함수형 프로그래밍

소유자	종수 김
태그	

프로그래밍 패러다임

프로그램을 구성하고, 구현하는 사상이나 접근법

- 명령형 프로그래밍
  - 절차지향 프로그래밍
  - 객체지향 프로그래밍
- 선언형 프로그래밍
  - 함수형 프로그래밍

명령형 프로그래밍

- 프로그램이 **어떻게** 동작해야하는지 세세한 제어 흐름을 통해 기술
- 절차지향과, 객체지향을 포함.
  - 절차지향 : 프로시저, 함수를 기반으로 로직을 절차적으로 구성
  - 객체지향 : 데이터(필드)와 함수(메서드)를 하나로 묶은 객체를 중심으로 설계

선언형 프로그래밍

- **무엇(What)을** 해야하는지에 초점을 맞추어, 목적만 선언하고 구현방식은 추상화
- 함수형 프로그래밍, SQL, HTML 등이 있음.
  - 함수형 프로그래밍 : 순수 함수를 조합하여, 부수 효과와 가변 상태를 최소화 하여 로직을 표현
  - SQL : 어떻게 가져올지는, 전혀 신경쓰지 않고 SELECT 라는 명령을 통하여 결과를 가져옴.

명령형 프로그래밍 (Imperative Programming)

어떻게(How)할 것인지 구체적으로 명령(Instruction)을 내리는 방식

특징

- 프로그램이 어떤 순서와 단계로 동작해야 하는지를 구체적인 제어 흐름(조건문, 반복문 등)으로 기술
- 변수의 값이 바뀌면서 상태(state)가 변해감
- CPU의 동작 방식(메모리 수정, 제어 흐름에 따른 실행)과 유사하여, 전통적인 하드웨어와의 직관적인 일치
- 예시: C, C++, Java 등 대부분의 언어가 명령형 특성을 지님

장단점

- 장점: 컴퓨터의 동작 방식과 매우 유사해 이해하기 직관적, 제어 흐름을 상세히 제어하기 쉽다.
- 단점: 프로그램 규모가 커지면 상태 변경에 따른 복잡도가 증

절차지향 프로그래밍 (Procedural Programming)

- **핵심 개념**: 명령형 프로그래밍의 대표적인 형태로, 프로그램을 절차(Procedure)나 함수(Function) 단위로 나누어 순서대로 실행
- **특징**
  - 명령형 패러다임의 하위 개념으로 볼 수 있음
  - 공통된 로직을 재사용하기 위해 함수나 프로시저를 만들어 사용
  - "데이터와 절차가 분리되어 있다"라는 말로도 자주 설명됨. 즉, 함수(절차)는 별도로 정의해 두고, 여러 데이터에 대해 같은 절차를 적용
  - 예시: C, Pascal 등
- **장단점**
  - 장점: 구조적 프로그래밍 기법(모듈화, 함수화)으로 코드 가독성 상승, 코드 재사용성 향상
  - 단점: 데이터와 로직이 명확히 분리되지 않을 때, 코드 유지 보수가 어렵고 대형 프로젝트에서 복잡성 증가

참고 - 함수와 프로시저

- **프로시저(Procedure)**: 일련의 명령문들을 하나의 단위로 묶은 것으로, 특정 작업이나 행동을 수행하는 데 중점을 둔다. 프로시저는 반드시 값을 반환할 필요가 없으며, 주로 상태 변경이나 특정 동작 수행에 초점을 맞춘다.
  - 자바로 쉽게 비유를 하자면 `void`를 반환하는 메서드를 떠올리면 된다.
- **함수(Function)**: 수학적 함수의 개념에서 유래했으며, 입력값을 받아 처리하고 결과값을 반환하는 것이 주 목적이다. 함수는 보통 값을 계산하고 반환하는 데 중점을 둔다.

