**객체 지향적 설계 원칙**

**SRP(Single Responsibility Principle) :** 단일 책임 원칙  
클래스는 단 하나의 책임을 가져야 하며 클래스를 변경하는 이유는 단 하나의 이유이어야 한다.

**OCP(Open-Closed Principle) :** 개방-폐쇄 원칙  
확장에는 열려 있어야 하고 변경에는 닫혀 있어야 한다.

**LSP(Liskov Substitution Principle) :** 리스코프 치환 원칙  
상위 타입의 객체를 하위 타입의 객체로 치환해도 상위 타입을 사용하는 프로그램은 정상적으로 동작해야 한다.

**ISP(Interface Segregation Principle) :** 인터페이스 분리 원칙  
인터페이스는 그 인터페이스를 사용하는 클라이언트를 기준으로 분리해야 한다.

**DIP(Dependency Inversion Principle) :** 의존 역전 원칙  
고수준 모듈은 저수준 모듈의 구현에 의존해서는 안된다.

객체지향의 기본원리

- 추상화 (abstraction)

- 캡슐화 (Encapuslation)

- 모듈화

- 계층화 (Hierarchy)

객체지향의 기본개념

- 객체 (Object)

- 클래스 ( Class)

- 다형성 (Polymorphism)

- 관계 (Relationship)

**자바의 특징**

**이식성이 높은 언어이다**

 이식성이란 서로 다른 실행 환경을 가진 시스템 간에 프로그램을 옮겨 실행할 수 있는 것을 말한다. 자바 언어로 개발된 프로그램은 소스 파일을 수정하지 않아도, 자바 실행 환경(JRE)이 설치되어 있는 모든 운영 체제에서 실행 가능하다.

**객체 지향 언어이다**

 객체 지향 프로그래밍(Object Oriented Programming, OOP)이란 프로그램을 개발하는 기법으로, 부품에 해당하는 객체들을 먼저 만들고 이것들을 하나씩 조립 및 연결해서 전체 프로그램을 완성하는 기법을 말한다. 자바는 100% 객체 지향 언어이며, 객체를 만들기 위해 설계도인 클래스를 작성하고 객체와 객체를 연결하여 목적에 맞는 프로그램을 만들어 낸다. 또한 객체 지향 언어가 가져야 할 캡슐화, 상속, 다형성 기능을 완벽하게 지원한다.

**함수적 스타일 코딩을 지원한다**

 함수적 프로그래밍은 대용량의 데이터 병렬 처리와 이벤트 지향 프로그래밍을 위해 적합하기 때문에 최근 들어 다시 주목받고 있다. 자바는 이 함수적 프로그래밍을 위해 람다식을 자바 8부터 지원한다. 람다식을 사용하면 컬렉션 요소를 필터링, 매핑, 집계 처리하는게 쉬워지고, 코드가 매우 간결해진다는 장점이 있다.

**메모리를 자동으로 관리한다**

 C++이 메모리 관리를 위해 개발자가 직접 코드를 작성해야 하는 반면, 자바는 개발자가 메모리에 직접 접근할 수 없으며 자바가 직접 메모리를 관리한다. 객체를 생성할 때 자동적으로 메모리 영역을 찾아서 할당하고, 사용이 완료되면 Garbage Collector를 실행시켜 자동적으로 사용하지 않는 객체를 제거한다. 따라서 개발자는 메모리 관리의 수고스러움을 덜고, 코딩에 좀 더 집중할 수 있다.

**다양한 애플리케이션을 개발할 수 있다**

 자바는 윈도우, 리눅스, 유닉스, 맥 등 다양한 운영체제(OS)에서 실행되는 프로그램을 개발할 수 있다. 또한 콘솔 프로그램, 클라이언트용 윈도우 애플리케이션, 서버용 웹 애플리케이션, 모바일용 안드로이드 앱에 이르기까지 거의 모든 곳에 실행되는 프로그램을 개발할 수 있다.

**멀티스레드를 쉽게 구현할 수 있다**

 자바는 스레드 생성 및 제어와 관련된 라이브러리 API를 제공하고 있기 때문에 실행되는 운영체제에 상관없이 멀티 스레드를 쉽게 구현할 수 있다.

**동적 로딩(Dynamic Loading)을 지원한다**

 애플리케이션이 실행될 때 모든 객체가 생성되지 않고, 각 객체가 필요한 시점에 클래스를 동적 로딩해서 생성한다. 또한 유지보수 시 해당 클래스만 수정하면 되기 때문에 전체 애플리케이션을 다시 컴파일할 필요가 없다. 따라서 유지보수가 쉽고 빠르다.

