프로그래밍 패러다임

- 프로그래밍의 관점을 갖게하고 결정하는 역할
- 절차적 → 객체지향 → 함수형 의 순서로 단점들을 극복하기 위해 나왔다고 함

목차

명령형 프로그래밍, 선언형 프로그래밍

명령형 프로그래밍

- **어떻게(How)** 목적을 달성할 지에 초점이 맞춰져 있고, 프로그래밍의 상태와 그것을 변경시키는 **구문**의 관점에서 연산을 설명하는 프로그래밍 패러다임
- 프로그래밍에 사용된 알고리즘에 대해서는 명시되어 있으나, 목표에 대해서는 명시되어 있지 않음
- 일반적으로 컴퓨터가 수행할 명령들을 **순서대로** 써내려감
- 예시로는 C, C++, Java, Pascal, Ruby 등이 있음
- 대부분의 절차형, 객체지향 언어들은 명령형 프로그래밍 언어

선언형 프로그래밍

- **무엇(What)**을 할지에 초점이 맞춰져 있는 프로그래밍 패러다임
- 알고리즘은 명시하지 않고, 목표에 대해서만 명시함
- 함수형, 논리형, 제한형 프로그래밍 언어로 쓰인 경우 선언형 프로그래밍이라고 함
- 일반적으로 명령형 언어와 대비되는 프로그래밍 언어들을 선언형으로 통칭함
- 단, 이 중 논리형, 제한형 프로그래밍 언어의 경우 명백히 알고리즘을 설명할 수 있고, 구현도 가능하기 때문에, 엄밀한 의미의 선언형 프로그래밍은 아님

절차적 프로그래밍

절차적 프로그래밍

- 실행 절차에 중점을 둔 프로그래밍 방식으로, 순차적인 처리방식으로 프로그램 전체가 유기적인 연결이 있음
- 순차 처리 방식은 컴퓨터의 동작 방식과 같아서 처리 시간이 빠르다고 함

특징

1. 하나의 큰 기능을 처리하기 위해 작은 단위의 기능들로 나누어 처리하는 **Top-Down** 방식
2. 비교적 작은 규모의 작업을 수행하는 함수 생성
3. 인수, 반환 값으로 명령을 전달하고 수행함
4. 데이터와 함수를 별개 취급함
5. 특정 기능을 수행하려면 해당 메소드를 직접 호출해야함

장단점

- 장점
  - 개인 프로젝트에 적합함
  - 객체 지향에 비해 빠름
- 단점
  - 대형 프로젝트에 부적합함
  - 실제 데이터에 대한 변수나 상수의 표현에는 어려움이 있어 비효율적으로 코딩을 하게 됨 → 유지보수, 분석과 디버깅이 어려움
  - 코드 순서가 중요하여, 순서 변경 시 다른 결과를 초래할 수 있음

객체지향 프로그래밍

객체지향 프로그래밍

- **객체** 라는 기본단위로 나누고, 객체들끼리 서로 상호작용을 하면서 문제를 해결해 나가는 것
- 클래스를 이용해 연관 있는 `처리부분(함수, 메소드)`과 `데이터 부분(변수, 필드)`을 하나의 `객체(인스턴스)`로 묶어서 생성하고 사용함

- 작은 문제를 해결하는 객체를 생성해 큰 문제를 해결하는 **Bottom-Up** 방식
- 네가지 특징(추상화, 캡슐화, 다형성, 상속)을 지켜야함
- 객체 간의 독립성이 생기고 중복코드의 양이 줄어듦
- 독립성으로 인해 유지보수가 쉬움

특징

1. **추상화(Abstraction)**
    - 현실세계 대상을 관찰하여 핵심적인 특징(속성과 행위)을 뽑아내는 과정
    - 이때 관련있는 것들을 묶게되면 캡슐화가 되고 묶은 개념은 추상화가 되고 주체는 클래스가 됨
    - `객체`는 클래스로 부터 실체화 된것으로 눈에 보이며 실재함
    - `클래스`는 추상적이기에 눈에 보이지 않고 개념적으로만 존재함
2. **캡슐화(Encapsulation)**
    - 속성과 행위를 관련 있는 것끼리 묶는 것
    - 접근이 필요한 부분을 제외하고 구체적인 로직은 내부로 숨김(⇒ 은닉성)
    - 높은 응집도와 낮은 결합도를 유지해야함
    - 구체적인 로직을 내부로 은닉하는 목적
        - 클래스 내부 구현의 '응집도' 증가
        - 외부 다른 클래스와의 '결합도' 감소(일반적으로 접근 제한자를 이용)