객체지향 프로그래밍 (Object-Oriented Programming)

- **핵심 개념**: 프로그램을 객체(Object)라는 추상화된 단위로 구성. 각 객체는 상태(필드, 속성)와 행동(메서드)을 갖고 있으며, 메시지 교환(메서드 호출)을 통해 상호작용
- **특징**
  - 캡슐화(Encapsulation), 추상화(Abstraction), 상속(Inheritance), 다형성(Polymorphism)과 같은 특징이 있음
  - 데이터와 해당 데이터를 처리하는 함수를 하나의 객체로 묶어서 관리해 유지보수성과 확장성을 높인다.
  - 예시: Java, C++, C#
- **장단점**
  - 장점: 객체라는 단위로 묶이므로 코드 재사용성, 확장성, 유지보수성 우수. 대규모 시스템 설계에 적합
  - 단점: 과도한 객체 분리나 복잡한 상속 구조 등으로 인해 오히려 복잡도가 증가할 수 있음

선언형 프로그래밍 (Declarative Programming)

> **무엇을(What) 할 것인지를 기술하고, 어떻게(How)는 위임하는 방식**

특징

- 구체적인 제어 흐름(조건문, 반복문 등)을 직접 작성하기보다, 원하는 결과나 조건을 선언적으로 표현
- 상태 변화보다는 결과에 초점을 맞추어 코드를 작성
- EX) SQL, HTML
  - 쿼리로 원하는 데이터나 조건을 선언
  - 화면 구조 / 콘텐츠만 기술하면 브라우저가 렌더링
- 함수형 프로그래밍 등이 선언형 패러다임에 속하거나, 밀접하게 관련

장단점

- 장점 : 구현의 복잡한 로직을 많이 숨길 수 있어, 높은 수준에서 문제 해결에 집중 가능. 비즈니스 로직을 직관적으로 표현하기 쉬움
- 단점 : 언어나 환경이 제공하는 추상화 수준에 의존적이며, 내부 동작이 보이지 않을 경우 디버깅이 어려울 수 있음. 낮은 수준의 최적화나 세밀한 제어가 필요한 상황에서는 제

약이 생길 수 있음.

함수형 프로그래밍(Funtional Programming)

> 무엇을(What)할 것인지를 수학점 함수(Function)들로 구성하고, 부수 효과(Side Effect) 최소화 및 불변성(Immutable State)을 강조하는 프로그래밍 방식

특징

- 선언형(Declaratvie) 접근에 가까움 : 어떻게가 아니라, 어떤 결과를 원한다고 선언
- 순수 함수(Pure Function)를 중시 : 같은 입력이 주어지면 항상 같은 출력
- 데이터는 불변(Immutable)하게 처리 : 재할당 대신, 새로운 데이터를 만들어 반환
- 함수가 일급 시민(First-Class Citizen)으로 취급 : 고차 함수(Higher-Order Function), 함수를 인자로 넘기거나 반환 가능.
- Ex) Haskell, Clojure, Scala, Java(람다와 함수형 인터페이스를 통한 부분지원)

장단점

- 장점 : 상태 변화가 없거나 최소화되므로 디버깅과 테스트 용이, 병렬 처리 및 동시성 처리가 간단해지는 경향
- 단점 : 명령형 사고방식에 익숙한 프로그래머에게는 초기 접근이 어려울 수 있음, 계산 과정에서의 메모리 사용이 증가할 수 있음.

정리

- **명령형 프로그래밍**: 컴퓨터가 실행할 단계별 명령을 직접 제어하고 기술
  - **절차지향 프로그래밍**: 명령형 패러다임 안에서 프로그램을 함수(프로시저) 단위로 분할하여 개발
  - **객체지향 프로그래밍**: 데이터와 함수를 객체라는 단위로 묶어 추상화하고 상호 작용
- **선언형 프로그래밍**: 무엇(What)을 해야 하는지에 집중하여, 구체적인 구현 방식이나 절차 흐름을 추상화하고 선언적으로 기술
  - **함수형 프로그래밍**: 순수 함수와 불변성에 기반하여, 선언적으로 로직을 기술

이러한 패러다임은 서로 대립되지 않으며, 문제 상황이나 설계 목표에 따라 적절히 선택하고 조합해 사용할 수 있다. 실제로 대부분의 언어들은 여러 패러다임을 부분적으로 섞어서 지원(멀티 패러다임)하므로, 상황에 맞추어 다른 스타일을 병행해 쓰는 능력이 중요하다. 쉽게 이야기해서 한 프로젝트 안에서도 특정 부분은 절차지향으로 특정 부분은 객체지향으로 특정 부분은 함수형 스타일을 사용한다. 따라서 문제를 해결하기 가장 나은 방법을 각각 고민하고 선택해야 한다.

