객체지향

객체지향이론의 기본개념은 '실제 세계는 사물(객체)로 이루어져 있으며, 발생하는 모든 사건들은 사물간의 상호작용이다'라는 것이다. 실제 사물의 속성과 기능을 분석한 다음, 데이터(변수)와 함수로 정의함으로써 실제 세계를 컴퓨터 속에 옮겨 놓기 위함이었다. 객체지향이론은 상속, 캡슐화, 추상화 개념을 중심으로 점차 구체적으로 발전되었다.

객체지향언어는 기존의 프로그래밍언어와 다른 전혀 새로운 것이 아니라, 기존의 프로그래밍 언어에 몇 가지 새로운 규칙을 추가한 보다 발전된 형태의 것이다. 이러한 규칙들을 이용해 코드 간에 서로 관계를 맺어 줌으로써 보다 유기적으로 프로그램을 구성하는 것이 가능해졌다.

객체지향언어의 주요 특징

1. 코드의 재사용성이 높다 : 객체지향언어의 가장 큰 장점으로, 새로운 코드를 작성할 때 기존의 코드를 이용하여 쉽게 작성할 수 있다. 그래서 프로그램의 개발과 유지보수에 드는 시간과 비용을 획기적으로 개선하였다.

2. 코드의 관리가 용이하다 : 코드간의 관계를 이용해서 적은 노력으로 쉽게 코드를 변경할 수 있다.

3. 신뢰성이 높은 프로그래밍을 가능하게 한다 : 제어자와 메서드를 이용해서 데이터를 보호하고 올바른 값을 유지하도록 하며, 코드의 중복을 제거하여 코드의 불일치로 인한 오동작을 방지할 수 있다.

객체지향의 상속, 다형성과 같은 객체지향개념을 학습할 때 1)재사용성과 2)유지보수 그리고 3)중복된 코드의 제거의 관점에서 보면 보다 쉽게 이해할 수 있다.

객체지향 프로그래밍은 프로그래머에게 거시적 관점에서 설계할 수 있는 능력을 요구한다.

너무 객체지향개념에 얽매여서 고민하기 보다는 일단 프로그램을 기능적으로 완성한 다음 어떻게 하면 보다 객체지향적으로 코드를 개선할 수 있을지를 고민하여 점차 개선해 나가는 것이 좋다.

클래스

클래스란 '객체를 정의해놓은 것' 또는 '객체의 설계도 또는 틀'이라고 정의할 수 있다. 클래스는 객체를 생성하는데 사용되며, 객체는 클래스에 정의된 대로 생성된다.

클래스의 정의 : 객체를 정의해 놓은 것, 객체의 설계도(또는 틀)

클래스의 용도 : 클래스는 객체를 생성하는데 사용된다. 즉, 객체는 클래스에 정의된 대로 생성된다.

객체는 사전적 정의로 '실제로 존재하는 것'이다. 하지만 우리가 주변에서 볼 수 있는 책상, 의자, 자동차 뿐만 아니라 개념이나 논리와 같은 무형적인 것들도 객체로 간주한다.

프로그래밍에서의 객체는 클래스에 정의된 내용대로 메모리에 생성된 것을 뜻한다.

객체의 정의 : 실제로 존재하는 것, 사물 또는 개념

객체의 용도 : 객체가 가지고 있는 기능과 속성에 따라 다르다

유형의 객체 : 책상, 의자, 자동차, TV와 같은 사물

무형의 객체 : 수학공식, 프로그램 에러와 같은 논리나 개념

클래스와 객체의 관계를 우리가 살고 있는 실생활에서 예를 들면 제품의 설계도와 제품의 관계라고 할 수 있다. 예를 들면, TV설계도(클래스)는 TV라는 제품(객체)을 정의한 것이며, TV(객체)를 만드는데 사용된다.

클래스는 단지 객체를 생성하는데 사용될 뿐, 객체 그 자체는 아니다.

