OOP

2023. 9 .7.

프로그래밍 패러다임

- 절차적 프로그래밍
- 객체지향 프로그래밍
- 함수형 프로그래밍

객체지향이란 무엇인가?

객체지향이란 무엇인가?

객체지향 프로그래밍의 시작

- 1960~ 클래스와 객체의 개념을 사용한 Simular 프로그래밍 언어에서 처음 등장
- 최초의 진정한 객체지향 프로그래밍 언어 Smalltalk

객체지향이란 무엇인가?

객체지향 프로그래밍과 객체지향 프로그래밍 언어

- 객체지향 프로그래밍
  - 프로그래밍 패러다임 또는 프로그래밍 스타일
  - 클래스 또는 객체가 기본 단위
  - 캡슐화, 추상화, 상속, 다형성 특성 사용
- 객체지향 프로그래밍 언어
  - 캡,추,상,다를 구현할 수 있도록 클래스 또는 객체 문법 지원
- 객체지향 언어에 대한 명확한 정의는 불필요하며 4대 특성을 모두 지원하지 않더라도 객체지향 언어라고 할 수 있다. 예를 들면 Go는 상속기능을 포함하지 않는다.

객체지향이란 무엇인가?
객체지향 분석과 객체지향 설계

- 객체지향 소프트웨어 개발의 3단계
  - 객체지향 분석
    - 무엇을 해야하는지 알아내는 것 (=요구사항 분석)
  - 객체지향 설계
    - 어떻게 만들어야 하는지 정의하는 것 (=시스템 설계)
  - 객체지향 프로그래밍

캡슐화, 추상화, 상속, 다형성이 등장한 이유

캡슐화, 추상화, 상속, 다형성이 등장한 이유

캡슐화 encapsulation

- 객체의 외부에서 접근 가능한 인터페이스를 제한
- 프로그래밍 언어 자체에서 문법 제공 (접근 제어)
  - Swift의 Access Control (open, public, internal, fileprivate, private)
  - Kotlin의 Visibility Modifier (private, protected, internal, public)
- 모든 속성에 접근이 열려있는 경우
  - 과도한 유연성 -> 가독성, 유지 보수성 악영향
- 필요한 속성에만 접근이 열려있는 경우
  - 모든 세부 사항을 알지 않아도 됨 -> 학습 비용과 실수할 가능성을 줄일 수 있음

캡슐화, 추상화, 상속, 다형성이 등장한 이유

추상화 abstraction

- 메서드의 내부 구현을 숨기는 것을 의미
  - 클래스 사용시에 구현 방식에 대한 고민을 줄이고 제공하는 기능에만 집중
- Swift - protocol / Kotlin - abstract class
- 기본 함수를 사용하는 것만으로도 추상화라고 할 수 있다
  - 세부 구현 사항을 **함수명**을 통해서 추상화 하고 있으므로
- 추상화 관점에서 올바른 네이밍 - 세부 구현사항이 변경되더라도 유효한 이름이 되도록

캡슐화, 추상화, 상속, 다형성이 등장한 이유

상속 inheritance

- 클래스 사이의 'is-a' 관계를 나타내는 데 사용
- 상속의 종류
  - 단일 상속 (Swift)
  - 다중 상속 : 하위 클래스가 여러 상위 클래스를 동시에 상속할 수 있음 (python)
    - 다이아몬드 상속 문제, 메서드 탐색 순서
- 코드 재사용성 증가
  - but 과도하게 사용하는 경우에 상속 계층 구조가 깊고 복잡해져 가독성과 유지 관리성 떨어지게 된다.

캡슐화, 추상화, 상속, 다형성이 등장한 이유

다형성 polymorhpism

- 런타임에 하위 클래스를 상위 클래스 대신 사용하고,하위 클래스의 메서드를 호출할 수 있는 특성
- 문법적으로 다형성을 구현할 수 있는 기능이 제공되어야 함
  - 상속과 메서드 재정의
  - 인터페이스 문법 제공
  - duck-typing
- 코드 확장성, 재사용성 향상

객체지향 분석/설계/프로그래밍 수행 방법

객체지향 분석/설계/프로그래밍 수행 방법

객체지향 분석 수행 방법

- 시작은 요구 사항 분석부터
- 추상적인 요구사항을 커뮤니케이션 등을 통해서 구체화 할 것
- 처음부터 완벽한 해결 방법 제시 X -> 기본 계획 부터 시작해서 단계를(4단계) 나누어 최적화 과정을 거쳐서 실제 구현 가능한 수준으로 명세를 정리해 나간다

객체지향 분석/설계/프로그래밍 수행 방법

객체지향 설계 방법

- 책임과 기능에 따라 필요한 클래스 리스트업
  - 요구 사항 명세에 따라 관련된 기능들을 나열
  - 단일 책임 기능으로 분해
- 클래스를 정의하고, 클래스의 속성과 메서드를 정의한다.
  - 실제 구현하는 것은 아직 X
  - 접근 제어 추가하여 내부에서만 사용할 속성과 메서드는 캡슐화

객체지향 분석/설계/프로그래밍 수행 방법

객체지향 설계 방법

- 클래스 간의 상호 작용 정의. (UML에 정의된 상호 작용들로 설명)
  - 일반화(generalization) - 일반적인 상속 관계
  - 실체화(realization) - 일반적으로 인터페이스와 구현 클래스 간의 관계
  - 합성(composition) - 포괄적인 관계, A에 B가 포함되며 A의 수명주기에 따라 B의 수명주기가 결정됨. B는 A와 관계없이 단독으로 존재 불가능.
  - 의존(dependency) - A가 B를 포함하거나, A의 메서드에서 B가 매개변수 또는 반환값으로 정의되어있는 경우
- 클래스 연결 및 실행 엔트리 포인트 제공

객체지향 분석/설계/프로그래밍 수행 방법

객체지향 프로그래밍 방법

- 기능 명세상의 문장단위로 구현해 나간다
- 모든 상세 기능을 한번에 구현하려고 하면 난이도가 증가함
  - 지속적으로 개선해 나가면서 완성해 나가기

객체지향 / 절차적 / 함수형 차이점

객체지향 / 절차적 / 함수형

절차적 프로그래밍

- 데이터와 메서드가 분리됨
- 프로그래밍의 단위는 **함수**
- 메서드를 순차 실행하며 데이터를 변경해 나가면서 기능을 구현하는 프로그래밍 스타일
- 요구사항이 간단하며 작업 흐름이 일직선 형태일 때 객체지향 대비 유리함
  - 복잡한 대규묘 프로그램 개발하는 경우 객체지향이 유리함
- 재사용, 확장, 유지 관리 용이

객체지향 / 절차적 / 함수형

함수형 프로그래밍

- 프로그램을 여러 함수의 조합으로 표현한다는 접근
- 프로그래밍의 단위는 **스테이트리스 함수**
- 함수는 상태를 가지지 않는다 (stateless function)
  - 함수 내부에서는 함수 외부의 멤버변수에 접근하지 않으며 필요한 값은 오직 매개변수로 전달받는다
  - 동일한 매개 변수를 실행하면 언제나 결과가 동일하다

객체지향을 절차적으로
잘못 사용하는 경우

객체지향을 절차지향으로 잘못쓰는 경우

- 모든 속성에 getter/setter 정의하는 경우 -> 객체지향의 캡슐화 특성 위반
- 전역 변수와 전역 메서드 남용하는 경우
  - 유지 보수성 악영향
  - 컴파일 시간 증가시킴
  - 재사용성 악화
- 데이터를 전용 클래스로 정의하고, 메서드는 다른 클래스로 정의되는 경우