> **응집도(Cohesion)**
> - 모듈(클래스) 내에서 요소들이 얼마나 밀접하게 관련이 있는지를 나타냄.
> - 정보 은닉을 통해 높일 수 있음
> - 내부에서만 사용하는 정보는 외부에서 접근하지 못하도록 제한하는 것
>
> **결합도(Coupling)**
> 어떠한 기능을 수행하는데에 다른 모듈(클래스)에 얼마나 의존적인지를 나타냄

3. **상속(Inheritance)**
    - 대상이 되는 클래스의 모든 특징을 물려받는 것
    - 이때 자식 클래스로 부터 부모 클래스가 대체되어도 의미가 성립되어야함
    - 계층형 구조
        - 부모 클래스와 자식클래스는 **'상속관계'**
        - 관계가 아래로 내려갈수록 **'구체화된다'**라고 말함 (=> 고유 특징이 증가)
        - 관계가 위로 올라올수록 **'일반화된다'**고 말함 (=> 더 많은 객체에 영향을 준다)
    - 장단점
        - 장점
            - 부모 클래스로 부터 물려 받은 특징으로 인해 같은 내용을 다시 구현할 필요가 없어 **'코드의 재사용성'이 향상된다.**
        - 단점
            - 재사용성만 고려하여 상속한다면 혼란이 생김
            - 상속은 '다형성'을 구현하기 위해 사용할 것을 권장함
                - 좋은 상속관계는 **is-a 관계 (~은 ~이다)** : 바나나는 과일이다 / 자동차는 바퀴가 있는 탈 것이다
                - 혼란을 줄 수 있는 상속관계는 **has-a 관계** : 사람-팔 / 자동차-바퀴 / 새-날개
4. **다형성(Polymorphism)**

- 하나의 속성이나 행위가 상황에 따라 다른 의미로도 해석될 수 있는 특성
- 오버로딩, 오버라이딩을 통해 다형성을 구현 할 수 있음
- **오버로딩(Overloading)**
    - 매개 변수에 따라 여러 종류의 타입을 받아 같은 기능으로 하도록 하기 위한 작업
    - 자바스크립트는 자유로운 언어이므로 기본 제공하지는 않는다
    - 함수는 변수로 취급하며 모든 변수는 전역 객체로 취급하기 때문에 같은 이름이라면 하나만 취급
    - 따라서 타입체크해서 기능을 구현하도록 함수를 만들 수는 있다
- **오버라이딩(Overriding)**
    - 상위 클래스의 메서드를 하위 클래스에서 재 정의하는 것(상속)

- **제네릭(Generics)**
  - T(Type parameter) : 어떤 타입이 와도 무관함

객체 지향 설계 원칙 SOLID

1. **SRP(단일 책임의 원칙 : Single Responsibility Principle)**
   - 작성된 클래스는 하나의 기능만 가지며 클래스가 제공하는 모든 서비스는 그 하나의 책임을 수행하는데 집중되어 있어야 한다는 원칙
2. **OCP(개방폐쇄의 원칙 : Open Close Principle)**
   - 소프트웨어의 구성요소는 확장에는 열려있고, 변경에는 닫혀있어야 한다는 원칙. 변경은 최소화 하고 확장을 최대화함
3. **LSP(리스코브 치환의 원칙 : The Liskov Substitution Principle)**
   - 서브타입은 언제나 기반 타입으로 교체할 수 있어야 한다는 원칙, 즉 항상 하위 클래스는 상위 클래스를 대신할 수 있음
   - 상위클래스가 할 수 있는 일들에 대해선 하위클래스는 당연히 할 수 있다는 원칙
4. **ISP(인터페이스 분리의 원칙 : Interface Segregation Principle)**
   - 인터페이스 분리 원칙은 클라이언트가 자신이 이용하지 않는 메서드에 의존하지 않아야 한다는 원칙
   - 인터페이스 분리 원칙은 큰 덩어리의 인터페이스들을 구체적이고 작은 단위들로 분리시킴으로써 클라이언트들이 꼭 필요한 메서드들만 이용할 수 있게 함
   - SRP 는 클래스의 단일 책임이라면 ISP는 인터페이스의 단일 책임을 강조하는 것
5. **DIP(의존성 역전의 원칙 : Dependency Inversion Principle)**
   - 자주 변화하는 구체적인 클래스 보다는 변화하기 어려운 인터페이스나 상위(추상) 클래스와 관계를 맺으라는 원칙

절차적 프로그래밍과 객체 지향 프로그래밍의 차이

- 명령형 프로그램의 하위 개념이기 때문에 공유하는 것이 많음
- 그렇다고해서 차이점이 없지는 않으며, 이는 아래와 같음
  - **절차적 프로그래밍**
    - 프로그램의 **순서와 흐름을 먼저** 세우고 필요한 **자료구조와 함수**를 설계
    - (=데이터 중심으로 절차적 실행에 초점)
  - **객체지향 프로그래밍**