우리는 TV를 보기 위해서 TV가 필요한 것이지 TV의 설계도가 필요한 것이 아니다. TV설계도는 단지 TV라는 제품을 만드는 데만 사용될 뿐이다.

즉, 객체를 사용하기 위해서는 먼저 클래스로부터 객체를 생성하는 과정이 선행되어야 한다.

클래스를 정의하고 클래스를 통해 객체를 생성하는 이유는 설계도를 통해서 제품을 만드는 이유와 같다. 설계도를 잘 만들어 놓으면 제품을 만드는 일이 쉬워진다. 즉, 클래스를 한번만 잘 만들어 놓기만 하면, 매번 객체를 생성할 때마다 어떻게 객체를 만들어야 할지 고민하지 않아도 된다.

객체와 인스턴스

클래스로부터 객체를 만드는 과정을 클래스의 인스턴스화(instantiate)라고 하며, 어떤 클래스로부터 만들어진 객체를 그 클래스의 인스턴스(instance)라고 한다.

예를 들어, TV클래스로부터 만들어진 객체를 TV클래스의 인스턴스라고 한다.

인스턴스는 객체와 같은 의미이지만, 객체는 모든 인스턴스를 대표하는 포괄적인 의미를 갖고 있으며 인스턴스는 어떤 클래스로부터 만들어진 것인지를 강조하는 보다 구체적인 의미를 갖고 있다. 인스턴스와 객체는 같은 의미이므로 두 용어의 사용을 엄격히 구분할 필요는 없다. 다만 문맥에 따라 구별하여 사용하는 것이 좋다.

[ 클래스 ---인스턴스화---> 인스턴스(객체) ]

객체의 구성요소

객체는 속성과 기능, 두 종류의 구성요소로 이루어져 있으며 다수의 속성과 다수의 기능을 갖는다. 즉, 객체는 속성과 기능의 집합이다. 그리고 객체가 가지고 있는 속성과 기능을 객체의 멤버(구성원, member)라 한다.

속성과 기능은 같은 뜻의 여러 용어가 있으나 속성은 멤버변수, 기능은 메서드란 용어로 많이 사용된다.

보다 쉽게 이해할 수 있게 TV를 예로 들어보면, TV의 속성은 전원상태, 크기, 길이 , 높이, 생상, 볼륨, 채널과 같은 것들이 있다. 기능은 켜기, 그기, 볼륨 높이기, 채널 변경하기 등이 있다.

이를 토대로 TV클래스를 만들어 보면 다음과 같다.

프로그래밍에 필요한 속성과 기능만 선택해서 클래스에 작성하면 된다. 또 변수의 자료형은 속성의 값에 알맞은 것을 선택해야 한다. 예를 들어 전원상태(power)의 경우 on, off 두 가지 값만 가질 수 있기 때문에 boolean형으로 선언했다. power() 메서드는 power의 값이 true이면 false로 false이면 true로 변경하는ㄴㄴ 일을 한다.

tv클래스를 선언한 것은 tv설계도를 작성한 것에 불과하므로 tv인스턴스를 생성해야 제품 tv를 사용할 수 있다. tv클래스로부터 인스턴스가 생성되고 메서드가 사용되는 것을 단계별로 자세히 알아보도록 하자.

JVM의 메모리 구조

응용프로그램이 실행되면 JVM은 시스템으로부터 프로그램을 수행하는데 필요한 메모리를 할당받고 JVM은 이 메모리를 용도에 따라 여러 영역으로 나누어 관리한다. 그 중 3가지 주요영역인 1) method area, 2) call stack, 3) heap에 대해 알아보자

1) method area(메서드 영역)

프로그램 실행 중 어떤 클래스가 사용되면, JVM은 해당 클래스의 클래스 파일(\*.class)을 읽어서 분석하여 클래스에 대한 정보(클래스 데이터, CV)를 이곳에 저장한다. 이 때, 그 클래스의 클래스변수(class variable)도 이 영역에 함께 생성된다.

