunctionalInterface // 애노테이션 추가
public interface SamInterface {
    void run();
}
```

- `@FunctionalInterface` 을 통해 함수형 인터페이스임을 선언해두면, 이후에 누군가 실수로 추상 메서드를 추가할 때 컴파일 오류가 발생한다.

```
package lambda.lambda1;

@FunctionalInterface // 애노테이션 추가
public interface SamInterface {
    void run();
    void gogo(); // 실수로 누군가 추가시 컴파일 오류 발생
}
```

람다와 시그니쳐

람다를 함수형 인터페이스에 할당할 때는 메서드의 형태를 정의하는 요소인 메서드 시그니처가 맞아야함.

1. 이름
2. 매개변수
3. 반환타입

- 시그니처에서 이름은 제외해도 됨. (람다는 익명함수니까.)
- 타입과 순서만 맞다면 매개변수 이름은 상관없음.

람다와 생략

람다는 간결한 코드 작성을 위해 다양한 문법 생략을 지원

```
package lambda.lambda1;

import lambda.MyFunction;

public class LambdaSimple1 {
    public static void main(String[] args) {
        // 기본
        MyFunction myFunction = (int a, int b) -> {
            return a + b;
        };
        System.out.println("function1 : " + myFunction.apply(1, 2));


        //단일 표현식인 경우 중괄호와 리턴 생략 가능
        MyFunction myFunction1 = (int a, int b) -> a + b;
        System.out.println("function2 : " + myFunction1.apply(1, 2));


        // 단일 표현식이 아닐 경우 중괄호와 리턴 모두 필수
        MyFunction myFunction2 = (int a, int b) -> {
            System.out.println("람다 실행");
```

```
            return a + b;
        };
        System.out.println("function3 : " + myFunction2.apply(1,2));

        // 매개변수 타입 생략 가능
        MyFunction myFunction3 = (a, b) -> a+b;
        System.out.println("function4 : " + myFunction3.apply(1, 2));
    }
}

package lambda.lambda1;

import java.util.function.DoubleUnaryOperator;

public class LambdaSimple4 {
    public static void main(String[] args) {
        MyCall call1 = (int value) -> value * 2;
        MyCall call2 = (value) -> value * 2;
        MyCall call3 = value -> value * 2; // 매개변수 1개만 있을 때 .
    }
    interface MyCall {
        int call(int value);
    }
}
```

정리

매개변수 타입*: 생략 가능하지만 필요하다면 명시적으로 작성할 수 있다.

반환 타입*: 문법적으로 명시할 수 없고, 식의 결과를 보고 컴파일러가 항상 추론한다.
람다는 보통 간략하게 사용하는 것을 권장한다.

단일 표현식이면 중괄호와 리턴을 생략하자.
타입 추론을 통해 매개변수의 타입을 생략하자. (컴파일러가 추론할 수 있다면, 생략하자)

람다의 전달

함수형 인터페이스를 통해 변수에 대입하거나, 메서드에 전달하거나 반환 가능.

```
package lambda.lambda2;

import lambda.MyFunction;

public class LambdaPassMain1 {
    public static void main(String[] args) {
        MyFunction add = (a, b) -> a + b;
        MyFunction sub = (a, b) -> a - b;

        System.out.println("add.apply(1,2) = " + add.apply(1, 2));
        System.out.println("sub.apply(1,2) = " + sub.apply(1, 2));

        MyFunction cal = add;
        System.out.println("cal.apply(1,2) = " + cal.apply(1, 2));

        cal = sub;
        System.out.println("cal.apply(1,2) = " + cal.apply(1, 2));
    }
}
```

- 람다 자체도 인스턴스화 된 참조 주소를 갖고 있기 때문에, 변수로 대입하는 것이 가능.