자바 언어의 패러다임

자바는 객체지향 프로그래밍 언어지만, 절차지향 방식을 전혀 사용하지 않는 것은 아님.

> 지향
> 그 방향을 주된 목표나 철학으로 삼고, 그에 맞춰 설계와 구현을 이끌어 가는 것.

그 언어의 기본 구조와 철학이 객체지향 개념(클래스, 객체, 캡슐화, 다형성)을 중심으로 설계되어있다는 것을 강조하는 표현이지, 그 언어가 다른 모든 패러다임을 전면 배제 한다는 의미는 아님.

함수형 프로그래밍이란 ?

프로그램을 함수(= 수학적 함수)를 조합해 만드는 방식에 초점을 두는 프로그래밍 패러다임.

> 어떻게(How)할 것인지(절차와 상태 변화를 명시)보다는, 필요한 결과를 얻기 위해 무엇(What)을 계산할 것인가를 강조.
>
> 함수를 일급 시민(First-class Citizen)으로 취급하고, 불변(Immutable) 상태를 지향하며, 순수 함수(Pure Function)를 중심에 두는 것이 주요 특징.

함수형 프로그래밍의 핵심 개념과 특징

- **순수 함수(Pure Function)**
  - 같은 인자를 주면 항상 같은 결과를 반환하는 함수이다.
  - 외부 상태(변할 수 있는 전역 변수 등)에 의존하거나, 외부 상태를 변경하는 부수 효과(Side Effect)가 없는 함수를 의미한다.
- **부수 효과(Side Effect) 최소화**
  - 함수형 프로그래밍에서는 상태 변화를 최소화하기 위해 변수나 객체를 변경하는 것을 지양한다.
  - 이로 인해 프로그램의 버그가 줄어들고, 테스트나 병렬 처리(Parallelism), 동시성(Concurrency) 지원이 용이해진다.
- **불변성(Immutable State) 지향**
  - 데이터는 생성된 후 가능한 한 변경하지 않고, 변경이 필요한 경우 새로운 값을 생성해 사용한다.
  - 가변 데이터 구조에서 발생할 수 있는 오류를 줄이고, 프로그램의 예측 가능성을 높여준다.
- **일급 시민(First-class Citizen) 함수**
  - 함수가 일반 값(숫자, 문자열, 객체(자료구조) 등)과 동일한 지위를 가진다.
  - 함수를 변수에 대입하거나, 다른 함수의 인자로 전달하거나, 함수에서 함수를 반환하는 고차 함수(Higher-order Function)를 자유롭게 사용할 수 있다.
- **선언형(Declarative) 접근**
  - **어떻게**가 아닌 **무엇**을 계산할지 기술한다.
  - 복잡한 제어 구조나 상태 관리를 함수의 합성과 함수 호출로 대체하여 간결하고 가독성 높은 코드를 작성한다.
- **함수 합성(Composition)**
  - 간단한 함수를 조합해 더 복잡한 함수를 만드는 것을 권장한다.
  - 작은 단위의 함수들을 체이닝(Chaining)하거나 파이프라이닝(Pipelining)해서 재사용성을 높이고, 코드 이해도를 높인다.
- **Lazy Evaluation(지연 평가) (선택적 특징)**
  - 필요한 시점까지 계산을 미루는 평가 전략이다.
  - 불필요한 계산 비용을 줄인다.

