객체지향의 패러다임 4원칙( 캡슐화, 다형성, 상속, 추상화)

다형성(Polymorphism)이란?

위키피디아에 따르면, 다형성을 아래와 같이 정의하고 있다.

프로그램 언어의 다형성은 그 프로그래밍 언어의 자료형 체계의 성질을 나타내는 것으로, 프로그램 언어의 각 요소들(상수, 변수, 식, 오브젝트, 함수, 메소드 등)이 다양한 자료형(type)에 속하는 것이 허가되는 성질을 가리킨다. 반댓말은 단형성으로, 프로그램언어의 각 요소가 한가지 형태만 가지는 성질을 가리킨다.

다형성이란 하나의 객체에 여러 가지 타입을 대입할 수 있다는 것을 의미한다.

반대로, 단형성은 하나의 객체에 하나의 타입만 대응할 수 있다. 다형성을 설명하기 전에 단형성을 사용한 코드를 보자.

예를 들어, 특정한 자료형을 문자열로 바꾼다고 가정해보자. 숫자나 날짜 자료형을 문자열로 바꾸기 위해 아래와 같은 메소드 들을 정의할 수 있다.

String age = stringFromNumber(number); // 숫자를 문자열로 바꾸는 경우.

String today = stringFromDate (date); // 날짜를 문자열로 바꾸는 경우

숫자를 문자열로 바꿀 때는 stringFromNumber(), 날짜를 문자열로 바꿀 때는 stringFromDate()를 사용하는 것이다. 이것은 단형성 체계의 언어에서는 함수의 이름이 중복 될 수 없기 때문이다.

여기서 눈치가 빠른 사람은 자바에서는 오버로딩을 사용하여 함수의 이름이 중복되어도 무방하다는 것을 알고 있다. 아무튼, 위와 같이 비슷한 기능을 하는 함수의 이름을 자료형에 따라 끝도 없이 나열한다면. 상상만해도 끔찍하다. 그래서 다형성 체계의 언어에서는 함수의 이름을 같게 하여

위의 작업을 아래와 같이 간결하게 만들수 있다.

String age = stringValue(number); //숫자를 문자열로 바꿀 때

String today =stringValue(date); // 날짜를 문자열로 바꾸는 경우.

물론 여러 자료형에 따라 문자열로 바꾸는 작업 자체가 줄어든 것은 아니지만 메소드 하나의 이름만 가지고도 기억하기 쉽고 메소드의 이름을 절약한다는 장점이 있다. 개발을 하다보면 메소드나 변수명의 이름을 짓기가 골치아픈데 그 부담을 덜어준다.

다형성을 구현하는 방법

오버로딩, 오버라이딩, 함수형 인터페이스가 존재한다.

1. 오버로딩

첫 번째는 위에서 언급했던 오버로딩이다. 메소드 오버로딩은 한 클래스내에 이미 사용하는 이름의 메소드가 있더라도 특정 규칙을 지킨다면 동일한 이름의 메소드를 잘 정의하도록 허용하는 기술이다.

규칙. 1 메소드의 이름이 같아야한다, 2. 매개변수의 개수 또는 타입 달라야한다.

3. 매개 변수는 같고, 리턴 타입이 다를 때는 성립하지 않는다.

4. 오버로딩된 메소드들은 매개 변수로만 구분 될 수있다.

4가지 장황하게 써놨지만, 핵심은 매개 변수의 타입, 개수, 순서 중에 하나 이상 달라야 합니다. 위 오버 로딩을 구현한 실제 JDK 코드로는 String.valueOf()가 있습니다. 제가 처음 다형성을 소개할 때 그 stringValue()와 똑같은 역할을 하는 메소드입니다.

제가 예시로 들었던 날짜를 문자열로 바꿔주는 메소드는 없지만, 그래도 상당히 많은 자료형을 문자열로 바꿔주는 작업을 해 주고 있습니다. 그리고 매개 변수가 동일한 메소드는 하나도 존재하지 않습니다. 이러한 오버로딩을 통해 우리는 다형성을 구현해 줄 수 있습니다.

하지만, 메소드 오버로딩을 사용하는 경우 요구 사항이 바뀌었을 때 모든 메소드를 수정할 수도 있으므로 꼭 필요한 경우에만 사용해야 합니다. 일반적인 메소드 보다는 생성자 오버로딩을 사용하는걸 추천.