```
package lambda.lambda2;

import lambda.MyFunction;

// 2. 람다를 메서드(함수)에 전달하기
public class LambdaPassMain2 {
    public static void main(String[] args) {
        MyFunction add = (a, b) -> a + b;
        MyFunction sub = (a, b) -> a - b;

        System.out.println("변수를 통해 전달");
        calc(add);
        calc(sub);

        calc((a, b) -> a + b);
```

```
        calc((a, b) -> a - b);
    }

    public static void calc(MyFunction myFunction) {
        int a = 1;
        int b = 2;

        System.out.println("계산 시작");
        int result = myFunction.apply(a, b);
        System.out.println("계산 결과" + result);
    }
}
```

- 매개 변수로 람다를 전달하는 것이 가능.
- 일반적인 참조를 매개 변수에 전달하는 것과 같음.

```
package lambda.lambda2;

import lambda.MyFunction;

public class LambdaPassMain3 {
    public static void main(String[] args) {
        MyFunction add = getOperation("add");
        System.out.println(add.apply(1, 2));

        MyFunction sub = getOperation("sub");
        System.out.println(sub.apply(1, 2));

        MyFunction x = getOperation("x");
        System.out.println(x.apply(1, 2));
    }
    static MyFunction getOperation(String operator) {
        switch (operator) {
            case "add":
                return (a, b) -> a + b;
            case "sub":
                return (a, b) -> a - b;
            default:
```

```
            return (a, b) -> 0;
        }
    }
}
```

- 람다를 반환 하는 것도 가능.

람다를 변수에 대입하거나, 함수의 매개변수로 전달하거나, 리턴 받거나 등.
마치 값 처럼 사용하는 것이 가능.

고차 함수

람다

함수형 인터페이스를 구현한 익명 클래스 인스턴스와 같은 개념

람다의 전달 정리

- 람다를 변수에 대입한다는 것은
  = 람다 인스턴스의 참조값을 대입
  - `MyFunction add = (a, b) -> a + b;` 처럼 함수형 인터페이스 타입의 변수에 람다 인스턴스의 참조를 대입한다.
- 람다를 메서드의 매개변수나 반환 값으로 넘기다는 것
  = 람다 인스턴스의 참조값을 전달, 반환
  - 메서드 호출 시 람다 인스턴스의 참조를 직접 넘기거나, 이미 람다 인스턴스를 담고 있는 변수를 전달한다.

```
//변수에 담은 후 전달
MyFunction add = (a, b) -> a + b;
calculate(add);
// 직접 전달
calculate((a, b) -> a + b);
```

- 코드의 간결성과 유연성이 높아짐.

고차함수(Higher-Order Function)

함수를 값처럼 다루는 함수.

- 함수를 인자로 받는 함수

```
// 함수(람다)를 매개변수로 받음
static void calculate(MyFunction function) { // MyFunction 내에 함수를 전달
// ...
}
```

- 함수를 반환하는 함수

```
/ 함수(람다)를 반환
static MyFunction getOperation(String operator) {
// ...
    return (a, b) -> a + b;
 }
```

즉, 매개변수나 반환 값에 함수(또는 람다)를 활용하는 함수가 고차함수.

자바에서 람다(익명 함수)는 함수형 인터페이스를 통해서만 전달 가능.

자바에서 함수를 주고 받는다는 것은
= 함수형 인터페이스를 구현한 어떤 객체(람다, 익명클래스)를 주고받는 것

(함수형 인터페이스는 인터페이스이므로 익명 클래스, 람다 둘 다 대입 가능)

> ***고차 함수(Higher-Order Function)***라는 이름은 ****함수를 다루는 추상화 수준***이 더 높다는 데에서 유래했다.
>
> 보통의 (일반적인) 함수는 ****데이터(값)****를 입력으로 받고, 값을 반환한다.
> 이에 반해, 고차 함수는 **
> **함수를 인자로 받거나 함수를 반환****한다.
> 쉽게 이야기하면 일반 함수는 값을 다루지만, 고차 함수는 함수 자체를 다룬다.

"값"을 다루는 것을 넘어, "함수"라는 개념 자체를 값처럼 다룬다는 점에서 추상화의 수준(계층, order)이 한 단계 높아진다고 해서 고차함수 라고 부름.

문제 풀이

1. Filter
    a. ***filter***: 조건(함수)을 인자로 받아, 리스트에서 필요한 요소만 추려내기

▼ 문 - 답

```
package lambda.ex2;

@FunctionalInterface
public interface MyPredicate {
    boolean test(int value);
}
```