함수형 프로그래밍의 장점

- 코드 가독성 및 유지보수성 : 순수 함수, 불변 데이터 구조를 활용하면, 프로그램 흐름이 명료
- 병렬 처리 및 동시성 : 데이터가 불변이므로, 여러 쓰레드가 같은 데이터를 다룰 때 충돌이 적어 병렬 실행이 유리하다.
- 테스트 용이성 : 순수 함수는 외부 환경에 의존하지 않으므로, 테스트가 간단.

함수형 프로그래밍의 단점

- 학습 곡선 : 기존 명령형 / 객체지향 스타일에 익숙한 개발자는 순수 함수, 고차 함수, 함수형 개념이 이해하기 쉽지 않음
- 성능 : 모든 데이터를 불변하게 다루는 경우, 매번 새로운 데이터를 생성해야하므로 메모리 사용량이 늘어날 수 있다.

주요 함수형 프로그래밍 언어 및 활용

대표적인 순수 함수형 프로그래밍 언어로는 Haskell이 있다. Java, JavaScript, Python 등의 전통적인 언어들은 순수한 함수형 프로그래밍 언어는 아니지만, 람다 표현식, 고차 함수 등 함수형 스타일을 점진적으로 지원함으로써, 함수형 프로그래밍의 장점을 활용한다.

정리하자면, 함수형 프로그래밍이란 **순수 함수, 불변성, 부수 효과 최소화**를 핵심으로 하여, **함수를 일급 시민으로 취급**하고, **합성(Composition)을 통해 문제를 해결**하는 프로그래밍 패러다임이다. 이러한 접근 방식은 프로그램을 이해하기 쉽게 만들고, 오류나 동시성 문제를 줄여 유지보수와 확장이 용이한 코드를 작성하도록 돕는다.

자바는 **명령형, 객체지향**이 주된 패러다임이고, 거기에 **람다 등 함수형 문법**이 일부 도입된 "멀티 패러다임(Multi-paradigm) 언어"이다. 따라서 자바는 객체지향 중심이지만, 부분적으로 함수형 특성을 지원하는 언어 정도로 이해하면 된다.

자바와 함수형 프로그래밍 - 1

함수형 프로그래밍은 다음과 같은 특징이 있다.

1. 순수 함수(Pure Function)
2. 부수 효과(Side Effect) 최소화
3. 불변성(Immutable State) 지향
4. 일급 시민(First-class Citizen) 함수
5. 선언형(Declarative) 접근
6. 함수 합성(Composition)
7. Lazy Evaluation(지연 평가) (선택적 특징)

순수 함수

- 같은 인자를 주면 항상 같은 결과를 반환하는 함수.
- 외부 상태(변할 수 있는 전역 변수 등)에 의존하거나, 외부 상태를 변경하는 부수 효과(Side Effect)가 없는 함수를 의미.

```
package functional;

import java.util.function.Function;

public class PureFunctionMain1 {
    public static void main(String[] args) {
        Function<Integer, Integer> func = x -> x * 2;
        System.out.println(func.apply(10));
        System.out.println(func.apply(10));
        System.out.println(func.apply(10));
        System.out.println(func.apply(10));
    }
}
```

- 입력 값이 동일하면 항상 동일한 결과를 반환.
- 외부 상태에 의존하거나 수정하지 않으므로, 10을 두 번 호출해도 결과는 모두 20.

부수 효과(Side Effect) 최소화

- 함수형 프로그래밍에서는 상태 변화를 최소화하기 위해 변수나 객체를 변경하는 것을 지양한다.
- 이로 인해 프로그램의 버그가 줄고, 테스트나 병렬 처리 동시성 지원이 용이해진다.

```
package functional;

import java.util.function.Function;

public class SideEffectMain1 {
    public static int count = 0;

    public static void main(String[] args) {
        System.out.println("before count = " + count);
        Function<Integer, Integer> func = x -> {
            count++;
            return x * 2;
        };
```

```
        func.apply(10);
        System.out.println("after count = " +count);
    }
}
```