1. 오버라이딩.

메소드 오버라이딩은 상위클래스의 메소드를 재정의하는 것을 의미한다. 주로, 클래스 상속이나 인터페이스 상속을 통해 구현할 수 있다. 또한, 개발하면서 다형성 구현이다 하면 이러한 상속을 사용 하는 편이 많다. 인터페이스 상속을 예시로 들어보자.

예를들어 운송 수단 종류인 자동차 비행기, 기차가 있다고 하자. 모두 움직일 수 있으므로 move() 메소드를 각각 아래와 같이 정의할 수 있다.

Public class Car {

Public void move(){

System.out.println(“도로로 달린다.”);

}}

**public** **class** **Airplane** {

**public** **void** **move**() {

System.out.println("하늘을 납니다.");

}

}

**public** **class** **Train** {

**public** **void** **move**() {

System.out.println("선로로 주행합니다.");

}

}

 각각의 운송수단들을 움직여보자

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** Car car = **new** Car();

**final** Airplane airplane = **new** Airplane();

**final** Train train = **new** Train();

car.move();

airplane.move();

train.move();

}

}

우리는 각각의 운송수단을 움직이기위하여 서로 다른 객체를 만들어 주었다. 그리고 move() 메소드를 각 객체마다 실행해 주었다.

만약 100가지의 운송수단이 생긴다면 move() 메소드를 손으로 100번을 쳐야한다. 왜냐하면 이들을 반복문으로 묶어줄 수도 없기 때문이다. 이때, 오버라이딩을 사용하면 위 문제를 해결할 수 있다.

Movable이라는 인터페이스를 만들어보자. 이 인터페이스의 메소드를 move()가 있고, Car Tarin, Airplane에 상속해준다. 그리고 하위 클래스들은 각각 move() 메소드를 오버라이딩 한다.

**public** **interface** **Movable** {

**void** **move**();

}

**public** **class** **Car** **implements** **Movable** {

**@Override**

**public** **void** **move**() {

System.out.println("도로로 달립니다.");

}

}

**public** **class** **Airplane** **implements** **Movable** {

**@Override**

**public** **void** **move**() {

System.out.println("하늘을 납니다.");

}

}

**public** **class** **Train** **implements** **Movable** {

**@Override**

**public** **void** **move**() {

System.out.println("선로로 주행합니다.");

}

}

이제 Movable이라는 객체는 Car가 될 수도 있고, Airplane이 될 수도있고, Train이 될 수도 있다. 맨 처음에 말했던 “한 객체에 여러가지 타입을 대응할 수 있다.”를 보여주는 예시이다. 그래서 아래와 같이 다형성을 이용하여 Car, Airplane, Train의 move() 메소드를 호출할 수 도 있다.

Public class Main {

Public static void main (String[] args) {

Final List<Movable> movables = Arrays.asList(new Car(), new Train(), new Airplane());

Movable.move();

}

}

}

위와 같이 오버라이딩을 이용하면 비슷한 로직을 줄일 수있다는 큰 장점이 존재함.

1. 함수형 인터페이스

함수형 인터페이스는 주로 Enum에서 빛을 발한다. 먼저 함수형 인터페이스란, 한 개의 추상 메소드를 가지고있는 인터페이스를 말한다.

위의 오버라이딩 예시에서 언급했던 Movable도 함수형 인터페이스에 속한다. 그리고 이것은 람다식으로 간단히 표현이 가능하다.

Public class Main {

Public static void main(String[] args) {

Final List<Movable> movables = Arrays.asList(new Car(), new Train(), new Airplane(), () -> System.out.println(“길을 걷는다”));

For (final Movable movable : movables) {

Movable.move();

}

}}

이런 람다식으로 하나 뿐인 추상 메소드를 정의하여 사용 할 수 잇다.

이렇게 우리가 만든 인터페이스도 함수형 인터페이스가 될 수 있지만

자바에서 제공하는 대표적인 함수형 인터페이스가 있다. 그 종류로는

Predicate, consumer, function, supplier XXX Operator 등이 있다. 앞의 XXX 붙은 것은 접두사를 의미한다.

2. 캡슐화(Encapsulation)란?

객체의 속성(data fields)과 행위(methods)를 하나로 묶고, 실제 구현 내용 일부를 외부에 감추어 은닉한다.