```
package lambda.ex2;

import java.util.ArrayList;
import java.util.List;

public class FilterExample {
    public static List<Integer> filter(List<Integer> list, MyPredicate pred
        List<Integer> result = new ArrayList<>();
        for (int val : list) {
            if (predicate.test(val)) {
                result.add(val);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> numbers = List.of(-3, -2, -1, 1, 2, 3, 5);
        System.out.println("원본 리스트 : " + numbers);
        filter(numbers, new MyPredicate() {
            @Override
            public boolean test(int value) {
                return value < 0;
            }
        });
```

```
filter(numbers, new MyPredicate() {
    @Override
    public boolean test(int value) {
        return value % 2 == 0;
    }
});

MyPredicate myPredicate1 = new MyPredicate() {
    @Override
    public boolean test(int value) {
        return value < 0;
    }
};

MyPredicate myPredicate2 = new MyPredicate() {
    @Override
    public boolean test(int value) {
        return value % 2 == 0;
    }
};

System.out.println("음수만: " + filter(numbers, myPredicate1));
System.out.println("짝수만: " + filter(numbers, myPredicate2));

System.out.println("음수만 : " + filter(numbers, new MyPredicate()
    @Override
    public boolean test(int value) {
        return value < 0;
    }
}));

System.out.println("짝수만 : " + filter(numbers, new MyPredicate()
    @Override
    public boolean test(int value) {
        return value % 2 == 0;
    }
}));
```

```
        System.out.println(filter(numbers, (value -> value < 0)));
        System.out.println(filter(numbers, (value -> value % 2 == 0)));
    }
}
```

2. Map
    a. ***map***: 변환 로직(함수)을 인자로 받아, 리스트의 각 요소를 다른 형태로 바꾸기

▼ 문 - 답

```
package lambda.ex2;

public interface MyFunction <T>{
    T apply(T value);
}
```

```
package lambda.ex2;

import java.util.ArrayList;
import java.util.List;

public class MapExample {
    public static void main(String[] args) {
        List<String> list = List.of("hello", "java", "lambda");
        System.out.println("원본 리스트 : " + list);

        System.out.println(map(list, (value) -> value.toUpperCase()));
        System.out.println(map(list, (value) -> "***"+value+"***"));
    }

    public static List<String> map(List<String> list, MyFunction<String>
        List<String> result = new ArrayList<>();
        for (String str : list) {
            String s = function.apply(str);
            result.add(s);
        }
```

```
        return result;
    }
}
```

3. Reducer
    a. ***reduce***: 누적 로직(함수)을 인자로 받아, 리스트의 모든 요소를 하나의 값으로 축약하기

▼ 문 - 답

```
package lambda.ex3;

@FunctionalInterface
public interface MyReducer {
    int reduce(int value1, int value2);
}
```

```
package lambda.ex3;

import java.util.List;

public class ReducerExample {
    public static void main(String[] args) {
        List<Integer> list = List.of(1, 2, 3, 4);
        System.out.println("리스트: " + list.toString());

        System.out.println(reduce(list, 0, (value1, value2) -> value1 + valu
        System.out.println(reduce(list, 1, (value1, value2) -> value1 * value

    }

    public static int reduce(List<Integer> list, int initial, MyReducer myR
        int result = initial;
        for (int x : list) {
            result = myReducer.reduce(result, x);
        }

        return result;
```

```
    }
}
```

4. Greeter
    a. ***함수를 반환***: 어떤 문자열/정수 등을 받아서, 그에 맞는 새로운 "함수"를 만들어 돌려주기

    ▼ 문 - 답

```
package lambda.ex3;

@FunctionalInterface
public interface MyBuildGreeter {
    String apply(String str);
}

package lambda.ex3;

public class BuildGreeterExample {
    public static MyBuildGreeter buildGreeter(String greeting){
        return (str -> greeting + ", " + str);
    }
    public static void main(String[] args) {
        MyBuildGreeter helloGreeter = buildGreeter("Hello");
        MyBuildGreeter hiGreeter = buildGreeter("Hi");

        System.out.println(helloGreeter.apply("Java"));
        System.out.println(hiGreeter.apply("Lambda"));


