2. 람다

#1.인강/0.자바/7.자바-고급3편

람다 정의

- 자바 8부터 도입된 람다는 자바에서 함수형 프로그래밍을 지원하기 위한 핵심 기능이다.
  - 함수형 프로그래밍에 대해서는 뒤에서 설명한다.
- **람다는 익명 함수**이다. 따라서 이름 없이 함수를 표현한다.

메서드나 함수는 다음과 같이 표현한다.

```
반환타입  메서드명(매개변수) {
      본문
}
```

람다는 다음과 같이 간결하게 표현한다

```
(매개변수) -> {본문}
```

- 람다는 익명 함수이다. 따라서 이름이 없다.

```
// 일반 함수 - 이름이 있음
public int add(int x) {
```

```
    return x + 1;
}

// 람다 - 이름이 없음
(int x) -> {return x + 1;}
```

- 자바는 독립적인 함수를 지원하지 않으며, 메서드는 반드시 클래스나 인터페이스에 속한다

> **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.

람다는 표현이 간결하다

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

- 익명 클래스를 사용하면 `new` 키워드, 생성할 클래스명, 메서드명, 반환 타입 등을 모두 나열해야 한다.

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

- 람다를 사용하면 이런 부분을 모두 생략하고, 매개변수와 본문만 적으면 된다.

람다는 변수처럼 다룰 수 있다

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

```
procedure.run(); // 변수를 통해 람다를 실행
```

- 람다를 `procedure`라는 변수에 담았다.
- `procedure` 변수를 통해 이곳에 담은 람다를 실행할 수 있다.

람다도 클래스가 만들어지고, 인스턴스가 생성된다

- 람다도 익명 클래스처럼 클래스가 만들어지고, 인스턴스가 생성된다.

다음 코드로 확인해보자.

```
package lambda.lambda1;

import lambda.Procedure;

public class InstanceMain {

    public static void main(String[] args) {
        Procedure procedure1 = new Procedure() {
            @Override
            public void run() {
                System.out.println("hello! lambda");
            }
        };
        System.out.println("class.class = " + procedure1.getClass());
        System.out.println("class.instance = " + procedure1);

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

실행 결과

```
class.class = class lambda.lambda1.InstanceMain$1
class.instance = lambda.lambda1.InstanceMain$1@506e6d5e
lambda.class = class lambda.lambda1.InstanceMain$$Lambda/0x00000008000c2618
```

```
lambda.instance = lambda.lambda1.InstanceMain$$Lambda/
0x00000008000c2618@3796751b
```

- 익명 클래스의 경우 `$`로 구분하고 뒤에 숫자가 붙는다.
- 람다의 경우 `$$`로 구분하고 뒤에 복잡한 문자가 붙는다.
- 실행 환경에 따라 결과는 다를 수 있다.

정리

- 람다를 사용하면 익명 클래스 사용의 보일러플레이트 코드를 크게 줄이고, 간결한 코드로 생산성과 가독성을 높일 수 있다.
- 대부분의 익명 클래스는 람다로 대체할 수 있다.
  - 참고로 람다가 익명 클래스를 완전히 대체할 수 있는 것은 아니다. 람다와 익명 클래스의 차이는 뒤에서 따로 정리하겠다.
- 람다를 사용할 때 `new` 키워드를 사용하지 않지만, 람다도 익명 클래스처럼 인스턴스가 생성된다.
- 지금은 람다를 익명 클래스의 구현을 간단히 표현할 수 있는 **문법 설탕**(Syntactic sugar, 코드를 간결하게 만드는 문법적 편의) 역할 정도로 생각하자. 람다와 익명 클래스의 차이는 뒤에서 설명한다.

함수형 인터페이스

- **함수형 인터페이스**는 정확히 하나의 추상 메서드를 가지는 인터페이스를 말한다.
- 람다는 추상 메서드가 하나인 **함수형 인터페이스**에만 할당할 수 있다.
- 단일 추상 메서드를 줄여서 **SAM**(Single Abstract Method)이라 한다.
- 참고로 람다는 클래스, 추상 클래스에는 할당할 수 없다. 오직 단일 추상 메서드를 가지는 인터페이스에만 할당할 수 있다.

여러 추상 메서드

```
package lambda.lambda1;

public interface NotSamInterface {
    void run();
    void go();
}
```

- 인터페이스의 메서드 앞에는 `abstract`(추상)이 생략되어 있다. (자바 기본!)
- 여기에는 `run()`, `go()` 두 개의 추상 메서드가 선언되어 있다.
- 단일 추상 메서드(SAM)가 아니다. 이 인터페이스에는 람다를 할당할 수 없다.

단일 추상 메서드

```
package lambda.lambda1;

public interface SamInterface {
    void run();
}
```

- 여기에는 `run()` 한 개의 추상 메서드만 선언되어 있다.
- 단일 추상 메서드(SAM)이다. 이 인터페이스에는 람다를 할당할 수 있다.

```
package lambda.lambda1;

public class SamMain {

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

        // 컴파일 오류
        /*
        NotSamInterface notSamInterface = () -> {
            System.out.println("not sam");
        };
        notSamInterface.run(); // ?
        notSamInterface.go(); // ?
        */
    }
}
```

정상 - 실행 결과

```
sam
```

컴파일 오류 - 실행 결과

```
java: incompatible types: lambda.lambda1.NotSamInterface is not a functional
interface
    multiple non-overriding abstract methods found in interface
lambda.lambda1.NotSamInterface
```

- `NotSamInterface`이 함수형 인터페이스가 아니라는 컴파일 오류 메시지가 나온다.
- 오류를 확인했으면 컴파일 오류 부분을 다시 주석 처리하자.

자바는 왜 다음 코드를 허용하지 않을까?