여기서 은닉이라는 단어 때문에 캡슐화와 은닉화를 혼동하는 경우가 많다.

은닉화는 캡슐화를 통해 얻어지는 “실제 구현 내용 일부를 외부에 감추는’ 효과입니다.

객체의 속성과 행위를 묶으면 응집도가 올라가므로 자율적인 객체가 된다는 장점이 있다. 자율적인 객체가 되면 단순히 데이터 전달자 역할이 아니라, 자신의 상태를 스스로 처리할 수 있다.

그런데 이 상황에서 은닉화가 이루어지지 않는다면 어떨까? 객체가 스스로의 상태를 처리하지 못하고, 외부에서 속성을 꺼내와서 상태를 수정하게 되면 높은 결합도와 낮은 응집도를 가진 수동적인 객체가된다

이렇게 수동적인 객체는 유지보수를 어렵게 하므로 은닉화를 통해 내부 구현을 감춰야 한다.

또한 돈과 같이 민감한 부분을 외부에 공개하게 되면, 악의적인 사용자가 돈의 양을 바꿔 버릴수도있음.

보안적인 프로그램을 위해 캡슐화를 지켜야한다.

은닉화는 크게 필드 데이터의 은닉화와 기능의 은닉화가 존재한다. 그리고

이러한 은닉화는 주로 접근 제한자를 통해 설정할 수 있다. 우선 은닉화를 지키지 않는 코드를 보자

이산이라는 남자 아이가 있고, 그 친구가 배가 고픈지 안고픈지 알아보는 코드를 작성

**public** **class** **YeeSan** {

**public** **final** Stomach stomach;

**public** **YeeSan**(**int** capacity) {

**this**.stomach = **new** Stomach(capacity);

}

**public** Stomach **getStomach**() {

**return** stomach;

}

}

**public** **class** **Stomach** {

**public** **final** **int** capacity;

**public** **Stomach**(**int** capacity) {

**this**.capacity = capacity;

}

**public** **int** **getCapacity**() {

**return** capacity;

}

}

Yeesan 클래스 안에는 위장을 뜻하는 Stomach 클래스가 있고, Stomach 클래스의 필드로는 위장의 들어있는 음식물의 양을 뜻하는 capacity가 있다. 이제, 사용자 입장에서는 “이산아 너 배고프니, 배부르니?” 물어볼 때 다음과 같은 코드를 작성할 수 있다.

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** YeeSan yeeSan = **new** YeeSan(0);

**if** (yeeSan.getStomach().getCapacity() == 0) {

System.out.println("이산이는 배가 고프다.");

} **else** **if** (yeeSan.getStomach().getCapacity() <= 5) {

System.out.println("이산이는 약간 배가 고프다");

} **else** {

System.out.println("이산이는 배부르다.");

}

}

}

처음에 YeeSan 객체를 만들고, YeeSan의 속성인 Stomach를 가져오고, 또 Stomach의 속성인 capacity를 가져와서 행위를 정의해 주었다.

그리고 YeeSan의 속성인 Stomach, 그리고 Stomach의 속성인 capacity 모두 public 이므로 생성자로 초기화해 준 이후 임의의 값을 변경할 수도 있다. 즉 내부 구현이 외부에 노출되었고, 외부에서 이들 맘대로 사용하고 수정할 수 있게된 셈이다.

자, 이러한 상황에서 이산이는 더 이상 사람이 아니고 사이보그가 되었다고 가정해보자. 사이보그는 위장 대신 바이오 전지를 사용할 수도 있을 것입니다. 그러면 YeeSan 클래스는 다음과 같이 바뀝니다.

4.추상화(Abstraction)란? 컴퓨터 과학에서 추상화는 복잡한 자료를 핵심적인 개념 또는 기능을 간추려 내는 것을 말한다.

추상화는 이처럼 구체적인 사물들간의 공통점을 취하고 차이점을 버리는 일반화를 사용하거나, 중요한 부분을 강조하기 위해 불필요한 세부사항을 제거함으로써 단순하게 만든다. 결국 핵심은 불필요한 코드를 제거, 중요한 부분을 살린다.

예를들어 for, while, do~ while 등과 같은 문법은 반복문을 추상화한 것이다. 실제로는 CPU의 명령을 통해서 반복이 구현되겠지만, 이 구현으로부터 ‘반복’이라는 개념을 뽑아내서, FOR while, do ~ while등으로 추상화 한 것이다.