    }
}
```

5. Compose
    a. ***함수합성***:두함수를이어붙여,한번에변환로직을적용할수있는새함수를만들기

    ▼ 문 - 답

```java
package lambda.ex3;

@FunctionalInterface
public interface MyCompose {
    String transform(String str);
}

package lambda.ex3;

public class ComposeExample {
    public static MyCompose compose(MyCompose myCompose1, MyC
        return (str) -> {
            str = myCompose1.transform(str);
            return myCompose2.transform(str);
        };

//        return new MyCompose() {
//            @Override
//            public String transform(String str) {
//                str = myCompose1.transform(str); // 중간 값
//                return myCompose2.transform(str); // 마지막 값
//            }
//        };
    }
    public static void main(String[] args) {
        MyCompose myCompose1 = (str) -> str.toUpperCase();
        MyCompose myCompose2 = (str) -> "***" + str + "***";

        System.out.println(compose(myCompose1, myCompose2));

        MyCompose myCompose = compose(myCompose1, myCompose
        System.out.println(myCompose.transform("Hello"));

    }
}
```

지금까지 진행한 **5가지 문제**는 자바에서 고차 함수를 구현할 때 자주 등장하는 패턴으로 구성되어 있다.

이 문제들을 통해 다음 내용들을 깊이있게 이해할 수 있다.
자바에서 **
함수형 인터페이스를 이용해 함수를 표현하고, 이를 매개변수/반환값으로 활용하는 방식
**
익명 클래스 또는 **__람다__**를 활용해, 간결하게 고차 함수를 구현하는 방법
filter-map-reduce등, 컬렉션/스트림 라이브러리에서도 흔히 볼 수 있는 고차 함수 패턴(이 부분은 뒤에서 다룬
다.)

처음에는 람다 문법이 익숙하지 않기 때문에, 처음부터 바로 문제를 풀기는 쉽지 않을 것이다.
**
**지금까지 설명한 문제들은 반드시 이해가 될 때 까지 반복해서 풀어봐야 한다! 그리고 반복을 통해 어느정도 람다에 익숙
해지는 시간을 만들어야 한다!**
**

정리

- ***람다란?***
  자바 8에서 도입된 **
  익명 함수*로, 이름 없이 간결하게 함수를 표현한다.
  예: `(x)->x+1`
  익명 클래스보다 보일러플레이트 코드를 줄여 생산성과 가독성을 높이는 **
  문법 설탕* 역할.
- ***함수형 인터페이스***
  람다를 사용할 수 있는 기반으로, **
  단일 추상 메서드(SAM)*만 포함하는 인터페이스.
  예: `@FunctionalInterface` 로 보장하며, 하나의 메서드만 정의.
  여러 메서드가 있으면 람다 할당 불가(모호성 방지).
- ***람다 문법***
  기본 형태: `(매개변수) -> {본문}`
  생략 가능

단일표현식(본문, 반환생략): `x→x+1`
타입 추론: `(int x) -> x` `(x) → x`
매개변수 괄호(단일 매개변수일 때): `x → x`

시그니처(매개변수 수/타입, 반환 타입)이 함수형 인터페이스와 일치해야 함.
**
람다 활용

- ***변수 대입***: `MyFunction f = (a, b) → a + b;` 처럼 람다 인스턴스를 변수에 저장.
  **
  메서드 전달*: `calculate((a, b) → a + b)` 로 함수처럼 전달 가능.
  **
  반환*: `return (a, b) → a + b;` 로 메서드에서 람다를 반환.
- ***고차 함수***
  함수를 인자나 반환값으로 다루는 함수(예: `filter` , `map` , `reduce`).
  자바에서는 함수형 인터페이스와 람다로 구현하며, 코드의 유연성과 추상화 수준을 높임.
  예: `List<Integer> filter(List<Integer> list, MyPredicate p)` 는 조건 함수를 받아 동
  작.
- ***기타***
  람다는 익명 클래스를 간소화한 도구지만, 내부적으로 인스턴스가 생성됨.
  반복 연습으로 문법과 활용법을 익히는 것이 중요!