[개요]

생코 강의를 들으면서 모든 내용을 워드에 정리하려고 하다보니 너무도 많은 시간이 걸리고, 수업에도 집중하기 힘들었다. 그리고 그렇게 고생해서 적은 내용들을 나중에 한 달, 두 달 뒤에 찾아보면 다시 이해하는데 또 시간이 걸릴뿐더러 텍스트 투성이의 공부 내용을 줄줄이 읽기가 참 힘들었다. 분명 읽기 쉽도록 구어체로, 나름 농담도 섞어가면서 적었음에도 불구하고 말이다.

이러한 문제를 인지하고 앞으로 수업을 들을 때에는 메모보다는 강의내용에 집중하고, 정말 강의에서 중요하다, 잊고 싶지 않다 싶은 내용들만 필기하기로 했다.

[클래스와 인스턴스의 변수]

클래스는 변수와 메소드로 선언됨은 알고 있을 것이다.

인스턴스는 그러한 클래스로부터 만들어진 클래스의 이른바 복제본이라고 말할 수 있겠다.

class MyClass {

static int 객체에\_관계없이\_고정값 = 123;

int A;

int B;

void setAB(int A, int B) {

this.A = A;

this.B = B;

}

void sum() {

System.out.println(this.A + this.B);

}

}

public class Test4 {

public static void main(String[] args) {

MyClass mc1 = new MyClass();

mc1.setAB(10, 20); // 30 출력

mc1.sum();

System.out.println(mc1.객체에\_관계없이\_고정값); // 123 출력

MyClass.객체에\_관계없이\_고정값 = 100;

System.out.println(mc1.객체에\_관계없이\_고정값); // 100 출력

mc1.객체에\_관계없이\_고정값 = 321;

System.out.println(mc1.객체에\_관계없이\_고정값); // 321 출력

System.out.println(MyClass.객체에\_관계없이\_고정값); // 321 출력

}

}

위 코드를 보면 클래스를 통해서 인스턴스를 생성하고 있다.

클래스에선 인스턴스 변수 A, B와 클래스 변수 “객체에\_관계없이\_고정값”(이하 객관고라 부름) 이라는 변수를 123으로 가지고 있다.

이 값은 클래스 자체가 고유하게 가지고 있는 변수로, 해당 클래스를 통해 어떠한 인스턴스를 만들건 똑같이 갖게 된다. 사실 갖는다기보다는 각 인스턴스들이 클래스 변수인 객관고를 똑같이 참조하게 된다.

즉 인스턴스를 아무리 만들어도 객관고가 차지하는 메모리는 객관고 1개 만큼만 차지한다는 말이다.

무슨 말이냐?

원래 인스턴스를 만들면 인스턴스 만큼 메모리를 차지하게 될것이다.

생각하면 당연한 얘기다. 클래스 A가 있고, A타입의 인스턴스 10개를 만든다 치면 인스턴스 변수들은 10개만큼 메모리를 차지하게 된다. 이 말은 인스턴스 변수들이 메모리를 뭐 2byte 차지한다 치면 인스턴스를 10개 만들었으니 20byte를 차지할 것이라는 말이다.

그런데 클래스 변수인 static이면 어떨까? 클래스 변수는 인스턴스에 상관없이 클래스가 고유하게 가지고 있는 값이므로, 모든 인스턴스가 동일하게 참조한다.

이 말은 클래스 변수가 2byte를 차지한다고 한다면 인스턴스가 1억개가 되어도 클래스 변수는 2byte를 차지할 것이라는 말이다.(대신 인스턴스 변수는 1억개만큼 차지할 것이다)

마치 벤츠가 다양한 자동차 모델을 만들지만 보닛위에 달린 엠블렘은 똑같은 것처럼, 인스턴스가 아무리 많아도 클래스 변수는 동일하게 참조된다.

따라서 클래스 변수를 바꾸면 해당 클래스 타입인 모든 인스턴스의 클래스 변수는 바뀐 클래스 변수를 참조하게 된다.

코드를 보면 인스턴스가 담긴 변수인 mc1에서도 객관고를 호출할 수 있고, MyClass라는 클래스 자체에서도 객관고를 호출할 수 있다.