2) heap(힙)

인스턴스가 생성되는 공간. 프로그램 실행 중 생성되는 인스턴스는 모두 이곳에 생성된다. 즉, 인스턴스 변수(instance variable)들이 생성되는 공간이다.

3) call stack(또는 execution stack, 호출스택)

호출스택은 메서드의 작업에 필요한 메모리 공간을 제공한다. 메서드가 호출되면, 호출스택에 호출된 메서드를 위한 메모리가 할당되며, 이 메모리는 메서드가 작업을 수행하는 동안 지역변수(매개변수-parameter 포함)들과 연산의 중간결과 등을 저장하는데 사용된다. 그리고 메서드가 작업을 마치면 할당되었던 메모리공간은 반환되어 비워진다.

각 메서드를 위한 메모리상의 작업공간은 서로 구별되며, 첫 번째로 호출된 메서드를 위한 작업공간기 호출스택의 맨 밑에 마련되고 첫 번째 메서드 수행 중에 다른 메서드를 호출하면, 첫 번째 메서드의 바로 위에 두 번째로 호출된 메서드를 위한 공간이 마련된다. 이때 첫 번째 메서드는 수행을 멈추고, 두 번째 메서드가 수행되기 시작한다. 두 번째로 호출된 메서드가 수행을 마치게 되면, 두 번재 메서드를 위해 제공되었던 호출스택의 메모리공간이 반환되며, 첫 번째 메서드는 다시 수행을 계속하게 된다. 첫 번째 메서드가 수행을 마치면, 역시 제공되었던 메모리 공간이 호출스택에서 제거되며 호출스택은 완전히 비워지게 된다. 호출스택의 제일 상위에 위치하는 메서드가 현재 실행 중인 메서드이며, 나머지는 대기상태에 있게 된다.

호출스택의 특징을 다시 정리해보면 다음과 같다.

- 메서드가 호출되면 수행에 필요한만큼의 메모리를 반환하고 스택에서 제거된다.

- 메서드가 수행을 마치고나면 사용했던 메모리를 반환하고 스택에서 제거된다.

- 호출스택의 제일 위에 있는 메서드가 현재 실행 중인 메서드이다.

- 아래에 있는 메서드가 바로 위의 메서드를 호출한 메서드이다.

반환타입(return type)이 있는 메서드는 종료되면서 결과값을 자신을 호출한 메서드(caller)에게 반환한다. 대기상태에 있던 호출한 메서드(caller)는 넘겨받은 반환값으로 수행을 계속 진행하게 된다. 대기상태에 있던 호출한 메서드(caller)는 넘겨받은 반환값으로 수행을 계속 진행하게 된다.

클래스 메서드(static 메서드)와 인스턴스 메서드

클래스 메서드도 클래스 변수처럼 객체를 생성하지 않고도 호출이 가능하다. 반면 인스턴스 메서드는 반드시 객체를 생성해야만 호출할 수 있다.

그렇다면 클래스를 정의할 때, 어느 경우에 static을 사용해서 클래스 메서드로 정의해야 하는 것일까?

인스턴스 메서드는 인스턴스 변수와 관련된 작업을 하는, 즉 메서드의 작업을 수행하는데 인스턴스 변수를 필요로 하는 메서드이다. 그런데 인스턴스 변수는 인스턴스(객체)를 생성해야만 만들어지므로 인스턴스 메서드 역시 인스턴스를 생성해야만 호출 할 수 있는 것이다.

반면에 메서드 중에서 인스턴스 변수나 인스턴스 메서드를 사용하지 않는 즉, 인스턴스와 관계없는 메서드를 클래스 메서드(static)메서드로 정의한다.

물론, 인스턴스 변수를 사용하지 않는다고 해서 반드시 클래스 메서드로 정의해야하는 것은 아니지만 특별한 이유가 없는 한 그렇게 하는 것이 일반적이다.

1. 클래스를 설계할 때, 멤버변수 중 모든 인스턴스에 공통으로 사용하는 것에 static을 붙인다.