조금 더 직관적인 예시를 들어보자. 만약 은행 관련 애플리케이션 만든다고 가정해보자. 은행을 이용하는 고객의 정보가 필요한데, 고객의 정보라고 하면 이름, 주소, 휴대폰 번호, 주소 등이 있을 수 있다. 그런데 사실 정보라는 범위가 모호하므로 위의 정보외에도 좋아하는 음식, 취미 특기등까지도 포함할 수 있다. 다만 은행 애플리케이션에서의 고객 정보로 좋아하는 음식, 취미, 특기는 필요하지 않다. 그래서 이러한 불필요한 정보를 제거함으로써 중요한 정보만 남기는 것 자체도 추상화라고 할 수있다.

추상화 예제.

일반화는 구체적인 사물간의 공통점을 취하고 차이점은 버린다.

SuperCar, SnowCar, FastCar가 있다고 해보자.

**public** **class** **SuperCar** {

**public** **void** **move**() {

System.out.println("슈퍼하게 달립니다.");

}

}

**public** **class** **SnowCar** {

**public** **void** **move**() {

System.out.println("겨울에 잘 달립니다.");

}

}

**public** **class** **FastCar** {

**public** **void** **move**() {

System.out.println("빠르게 달립니다.");

}

}

그리고 이들을 움직이기 위하여 사용자 입장에서는 이렇게 코드 작성

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** SuperCar superCar = **new** SuperCar();

**final** SnowCar snowCar = **new** SnowCar();

**final** FastCar fastCar = **new** FastCar();

superCar.move();

snowCar.move();

fastCar.move();

}

}

그런데 움직인다는 행위는 같으므로 차의 공통된 속성인 ‘move’를 상위 클래스로 뽑아낼 수 있다. 똑 같은 자동차 이므로 Car라는 상위 클래스를 두어 상속 관계를 만들어도 무방하다. 이때, 저번 시간에 배운 캡슐화를 지키기 위해서 추상 클래스로 만들었다.

**public** **abstract** **class** **Car** {

**public** **abstract** **void** **move**();

}

**public** **class** **FastCar** **extends** **Car**{

**@Override**

**public** **void** **move**() {

System.out.println("빠르게 달립니다.");

}

}

**public** **class** **SnowCar** **extends** **Car** {

**@Override**

**public** **void** **move**() {

System.out.println("겨울에 잘 달립니다.");

}

}

**public** **class** **SuperCar** **extends** **Car** {

**@Override**

**public** **void** **move**() {

System.out.println("슈퍼하게 달립니다.");

}

}

그리고 하위 클래스들은 추상메소드를 재정의하면 된다.

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** List<Car> cars = Arrays.asList(**new** SuperCar(), **new** SnowCar(), **new** FastCar());

**for** (**final** Car car : cars) {

car.move();

}

}

}

일반화를 통해 추상화를 구현하고, 그 목적을 위한 방식은 바로 type이다. 좀 더 상위 타입을 정의 함으로써 추상화와 동시에 다형성을 구현할 수 있게된다. 그리고 상위 타입의 방식으로 상속을 사용하였다.

더불어, 상위 타입은 추상 클래스로 선언함으로써 캡슐화까지도 지킬 수 있다.

상속이란? 객체들간의 관계를 구축하는 방법이다.

실생활에서의 상속은 부모가 자식에게 무언가 물려준다는 의미가 있다

사실 OOP에서의 상속도 크게 다르지 않다. 부모 클래스의 메소드 혹은 필드를 자식 클래스에게 물려준다.

그래서 상속을 위키피디아 설명에 덧붙여서, ‘기존 클래스에 기능을 추가하거나 재정의 하며 새로운 클래스를 정의한다.’

하위 클래스에서 이용할 공통된 메소드를 상위 클래스가 제공해 주고,

하위 클래스는 상위 클래스에서 없는 기능을 추가하거나 수정하는 것이다.

**(1) 코드를 재사용할 수 있다.**

이 장점은 실생활에서의 상속의 의미를 가장 잘 살린다고 생각합니다. 앞서 말했듯, 하위 클래스는 상위 클래스의 메소드나 필드를 이용할 수 있습니다. 다만, protected 혹은 public 접근 제한자를 주어야 합니다. 아무리 하위 클래스라도 상위 클래스의 해당 기능에 private을 설정한다면 접근할 수 없습니다. 우선, 저는 상속을 적용하지 않은 비효율적인 코드부터 작성해 보겠습니다.