```
NotSamInterface notSamInterface = () -> {
    System.out.println("not sam");
}
notSamInterface.run(); // ?
notSamInterface.go(); // ?
```

- 람다는 하나의 함수이다. 따라서 람다를 인터페이스에 담으려면 하나의 메서드(함수) 선언만 존재해야 한다.
- 인터페이스는 여러 메서드(함수)를 선언할 수 있다. 여기서는 `run()`, `go()` 두 메서드가 존재한다.
- 이 함수를 `NotSamInterface`에 있는 `run()` 또는 `go()` 둘 중에 하나에 할당해야 하는 문제가 발생한다.

자바는 이러한 문제를 해결하기 위해, 단 하나의 추상 메서드(SAM: Single Abstract Method)만을 포함하는 **함수형 인터페이스에만 람다를 할당할 수 있도록 제한**했다.

`SamInterface`은 `run()`이라는 단 하나의 추상 메서드만을 포함한다. 따라서 문제 없이 람다를 할당하고 실행할 수 있다.

@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
}
```

컴파일 오류 메시지

```
java: Unexpected @FunctionalInterface annotation
  lambda.lambda1.SamInterface is not a functional interface
    multiple non-overriding abstract methods found in interface
lambda.lambda1.SamInterface
```

- 함수형 인터페이스가 아니라는 컴파일 오류 메시지
- 오류를 확인했으면 `gogo()` 메서드는 삭제하자.

따라서 람다를 사용할 함수형 인터페이스라면 `@FunctionalInterface`를 필수로 추가하는 것을 권장한다.
앞서 우리가 사용한 `MyFunction`, `Procedure`에 해당 애노테이션을 추가하자.

```
package lambda;

@FunctionalInterface // 추가
public interface MyFunction {
    int apply(int a, int b);
}
```

```
package lambda;

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

람다와 시그니처

람다를 함수형 인터페이스에 할당할 때는 메서드의 형태를 정의하는 요소인 메서드 시그니처가 일치해야 한다.
메서드 시그니처의 주요 구성 요소는 다음과 같다.

1. 메서드 이름
2. 매개변수의 수와 타입(순서 포함)
3. 반환 타입

MyFunction 예시

예를 들어 `MyFunction`의 `apply` 메서드를 살펴보자.

```
@FunctionalInterface
public interface MyFunction {
    int apply(int a, int b);
}
```

이 메서드의 시그니처

- 이름: `apply`
- 매개변수: `int`, `int`
- 반환 타입: `int`

```
MyFunction myFunction = (int a, int b) -> {
    return a + b;
};
```

람다는 익명 함수이므로 시그니처에서 이름은 제외하고, **매개변수, 반환 타입이 함수형 인터페이스에 선언한 메서드와 맞아야 한다.**

이 람다는 매개변수로 `int a`, `int b`, 그리고 반환 값으로 `a + b`인 `int` 타입을 반환하므로 시그니처가 맞다. 따라서 람다를 함수형 인터페이스에 할당할 수 있다.

참고로 람다의 매개변수 이름은 함수형 인터페이스에 있는 메서드 매개변수의 이름과 상관없이 자유롭게 작성해도 된다. 타입과 순서만 맞으면 된다.

```
MyFunction myFunction = (int xxx, int yyy) -> {return xxx + yyy;};
```

Procedure 예시

간단한 다른 예시도 살펴보자.

```
@FunctionalInterface
public interface Procedure {
    void run();
}
```

이 메서드의 시그니처

- 이름: `run`
- 매개변수: 없음
- 반환 타입: 없음

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

이 람다는 매개변수가 없고, 반환 타입이 없으므로 시그니처가 맞다. 따라서 람다를 함수형 인터페이스에 할당할 수 있다.

람다와 생략

람다는 간결한 코드 작성을 위해 다양한 문법 생략을 지원한다.

단일 표현식1

```
package lambda.lambda1;

import lambda.MyFunction;

public class LambdaSimple1 {

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

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

        // 단일 표현식이 아닐 경우 중괄호와 리턴 모두 필수
        MyFunction function3 = (int a, int b) -> {
```

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

실행 결과

```
function1: 3
function2: 3
람다 실행
function3: 3
```

생략 전

```
(int a, int b) -> {return a + b;};
```

- `a + b`와 같이 간단한 단일 표현식은 중괄호(`{}`)와 `return`을 함께 생략할 수 있다.

생략 후

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

- 생략한 코드는 생략 전 코드와 같은 코드이다. `return` 문이 보이지 않지만 결과를 반환한다.

표현식(expression)이란?

- 하나의 값으로 평가되는 코드 조각을 의미한다.
- 표현식은 산술 논리 표현식, 메서드 호출, 객체 생성등이 있다.
  - 예) `x + y`, `price * quantity`, `calculateTotal()`, `age >= 18`
- 표현식이 아닌것은 제어문, 메서드 선언 같은 것이 있다.
  - 예) `if (condition) { }`

람다 - 단일 표현식(single expression)인 경우

- 중괄호 `{}`와 `return` 키워드를 함께 생략할 수 있음
  - 표현식의 결과가 자동으로 반환값이 됨
- 중괄호를 사용하는 경우에는 반드시 `return` 문을 포함해야 한다.

- `return` 문을 명시적으로 포함하는 경우 중괄호를 사용해야 한다.
- 반환 타입이 `void`인 경우 `return` 생략 가능

단일 표현식이 아닌 경우

```
(int a, int b) -> {
    System.out.println("람다 실행");
    return a + b;
};
```

- 단일 표현식이 아닌 경우 중괄호(`{}`)를 생략할 수 없다. 이 경우 반환 값이 있으면 `return` 문도 포함해야 한다.

단일 표현식2

이번에는 매개변수와 반환 값이 없는 경우를 살펴보자.

```
package lambda.lambda1;

import lambda.Procedure;

public class LambdaSimple2 {

    public static void main(String[] args) {
        // 매개변수, 반환 값이 없는 경우
        Procedure procedure1 = () -> {
            System.out.println("hello! lambda");
        };
        procedure1.run();

        // 단일 표현식은 중괄호 생략 가능
        Procedure procedure2 = () -> System.out.println("hello! lambda");
        procedure2.run();
    }
}
```

실행 결과

```
hello! lambda
```

```
hello! lambda
```

매개변수와 반환 값이 없는 경우도 동일하다.

`Procedure.run()`의 경우 반환 타입이 `void`이기 때문에 중괄호를 사용해도 `return`은 생략할 수 있다.

타입 추론

다음과 같은 람다 코드를 작성한다고 생각해보자.