- 함수 호출 전 후로 count가 0에서 1로 바뀌는데, 이런 외부 상태 변화가 부수 효과의 대표적인 예시
- 함수형 프로그래밍에서는 이러한 외부 상태 변경을 최소화하는 것을 권장.

```
package functional;

import java.util.function.Function;

public class SideEffectMain2 {
    public static int count = 0;

    public static void main(String[] args) {
        System.out.println("before count = " + count);
        Function<Integer, Integer> func = x -> {
            int result = x * 2;
            // 부수 효과(Side Effect)
            System.out.println("x = " + x + ", result = " + (x * 2));
            return result;
        };

        func.apply(10);
        func.apply(10);
    }
}
```

- 콘솔에 출력을 하는 동작도 부수 효과.
- 순수 함수로 보기는 어려우나, 뭐 이정도까지 ...

```
package functional;

import java.util.function.Function;
```

```
public class SideEffectMain3 {
    public static int count = 0;

    public static void main(String[] args) {
        Function<Integer, Integer> func = x -> x * 2;

        int x = 10;
        Integer result = func.apply(10);
        // 부수 효과는 순수 함수와 분리해서 실행
        // 출력은 별도로 처리해 순수성을 유지
        System.out.println("x = " + x + ", result = " + result);
    }
}
```

- 연산을 담당하는 함수가 외부 상태를 전혀 수정하지 않는 순수 함수.
- 연산(순수 함수)과 외부 동작(부수 효과)을 명확히 분리함으로써 순수 함수를 유지.

▼ 부수 효과 컬렉션 예제

```
package functional;

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

public class SideEffectListMain {
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<>();
        list1.add("apple");
        list1.add("banana");
        System.out.println("before list1 = " + list1);
        changeList1(list1);
        System.out.println("after list1 = " + list1);
        List<String> list2 = new ArrayList<>();
        list2.add("apple");
        list2.add("banana");
        System.out.println("before list2 = " + list2);
        List<String> result = changeList2(list2);
        System.out.println("after list2 = " + list2);
```

```
        System.out.println("result = " + result);
    }
    private static void changeList1(List<String> list) {
        for (int i = 0; i < list.size(); i++) {
            list.set(i, list.get(i) + "_complete");
        }
    }
    private static List<String> changeList2(List<String> list) {
        List<String> newList = new ArrayList<>();
        for (String s : list) {
            newList.add(s + "_complete");
        }
        return newList;
    }
}
```

- changeList1()
  - 리스트 원본을 직접 변경함으로써, 사이드 이펙트를 일으킴
- changeList2()
  - 새로운 리스트를 생성해서 반환함으로써, 원본 리스트를 변경하지 않음.
- 함수형 프로그래밍에서는 changeList2()와 같은 방식이 권장

자바와 함수형 프로그래밍 - 2

불변성 지향

- 데이터는 생성된 후 가능한 한 변경하지 않고, 변경이 필요한 경우 새로운 값을 생성해서 사용
- 가변 데이터 구조에서 발생할 수 있는 오류를 줄이고, 프로그램의 예측 가능성을 높여줌
- 불변성은 데이터를 변경하지 않기 때문에 부수 효과와도 관련이 없다.

```
package functional;

public class MutablePerson {
    private String name;
    private int age;
```

```
    public MutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "MutablePerson{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- setter 메서드로 인해, 객체 생성 후에도 상태(필드 값)을 언제든 변경 가능

```
package functional;

public class ImmutablePerson {
    private final String name;
    private final int age;
```

```
    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "ImmutablePerson{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // 변경이 필요한 경우 기존 객체를 수정하지 않고, 새 객체를 반환
    public ImmutablePerson withName(String newName) {
        return new ImmutablePerson(newName, this.age);
    }
}
```

- 필드가 final이며, 생성 후에는 상태를 변경할 수 없도록 설계
- 이름을 변경해도 새 객체를 생성하므로, 원본 객체는 변하지 않고 부수 효과가 최소화

```
package functional;

public class ImmutableMain1 {
    public static void main(String[] args) {
        MutablePerson m1 = new MutablePerson("Kim", 10);
        MutablePerson m2 = m1;
        m2.setAge(11);
```

