**<객체지향 문법정리>**

**Ⅰ. 객체지향을 하는 이유**

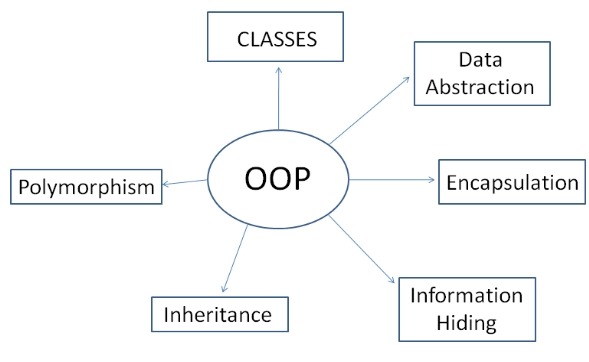
이름, 국어점수, 수학점수, 영어점수, 총점, 평균이 적힌 30명 인원의 성적리스트를 만든다고 생각해보자.

그냥 변수를 직접 선언하는 코딩을 한다면 한 사람당 선언해야 하는 변수의 개수는 6개이므로, 총 120개의 변수를 선언해야 하며, 필요한 변수명도 180개나 된다. 때문에 이 방법은 개발자가 변수명을 일일이 기억하기도 어렵고, 변수를 선언하는데 불필요한 시간과 노력이 투자되므로 너무 비효율적이다. 따라서 좀 더 효율적인 코딩을 하고자 변수들을 하나의 배열로, 배열을 이차원 배열로, 이차원 배열을 ArrayList로 넣어보았는데, ArrayList를 사용하다 보니 이름과 점수들을 함께 뽑아서 쓰려면 이름의 위치, 즉 index를 알아야 하기 때문에 불편!!! 그래서 객체가 생겨났다.

즉, 효율적인 코딩을 하고자 객체의 개념이 생겨났다. VO클래스를 이용하여 다양한 변수들을 하나의 이름으로 관리하기가 쉬워지고, 내가 만든 클래스의 이름이 변수명(변수타입)이 된다.

|  |  |
| --- | --- |
| **변수선언방법** | **예제 코드** |
| 일일이 변수 선언 | String studentName = "조현아";  **int** studentKor = 100;  **int** studentEng = 95;  **int** studentMath = 85;  **int** studentTotal = studentKor + studentEng + studentMath;  **int** studentAvg = studentTotal / 3; |
| 배열을 이용 | String[] name = {"조현아"};  **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 = {"조현아"};  **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("조현아");  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;  } |

**Ⅱ. 객체지향 프로그램이란?**



**객체 지향 프로그래밍(OOP: Object Oriented Programming)**은 부품에 해당하는 객체들을 먼저 만들고, 이것들을 하나씩 조립해서 완성된 프로그램을 만드는 기법이다. 즉, 캡슐화, 다형성, 상속 등을 이용하여 코드 재사용을 증가시키고, 유지보수를 감소시키는 장점을 얻기 위해서 객체들을 연결 시켜 프로그래밍 하는 것이다. (출처: 이것이 자바다 책 / 웹사이트 <http://vandbt.tistory.com/10>)

**Ⅲ. 객체지향 프로그램의 특징**

1. **캡슐화(Encapsulation)**

|  |
| --- |
| => 인터넷검색: **캡슐화 (Encapsulation)**  **1. 캡슐화란?**  ▶ 여러 속성과 여러 오퍼레이션을 함께 묶어 클래스로 취급하는 것과 클래스 내부를 외부에서 접근하지 못하도록 보호하는 것  **2. 캡슐화는 왜 해야 할까?**  ▶ 첫째, 묶음으로 인해 프로그램을 바라보는 단위가 커진다. 이전의 프로그래밍 언어인 C언어는 프로그램을 함수 단위로 구조화할 수 있으나, 프로그램 소스가 커지면 이해하기 어렵고 관리가 힘들어 질 수 있었다. 그러나 객체지향 프로그램에서는 프로그램 소스를 클래스 단위로 바라보게 됨으로써 좀더 복잡하고 커다란 소스코드도 쉽게 이해하게 되었다. 왜냐하면 클래스 내부에 여러 함수를 내포할 수 있기 때문에 프로그램 소스 코드를 바라보는 단위가 커졌으며, 그로 인해 프로그램 관리가 좀 더 수월해진 것이다.  ▶ 두번째, 내부를 숨김으로써 내부를 좀더 자유롭게 변경할 수 있게 되었다. 이전의 함수 중심적인 구조적 프로그래밍 언어에서는 프로그램 내부에서 데이터가 어디서 어떻게 변경되는지 파악하기 어려웠고, 그로 인해 유지 보수가 힘들었기 때문에 자료를 중심으로 함수가 종속되는 구조가 되기도 하였다. 객체 지향에서는 클래스 내부의 데이터를 외부에서 참조하지 못하도록 차단하여 이러한 폐단을 없앨 수 있다. 이렇게 내부의 데이터나 함수를 외부에서 참조하지 못하도록 차단하는 개념을 정보 은닉(Information Hiding)이라고 하며 이것이 바로 캡슐화라는 개념이다. |
| 출처: http://searchstory.tistory.com/entry/객체지향-기본-개념 |