```
MyFunction function1 = (int a, int b) -> a + b;
```

- 여기서 매개변수에 해당하는 `(int a, int b)` 부분을 집중해보자.
- 함수형 인터페이스인 `MyFunction`의 `apply()` 메서드를 보면 이미 `int a, int b`로 매개변수의 타입이 정의되어 있다.
- 따라서 이 정보를 사용하면 람다의 `(int a, int b)`에서 타입 정보를 생략할 수 있다.

```
@FunctionalInterface
public interface MyFunction {
    int apply(int a, int b);
}
```

다음 예제 코드로 확인해보자.

```
package lambda.lambda1;

import lambda.MyFunction;

public class LambdaSimple3 {

    public static void main(String[] args) {
        // 타입 생략 전
        MyFunction function1 = (int a, int b) -> a + b;

        // MyFunction 타입을 통해 타입 추론 가능, 람다는 타입 생략 가능
        MyFunction function2 = (a, b) -> a + b;
```

```
        int result = function2.apply(1, 2);
        System.out.println("result = " + result);
    }
}
```

실행 결과

```
result = 3
```

타입 생략 전 후

```
MyFunction function1 = (int a, int b) -> a + b; // 타입 직접 입력
MyFunction function2 = (a, b) -> a + b; // 타입 추론 사용
```

- 자바 컴파일러는 람다가 사용되는 함수형 인터페이스의 메서드 타입을 기반으로 람다의 매개변수와 반환값의 타입을 추론한다. 따라서 람다는 타입을 생략할 수 있다.
- 반환 타입은 문법적으로 명시할 수 없다. 대신에 컴파일러가 자동으로 추론한다.

매개변수의 괄호 생략

```
package lambda.lambda1;

public class LambdaSimple4 {

    public static void main(String[] args) {
        MyCall call1 = (int value) -> value * 2; // 기본
        MyCall call2 = (value) -> value * 2; // 타입 추론
        MyCall call3 = value -> value * 2; // 매개변수 1개, () 생략 가능

        System.out.println("call3 = " + call3.call(10));
    }

    interface MyCall {
        int call(int value);
    }
```

```
}
```

- 매개변수가 정확히 하나이면서, 타입을 생략하고, 이름만 있는 경우 소괄호 `()` 를 생략할 수 있다.
- 매개변수가 없는 경우에는 `()` 가 필수이다.
- 매개변수가 둘 이상이면 `()` 가 필수이다.

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

람다는 함수형 인터페이스를 통해 변수에 대입하거나, 메서드에 전달하거나 반환할 수 있다.

예제로 확인해보자.

람다를 변수에 대입하기

```
package lambda.lambda2;

import lambda.MyFunction;

// 1. 람다를 변수에 대입하기
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(add).apply(1, 2) = " + cal.apply(1, 2));

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

실행 결과

```
add.apply(1, 2) = 3
sub.apply(1, 2) = -1
cal(add).apply(1, 2) = 3
cal(sub).apply(1, 2) = -1
```

```
MyFunction add = (a, b) -> a + b;
```

- 이 대입식에서 변수 `add`의 타입은 `MyFunction` 함수형 인터페이스이다. 따라서 `MyFunction` 형식에 맞는 람다를 대입할 수 있다. (메서드 시그니처가 일치한다)

자바에서 기본형과 참조형은 다음과 같이 변수에 값을 대입할 수 있다.

기본형의 값 대입

```
int a = 10;
int c;
c = a;
```

참조형의 값 대입

```
Member newMember = new Member();
Member target;
target = newMember;
```

클래스나 인터페이스로 선언한 변수에 값을 대입하는 것은 인스턴스의 참조값을 대입하는 것이다.

람다의 대입

```
MyFunction add = (a, b) -> a + b;
MyFunction cal = add;
```

```
// 람다의 대입 분석
MyFunction add = (a, b) -> a + b; // 1. 람다 인스턴스 생성
MyFunction add = x001; // 2. 참조값 반환, add에 x001 대입

MyFunction cal = add;
MyFunction cal = x001; // 3. cal에 참조값 대입
```

람다도 마찬가지다. 함수형 인터페이스로 선언한 변수에 람다를 대입하는 것은 람다 인스턴스의 참조값을 대입하는 것이다.
이해가 잘 안된다면 익명 클래스의 인스턴스를 생성하고 대입한다고 생각해보자.
참고로 함수형 인터페이스도 인터페이스이다.

람다도 인터페이스(함수형 인터페이스)를 사용하므로, 람다 인스턴스의 참조값을 변수에 전달할 수 있다.
변수에 참조값을 전달할 수 있으므로 다음과 같이 사용할 수 있다.

- 매개변수를 통해 메서드(함수)에 람다를 전달할 수 있다. (정확히는 람다 인스턴스의 참조값을 전달)
- 메서드가 람다를 반환할 수 있다. (정확히는 람다 인스턴스의 참조값을 반환)

람다를 메서드(함수)에 전달하기

앞서 본 것과 같이 람다는 변수에 전달할 수 있다.
같은 원리로 람다를 매개변수를 통해 메서드(함수)에 전달할 수 있다.

```
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");
        calculate(add);
        calculate(sub);

        System.out.println("람다를 직접 전달");
        calculate((a, b) -> a + b);
        calculate((a, b) -> a - b);
    }

    static void calculate(MyFunction function) {
        int a = 1;
        int b = 2;

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

실행 결과

```
변수를 통해 전달
계산 시작
계산 결과: 3
계산 시작
계산 결과: -1
람다를 직접 전달
계산 시작
계산 결과: 3
계산 시작
계산 결과: -1
```

```
void calculate(MyFunction function)
```

- `calculate()` 메서드의 매개변수는 `MyFunction` 함수형 인터페이스이다. 따라서 람다를 전달할 수 있다.

람다를 변수에 담은 후에 매개변수에 전달

```
MyFunction add = (a, b) -> a + b;
```

```
calculate(add);
```

```
// 람다를 변수에 담은 후에 매개변수에 전달 분석
MyFunction add = (a, b) -> a + b; // 1. 람다 인스턴스 생성
MyFunction add = x001; // 2. 참조값 반환
add = x001; // 3. 참조값 대입

calculate(add);
calculate(x001);

