* **Java**
* **객체지향 프로그래밍 개념**

현실 세계에 존재하는 사물이나 개념을 그대로 프로그램을 옮겨 놓은 것

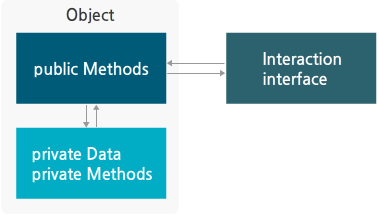
**객체:** 실제 사물이 가진 속성과 행동을 분석해 프로그램으로 구현한 것으로 상태와 행동을 가진다.

1. **객체지향의 특성**

* **은닉성 (Encapsulation)**

클래스의 사용자에게 필요한 최소의 기능만을 노출하고 내부를 숨기는 것

클래스에 선언되어 있는 필드와 메소드 중에 사용자 노출 여부 결정 가능 (Private 선언)



* **상속성**

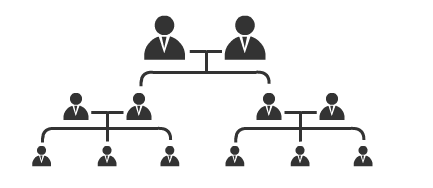
기존에 있는 클래스의 코드를 이용하여 새로운 코드를 생성하거나 재정의

프로그램의 개발 속도 개선 및 코드 중복 방지

기존에 정의된 클래스의 모든 멤버와 기능을 물려받아 사용하는 것으로 기존에 정의된 클래스는 부모 클래스, 물려 받은 클래스는 자식 클래스라고 한다. 자식 클래스에서 extends 키워드를 사용하여 상속 받는다.

public class Parent {}

public class Child extends Parent{}}

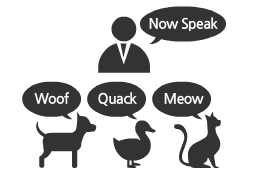


* **다형성**

서로 다른 클래스와 객체들이 같은 메시지에 대해 각자의 방법으로 작동할 수 있도록 하는 방법

하나의 인터페이스를 제공하고 여러 개의 클래스가 그 인터페이스를 각자의 방법으로 구현

상위 클래스에서 공통 규약을 만들고 서로 다른 하위 클래스의 객체에 각각 정의한 메소드를 동일한 방식으로 호출하여 사용 (Overriding)



Java에서는 다중 상속이 지원 되지 않는데 Interface를 사용하여 해결 한다. Interface는 추상 메소드만을 가진다.

Interface 키워드로 클래스를 선언하며 abstract으로 메소드를 추상화 한다. Abstract 키워드가 없어도 컴파일 시 자동 추가.

Interface Transport {

abstract void move();}

Implements 키워드를 통해 구현하며 Transport에 정의된 추상화 메소드인 move를 재정의 해주어야 한다.

public class Car implements Transport{

@Override

public void move() {

System.out.println("Car is moving");

}}

public class Train implements Transport{

@Override

public void move() {

System.out.println("Train is moving");

}}

public class Test {

public static void main(String args[]) {

Car c1 = new Car();

Train t1 = new Train();

Transport p1;

p1 = (Transport)t1;

if(p1 instanceof Car) {

p1.move();

}

else {

p1.move();

}

}}

* **Java API**

1. **String 클래스**

String, StringBuffer, StringBuilder의 비교

String – 문자열 조합 시 느리고, 동기화를 지원하지 않는다.

StringBuffer – 문자열 조합 시 빠르고, 동기화 지원

StringBuilder – 문자열 조합 시 빠르지만, 동기화를 지원하지 않는다.

String 사용시

String str = "Hello"; <- 객체 1

str = str + " World"; <- 객체 2

String 사용시 문자열 조합이 느린 이유는 String은 최초 생성된 데이터 대해서 변경이 이루이지지 않기 때문이다. 위와 같이 선언 하면 Hello, Hello World 두 개의 문자열 객체가 생성 된다.

StringBuffer 사용시

StringBuffer sb = new StringBuffer("Hello");

sb.append("World");

StringBuffer 사용시에는 Hello 객체를 만들고 append를 메소드를 통해 World를 붙인다. 위와 같이 선언 하면 Hello World하나의 객체만 사용한다.

객체를 생성하는 것은 꽤 무거운 연산에 속하므로 StringBuffer를 사용하는 것이 더 빠르다.