* 1. **캡슐화방법**

만약 다른 클래스의 메인함수에서 내가 만든 VO클래스 타입의 객체를 선언하고 싶다면, 생성자라는 것이 필요하다. 즉, 인스턴스 객체는 생성자 메소드를 통해서 태어나는데, 기본 생성자는 jvm이 자동적으로 만들어준다. 하지만, 개발자가 생성자 메소드를 하나라도 만든다면 jvm이 기본 생성자를 만들지 않기 때문에, 다른 생성자와 기본생성자 둘 다 필요 시, 개발자가 두 생성자 모두 직접 정의를 해주어야 한다. 또한, 생성자는 오버로딩이 가능하기 때문에, 필요에 의해 개발자가 개별적으로 정의를 하면 된다. (단, 생성자 메소드명은 이름대신 this()를 사용한다.)

또한, 사용자가 해야 할 일을 묶어서 VO클래스에서 처리할 수 있는데, 이 것을 메소드라 부른다. 수업시간 예제에서는 총점과 평균 점수를 메소드로 구현해보았다. 메소드를 만들 때, 클래스 안쪽의 멤버변수가 사용된다면, 이 때 변수 앞에는 this를 붙여도 되고 생략해도 된다. 그리고 메인함수에서 호출할 때, 변수와 똑같이 호출가능하나, ( )를 뒤에 붙여줘야한다.

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

마지막으로, VO클래스만 만들어서 Main함수에서 변수선언을 하여 사용한다면, 사용자가 마음대로 변수들을 바꿔버릴 수가 있기 때문에 DAO클래스를 만들기 시작했다. DAO클래스는 VO변수를 private로 만들고, 생성자를 이용하여 VO에 접근한다. 이로써 사용자는 DAO에만 접근 할 수 있고, DAO가 VO에 접근할 수 있게 되었다. (단, getter/setter 메소드 필요 – 이 것이 바로 캡슐화임)

1. **상속(Inheritance)**

|  |  |  |
| --- | --- | --- |
| => 인터넷검색: **상속 (Inheritance)**  **1. 상속이란?**  ▶ 클래스의 속성과 오퍼레이션을 하위 클래스에 물려주거나, 상위클래스에서 물려받는 것  예) 자바 API에서 최상위 클래스인 Object 클래스 내부에는 다음과 같은 메소드가 존재한다.   |  | | --- | | Protected void finalize() / Class getClass() / int hashCaode() / void notify() /  void notifyAll() / String toString() / void wait() / void wait(long timeout) /  void wait(long timeout, int nanos) |   자바의 모든 클래스는 Object 클래스의 하위 클래스가 되므로 어떤 클래스에서든 Object 클래스에 정의된 위와 같은 메소드를 사용할 수 있다. 이와 같이 상속이란 하위에게 사용할 수 있도록 물려주는 것이다.  **2 상속의 장점은?**  ▶ 재사용으로 인해 코드가 줄어든다. 하위 클래스에서 속성이나 오퍼레이션을 다시 정의하지 않고 상속받아서 사용함으로써 코드가 줄어든다. 그리고 좀 더 범용성있게 사용할 수 있다. 참고로 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.  **3. 상위 클래스와 하위 클래스의 생성자**  ▶ 생성자라는 것은 일종의 오퍼레이션이며, 클래스가 객체화될 때 실행된다. 그런데 하위 클래스의 생성자를 호출할 경우, 자동적으로 상위의 생성자가 호출됨을 반드시 기억해야 한다. 하위의 생성자가 호출될 때 묵시적으로 상위의 생성자를 자동으로 호출하는 것이다. | |
| http://cfile24.uf.tistory.com/image/2403CB36536EF3E422C06E  3개에 동일한 메소드가 있을시 상속을 사용하여  오른쪽 그림과 같이 하나의 부모클래스에 동일한 메소드를 넣어 코드가 중복되는것을 막을수 있음 | http://cfile5.uf.tistory.com/image/23741E4B536EF4811B3C2D |
| 출처: http://searchstory.tistory.com/entry/객체지향-기본-개념 | |