// 메서드 호출, 매개변수에 참조값 대입
void calculate(MyFunction function = x001)
```

람다를 직접 전달

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

```
// 람다를 직접 전달 분석
calculate((a, b) -> a + b); // 1. 람다 인스턴스 생성
calculate(x001); // 2. 참조값 반환 및 매개변수에 전달

// 메서드 호출, 매개변수에 참조값 대입
void calculate(MyFunction function = x001)
```

- 람다 인스턴스의 참조를 매개변수에 전달하는 것이기 때문에 이해하는데 어려움은 없을 것이다.
- 일반적인 참조를 매개변수에 전달하는 것과 같다.

람다를 반환하기

```
package lambda.lambda2;

import lambda.MyFunction;
```

```java
// 3. 람다를 반환하기
public class LambdaPassMain3 {

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

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

        MyFunction xxx = getOperation("xxx");
        System.out.println("xxx.apply(1, 2) = " + xxx.apply(1, 2));
    }

    // 람다를 반환하는 메서드
    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;
        }
    }
}
```

실행 결과

```
add.apply(1, 2) = 3
sub.apply(1, 2) = -1
xxx.apply(1, 2) = 0
```

```java
MyFunction getOperation(String operator){}
```

- `getOperation` 메서드는 반환 타입이 `MyFunction` 함수형 인터페이스이다. 따라서 람다를 반환할 수 있다.

분석

```
// 1. 메서드를 호출한다.
MyFunction add = getOperation("add");

// 2. getOperation() 메서드 안에서 다음 코드가 호출된다.
MyFunction getOperation(String operator) {} // 반환 타입이 MyFunction 함수형 인터페이스이다.
return (a, b) -> a + b; // 2-1. 람다 인스턴스를 생성한다.
return x001; // 2-2. 람다 인스턴스의 참조값을 반환한다.

// 3. main 메서드로 람다 인스턴스의 참조값이 반환된다.
MyFunction add = x001; // 3-1. 람다 인스턴스의 참조값을 add에 대입한다.
```

고차 함수

람다의 전달 정리

앞서 배운 내용을 다시 한번 정리해보자.

람다는 함수형 인터페이스를 구현한 익명 클래스 인스턴스와 같은 개념으로 이해하면 된다. 즉, 람다를 변수에 대입한다는 것은 **람다 인스턴스의 참조값을 대입**하는 것이고, 람다를 메서드(함수)의 매개변수나 반환값으로 넘긴다는 것 역시 **람다 인스턴스의 참조값을 전달, 반환**하는 것이다.

- **람다를 변수에 대입**: `MyFunction add = (a, b) -> a + b;` 처럼 함수형 인터페이스 타입의 변수에 람다 인스턴스의 참조를 대입한다.
- **람다를 메서드 매개변수에 전달**: 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);
```

- **람다를 메서드에서 반환**: `return (a, b) -> a + b;` 처럼 함수형 인터페이스 타입을 반환값으로 지정해 람다 인스턴스의 참조를 돌려줄 수 있다.

이런 방식으로 람다를 자유롭게 전달하거나 반환할 수 있기 때문에, 코드의 간결성과 유연성이 높아진다. 만약 익명 클

래스를 작성하고 전달했다면 매우 번잡했을 것이다.

고차 함수(Higher-Order Function)

고차 함수는 함수를 값처럼 다루는 함수를 뜻한다.

일반적으로 다음 두 가지 중 하나를 만족하면 고차 함수라 한다.

- 함수를 인자로 받는 함수(메서드)
- 함수를 반환하는 함수(메서드)

함수를 인자로 받는 경우

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

함수를 반환하는 경우

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

- 즉, 매개변수나 반환값에 함수(또는 람다)를 활용하는 함수가 고차 함수에 해당한다.
- 자바에서 람다(익명 함수)는 함수형 인터페이스를 통해서만 전달할 수 있다.
- **자바에서 함수를 주고받는다는 것**은 "함수형 인터페이스를 구현한 어떤 객체(람다든 익명 클래스든)를 주고받는 것"과 동의어이다. (함수형 인터페이스는 인터페이스이므로 익명 클래스, 람다 둘다 대입할 수 있다. 하지만 실질적으로 함수형 인터페이스에는 람다를 주로 사용한다.)

용어 - 고차 함수

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.

즉, "값"을 다루는 것을 넘어, "함수"라는 개념 자체를 값처럼 다룬다는 점에서 **추상화의 수준(계층, order)이 한 단계**

높아진다고 해서 Higher-Order(더 높은 차원의) 함수라고 부른다.

문제와 풀이1

문제 1. 중복되는 메시지 출력 로직 리팩토링

문제 설명

다음 코드는 화면에 여러 종류의 인삿말 메시지를 출력하지만, 모든 메서드마다 `=== 시작 ===`과 `=== 끝 ===`을 출력하는 로직이 중복되어 있다. 중복되는 코드를 제거하고, **변하는 부분(인삿말 메시지)**만 매개변수로 받도록 리팩토링 해라.

예시 코드

```
package lambda.ex1;

public class M1Before {

    public static void greetMorning() {
        System.out.println("=== 시작 ===");
        System.out.println("Good Morning!");
        System.out.println("=== 끝 ===");
    }

    public static void greetAfternoon() {
        System.out.println("=== 시작 ===");
        System.out.println("Good Afternoon!");
        System.out.println("=== 끝 ===");
    }

    public static void greetEvening() {
        System.out.println("=== 시작 ===");
        System.out.println("Good Evening!");
        System.out.println("=== 끝 ===");
    }

    public static void main(String[] args) {
        greetMorning();
        greetAfternoon();
```

```
        greetEvening();
    }
}
```

```
=== 시작 ===
Good Morning!
=== 끝 ===
=== 시작 ===
Good Afternoon!
=== 끝 ===
=== 시작 ===
Good Evening!
=== 끝 ===
```

정답

```
package lambda.ex1;

public class M1After {

    // 하나의 메서드로 합치고, 매개변수(문자열)만 다르게 받아 처리
    public static void greet(String message) {
        System.out.println("=== 시작 ===");
        System.out.println(message);
        System.out.println("=== 끝 ===");
    }

    public static void main(String[] args) {
        greet("Good Morning!");
        greet("Good Afternoon!");
        greet("Good Evening!");
    }
}
```

문제 2. 값 매개변수화 - 다양한 단위를 매개변수로 받기

문제 설명

다음 코드는, 주어진 숫자(예: 10)를 특정 단위(예: "kg")로 출력하는 간단한 메서드를 작성한 예시이다.