: 생성된 각 인스턴스는 서로 독립적이다. 그래서 각 인스턴스 변수는 서로 다른 값을 유지한다. 그러나 모든 인스턴스에서 같은 값이 유지되어야 하는 변수는 static을 붙여서 클래스 변수로 정의해야 한다.

2. 클래스 변수(static 변수)는 인스턴스를 생성하지 않아도 사용할 수 있다.

: static이 붙은 변수(클래스 변수)는 클래스가 메모리에 올라갈 때 이미 자동적으로 생성되기 때문이다.

3. 클래스 메서드(static 메서드)는 인스턴스 변수를 사용할 수 없다.

: 인스턴스 변수는 인스턴스가 반드시 존재해야만 사용할 수 있는데, 클래스 메서드(static 메서드)는 인스턴스 생성 없이 호출가능하므로 클래스 메서드가 호출되었을 때 인스턴스가 존재하지 않을 수도 있다. 그래서 클래스 메서드에서 인스턴스 변수의 사용을 금지한다. 반면에 인스턴스 변수나 인스턴스 메서드에서는 static이 붙은 멤버들을 사용하는 것이 언제나 가능하다. 인스턴스 변수가 존재한다는 것은 static변수가 이미 메모리에 존재한다는 것을 의미하기 때문이다.

4. 메서드 내에서 인스턴스 변수를 사용하지 않는다면, static을 붙이는 것을 고려한다.

: 메서드의 작업내용 중에서 인스턴스 변수를 필요로 한다면, static을 붙일 수 없다. 반대로 인스턴스 변수를 필요로 하지 않는다면 static을 붙이자. 메서드 호출시간이 짧아지므로 성능이 향상된다. static을 안 붙인 메서드(인스턴스 메서드)는 실행 시 호출되어야할 메서드를 찾는 과정이 추가적으로 필요하기 때문에 시간이 더 걸린다.

ex) Math클래스의 메서드는 모두 클래스 메서드이다. Math클래스의 경우 인스턴스 변수가 하나도 없고 수행하는데 필요한 값들을 모두 매개변수로 받아서 처리하기 때문이다.

정리

1. 클래스의 멤버변수 중 모든 인스턴스에 공통된 값을 유지해야하는 것이 있는지 살펴보고 있으면 static을 붙여준다.

2. 작성한 메서드 중에서 인스턴스 변수나 인스턴스 메서드를 사용하지 않는 메서드에 static을 붙일 것을 고려한다.

클래스 멤버(클래스 변수, 클래스 메서드)는 언제나 참조 또는 호출이 가능하기 때문에 인스턴스 멤버가 클래스 멤버를 사용하는 것은 아무런 문제가 안된다. 클래스 멤버 간의 참조 또는 호출 역시 아무런 문제가 없다. 그러나 인스턴스 멤버(인스턴스 변수와 인스턴스 메서드)는 반드시 객체를 생성한 후에만 참조 또는 호출이 가능하기 때문에 클래스 멤버가 인스턴스 멤버를 참조, 호출하기 위해서는 객체를 생성해야만 한다. 하지만 인스턴스 멤버간의 호출에는 아무런 문제가 없다. 하나의 인스턴스 멤버가 존재한다는 것은 인스턴스가 이미 생성되어 있다는 것을 의미하며, 즉 다른 인스턴스 멤버들도 모두 존재하기 때문이다. 실제로 같은 클래스 내에서 클래스 멤버가 인스턴스 멤버를 참조 또는 호출해야하는 경우는 드물다. 만일 그런 경우가 발생한다면, 인스턴스 메서드로 작성해야할 메서드를 클래스 메서드로 한 것은 아닌지 한 번 더 생각해봐야 한다.

추상화

추상화는 우리가 사는 삶에서도 중요하다.