**오픈소스 라이브러리가 풍부하다**

 자바는 오픈소스 언어이기 때문에 자바 프로그램에서 사용하는 라이브러리 또한 오픈소스가 많다. 오픈소스 라이브러리를 사용하면 개발 시간을 단축하면서 안정성이 높은 애플리케이션을 쉽게 개발할 수 있다.

**속도가 느리다**

 자바는 한 번의 컴파일링으로 실행 가능한 기계어가 만들어지지 않고 JVM에 의해 기계어로 번역되고 실행하는 과정을 거치기 때문에 C나 C++의 컴파일 단계에서 만들어지는 완전한 기계어보다는 속도가 느리다. 그러나 바이트 코드를 기계어로 변환해주는 JIT 컴파일러 같은 기술 적용으로 JVM의 기능이 향상되어 속도의 격차가 많이 줄어들었다.

**예외처리가 불편하다**

 자바는 다른 언어들과 달리 프로그램 실행 시 발생할 수 있는 예외(Exception)들을 개발자가 직접 선언하여 처리해야 한다. 그렇지 않으면 아예 컴파일이 되지 않는다.

**- JAVA** "Write Once , Run Anywhere"

1) 플랫폼 독립적 - JVM 이 통역자 역할

2) 객체 지향(Object-Oriented) 언어이다

3) 가비지 컬렉션 ( jvm이 메모리 관리 )

4) Multi Threading   활용도 좋음

**- 객체지향 프로그래밍 ( OOP )**

OOP ( Object Oriented Programming , 객체 지향 프로그래밍 )

- 객체의 관점에서 프로그래밍 하는 것을 의미.

- 관계성 있는 객체들의 집합 객체들의 유기적인 관계를 통해 프로세스 진행

즉, 어플리케이션을 구성하는 요소들을 객체로 바라보고 객체들을 유기적으로 연결하여 프로그래밍 하는 것.

- 코드의 재사용성과 중복제거가 가장 큰 목적

**- 객체( Object )**

개념

- 구현할 대상을 나타냄 ( 사람, 사물 등..) - 자신의 고유 이름과 상태, 행동을 갖는다.

- 클래스에 선언된 모양 그대로 생성된 실체.

특징

- 클래스의 인스턴스

- 객체는 모든 인스턴스를 대표하는 포괄적인 의미를 갖는다.

- 클래스의 타입으로 선언되었을 때 '객체'라고 부른다.

* **Instance ( 인스턴스 )**

개념

- 설계도를 바탕으로 소프트웨어 세계에 구현된 구체적인 실체

- 객체를 소프트웨어에 실체화하면 그것을 'Instance'라고 부른다.

- 실체화 된 인스턴스는 메모리에 할당된다.

특징

- 인스턴스는 객체에 포함된다고 볼 수 있다.

- OOP의 관점에서 객체가 메모리에 할당되어 실제 사용될 때 '인스턴스'라고 부른다.

- 추상적인 개념과 구체적인 객체 사이의 관계에 초점을 맞출 경우에 사용한다.

\* '~의 인스턴스'의 형태로 사용된다.

\* 객체는 클래스의 인스턴스다.

\* 객체 간의 링크는 클래스 간의 연관관계의 인스턴스다.

\* 실행 프로세스는 프로그램의 인스턴스다.

**3. Class**

- 객체를 만들어내기 위한 설계도 혹은 틀

- 객체를 정의하기 위한 상태(맴버변수)와 메서드(함수)로 구성된다.

- 연관되어 있는 변수와 메서드의 집합

- 클래스는 '설계도' / 객체는 '설계도로 구현한 모든 대상'을 의미.

- 클래스는 전부 혹은 일부를 그 클래스 특성으로부터 상속받는 서브클래스를 가질 수 있으며, 클래스는 각 서브클래스에 대해 수퍼클래스가 된다.

- 서브클래스는 자신만의 메소드와 변수를 정의할 수도 있다. 이러한 클래스와 그 서브클래스 간의 구조를 "클래스 계층(hierarchy)"이라 한다.

class 클래스명 {

필드 // 상태 속성 정보( 맴버필드 데이터 선언, 저장 )

생성자 // 멤버필드 초기화

메소드 // (행위)데이터 입력, 연산, 출력

}

**4. abstraction ( 추상화 )**

- 객체에서 공통된 속성과 행위를 추출하여 이름을 붙이는 것

- 객체 지향적 관점에서 공통된 속성들 중 필요한 부분을 클래스로 정의하는 것을 추상화라고 한다.

- 계층화(Hierarchy) 높은 레벨은 추상화 수준이 높고 아래 레벨일수록 구체적이다.