숫자와 단위를 나누고 재사용 가능한 메서드를 사용하도록 코드를 수정해라.

예시 코드

```
public class M2Before {

    public static void print1() {
        System.out.println("무게: 10kg");
    }

    public static void print2() {
        System.out.println("무게: 50kg");
    }

    public static void print3() {
        System.out.println("무게: 200g");
    }

    public static void print4() {
        System.out.println("무게: 40g");
    }

    public static void main(String[] args) {
        print1();
        print2();
        print3();
        print4();
    }
}
```

목표 출력 결과

```
무게: 10kg
무게: 50kg
무게: 200g
무게: 40g
```

정답

```
package lambda.ex1;

public class M2After {

    // 숫자(무게)와 단위 모두 매개변수화
    public static void print(int weight, String unit) {
        System.out.println("무게: " + weight + unit);
    }

    public static void main(String[] args) {
        print(10, "kg");
        print(50, "kg");
        print(200, "g");
        print(40, "g");
    }
}
```

문제 3. 동작 매개변수화 - 익명 클래스로 다른 로직 전달

문제 설명

1부터 N까지 더하는 로직과, 배열을 정렬하는(`Arrays.sort()`) 로직을 각각 실행하고, 이 두 가지 로직 모두 "실행에 걸린 시간을 측정"하고 싶다.

- "실행 시간 측정" 로직은 변하지 않는 부분
- "실행할 로직"은 바뀌는 부분(1부터 N 합 구하기 vs 배열 정렬)

이 문제는 람다를 사용하지 말고 익명 클래스를 사용해서 풀어라

문제

1. 앞서 정의한 `Procedure`(추상 메서드 `run()`) 함수형 인터페이스를 사용해라.
2. `measure(Procedure p)` 메서드 안에서
    - 실행 전 시간 기록
    - `p.run()` 실행
    - 실행 후 시간 기록

    - 걸린 시간 출력
3. `main()`에서 **익명 클래스** 두 가지를 만들어 각각 실행 시간을 측정해라.
    - (1) 1부터 N까지 합을 구하는 로직 (`measure` 메서드 호출)
    - (2) 배열을 정렬하는 로직 (`measure` 메서드 호출)
    - `measure` 메서드는 총 2번 호출된다.

(1) 1부터 N까지 합을 구하는 로직 (`measure` 메서드 호출)

```
int N = 100;
long sum = 0;
for (int i = 1; i <= N; i++) {
    sum += i;
}
```

(2) 배열을 정렬하는 로직 (`measure` 메서드 호출)

```
int[] arr = { 4, 3, 2, 1 };
System.out.println("원본 배열: " + Arrays.toString(arr));
Arrays.sort(arr);
System.out.println("배열 정렬: " + Arrays.toString(arr));
```

예시 출력

```
[1부터 100까지 합] 결과: 5050
실행 시간: 4592542ns

원본 배열: [4, 3, 2, 1]
배열 정렬: [1, 2, 3, 4]
실행 시간: 301083ns
```

정답

예시 함수형 인터페이스

```
package lambda;

@FunctionalInterface
```

```java
public interface Procedure {
    void run();
}
```

```java
package lambda.ex1;

import lambda.Procedure;

import java.util.Arrays;

public class M3MeasureTime {

    // 공통: 실행 시간 측정 메서드
    public static void measure(Procedure p) {
        long startNs = System.nanoTime();
        p.run(); // 바뀌는 로직 실행 (익명 클래스 or 람다로 전달)
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }

    public static void main(String[] args) {

        // 1. 익명 클래스로 1부터 N까지 합 구하기
        measure(new Procedure() {
            @Override
            public void run() {
                int N = 100;
                long sum = 0;
                for (int i = 1; i <= N; i++) {
                    sum += i;
                }
                System.out.println("[1부터 " + N + "까지 합] 결과: " + sum);
            }
        });

        // 2. 익명 클래스로 배열 정렬
        measure(new Procedure() {
            @Override
            public void run() {
                int[] arr = { 4, 3, 2, 1 };
                System.out.println("원본 배열: " + Arrays.toString(arr));
```

```
                Arrays.sort(arr);
                System.out.println("배열 정렬: " + Arrays.toString(arr));
            }
        });
    }
}
```

문제 4. 람다로 변경 - 간결하게 코드 작성하기

문제 설명

이전 문제에서 익명 클래스로 작성한 부분을 **람다**로 변경해라.

- `measure()` 메서드와 `Procedure` 인터페이스는 그대로 둔다.
- `main()`에서 **익명 클래스를 사용하지 말고**, 람다를 이용하여 더욱 간결하게 코드를 작성해라.

정답

```
package lambda.ex1;

import lambda.Procedure;

import java.util.Arrays;

public class M4MeasureTime {

    // 공통: 실행 시간 측정 메서드
    public static void measure(Procedure p) {
        long startNs = System.nanoTime();
        p.run(); // 바뀌는 로직 실행 (익명 클래스 or 람다로 전달)
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns\n");
    }

    public static void main(String[] args) {

        // 1. 람다로 1부터 N까지 합 구하기
        measure(() -> {
            int N = 100;
            long sum = 0;
```

```
            for (int i = 1; i <= N; i++) {
                sum += i;
            }
            System.out.println("[1부터 " + N + "까지 합] 결과: " + sum);
        });

        // 2. 람다로 배열 정렬
        measure(() -> {
            int[] arr = { 4, 3, 2, 1 };
            System.out.println("원본 배열: " + Arrays.toString(arr));
            Arrays.sort(arr);
            System.out.println("[배열 정렬] 결과: " + Arrays.toString(arr));
        });
    }
}
```