```
        System.out.println("m1 = " + m1);
        System.out.println("m2 = " + m2);

        ImmutablePerson i1 = new ImmutablePerson("Lee", 20);
        ImmutablePerson i2 = i1.withName("Park"); // i2는 새로운 객체
        System.out.println("i1 = " + i1); // 20
        System.out.println("i2 = " + i2); // 21
    }
}
```

- m1과 m2가 동일 객체를 참조하므로, 하나를 변경하면 원본도 변화
- 하지만, ImmutablePerson은 새로운 객체를 생성하기 때문에 i1은 원본 유지, i2만 변경된 값

```
package functional;

import java.util.List;

public class ImmutableMain2 {
    public static void main(String[] args) {
        MutablePerson m1 = new MutablePerson("Kim", 10);
        MutablePerson m2 = new MutablePerson("Lee", 20);
        List<MutablePerson> originList = List.of(m1, m2);
        System.out.println("originList = " + originList);
        List<MutablePerson> resultList = originList.stream()
                .map(p -> {
                    p.setAge(p.getAge() + 1);
                    return p;
                })
                .toList();

        System.out.println("=== 실행 후 ===");
        System.out.println("originList = " + originList);
        System.out.println("resultList = " + resultList);
    }
}
```

```
originList = [MutablePerson{name='Kim', age=10}, MutablePerson{name='Lee
=== 실행 후 ===
originList = [MutablePerson{name='Kim', age=11}, MutablePerson{name='Lee
resultList = [MutablePerson{name='Kim', age=11}, MutablePerson{name='Lee
```

- stream에서 setAge를 사용하면서 원본 리스트(originList)의 데이터까지 변경.
- 가변 상태는 예상치 못한 곳에서 영향을 미치므로, 버그나 유지보수의 어려움

```
package functional;

import java.util.List;

public class ImmutableMain3 {
    public static void main(String[] args) {
        ImmutablePerson i1 = new ImmutablePerson("Kim", 10);
        ImmutablePerson i2 = new ImmutablePerson("Lee", 20);
        List<ImmutablePerson> originList = List.of(i1, i2);
        System.out.println("originList = " + originList);
        List<ImmutablePerson> resultList = originList.stream()
                .map(p → new ImmutablePerson(p.getName(), p.getAge() + 1))
                .toList();
        System.out.println("=== 실행 후 ===");
        System.out.println("originList = " + originList);
        System.out.println("resultList = " + resultList);
    }
}
```

- 불변 객체를 사용하므로, 스트림에서 나이를 1씩 증가해도 원본 리스트는 전혀 변경되지 않음.
- 결과적으로 새로운 객체 리스트가 생성되어 반환 되므로 부수 효과 없는 안전한 처리가 가능.

```
package functional;

import java.util.List;

public class ImmutableMain3 {
```

```
    public static void main(String[] args) {
        ImmutablePerson i1 = new ImmutablePerson("Kim", 10);
        ImmutablePerson i2 = new ImmutablePerson("Lee", 20);
        List<ImmutablePerson> originList = List.of(i1, i2);
        System.out.println("originList = " + originList);
        List<ImmutablePerson> resultList = originList.stream()
                .map(p -> p.withName("newName")) // 새로운 객체가 생성되어 반환됨.
                .toList();
        System.out.println("=== 실행 후 ===");
        System.out.println("originList = " + originList);
        System.out.println("resultList = " + resultList);
    }
}

originList = [ImmutablePerson{name='Kim', age=10}, ImmutablePerson{name=
=== 실행 후 ===
originList = [ImmutablePerson{name='Kim', age=10}, ImmutablePerson{name=
resultList = [ImmutablePerson{name='newName', age=10}, ImmutablePerson{
```

- 불변 객체를 사용하므로, 스트림에서 이름을 변경해도 원본 리스트는 전혀 변경되지 않음.
- 결과적으로, 새로운 객체 리스트가 생성되어 반환.

자바와 함수형 프로그래밍 - 3

일급 시민 함수

- 함수가 일반 값(숫자, 문자열, 객체(자료구조)등)과 동일한 지위를 가짐.
- 함수를 변수에 대입하거나, 다른 함수의 인자로 전달하거나, 함수에서 함수를 반환하는 고차 함수를 자유롭게 사용할 수 있음.