- ex) 물고기, 사자, 토끼, 뱀이 있을 때 우리는 이것들을 각각의 객체라 하며 이 객체들을 하나로 묶으려 할 때, 만약 동물 또는 생물이라는 어떤 추상적인 객체로 크게 정의한다고 하자. 이때 동물 또는 생물이라고 묶는 것을 추상화라고 한다.

검은색 보드마카, 청색 보드마카, 적색보드마카 --> 보드마카

**5. Generalization ( 일반화 ) / Inheritance ( 상속 )**

- 코드의 중복을 없애기 위함

- 상위 개념의 특징을 하위 개념이 물려받는 것

- 일반화 관계 "is a kind of"의 관계

- ex) '한국인' 이라는 집단(클래스)에 "홍길동"이라는 한국인(인스턴스)을 소속 시킨다.

- 검은색 보드마카, 청색 보드마카, 적색보드마카 --> 보드마카

**6. Specialization ( 구체적 ) - 특수화**

- 서브클래스가 슈퍼클래스와 비교했을 때 특수화 하는 경우

- 일반화의 반대

- 추상화 상태를 상속을 통하여 좀 더 구체적인 형태로 발전시키는 과정을 말함

- 보드마카 --> 검은색 보드마카, 청색 보드마카, 적색보드마카

**7. Realization**

- 추상화형태의 객체를 상속하여 자신이 사용하고자하는 구현방법에 맞게 오버라이딩하여

내용을 완성되어진 것을 의미

- 인터페이스의 추상메소드를 오버라이딩을 통해 완성시킴

**8. Association**

- ~ has a 관계 포함관계

- 연관은 구조적 관계로 정의되며 개념적으로 두 구성 요소가 서로 연결되어 있음을 의미.

이러한 종류의 관계를 사용 관계라고도한다. 여기서 한 클래스 인스턴스는 다른 클래스 인스턴스를 사용하거나 그 반대의 경우도 마찬가지입니다. 또는 둘 다 서로를 사용할 수 있습니다. 그러나 요점은 두 클래스의 인스턴스 수명이 서로 독립적이며 두 클래스간에 소유권이 없다는 것입니다.

- ex) 학생-교사 관계 / 각 학생은 여러 교사와 연관 될 수 있고 각 교사는 여러 학생과 연관 될 수 있다.

**9. dependency ( 종속성, 의존성 )**

- 클래스가 어떤 기능을 수행하려고 할 때 다른 클래스의 서비스가 필요한 경우 객체 사이의 도움을 주고 받는 것

- 필요할때 인스턴스를 생성하여 사용

- ex) 사람을 객체화하였을 떄 야구를 할때만 배트객체를 생성하여 사용하고 버리는 것

**10. Polymorphism ( 다형성 )**

- 부모클레스에서 물려받은 가상 함수를 자식 클래스 내에서 오버라이딩 되어 사용되는 것

- ex) 군대에서 K2 소총수 K1 소총수

사격 훈련이 있을 때 중대장이 '준비된 사수부터 발사!'라고 외치면 모든 소총수들은 명령을 받고 앞의 사로를 향해 총을 쏜다. 이때 중대장은 추상적 객체를 상속받은 모든 객체들에게 명령을 내린것이고 그 병사가 총이 뭐든간에 그냥 발사를 하라는 명령을 한것이다. 즉, 다형성이 없다면 K1 소총을 든 병사 발사, K2 소총을 든 병사 발사 라며 명령을 하나하나 내려야 할 것이다

**11. Encapuslation / 절차은닉 / 정보은닉 ( 캡슐화 )**

- 코드의 수정없는 재활용이 목적

- 데이터 구조와 데이터를 다루는 방법들을 결합 시켜 묶는 것, 변수와 함수를 하나로 묶는것을 말한다. 하지만 무작정 한대 묶으면 되는 것이 아니라 객체가 맡은 역할을 수행하기 위한 하나의 목적을 한데 묶는 것.

- 데이터를 절대로 외부에서 직접 접근을 하면 안되고 오로지 함수를 통해서만 접근해야하는데 이를 가능하게 해주는 것이 바로 캡슐화이다.

- 따라서 캡슐화에 성공하면 당연히 은닉화도 자연스럽게 효력이 나타난다.

- ex) 자동차라는 부모클래스 / 기름을 먹거나 달리는 기능을 하는 자동차인데,

만약 지붕뚜껑이 열리는 특수한 기능을 추가하고 싶다면 기존의 자동차에서 스포차카를 생성한다. 그러면 스포츠카는 기름도 먹고 달리면서 지붕두껑이 열리는 기능도 갖춘 자동차가 되는 것

**Overriding** = 부모에게 물려받아 변형한 함수

\*오버라이딩 = 상위클래스가 가지고 있는 메소드를 하위클래스가 재정의해서 사용한다.