문제 5. 고차 함수(High-Order Function) - 함수를 반환하기

문제 설명

"함수를 반환"하는 방식도 연습해보자. 두 정수를 받아서 연산하는 `MyFunction` 인터페이스를 사용해보자.

```
package lambda;

@FunctionalInterface
public interface MyFunction {
    int apply(int a, int b);
}
```

- `static MyFunction getOperation(String operator)` 라는 정적 메서드를 만들어라.
- 매개변수인 `operator` 에 따라 다음과 같은 내용을 전달하고 반환해라.
  - `operator` 가 "add"면, `(a, b)` 를 받아 `a + b` 를 리턴하는 람다를 반환해라.
  - "sub"면, `a - b` 를 리턴하는 람다를 반환해라.
  - 그 외의 경우는 항상 0을 리턴하는 람다를 반환해라.
- `main()` 메서드에서 `getOperation("add")`, `getOperation("sub")`, `getOperation("xxx")` 를 각각 호출해서 반환된 람다를 실행해라.

예시 출력

```
add(1, 2) = 3
sub(1, 2) = -1
xxx(1, 2) = 0 // 그 외의 경우
```

정답

```
package lambda.ex1;

import lambda.MyFunction;

public class M5Return {

    // operator에 따라 다른 람다(=함수)를 반환
    public static void main(String[] args) {
        MyFunction add = getOperation("add");
        System.out.println("add(1, 2) = " + add.apply(1, 2));

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

        MyFunction xxx = getOperation("xxx");
        System.out.println("xxx(1, 2) = " + xxx.apply(1, 2));
    }

    public 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; // 잘못된 연산자일 경우 0 반환
        }
    }
}
```

문제와 풀이2

이번 문제들은 이후에 설명할 스트림은 물론이고, 함수형 프로그래밍의 개념을 이해하기 위해 **반드시 반복해서 풀어보고, 또 이해해야 한다.**

이번 문제들은 고차 함수(Higher-Order Function) 개념을 직접 실습해볼 수 있도록 구성했다. 각 문제에서 요구하는 핵심 사항은 **"함수를 매개변수로 받거나, 함수를 반환"** 하는 구조를 구현하는 것이다. 람다가 아직 익숙하지 않을 것이니 먼저 익명 클래스로 구현해보고 그 다음에 람다로 구현해보자.

참고: 고차 함수(Higher-Order Function)란?

- **함수를 인자로 받거나, 함수를 반환**하는 함수
- 자바에서는 **함수형 인터페이스**에 익명 클래스나 람다를 담아서 주고받음으로써 고차 함수를 구현할 수 있다.

문제 1. filter 함수 구현하기

요구 사항

1. 정수 리스트가 주어졌을 때, 특정 조건에 맞는 요소들만 뽑아내는 `filter` 함수를 직접 만들어보자.
2. `filter(List<Integer> list, MyPredicate predicate)` 형식의 정적 메서드를 하나 작성한다.
    - `MyPredicate`는 함수형 인터페이스이며, `boolean test(int value);` 같은 메서드를 가진다.
3. `main()`에서 예시로 다음과 같은 상황을 실습해보자.
    - 리스트: `[-3, -2, -1, 1, 2, 3, 5]`
    - 조건 1: 음수(negative)만 골라내기
    - 조건 2: 짝수(even)만 골라내기

예시 실행

```
원본 리스트: [-3, -2, -1, 1, 2, 3, 5]
음수만: [-3, -2, -1]
짝수만: [-2, 2]
```

함수형 인터페이스 예시

```
package lambda.ex2;

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

기본 코드 예시

```
package lambda.ex2;

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

public class FilterExample {

    // 고차 함수, 함수를 인자로 받아서 조건에 맞는 요소만 뽑아내는 filter
    public static List<Integer> filter(List<Integer> list, MyPredicate
predicate) {
        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);

        // 1. 음수(negative)만 뽑아내기
        // 코드 작성

        // 2. 짝수(even)만 뽑아내기
        // 코드 작성
    }
}
```

정답 - 익명 클래스

```
package lambda.ex2;
```

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

public class FilterExampleEx1 {

    // 고차 함수, 함수를 인자로 받아서 조건에 맞는 요소만 뽑아내는 filter
    public static List<Integer> filter(List<Integer> list, MyPredicate
predicate) {
        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);

        // 1. 음수(negative)만 뽑아내기
        List<Integer> negatives = filter(numbers, new MyPredicate() {
            @Override
            public boolean test(int value) {
                return value < 0;
            }
        });
        System.out.println("음수만: " + negatives);

        // 2. 짝수(even)만 뽑아내기
        List<Integer> evens = filter(numbers, new MyPredicate() {
            @Override
            public boolean test(int value) {
                return value % 2 == 0;
            }
        });
        System.out.println("짝수만: " + evens);
    }
}
```

정답 - 람다

```
package lambda.ex2;

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

public class FilterExampleEx2 {

    // 고차 함수, 함수를 인자로 받아서 조건에 맞는 요소만 뽑아내는 filter
    public static List<Integer> filter(List<Integer> list, MyPredicate
predicate) {
        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);

        // 1. 음수(negative)만 뽑아내기
        List<Integer> negatives = filter(numbers, value -> value < 0);
        System.out.println("음수만: " + negatives);

        // 2. 짝수(even)만 뽑아내기
        List<Integer> evens = filter(numbers, value -> value % 2 == 0);
        System.out.println("짝수만: " + evens);
    }
}
```

- `filter()` 메서드가 `MyPredicate`라는 "조건 함수"를 받아서, `test()`가 `true`일 때만 결과 리스트에 추가한다.
- 이처럼 **함수를 인자로 받아서 로직을 결정**하는 형태가 전형적인 고차 함수이다.

