**Day02.객체지향 문법정리**

**।. 객체지향 문법**

|  |  |
| --- | --- |
| **변수선언방법** | **예제 코드** |
| 일일이 변수 선언 | String studentName = "홍길동";  **int** studentKor = 100;  **int** studentEng = 95;  **int** studentMath = 85;  **int** studentTotal = studentKor + studentEng + studentMath;  **int** studentAvg = studentTotal / 3; |
| 배열을 이용 | String[] name = {" 홍길동 "};  **int**[] cho = {100, 95, 85, 0, 0};  cho[3] = cho [0] + cho [1] + cho [2];  cho[4] = cho [3] / 3; |
| 이차원배열을 이용  (여러명의 변수들을 한꺼번에 사용하기 위해) | String[] name = {" 홍길동 "};  **int**[][] jumsu = {{90, 75, 85, 0, 0}};  **for** (**int** i = 0; i < jumsu.length; i++) {  **for** (**int** j = 0; j < 3; j++) {  jumsu[i][3] += jumsu[i][j];  }  jumsu[i][4] = jumsu[i][3] / 3;  } |
| ArrayList 이용 | ArrayList<String> name = **new** ArrayList<>();  name.add("홍길동 ");  ArrayList<Integer> cho = **new** ArrayList<>();  cho.add(90);  cho.add(75);  cho.add(85);  **for** (**int** i = 0; i < jumsu.size(); i++) {  ArrayList<Integer> su = jumsu.get(i);  **int** total = 0;  **for** (**int** j = 0; j < su.size(); j++) {  total = su.get(0)+su.get(1)+su.get(2);  }  su.add(total);  su.add(total / 3);  } |
| VO클래스를 이용 | **public** **class** Jumsu {  **private** String name;  **private** **int** kor;  **private** **int** eng;  **private** **int** math;  **private** **int** totalScore;  **private** **int** average;  } |

객체지향의 목적: 효율적인 코딩을 위함. VO클래스를 이용하여 다양한 변수들을 하나의 이름으로 관리하기가 쉬워지고, 내가 만든 클래스의 이름이 변수명(변수타입)이 됨

**Ⅲ. 객체지향 프로그램의 특징**

1. **캡슐화(Encapsulation)**
   1. **캡슐화란?**

여러 속성과 여러 오퍼레이션을 함께 묶어 클래스로 취급하는 것과 클래스 외부를 외부에서 접근하지 못하도록 보호하는 것

* 1. **캡슐화방법**

|  |  |
| --- | --- |
| **< this 사용목적 >** | |
| 1. 자기의 생성자를 호출할 때 사용 | **public** A05\_Jumsu(String name, **int** kor, **int** eng, **int** math) {  **this**(name, kor);  **this**.eng = eng;  **this**.math = math;  } |
| 2. 멤버변수와 매개변수를 구분 | **void** onTotal(){  total = **this**.kor + eng + math;  } |

**상속(Inheritance)**

1. **상속이란?**

▶ 클래스의 속성과 오퍼레이션을 하위 클래스에 물려주거나, 상위클래스에서 물려받는 것

▶ 재사용으로 인해 코드가 줄어드는 장점이 있음. 하위 클래스에서 속성이나 오퍼레이션을 다시 정의하지 않고 상속받아서 사용함으로써 코드가 줄어든다. 그리고 좀 더 범용성있게 사용할 수 있다. 하위 클래스는 상위 클래스가 가지고 있는 모든 자료와 메소드를 물려받아 자유롭게 사용할 수 있지만, 또한 자신만의 자료와 메소드를 추가적으로 덧붙임으로써 새로운 형태의 클래스로 발전하게 된다.