(2) 다형성 구현

객체와의 관계를 만들어 준다는 것은 다시 말해 다형성을 살린다는 의미이기도 하다. 앞서 다형성 구현 방식으로 오버로딩, 오버라이딩, 함수형 인터페이스를 이야기 하였지만, 사실 오버라이딩 없는 상속만 사용해도 다형성의 의미를 살릴 수 있다.

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** Boy boy = **new** Boy("철수");

**final** Girl girl = **new** Girl("영희");

boy.study();

girl.study();

System.out.println();

boy.breathe();

girl.breathe();

}

}

다형성이 없다면, 위와 같이 Student로 추출한 메소드들을 하나 하나 코드를 작성해야 한다.

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** Boy boy = **new** Boy("철수");

**final** Girl girl = **new** Girl("영희");

**final** List<Student> students = Arrays.asList(boy, girl);

**for** (**final** Student student : students) {

student.study();

student.breathe();

System.out.println();

}

}

}

대신 이렇게 상속을 통해 다형성 구현하면, 반복문을 통해 공통된 메소드들을 딱 한번 씩만 코드로 작성하면된다.

그러나 상속의 문제점은. 상위 클래스에 강하게 결합한다.

문제점을 사실 여러 개 쪼갤수는 있지만, 가장 핵심은 하위 클래스가 상위 클래스에 강하게 결합하게 된다. 하위 클래스는 상위 클래스의 메소드나 필드를 사용할 수 있다. 다르게 보면, 하위 클래스는 상위 클래스와 강하게 결합하므로 응집력이 낮아 수동적인 객체로 됨. 수동적인 객체가 되면 결합된 객체에 영향을 크게 받으므로 변화에 대처하기 어려워진다.

**public** **class** **Document** {

**public** **int** **length**() {

**return** **this**.content().length();

}

**public** String **content**() {

*// ...*

}

}

예를 들어 Document 라는 상위 클래스가 있다. 여기서 content() 메소드의 리턴 타입이 String이 아니고 char[]로 바뀐다면 어떨까?

하위 클래스에서 해당 메소드를 사용하고 있다면 모조리 고쳐야한다.

이렇듯, 상속은 상위 클래스의 강하게 결합하여 변경에 취약하며 캡슐화를 깨뜨리는 프로그램을 만들 여지가 있습니다. 추가로 더 이야기하자면, 상위 클래스의 결함이 있다면 하위 클래스도 그 결함을 그대로 넘겨 받게 됩니다. (ex. Vector에 상속을 받은 Stack). Vector가 왜 결함이 있는 클래스인지

처음에는 상속 찬양을 하다가 이제는 상속 비난을 하는 글을 보시면서, "아니, 그러면 상속은 쓰라는 거야 쓰지 말라는 거야?" 라는 생각이 드실 수 있습니다. 제가 말씀드리고 싶은 것은 상속은 **잘** 써야 한다는 것입니다. 이 잘 쓰기 위한 전제 조건을 지금부터 설명드리겠습니다.

상속은 **클래스의 행동을 확장할 때가 아니라 정제할 때 사용하는 편이 좋습니다. 확장이란 새로운 행동을 덧붙여 기존의 행동을 부분적으로 보완하는 것을 의미하고 정제란 부분적으로 불완전한 행동을 완전하게 만드는 것을 의미한다. 객체 지향 초기에 가장 중요시 여기는 개념은 재사용성이었지만,**

**지금은 워낙 시스템이 방대해지고 잦은 변화가 발생하다 보니 유연성이 더 중요한 개념이 되었다.**

**즉 상속은 정제할 때 사용하는 것이 바람직하다. 인터페이스와 추상 클래스를 설명한 적이있다. 그 당시에는 별로 필요도 없어 보이는 인터페이스나 추상 클래스를 왜 배우는지도 몰랐고, 실제로 프로젝트에도 거의 적용하지 않은 경우가 많을 것으로 예상된다.**

**추상 클래스나 인터페이스는 추상메소드가 존재하며, 하위 클래스에서는 해당 추상 메소드를 재정의해야한다. 추상 메소드는 불완전한 것이고, 하위 클래스가 불완전한 것을 재정의함으로써 완전한 것으로 탈바꿈 시킨다.**