문제 2. map 함수 구현하기

요구 사항

1. 문자열 리스트를 입력받아, 각 문자열을 어떤 방식으로 변환(**map**, mapping)할지 결정하는 함수(`map`)를 만들어보자
2. `map(List<String> list, StringFunction func)` 형태로 구현한다.
    - `StringFunction`은 함수형 인터페이스이며, `String apply(String s);` 같은 메서드를 가진다.
3. `main()`에서 다음 변환 로직들을 테스트해보자.
    - 변환 1: 모든 문자열을 **대문자**로 변경
    - 변환 2: 문자열 앞 뒤에 `***`를 붙여서 반환(예: `"hello" → "***hello***"`)

예시 실행

```
원본 리스트: [hello, java, lambda]
대문자 변환 결과: [HELLO, JAVA, LAMBDA]
특수문자 데코 결과: [***hello***, ***java***, ***lambda***]
```

함수형 인터페이스

```
package lambda.ex2;

@FunctionalInterface
public interface StringFunction {
    String apply(String s);
}
```

코드 예시

```
package lambda.ex2;

import java.util.List;

public class MapExample {

    // 고차 함수, 함수를 인자로 받아, 리스트의 각 요소를 변환
    public static List<String> map(List<String> list, StringFunction func) {
        // 코드 작성
        return null; // 제거하고 적절한 객체를 반환
    }
```

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

        // 1. 대문자 변환
        // 코드 작성

        // 2. 앞뒤에 *** 붙이기 (람다로 작성)
        // 코드 작성
    }
}
```

정답

```
package lambda.ex2;

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

public class MapExample {

    // 고차 함수, 함수를 인자로 받아, 리스트의 각 요소를 변환
    public static List<String> map(List<String> list, StringFunction func) {
        List<String> result = new ArrayList<>();
        for (String str : list) {
            result.add(func.apply(str));
        }
        return result;
    }

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

        // 1. 대문자 변환
        List<String> upperList = map(words, s -> s.toUpperCase());
        System.out.println("대문자 변환 결과: " + upperList);

        // 2. 앞뒤에 *** 붙이기 (람다로 작성)
        List<String> decoratedList = map(words, s -> "***" + s + "***");
```

```
        System.out.println("특수문자 데코 결과: " + decoratedList);
    }
}
```

문제와 풀이3

문제 3. reduce(또는 fold) 함수 구현하기

요구 사항

1. 정수 리스트를 받아서, 모든 값을 하나로 **누적(reduce)**하는 함수를 만들어보자.
2. `reduce(List<Integer> list, int initial, MyReducer reducer)` 형태로 구현한다.
   - `MyReducer`는 `int reduce(int a, int b);` 같은 메서드를 제공하는 함수형 인터페이스이다.
   - `initial`은 누적 계산의 초깃값(예: 0 또는 1 등)을 지정한다.
3. `main()`에서 다음 연산을 테스트해보자.
   - 연산 1: 리스트 `[1, 2, 3, 4]`를 모두 더하기(`+`)
   - 연산 2: 리스트 `[1, 2, 3, 4]`를 모두 곱하기(`*`)

예시 실행

```
리스트: [1, 2, 3, 4]
합(누적 +): 10
곱(누적 *): 24
```

함수형 인터페이스

```
package lambda.ex2;

@FunctionalInterface
public interface MyReducer {
    int reduce(int a, int b);
}
```

```
```

```
package lambda.ex2;

import java.util.List;

public class ReduceExample {

    // 함수를 인자로 받아, 리스트 요소를 하나로 축약(reduce)하는 고차 함수
    public static int reduce(List<Integer> list, int initial, MyReducer
reducer) {
        // 코드 작성
        return 0; // 적절한 값으로 변경
    }

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

        // 1. 합 구하기 (초깃값 0, 덧셈 로직)
        // 코드 작성

        // 2. 곱 구하기 (초깃값 1, 곱셈 로직)
        // 코드 작성
    }
}
```

- 고차 함수: `MyReducer.reduce` 메서드가 "함수를 인자로 받아서" 내부 로직(합산, 곱셈 등)을 다르게 수행한다.
- 곱은 초깃값을 1로 한 것에 주의하자. 어떤 수 든지 `0`을 곱하면 그 결과가 `0`이 된다.

정답

```
package lambda.ex2;

import java.util.List;

public class ReduceExample {

    // 함수를 인자로 받아, 리스트 요소를 하나로 축약(reduce)하는 고차 함수
    public static int reduce(List<Integer> list, int initial, MyReducer
reducer) {
        int result = initial;
```

```
        for (int val : list) {
            result = reducer.reduce(result, val);
        }
        return result;
    }

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

        // 1. 합 구하기 (초깃값 0, 덧셈 로직)
        int sum = reduce(numbers, 0, (a, b) -> a + b);
        System.out.println("합(누적 +): " + sum);

        // 2. 곱 구하기 (초깃값 1, 곱셈 로직, 람다로 작성)
        int product = reduce(numbers, 1, (a, b) -> a * b);
        System.out.println("곱(누적 *): " + product);
    }
}
```

용어 - reduce, fold

이렇게 여러 값을 계산해서 하나의 최종 값을 반환하는 경우 `reduce(축약하다)`, `fold(접는다)` 같은 단어를 사용한다.

- `reduce`: 1, 2, 3, 4라는 숫자를 하나씩 계산하면서 축약하기 때문에 축약하다는 의미의 `reduce`를 사용한다.
- `fold`: 마치 종이를 여러 번 접어서 하나의 작은 뭉치로 만드는 것처럼, 초깃값과 연산을 통해 리스트의 요소를 하나씩 접어서 최종적으로 하나의 값으로 축약한다는 의미이다.

문제 4. 함수를 반환하는 `buildGreeter` 만들기

요구 사항

1. 문자열을 입력받아, **새로운 함수를 반환**해주는 `buildGreeter(String greeting)`라는 메서드를 작성하자.
    - 예) `buildGreeter("Hello")` ➜ `"Hello"`를 사용하는 새로운 함수 반환
    - 새로운 함수는 입력받은 문자열에 대해 `"Hello"(greeting) + ", " + (입력받은 문자열)` 형태로 결과를 반환
2. 함수를 반환받은 뒤에, 실제로 그 함수를 호출해 결과를 확인해보자.

함수형 인터페이스 - 이전에 작성한 코드를 사용하자.