[클래스 메소드와 인스턴스 메소드에서 인스턴스 멤버와 클래스 멤버로의 접근 가능 여부]

사실 아주 당연한 얘기다.

인스턴스의 메소드는 당연히 클래스의 멤버, 즉 클래스 변수나 클래스 메소드에 접근이 가능하다.

반대로 클래스 메소드는 인스턴스의 멤버에 접근할 수가 없다. 이미 손을 떠나버렸기 때문이다.

[Non-Static Field, Static Field, 인스턴스 변수, 클래스 변수]

구글링 하거나 다른 사람들이 말하는 걸 보면 알 수 있는 내용이다.

인스턴스 변수는 Non-Static Field 라고 부른다.

클래스 변수는 Static Field 라고 부른다.

왜 그러는지는 이미 알고 있을 수 있겠지만, 당연히 클래스 변수는 static으로 선언되고, (public static void 마냥) 인스턴스 변수는 그렇지 않기 때문이다.

[정적인 유효범위와 동적인 유효범위]

이고잉은 중요하지 않다고 했지만 일단 내가 몰랐던 것이 등장했으므로 메모한다.

scope에 대해서 이미 알고 있을 것이다.

결론적으로 말하자면 자바는 정적인 유효범위 방식을 채택하고 있는 프로그래밍 언어이고, 이것만 숙지해도 문제는 없다.

그런데 정적인 유효범위의 반댓말인 동적인 유효범위라는 말도 알아가고자 이런 메모를 적는다.

자바에서 스코프란 직관적으로 정적이다.

이를 테면 메소드 내의 변수를 참조하고, 없다면 클래스 내의 변수를 참조한다.(같은 레벨에 있는 다른 메소드를 참조하지 않는다!) 없으면? 에러를 뿜는다.

이런 것을 정적이라고 한다.

동적인 유효범위는 어떤 스코프에서 호출했다면 해당 스코프도 참조한다.

그림으로 설명해볼까.

화이트보드이(가) 표시된 사진

자동 생성된 설명

그림판이라 그림의 질이 낮지만 한번 설명해보자.

왼쪽은 자바의 스코프, 즉 정적인 유효범위 = 정적인 스코프를 말한다.

여기서 빨간색 메소드에서 변수 a를 호출한다면 jvm은 변수 a를 빨간색 메소드에서 우선 찾는다.

없다면? 보라색 영역에서 찾는다.

설령 다른 메소드에서 빨간색 메소드를 호출했다고 하더라도 변수 a를 빨간색 영역 > 보라색 영역 순으로 찾는다. 없으면? 에러고.

그런데 우측 동적인 스코프는 살짝 다르다.

만약 파란색 메소드가 실행되었다고 치고 파란색 메소드가 변수 a를 찾는다면 파란색 메소드에서 변수 a를 찾는 시도를 하고, 없다면 보라색 영역에서 변수 a를 찾을 것이다.

그런데 만약 파란색 메소드가 빨간색 메소드를 호출했다면 어떻게 할까.

빨간색 메소드에서도 역시 변수 a를 찾는 코드가 있다고 한다면, 빨간색 메소드가 실행될 때 변수 a는 어디서 찾을까? 가 문제다.

정적인 스코프에서는 당연히 빨간색 메소드에서 변수 a를 찾고, 없다면 보라색 영역을 찾는다.

그런데 동적인 스코프에서는

빨간색 메소드에서 변수 a를 찾고, 그 다음 빨간색 메소드를 호출한 파란색 메소드에서 변수 a를 찾고, 그 다음 클래스(보라색 영역)에서 찾고… 그래도 없으면 파란색 메소드를 호출한 곳에서 찾고… 이런 식이다.

근대적인 프로그래밍 언어는 동적인 유효범위를 가지고 있다고 한다.

[생성자 정의하기]

1-1. 생성자 정의 규칙  
1) 생성자 헤더에 리턴 타입을 넣지 않는다. (void 명시하지 않음)  
2) 생성자 이름과 클래스 명이 같아야 한다.  
3) 생성자를 아무것도 정의하지 않을 경우에는 기본 생성자가 자동으로 생성되며,   
     만약 하나의 생성자를 직접 정의한 경우에는 기본 생성자가 자동으로 생성되지 않는다. 