오토바이를 탈 때 오토바이의 내부가 어떻게 돌아가는지 알 필요가 없고 열쇠를 구멍에 꽂고 시동 거는 법 그리고 기어와 액셀 사용법만 알아도 충분하다. 엔진 내부가 어떻게 작동하는지는 알필요가 없다. 물론 고장난 기계를 수리해야하는 정비공이라면 다르지만 특이한 경우가 아닌 이상 오토바이의 내부의 세세한 것까지 이해할 필요가 없다. 프로그래밍도 마찬가지다. 우리가 자바를 사용할 때 이진법을 알 필요가 없다. 자바 바이트코드를 알필요가 없다. 자바 컴파일러가 우리가 작성한 코드를 바이트코드로 변환시켜주고 JVM이 실행시켜주기 때문이다. 특히 객체지향프로그래밍에서 중요하다.

캡슐화(Encapsulation)

: 연관된 목적을 가지는 변수와 함수를 하나의 클래스로 묶어 외부에 쉽게 접근하지 못하도록 은닉하는 것이다. 즉, 캡슐화는 중요한 데이터를 쉽게 바꾸지 못하도록 할 때 사용한다. 캡슐화를 통해 외부에서 내부의 정보에 접근하거나 변경할 수 없게 직접적인 접근을 막고 객체가 제공하는 필드와 메서드를 통해서만 접근이 가능하다. 이렇게 접근을 제한함으로써 유지보수나 확장 시 오류의 범위를 최소화할 수 있고 객체 내의 정보 손상과 오용을 방지하고 데이터가 변경되어도 다른 객체에 영향을 주지 않아 독립성이 좋다. 또한 중복된 코드의 구현을 막고 구현했던 코드를 다시 사용하여 쉽게 로직을 바꿀 수 있다.

캡슐화는 접근제어자를 통해 이루어진다.

public : 접근 제한 없음

protected : 동일한 패키지 내에 존재하거나 파생 클래스에서만 접근 가능

default: 아무런 접근 제한자를 명시하지 않으면 default값이 되며 동일한 패키지 내에서만 접근 가능

private : 자기 자신의 클래스 내에서만 접근 가능

캡슐화와 추상화는 연관이 깊다. 특정 개체에 속하는 데이터를 숨겨 접근을 차단하고 연산만 허용한다. 추상화는 훨씬 포괄적인 개념으로, 모든 복잡한 내용을 숨기고 응용프로그램 인터페이스 유저는 복잡한 세부내용을 알 필요 없다. 그냥 사용하면 된다.

오버로딩

: 메서드도 변수와 마찬가지로 같은 클래스 내에서 서로 구별될 수 있어야 하기 때문에 각기 다른 이름을 가져야 한다. 하지만 이름이 같은 메서드라도 매개변수의 개수, 타입이 다르면 같은 이름을 사용해서 메서드를 정의할 수 있다. 이처럼, 한 클래스 내에 같은 이름의 메서드를 여러 개 정의하는 것을 메서드 오버로딩 또는 간단하게 오버로딩이라고 한다. 보통 하나의 메서드 이름은 하나의 기능만 구현하지만 하나의 메서드 이름으로 여러 기능을 구현하기 때문에 오버로딩이란 이름이 붙여졌다.

오버로딩의 조건

: 같은 이름의 메서드를 정의한다고 해서 무조건 오버로딩인 것은 아니다. 오버로딩이 성립하기 위해서는 다음의 조건들이 있다.

1. 메서드의 이름이 같아야 한다.

2. 매개변수의 개수 또는 타입이 달라야 한다.

즉, 메서드 이름이 같더라도 매개변수가 다르면 서로 구별될 수 있다는 것이다.

위 조건을 만족하지 못하면 컴파일시 에러가 발생한다. 또한 오버로딩된 메서드들은 매개변수에 의해서만 구별될 수 있기 때문에 반환타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다는 사실에 주의하자!

오버로딩의 대표적인 예시 println()메서드

오버로딩이 성립하지 않는 경우

1) int add(int a, int b)와 int add(int x, int y)

위 두 메서드는 매개변수의 이름만 다를 뿐 매개변수의 타입이 같기 때문에 오버로딩이 성립하지 않는다. -> 매개변수의 이름이 다른 건 메서드 내에서 사용되는 변수의 이름이 달라질 뿐이다.