```
package lambda.ex2;

@FunctionalInterface
public interface StringFunction {
    String apply(String s);
}
```

문제 예시

```
package lambda.ex2;

public class BuildGreeterExample {

    // 고차 함수, greeting 문자열을 받아, "새로운 함수를" 반환
    public static StringFunction buildGreeter(String greeting) {
        // 코드 작성
        return null; // 적절한 람다 반환
    }

    public static void main(String[] args) {
        // 코드 작성
    }
}
```

실행 결과

```
Hello, Java
Hi, Lambda
```

정답

```
package lambda.ex2;

public class BuildGreeterExample {
```

```
    // 고차 함수: greeting 문자열을 받아, "새로운 함수를" 반환
    public static StringFunction buildGreeter(String greeting) {
        // 람다로 함수 반환
        return name -> greeting + ", " + name;
    }

    public static void main(String[] args) {
        StringFunction helloGreeter = buildGreeter("Hello");
        StringFunction hiGreeter = buildGreeter("Hi");

        // 함수가 반환되었으므로, apply()를 호출해 실제로 사용
        System.out.println(helloGreeter.apply("Java"));  // Hello, Java
        System.out.println(hiGreeter.apply("Lambda"));   // Hi, Lambda
    }
}
```

문제 5. 함수 합성하기 (`compose`)

이번에는 람다를 전달하고 또 람다를 반환까지 하는 복잡한 문제를 풀어보자.

요구 사항

1. 문자열을 변환하는 함수 두 개(`MyTransformer` 타입)를 받아서, **f1을 먼저 적용**하고, 그 결과에 **f2를 적용**하는 새로운 함수를 반환하는 `compose` 메서드를 만들어보자. 예) `f2(f1(x))`
2. 예시 상황:
    - `f1` : 대문자로 바꿈
    - `f2` : 문자 앞 뒤에 `"**"` 을 붙임
    - 합성 함수(`compose()`)를 `"hello"` 에 적용하면 → `"**HELLO**"`

함수형 인터페이스

```
package lambda.ex2;

@FunctionalInterface
public interface MyTransformer {
    String transform(String s);
}
```

```
package lambda.ex2;

public class ComposeExample {

    // 고차 함수, f1, f2라는 두 함수를 인자로 받아, "f1을 먼저, f2를 나중"에 적용하는 새 함수 반환
    public static MyTransformer compose(MyTransformer f1, MyTransformer f2) {
        // 코드 작성
        return null; // 적절한 람다 반환
    }

    public static void main(String[] args) {
        // f1: 대문자로 변환
        MyTransformer toUpper = s -> s.toUpperCase();

        // f2: 앞 뒤에 "**" 붙이기
        MyTransformer addDeco = s -> "**" + s + "**";

        // 합성: f1 → f2 순서로 적용하는 함수
        MyTransformer composeFunc = compose(toUpper, addDeco);

        // 실행
        String result = composeFunc.transform("hello");
        System.out.println(result); // "**HELLO**"
    }
}
```

실행 결과

```
**HELLO**
```

이번에 만나볼 고차 함수는 함수를 인자로 받아서, 또 다른 함수를 반환하는 형태이다.

힌트

문제를 풀기 쉽지 않을 것이다. `compose()` 메서드 안에서 `MyTransformer`를 반환해야 한다. 처음에는 익명 클래스를 사용해보자.

정답 - 익명 클래스

```
package lambda.ex2;

public class ComposeExampleEx1 {

    // 고차 함수, f1, f2라는 두 함수를 인자로 받아, "f1을 먼저, f2를 나중"에 적용하는 새 함수
반환
    public static MyTransformer compose(MyTransformer f1, MyTransformer f2) {
        return new MyTransformer() {
            @Override
            public String transform(String s) {
                String intermediate = f1.transform(s);
                return f2.transform(intermediate);
            }
        };
    }

    public static void main(String[] args) {
        // f1: 대문자로 변환
        MyTransformer toUpper = s -> s.toUpperCase();

        // f2: 앞 뒤에 "**" 붙이기
        MyTransformer addDeco = s -> "**" + s + "**";

        // 합성: f1 → f2 순서로 적용하는 함수
        MyTransformer composeFunc = compose(toUpper, addDeco);

        // 테스트
        String result = composeFunc.transform("hello");
        System.out.println(result); // "**HELLO**"
    }
}
```

정답 - 람다

```
package lambda.ex2;

public class ComposeExampleEx2 {
```

```
    // 고차 함수, f1, f2라는 두 함수를 인자로 받아, "f1을 먼저, f2를 나중"에 적용하는 새 함수
반환
    public static MyTransformer compose(MyTransformer f1, MyTransformer f2) {
        return s -> {
            String intermediate = f1.transform(s);
            return f2.transform(intermediate);
        };
    }

    public static void main(String[] args) {
        // f1: 대문자로 변환
        MyTransformer toUpper = s -> s.toUpperCase();

        // f2: 앞 뒤에 "**" 붙이기
        MyTransformer addDeco = s -> "**" + s + "**";

        // 합성: f1 → f2 순서로 적용하는 함수
        MyTransformer composeFunc = compose(toUpper, addDeco);

        // 테스트
        String result = composeFunc.transform("hello");
        System.out.println(result); // "**HELLO**"
    }
}
```

정리

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

1. **filter**: 조건(함수)을 인자로 받아, 리스트에서 필요한 요소만 추려내기
2. **map**: 변환 로직(함수)을 인자로 받아, 리스트의 각 요소를 다른 형태로 바꾸기
3. **reduce**: 누적 로직(함수)을 인자로 받아, 리스트의 모든 요소를 하나의 값으로 축약하기
4. **함수를 반환**: 어떤 문자열/정수 등을 받아서, 그에 맞는 새로운 "함수"를 만들어 돌려주기
5. **함수 합성**: 두 함수를 이어 붙여, 한 번에 변환 로직을 적용할 수 있는 새 함수를 만들기

이 문제들을 통해 다음 내용들을 깊이있게 이해할 수 있다.

- 자바에서 **함수형 인터페이스**를 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 다룬다.)

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 익숙해지는 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
    - 단일 표현식(본문, 반환 생략): `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 동작.
- **기타**
  - 람다는 익명 클래스를 간소화한 도구지만, 내부적으로 인스턴스가 생성됨.
  - 반복 연습으로 문법과 활용법을 익히는 것이 중요!