**2-1. 상속 구현 예제**

|  |
| --- |
| **class** Person {  **int** age;  **public** String name;  **protected** **int** height;  **private** **int** weight;  **public** **void** setWeight(**int** weight){  **this**.weight= weight;  }  **public** **int** getWeight(){  **return** weight;  }  }  **class** Student **extends** Person {  **void** set(){  age = 30;  name = "홍길동";  height = 175;  setWeight (99);  }  **public** **static** **void** main(String[] args) {  Student s = **new** Student(); s.set();  System.*out*.println("나이 : "+s.age+" 이름 : "+s.name+" 키 : "+s.height+" 몸무게 : "+s.getWeight());  }  } |

1. **다형성**

|  |
| --- |
| ﻿**1. 다형성(Polymorphism)이란?**  ▶ 여러 클래스들이 동일한 이름의 오퍼레이션을 서비스하도록 하는 것  실제의 코드에서는 '하나의 클래스 내부에 같은 이름의 오퍼레이션을 여럿 정의하거나, 상위 클래스의 오퍼레이션을 하위 클래스에서 다시 정의함'으로써 구현한다.  예) 상위클래스인 Object클래스에 toString(), equals() 메소드가 존재하지만, 하위 클래스인 String과 Date 클래스에도 toString(), equals() 메소드가 존재함으로 볼 수 있다.  이렇게 상위 클래스에 있고 상속받았으나 하위 클래스에서 다시 정의하는 것을 메소드 오버라이딩(Method Overriding)이라고 하며, 메소드 오버라이딩이 다형성이다.﻿ ﻿그리고 동일한 이름의 오퍼레이션이 여러개 정의되어 있는데 단지, 매개변수의 타입에 따라 서로 구분될 수 있다. |
| 출처: http://searchstory.tistory.com/entry/객체지향-기본-개념 |

* 1. **다형성 구현 방법**

(출처: <http://chs02.tistory.com/10>)

1. **메소드 오버로딩 (Method Overloading)**
2. 오버로딩이란?
   * 1. 클래스 내에 있는 변수나, 메소드는 서로간의 기능에 따라 구별되는 이름이 있어야 하지만 메소드의 매개변수의 갯수와 타입이 다르다면 같은 이름으로 메소드를 여러개 지정할 수 있으며 이를 메소드 오버로딩이라고 합니다.
3. 장점
   * 1. 메소드 이름의 낭비 방지
     2. 같은 메소드에 여러 종류의 매개변수를 받을 수 있음

|  |
| --- |
| **오버로딩 예제** |
| **class** Product {  **void** show(**int** price) {  System.*out*.println("가격:" + price);  }  **void** show(String name) {  System.*out*.println("제품:" + name);  }  **void** show(String name, **int** price) {  System.*out*.println("제품:" + name);  System.*out*.println("가격:" + price);  }  }  **public** **class** ProductStatus {  **public** **static** **void** main(String[] args) {  Product p = **new** Product();  p.show("노트북");  p.show(1000);  p.show("컴퓨터", 500);  }  } |
| -결과-    (void show 라는 메소드가 여러개 지정 됐음에도 매개변수가 다르기 때문에 오류가 나지 않는다. – 오버로딩) |

1. **메소드 오버라이딩 (Method Overriding)**
2. 오버로딩이란?
3. 부모클래스로 부터 상속받은 메소드를 자식클래스에서 재정의하여 사용하는 것
4. 예를 들면 아버지로 부터 물려받은 연식이 오래된 차를 내가 원하는대로 튜닝해서 새로 타고 다니는 것과 같은 맥락입니다.
5. 장점
   * 1. 코드의 재사용성이 향상됨

|  |
| --- |
| **오버라이딩 예제** |
| **class** Admin {  **void** printSum() {  System.*out*.println("가격 :1000원");  }  }  **class** Product **extends** Admin {  **void** printSum() {  System.*out*.println("가격 :2000원");  }  }  **public** **class** ProductStatus {  **public** **static** **void** main(String[] args) {  Product p = **new** Product();  p.printSum();  }  }  -결과- |
| **class** Admin {  **void** printSum() {  System.*out*.println("가격 :1000원");  }  }  **class** Product **extends** Admin {  **void** printSum(**boolean** stats) {  **if** (stats) {  System.*out*.println("가격 :2000원");  } **else** {  **super**.printSum();  }  }  }  **public** **class** ProductStatus {  **public** **static** **void** main(String[] args) {  Product p = **new** Product();  **boolean** stats = **false**;  p.printSum(stats);  }  } |