**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 {  // Student 클래스가 Person 클래스를 상속받음  **void** set(){  age = 30;  name = "홍길동";  height = 175;  setWeight (99);  }  **public** **static** **void** main(String[] args) {  Student s = **new** Student(); // Student 객체 생성  s.set(); //Person클래스를 상속 받았기 때문에 Person클래스에 있는 매개변수를 사용할 수 있음.  System.*out*.println("나이 : "+s.age+"\n이름 : "+s.name+"\n키 : "+s.height+"\n몸무게 : "+s.getWeight());  }  } |
| 결과:  http://cfile9.uf.tistory.com/image/2452293D536EF75D01D92A  Person 클래스와 Student 클래스가 있는데 Student 클래스는 Person 클래스를 상속 받았기 때문에 Person클래스는 Student클래스의 상위클래스가 되고 Student 클래스는 Person클래스의 하위 클래스가 된다. Student 클래스는 Person클래스가 가지고 있는 메소드와 매개변수를 자신의 것 처럼 사용할 수 있다. |

1. **다형성**

|  |
| --- |
| => 인터넷검색: **다형성(Polymorphism)**  ﻿**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**

다형성을 구현하는 방법에는 많은 방법들이 있지만, 이 곳에서는 오버로딩과 오버라이딩에 대해서만 알아보고자 한다. (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
     2. 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

|  |
| --- |
| **메소드 오버로딩 코드 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

|  |
| --- |
| **메소드 오버라이딩 코드 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);  }  }※ super : 현재 클래스에서 부모 클래스의 요소를 호출  -결과- |
| Product 객체 생성 후 boolean stats 변수에 false 값을 매개변수로 하여 printSum을 호출하고 이를 받은 Product 클래스에서 stats 값이 true면 오버라이딩 된 메소드를  실행하고 값이 false면 super를 이용하여 상속받은 클래스의 printSum을 실행함 |

1. **메소드 오버로딩과 메소드 오버라이딩 요약 비교**

|  |  |  |
| --- | --- | --- |
| 구분 | 오버로딩 | 오버라이딩 |
| 메소드 이름 | 동일 | 동일 |
| 매겨변수, 타입 | 다름 | 동일 |
| 리턴 타입 | 상관없음 | 동일 |

**Ⅲ. 인터페이스**

1. **인터페이스란?**
2. 하위 클래스들의 공통적인 기능이나 요소를 수집하여 정의해 놓은 것으로 하위클래스에서의 기능 구현을 보다 편리하게 하기 위한 용도로 사용

2) 추상클래스의 일종이나 추상클래스와 달리 인터페이스 내에 일반 메소드를 정의할 수 없고 구현부가 없는 추상메소드만 가능

3) 변수 선언시 Public static final, 메소드 선언시 public abstract 의 제어자가 자동으로 선언됨(접근제어자의 default와 같은 개념)

4) 상속과 달리 상수 그리고 메소드의 선언부가 구현되어 있지 않아 한개의 클래스를 인터페이스를 전환하여 다중으로 상속이 가능

5) 선언한 메소드를 상속 클래스가 오버라이딩 하여 사용

**2. 추상클래스와의 차이점**

|  |  |  |
| --- | --- | --- |
|  | **인터페이스** | **추상클래스** |
| 다중상속 | O | X |
| 일반메소드 선언 | X | O |
| 일반변수 선언 | X(상수만 가능) | O |
| 사용 방법 | implements | extends |

**3. 장점**

  1) 공통적인 요소를 미리 정의함으로써 보다 빠른 프로그래밍과 표준화가 가능

2) 다중상속이 되지 않는 Java에서 인터페이스를 통해 구현 가능

**4. 인터페이스 구현 방법**

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  /\* 각클래스별로 객체생성 \*/  Television tv = **new** Television();  Refrigerator r = **new** Refrigerator();  /\* 상속받은 메소드 호출 \*/  tv.turnOn();  r.turnOn();  tv.turnOff();  r.turnOff();  }  } |
| **interface** RemoteControl { // 인터페이스 - 단지 부모 역할만 할 껍데기 클래스, 다형성을 위해 자식들이 호출되어 사용될 껍데기 메소드만 정의한다.  **public** **void** turnOff(); // 메소드선언  **public** **void** turnOn(); // 메소드선언  } |
| **class** Television **implements** RemoteControl {  /\* 상속받은 메소드 재정의 \*/  **public** **void** turnOff() {System.*out*.println("TV를 끕니다.");}  **public** **void** turnOn() {System.*out*.println("TV를 켭니다.");}  } |
| **class** Refrigerator **implements** RemoteControl {  /\* 상속받은 메소드 재정의 \*/  **public** **void** turnOff() {System.*out*.println("냉장고를 끕니다.");}  **public** **void** turnOn() {System.*out*.println("냉장고를 켭니다.");}  } |
| 결과: |