**Overloading** = 이름은 같지만 그냥 전혀 다른 함수 // 자식이 변형하여 새로 정의한게 더 중요시 된다.

\*오버로딩 = 같은 이름의 메소드를 여러 개 가지면서 매개변수의 유형과 개수가 다르도록하는 기술

**객체지향 기본개념**

1. 객체

2. 클래스

3. 다형성

4. 관계 ( RelationShip )

- Generalization(추상화 수준 높음)

- 속성과 행위의 공유

- ~is a ~ 관계의 성립 ( 하위는 상위를 대신할 수 있다. ) / (상위는 하위를 대신할 수 없다.)

**primitive 타입**은 물리적으로 크기 결정 (레퍼런스 변환보다 상대적으로 훨씬 쉽다.)

double a = 10.0; a = 10 (O)

int a = 10; a= 10.0 (X)

**Reference**는 개념적으로 크기 결정

동물 a = new 동물(); (O)

= new 사람(); (O)

사람 b = new 사람(); (O)

= new 동물() (X)

**ㅁ생성자**

어떠한 클래스를 사용하고자 할 때는

new 연산자를 사용해 메모리 상에 클래스를 생성(인스턴스화) 하고

이 생성된 주소(인스턴스의 주소)를 리턴해 레퍼런스 변수에 저장하게 됩니다.

생성자는

1. 클래스의 이름과 같은 메서드이다.

2. 유일하게 메서드의 리턴타입이 없다.

3. 인스턴스 변수를 초기화 하는 역할을 한다.

ex) 클래스명 레퍼런스변수 = new 클래스명();

new 클래스명();

==> new 연산자에 의해 클래스가 힙 메모리 상에 인스턴스로 생성됩니다.

그리고 이 인스턴스는 생성자에 의해 초기화되게 됩니다. (정확히 말하면 인스턴스변수 초기화)

이 초기화된 인스턴스의 주소가 레퍼런스 변수에 저장되게 되는 것.

new라는 생성자를 통해 (설정해주지 않으면)자동으로 호출되는 생성자함수는 메소드 오버로딩 기술을 통해 다양하게 정의가능.

그런데 생성자에는 조건이 있다.

1. 생성자명은 클래스명과 같아야 한다.

2. 리턴타입이 없어야 한다.

생성자는 오버로딩이라는 기술을 통해 추상화 시킨 객체에 접근해서 다양한 값을 만들어 내는 메소드 // 인스턴스가 생성될 때 호출해야하는 인스턴스 초기화 메서드

**실행 = Application Class**

// classpath설정 -> Bean class찾기 ->인스턴스 생성해서 메모리에 적재

**실행 도움 = Bean**

리얼월드를 캡슐화 추상화 한 Bean

// 데이터 표현을 목적으로 하는 자바 클래스, Object Modeling

정적모델링 Static Modeling + 동적모델링Dynamic Modeling = Object Modeling

**-break는 해당 ‘loop만’ 종료 시키지만, return은 해당 루프문을 가진 ‘메소드’를 종료시킨다**

**-Continue는 해당 실행문 값을 건너뛰고 loop를 실행시킨다**

- path :

파일이나 디렉터리의 일반적인 형태로서 파일 시스템에서 고유한 위치를 지정한다.

경로는 문자열로 표현되는 디렉터리 트리 계층을 따라 시스템 위치를 가리키며 여기에서 구분자로 분리되는

경로 구성 요소를 통해 각각의 디렉터리를 대표하게 된다.

- classpath :

클래스를 찾기위한 경로이다. 즉, JVM이 프로그램을 실행할 때, 클래스파일을 찾는 데 기준이 되는 파일 경로를 말하는 것이다.

classpath에 지정된 경로를 모두 검색해서 특정 클래스에 대한 코드가 포함된 .class 파일을 찾는다.

찾으려는 클래스 코드가 포함된 .class 파일을 찾으면 첫 번째로 찾은 파일을 사용한다.

Generalization , Specialization 을 통해 공유를 하면서 효율적인 object modeling을 지원해주는 키워드 = extends

super() : 슈퍼 메서드 //

this() : 디스 메서드 //

super : 슈퍼 레퍼런스 //

this : 디스 레퍼런스 //

메서드 시그니처(Method signature)

- 메서드 명과 파라미터의 순서, 타입, 개수를 의미

- Java에서 메서드 시그니처는 메서드의 정의에서 메서드 이름과 매개변수 리스트의 조합을 말한다

인터페이스 = 순수한 추상클래스

= implements 를통해 구현