```
package functional;

import java.util.function.Function;

public class FirstClassCitizenMain {
    public static void main(String[] args) {
        //함수를 변수에 담음.
```

```
        Function<Integer, Integer> func = x -> x * 2;

        //함수를 인자로 전달
        applyFunction(10, func);

        //함수를 반환
        getFunc().apply(10);

    }
    // 고차 함수 : 함수를 인자로 받음
    private static Integer applyFunction(int input, Function<Integer, Integer> fur
        return func.apply(input);
    }
    // 고차 함수 : 함수를 반환
    private static Function<Integer, Integer> getFunc() {
        return x -> x * 2;
    }
}
```

- 함수를 변수처럼 취급.
- 함수를 인자로 전달하거나, 반환함으로써 함수가 일급 시민인 모습을 확인.
- 이는 고차 함수를 구현하는 기반

선언형 접근

- 어떻게가 아닌, 무엇을 계산할지를 기술
- 복잡한 제어 구조나, 상태 관리를 함수의 합성과 함수 호출로 대체하여 간결하고 가독성 높은 코드를 작성

```
package functional;

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

// 짝수면 값을 곱해라
public class DeclarativeMain {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
        // 명령형 : for문과 조건 검사로 처리
        ArrayList<Object> result1 = new ArrayList<>();
        for (Integer number : numbers) {
            if (number % 2 == 0) {
                result1.add(number);
            }
        }
        System.out.println("Imperative Result: " + result1);

        // 선언형 : 스트림 API로 처리
        List<Integer> result2 = numbers.stream()
                .filter(number -> number % 2 == 0)
                .map(number -> number * number)
                .toList();
        System.out.println("Declarative Result: " + result2);
    }
}
```

- 명령형 방식은 for 문과 조건문으로 어떻게 처리 할지를 구체적으로 작성.
- 선언형 방식은 스트림의 filter, map 같은 함수를 조합하여 무엇을 할지에 집중.

함수 합성

- 간단한 함수를 조합해 더 복잡한 함수를 만드는 것을 권장
- 작은 단위의 함수들을 체이닝하거나, 파이프라이닝해서 재사용성을 높이고, 코드 이해도를 높임

```
package functional;

import java.util.function.Function;

public class CompositionMain1 {
    public static void main(String[] args) {
        // 1. x 를 입력받아 x * x
        Function<Integer, Integer> square = x -> x * x;
        // 2. x 를 입력받아 x  + 1
        Function<Integer, Integer> add = x -> x + 1;
```

```
        // 함수 합성
        // 1. compose()를 사용한 새로운 함수 생성
        Function<Integer, Integer> newFunc1 = square.compose(add);
        // square(add(2))
        System.out.println("newFunc1 = " + newFunc1.apply(2));

        Function<Integer, Integer> newFunc2 = square.andThen(add);
        System.out.println("newFunc2 = " + newFunc2.apply(2));
    }
}
```

- 작은 함수를 여러개 조합해서 새로운 함수를 만들어내는 것.
- compose : 뒤쪽 함수 → 앞쪽 함수 순으로 적용
- andThen : 앞쪽 함수 → 뒤쪽 함수 순으로 적용

```
package functional;

import java.util.function.Function;

public class CompositionMain2 {
    public static void main(String[] args) {
        // 1. String → Integer
        Function<String, Integer> parseInt = Integer::parseInt;

        // 2. Integer → String
        Function<Integer, Integer> square = x → x * x;

        // 3. Integer → String
        Function<Integer, String> toString = x → "결과 : " + x;

        // compose() 혹은 andThen()으로 합성
        // parseInt → square → toString
        Function<String, String> finalFunc = parseInt
                .andThen(square)
                .andThen(toString);
```

```
        // 문자열 5 = 파싱 -> 제곱 -> 문자열
        String result1 = finalFunc.apply("5");
        System.out.println(result1);