출처: <http://chs02.tistory.com/13> / <http://enter.tistory.com/122>

**Ⅳ. Factory의 사용 – Spring과 관련**

|  |
| --- |
| **public** **interface** Radio {  **public** **void** powerOn();  **public** **void** powerOff();  **public** **void** volumnUp();  **public** **void** volumnDown();  } |
| **public** **class** SamsungRadio **implements** Radio {  **public** **void** powerOn(){  System.*out*.println("SamsungRadio => 전원 켜기");  }  **public** **void** powerOff(){  System.*out*.println("SamsungRadio => 전원 끄기");  }  **public** **void** volumnUp(){  System.*out*.println("SamsungRadio => 소리 올리기");  }  **public** **void** volumnDown(){  System.*out*.println("SamsungRadio => 소리 내리기");  }  } |
| **public** **class** LGRadio **implements** Radio {  **public** **void** powerOn() {  System.*out*.println("LGRadio => 전원 켜기");  }  **public** **void** powerOff() {  System.*out*.println("LGRadio => 전원 끄기");  }  **public** **void** volumnUp() {  System.*out*.println("LGRadio => 소리 올리기");  }  **public** **void** volumnDown() {  System.*out*.println("LGRadio => 소리 내리기");  }  } |
| **public** **class** RadioFactory {  // 어떤 클래스를 사용할 지 정해서 그 클래스를 생성해서 리턴하는 목적으로 만드는 클래스  //Object Type은 모든 객체의 부모 타입  //대신, 사용할 때 down casting을 해야 함  **public** Object getBean(String radio){  **if**(tv.equals("Samsung")){  **return** **new** SamsungRadio();  }**else** **if**(tv.equals("LG")){  **return** **new** LGRadio();  }  **return** **null**;  }  } |
| **public** **class** RadioUser {  **public** **static** **void** main(String[] args) {  //사용할 객체는 Factory로 부터 주입받아(Dependency injection) 사용 -> 의존성 주입  RadioFactory factory = **new** RadioFactory();  Radio radio = (Radio) factory.getBean(args[0]);  radio.powerOn();  radio.volumnUp();  radio.volumnDown();  radio.powerOff();  }  } |

**1. Factory 사용 방법**

1. 인터페이스를 만들고, 그 인터페이스를 구현(implements)하는 클래스들을 만든다.
2. 어떤 클래스를 사용할 지 정해서 그 클래스를 생성해서 리턴하는 목적으로 만드는 클래스인 Factory 클래스를 만든다. 리턴타입은 가장 상위클래스인 Object타입이다.
3. 사용할 객체는 Factory로부터 주입 받아 사용하기 때문에 사용자는 코드의 변환이 전혀 없이 사용할 수 있다. (factory를 가져다가 쓰는 부분에서는 new 객체()와 같은 식으로 변수를 선언할 필요가 없음.) 단, Object타입으로 return받았기 때문에 down casting이 필요하다.

**2. Factory와 interface는 왜 사용할까?**

1. 유지보수가 쉬워진다. 각 클래스에서 따로따로 변수를 선언하여 사용하면, 사용하는 변수의 타입을 바꾸고자 할 때 일일이 다 번거롭게 바꿔줘야 한다. 또한, interface를 사용하지 않으면 각 클래스 마다 사용하는 메소드명이 다 다르게 쓰일 수 있기 때문에, 메소드명도 번거롭게 바꿔줘야 한다. 하지만 Factory를 사용함으로써 Factory의 내용만 일부 변경해주면 되므로 유지보수하기 편리하다.
2. 사용자들은 코드에 아무런 변경 없이 그냥 사용하면 되기 때문에 무척 편리하며, 정확히 어떤 클래스의 인스턴스인지 신경쓰지 않고 구현할 수 있는 장점이 있다. (객체 타입이 굉장히 유연해 질 수 있음)

|  |
| --- |
| => 인터넷검색: **Factory Method**  ﻿**1. Factory란?**  ▶ 객체 생성을 처리하는 클래스  **2. Factory Method 사용의 의도**  ▶ 객체를 생성하기 위해 인터페이스를 정의하지만, 어떤 클래스의 객체를 생성할 지에 대해서는 하위 클래스에서 결정 ==> 클래스 상속을 이용함  **3. Factory Method 활용 예**  - 생성할 객체 타입을 예측할 수 없을 때 ==> 부모 클래스 타입을 이용  - 생성할 객체의 명세를 하위 클래스에서 정의하고자 하는 경우  - 객체 생성의 책임을 하위 클래스에 위임시키고 어느 하위 클래스가 위임했는지에 대한 정보를 은닉하고자 하는 경우 |
| 출처: <http://dextto.tistory.com/140> & <http://wiki.gurubee.net/pages/viewpage.action?pageId=1507401> |