2) int add(int a, int b)와 long add(int a, int b)의 경우

: 이 경우는 리턴타입만 다른 경우이다. 매개변수의 타입과 개수가 일치하기 때문에 add(3, 3)과 같이 호출했을 때 어떤 메서드가 호출된 것인지 결정할 수 없다. 따라서 오버로딩으로 간주되지 않는다.

오버로딩이 성립하는 경우

1) long add(int a, long b)와 long add(long a, int b)의 경우

: 두 메서드 모두 int형과 long형 매개변수가 하나씩 선언되었지만, 서로 순서가 다른 경우이다. 이 경우에는 호출 시 매개변수의 값에 의해 호출될 메서드가 구분될 수 있으므로 중복된 메서드 정의가 아닌, 오버로딩으로 간주한다. 하지만 매개변수 순서만 다른 것은 단점이 될 수 있다. add(3, 3)과 같은 경우에는 호출할 수 없기 때문이다.

2) int add(int a, int b), long add(long a, long b), long add(int[] a)의 경우

: 위 메서드들은 모두 바르게 오버로딩 되어 있다. 정의된 매개변수가 서로 다르긴 해도 모두 매개변수로 넘겨 받은 값을 더해서 그 결과를 돌려주는 일을 한다.

즉, 같은 일을 하지만 매개변수를 달리해야 하는 경우에, 이와 같이 이름은 같고 매개변수를 다르게 하여 오버로딩을 구현한다.

오버로딩의 장점

: 만일 오버로딩이 허용되지 않는다면 모두 근본적으로 같은 기능을 하는 메서드이더라도 메서드의 이름을 다르게 지어야 한다. 그래서 메서드를 작성하는 쪽에서는 이름 짓기가 어렵다. 메서드를 사용하는 쪽에서는 기능이 같지만 이름을 일일히 구분해서 사용해야 한다. 오버로딩은 이러한 단점들을 제거한다. 하나의 이름으로 정의할 수 있을 뿐만 아니라 하나의 이름만 기억해서 사용할 수 있기 때문이다. 또 이름이 같기 때문에 메서드 이름만 보고 기능을 쉽게 예측할 수 있다.

가변인자와 오버로딩

메서드의 매개변수의 개수를 동적으로 지정해줄 수 있는데 이 기능을 가변인자(variable arguments)라고 한다. 가변인자는 '타입... 변수명'과 같은 형식으로 선언하며, PrintStream 클래스의 printf()가 대표적인 예이다.

public PrintStream printf(String format, Object... args) {...}

만일 가변인자 외에도 매개변수가 더 있다면 반드시 위와 같이 가변인자를 제일 마지막에 선언해야만 한다. 그렇지 않으면 컴파일 에러가 발생한다. 왜냐하면 가변인자와 가변인자가 아닌 것을 구별할 방법이 없기 때문이다.

public PrintStream printf(Object... args, String format) {...}은 허용되지 않는다!

가변인자는 내부적으로 배열을 이용한다. 그래서 가변인자가 선언된 메서드를 호출할 때마다 배열이 새로 생성된다. 가변인자는 편리하지만 비효율이 있으므로 꼭 필요한 경우에만 사용하도록 하자.

그렇다면 가변인자와 매개변수 타입을 배열로 하는 것과의 차이점이 무엇일까?

매개변수 타입을 배열로하면 반드시 인자를 지정해줘야한다.

즉 가변인자처럼 인자를 생략하는 것이 불가능하다. 그래서 null이나 길이가 0인 배열을 인자로 지정해줘야 하는 불편함이 있다.

가변인자를 가지고 메서드 오버로딩을 하게 되면 메서드가 구분되지 않아 컴파일 오류가 발생할 수 있다. 가능하면 가변인자를 사용한 메서드 오버로딩은 하지 않는 것이 좋다.

ex) 자바의 정석 1권 289,290p