        // 문자열 10
        String result2 = finalFunc.apply("10");
        System.out.println(result2);

        Function<String, Integer> stringToSquareFunc = parseInt.andThen(squar
        Integer result3 = stringToSquareFunc.apply("5");
        System.out.println("result3 = " + result3);

    }
}
```

- 정수로 변환, 제곱, 문자열로 변환하는 과정을 서로 다른 함수로 분리.
- 함수들을 잘게 쪼개어 분리하면 result3 처럼, 필요한 부분까지만 사용할 수 있어 재사용성이 높아짐.

Lazy Evaluation(지연 평가) - 선택적 특징

- 필요한 시점까지 계산을 미루는 평가 전략.
- 불필요한 계산 비용을 줄임

- 자바 스트림도 중간 연산은 최종 연산이 호출되기 전 까지 실행되지 않음.
  이를 통해 필요한 시점까지 계산을 미루는 전략을 취할 수 있음.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
Stream<Integer> stream = numbers.stream()
                        .filter(n -> {
                            System.out.println("filter: " + n);
                            return n % 2 == 0;
});
// 아직 출력된 것이 없음 (중간 연산만 설정된 상태)
// 최종 연산을 호출할 때 실제 동작 시작
List<Integer> evens = stream.toList();
// 여기서야 filter가 실제로 동작하며 콘솔에 filter 로그가 찍힘
```

- 최종 연산이 실행될 때 한 번에 연산이 수행.
- 필요 없는 연산을 미리 실행하지 않으므로, 계산 효율을 높일 수 있음.

정리

1. **프로그래밍 패러다임**
    - 명령형(How)과 선언형(What)으로 크게 구분된다.
    - 명령형 프로그래밍: 절차지향, 객체지향 등을 포함하며, 상태 변화와 제어 흐름을 상세히 기술한다.
    - 선언형 프로그래밍: 필요한 결과(무엇)를 선언적으로 표현하고, 내부 구현(어떻게)은 추상화한다. 함수형 프로그래밍 등도 선언형 패러다임에 속한다.

2. **함수형 프로그래밍(Functional Programming)**
    - 순수 함수(Pure Function), 불변성(Immutable State), 부수 효과(Side Effect) 최소화 등을 핵심으로 하며, 선언형 스타일에 가깝다.
    - 함수가 일급 시민(First-class Citizen)으로 취급되어, 함수를 인자로 전달하거나 반환할 수 있는 고차 함수를 자유롭게 활용할 수 있다.
    - 이로 인해 병렬 처리나 동시성 프로그래밍에서 유리하며, 코드 가독성과 유지보수성이 높아진다.

3. **자바와 함수형 프로그래밍**
    - 자바는 객체지향이 주된 패러다임이지만, 동시에 절차지향적인 특징도 부분적으로 활용 가능하다.
    - 자바 8 이후로 람다, 함수형 인터페이스(`Function`, `Predicate` 등), 스트림 API를 지원하면서 함수형 스타일의 코드 작성이 한층 쉬워졌다.
    - 완벽한 함수형 언어는 아니지만, 불변 객체 설계나 순수 함수를 지향하면 자바에서도 함수형 프로그래밍의 이점을 누릴 수 있다.

정리

- 자바는 기본적으로 객체지향 언어이지만, 필요에 따라 절차지향 접근도 가능하고 함수형 요소도 도입되어 있어 **멀티 패러다임** 언어로 발전해왔다.
- 한 가지 패러다임만 고집하기보다는, 프로젝트의 목적과 상황에 따라 **절차지향**, **객체지향**, **함수형** 스타일을 조합해 사용하는 것이 실제 개발에서 더 효과적이다.
- 특히 함수형 프로그래밍에서 강조하는 순수 함수, 불변성, 부수 효과 최소화 개념은 **코드를 단순화**하고 **동시성 문제**를 줄이는 데 큰 도움이 되므로, 자바를 사용하는 개발자라면 적극적으로 활용할 가치가 있다.