- 반대로 **자료구조와 함수를 먼저** 설계하고 그 후 **실행순서와 흐름**을 설계
- (=객체간의 관계에 초점)

함수형 프로그래밍

함수형 프로그래밍

- 순수함수를 보조함수와 조합하고 소프트웨어를 만드는 방식
- 로직내에 복잡성을 해결하고 변수사용을 억제하여 상태변경을 피하려는 프로그래밍 패러다임
- 자료처리를 수학적 함수의 계산으로 취급하고 상태와 가변 데이터를 멀리하는 프로그래밍 패러다임의 하나
- 명령형 프로그래밍에서 상태를 바꾸는 것을 강조하는 것과는 달리 함수형프로그래밍은 함수의 응용을 강조함
- 따라서 프로그래밍이 식이나 선언으로 수행되는 선언형 프로그래밍 패러다임을 따르고 있음

함수형 프로그래밍 주요 개념

1. **순수 함수(pure function)**
   - 무조건 같은 입력이 주어지면 같은 출력을 반환하며 부작용이 없는 함수
2. **합성 함수(function composition)**
   - 새로운 함수를 만들기 위해 둘 이상의 함수를 조합하는 과정으로, $f(g(x))$로 이해하면 됨
   - 합성 함수는 함수형 프로그래밍을 이용하여 소프트웨어를 구성하는 중요한 방법
3. **공유 상태(shared state)**
   - 공유 범위(shared scope) 내에 있는 변수, 객체 또는 메모리 공간이거나 범위 간에 전달되는 객체의 속성
4. **불변성(Immutability)**
   - 변경할 수 없는 객체란 생성한 후에 수정할 수 없는 객체를 말함
   - 불변성을 빼면 프로그램의 데이터 흐름이 손실되기 때문에 함수형 프로그래밍의 핵심 개념

5. **부작용(side effects)** : 반환값 이외에 호출된 함수 외부에 영향을 끼치지는 것 → 부작용이 없는 순수한 함수는 스레드 측면에서 안전하고 병렬적인 계산이 가능함
    - 외부 변수 또는 객체 속성 수정
    - 콘솔에서 로깅
    - 화면에 쓰기 작업
    - 파일에 쓰기 작업
    - 네트워크에 쓰기 작업
    - 외부 프로세스를 트리거
    - 부작용을 동반한 다른 함수 호출
6. **고차함수(high order function)** : 함수를 인수로 취급하거나, 함수를 반환하거나 또는 둘 다인 함수
    - 콜백 함수, 프로미스, 모나드 등을 사용하여 액션, 효과 또는 비동기 흐름을 추상화하거나 분리
    - 다양한 데이터 타입에 대해 동작할 수 있는 유틸리티 만듬
    - 합성 함수나 재사용의 목적으로 커링 함수를 만들거나 인수를 함수에 부분적으로 적용
    - 함수 목록을 가져오고, 입력 함수의 합성을 반환

함수형 프로그래밍을 배워야하는 이유

- "일반적인 프로그래밍은 그냥 생각하면 되는 것이고, 함수형 프로그래밍은 기존과 다르게 생각하는 방법을 알려줄 것이다. 그러므로 당신은 아마도 예전 방식으로 절대 돌아가지 않을 것이다."
- 함수형 프로그래밍은 프로그래밍 언어나 방식을 배우는 것이 아니라 함수로 프로그래밍하는 사고를 배우는 것이다. 즉 기존의 사고방식을 전환하여 사고를 유연하게 문제 해결에 접근 하는 것이다.

객체지향 프로그래밍과 함수형 프로그래밍 차이

- 함수형 프로그래밍과 객체지향 프로그래밍은 약간의 차이가 있음
    - **객체지향 프로그래밍**: 관리하는 모듈함수에 의한 재사용성에 관심을 가지고 있으며 **클래스(또는객체,Object)가 일급객체**
        - 클래스와 객체들의 관계를 중심으로 코드 작성

        - 상태, 멤버변수, 메서드 간의 긴밀한 관계
        - 멤버변수의 상태에 따라 결과가 달라짐
    - **함수형 프로그래밍**: 순수함수와 일급객체에 관심을 가지고 있으며 **함수(Function) 자체가 일급객체**
        - 값의 연산 및 결과 도출 중심으로 코드 작성
        - 함수 내부로 넘겨받은 인자 값을 별도로 저장하지 않으며 간결한 과정으로 처리하고 매핑하는데 주 목적