1. **Collection 구조**

컬렉션에는 객체만 저장이 된다.

|  |  |  |  |
| --- | --- | --- | --- |
|  | LIST | SET | MAP |
| 값 중복 허용 | 중복 허용 | 중복 X | Key 데이터 중복 X |
| 값 저장 방법 | 순서 유지 | 내부적인 순서 | Key, Value 쌍 |
| 대표 클래스 | ArrayList, LinkedList | HashSet, TreeSet | HashMap, TreeMap |

<List>

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

ArrayList<String> list = **new** ArrayList<String>();

list.add("First");

list.add("Second");

list.add(2, "Third");

list.set(2, "Three");

**for** (**int** i = 0; i < list.size(); i++) {

System.***out***.println(list.get(i));

}

list.remove(1);

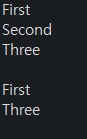
System.***out***.println();

**for** (**int** i = 0; i < list.size(); i++) {

System.***out***.println(list.get(i));

}

}



<Set>

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

HashSet<String> hset = **new** HashSet<String>();

hset.add("First");

hset.add("Second");

hset.add("First"); // Don't allow duplication

hset.add("Third");

// Output using Iterator

Iterator<String> iter = hset.iterator();

**while** (iter.hasNext()) {

System.***out***.println(iter.next());

}

// Output using for-each

**for** (String str : hset) {

System.***out***.println(str);

}

}



<Map>

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

HashMap<String, Integer> map = **new** HashMap<String, Integer>();

// Add elements <Name, Age>

map.put("David", 40);

map.put("Steve", 30);

map.put("David", 35);

**for** (String key : map.keySet()) {

System.***out***.println(key + " " + map.get(key));

}

// Check key value

**if** (map.containsKey("David")) {

System.***out***.println("David in map");

} **else** **if** (map.containsKey("Ko")) {

System.***out***.println("Ko in map");

}

// Remove

map.remove("David");

// Check contain Value

**if** (map.containsValue(30)) {

System.***out***.println("30 is in value");

}

**if** (map.containsValue(35)) {

System.***out***.println("35 is in value");

}

}



* **Exception 예외처리**

프로그램에서 발생하는 예외 상황에 대비하여 코드를 작성하여 비정상적인 종료를 막고 정상 실행 상태를 유지

오류의 종류

* **에러:** 프로그램 코드에 의해서 수습될 수 없는 오류 (메모리 부족, StackOverflow 등)
* **오류:** 프로그램 코드에서 수습될 수 있는 오류
* **예외처리 방법**

1. Try ~ Catch ~ Finally

코드 블록에서 예외처리

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

//Sample

**try** {

System.***out***.println("Step1");

System.***out***.println(1+0);

System.***out***.println("Step2");

System.***out***.println(0/0); //Occur Exception

System.***out***.println("Step3"); //Pass

System.***out***.println(1\*0);

}

**catch**(ArithmeticException e1) {

System.***out***.println("ArithmeticException"); //Handle Exception

}

**catch**(Exception e2) {

e2.printStackTrace();

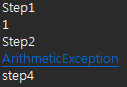
}

**finally** {

System.***out***.println("step4"); //Finaly Step

}

}



1. 메소드에 예외선언(thorws)

코드 블록 내의 예외 처리가 아닌 메소드에 예외 선언하여 메소드를 사용하는 측에서 예외처리 하도록 위임 처리

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

**int** num1, num2;

num1 = 1;

num2 = 0;

**try** {

System.***out***.println(*getSum*(num1,num2));

System.***out***.println(*getDiv*(num1,num2));

} **catch** (ArithmeticException e) {

System.***out***.println("Exception Handler");

} **catch** (Exception e) {

e.printStackTrace();

}

}

//Calculate Function

**static** **int** getSum(**int** a, **int** b) **throws** ArithmeticException, Exception{

**return** a+b;

}

**static** **int** getDiv(**int** a, **int** b) **throws** ArithmeticException, Exception{

**return** a/b;

}



* **예외 강제 발생**

thorw new Exception() 을 통해 강제로 예외처리 상황을 만들 수 있다.

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

System.***out***.println("Start Program");

**try** {

//Make Exception

**throw** **new** Exception("Make Exception e1");

}

**catch**(Exception e1) {

System.***out***.println(e1.getMessage());

}

**finally** {

System.***out***.println("End Program");

}

}



* **사용자 정의 Exception**

**class** CustomException **extends** Exception {

**static** **private** **int** *ERRCODE*;

CustomException(String msg, **int** errCode) {

**super**("CustomException: " + msg);

**this**.*ERRCODE* = errCode;

}

CustomException(String msg) {

**super**("CustomException: " + msg);

**this**.*ERRCODE* = 100;

}

**public** **int** getERRCode() {

**return** **this**.getERRCode();

}

}

* **Java 활용하기**

1. **Static**

별도의 객체 생성 없이 어디에서나 공통으로 사용하고자 하는 필드와 메소드를 표시할 때 사용

자바의 대표적 메모리 공간으로는 Static Area, Stack, Heap이 존재하고 Static으로 선언된 변수나 메소드는 Static Area에 저장 된다.

\*Static Area

접근 제한자를 public으로 선언할 경우 객체 생성 없이 어디에서나 접근 가능

Static area에 저장되는 모든 변수와 메소드들은 프로그램 로딩 시에 모두 메모리에 올라가고 종료 시까지 존재

\*Heap

New 키워드로 객체 생성시 인스턴스 변수가 저장되는 공간

클래스 로딩 후 stack 메모리 상에 존재하는 메소드와 참조 변수를 통해 접근 가능

\*Stack

프로그램의 시작점인 메소드가 호출 되는 공간

클래스 파일 로딩 시 static을 제외한 것들이 올라오는 공간