구글링하니 이런 글이 나온다.

class MyClass1 {

int left, right;

public MyClass1(int left, int right) {

this.left = left;

this.right = right;

}

}   
위와 같이 선언한다. 리턴 타입이 없다는 것을 알아두자.

[우리는 클래스를 호출하지 않고 있었다?]

MyClass1 mc1 = new MyClass1(10, 20);

점점 더 복잡해 지는 것인가 점점 더 명확해지는 것인가

우리는 지금까지 이렇게 인스턴스를 생성해왔다. 위 코드는 MyClass1 타입의 객체를 만드는 코드를 보여준다.

그런데 잘 보면 MyClass1에 괄호 열고 인자를 주고 있다. 이 말은 무엇인가?

관례적으로 클래스를 호출할 때 괄호 안에 인자를 주면 생성자가 실행될 때 그것을 파라미터로 하여 실행한다. 라고 이해하면 장땡이지만, 사실 저것은 메소드다.

무슨 말이냐? 우리는 관례적으로 new 클래스명() 하면 인스턴스가 생성되는 줄 알고 있었지만 사실은

new 생성자명() 하여 인스턴스를 생성하고 있는 것이다.

그러면 생성자가 없는 클래스에 대해선 무슨 말을 할 것인가?

생성자가 없는 클래스는 사실 내부적으로, 자동으로 클래스와 동일한 이름의 생성자를 만들어서 호출한다. 물론 인자가 없기 때문에 생성자인지 아닌지 인지하긴 힘들지만.

[상속할 때 생성자를 만들어줘도 되는 경우와 만들어줘야 하는 경우]

상속은 extends 로 하는 것을 알고 있을 것이다.

그런데 부모 클래스가 생성자가 명시되어 있다면 자식 클래스도 생성자를 명시해야 한다(보통 super로 생성자를 부른다) 왜냐하면 객체를 만들 때 부모객체부터 먼저 만들어야 하기 때문이다. 근데 부모객체는? 생성자로만든다.

따라서 자식객체가 생성자를 통해 만들어지기 전에 부모 클래스의 생성자가 먼저 호출되어야 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

상속 이해를 위해서 대충 예제를 만들어 봤다.

아무튼 저렇게 BenzSports는 생성자가 필요하다.(생성자를 꼭 써줘야 한다)

텍스트이(가) 표시된 사진

자동 생성된 설명

반대로 BMW는 생성자가 없이 만들었는데, 똑 같은 구조로 상속한 BMWSports 에서는 생성자가 필요없었다.

[왜 부모 클래스에 생성자가 있으면 자식 클래스도 생성자를 재선언 해줘야 하는가?]

가장 중요한 것

1. 기본 생성자는 파라미터가 없는 생성자를 말한다.
2. 만약 클래스에서 아무런 생성자가 정의되지 않으면 자바는 기본적으로 파라미터도 없고, 내용도 없는 기본 생성자를 임의로 만들어서 인스턴스를 만든다.
3. 만약 기본 생성자든 커스텀 생성자(파라미터가 있는 생성자)든 사용자가 어떻게든 키보드로 코드를 쳐서 정의를 했다면 자바는 2의 행동을 해주지 않는다.
4. 부모 생성자는 자식 생성자가 호출되기 전에 먼저 호출되어야 한다.

이것에 대해서 말하기 전에 우선 정의를 먼저 하고 가자.

기본 생성자와 사용자 정의 생성자(= 커스텀 생성자)에 대해서 말이다.

우리가 클래스를 만들고 따로 생성자를 만들어주지 않고 그 클래스의 인스턴스를 만들면 자바가 자동으로 생성자를 만들고 호출한다고 했다. 그리고 이런 기본 생성자는 당연하게도 매개변수도 존재하지 않고, 아무런 내용도 없다.

기본 생성자는 위와 같은 생성자 혹은 “매개변수가 없는 생성자” 라고 이해하자.

그런데 우리가 커스텀 생성자를 만들면 자바는 기본 생성자를 만들어주지 않는다. 왜냐하면 사용자 정의 생성자가 있기 때문이다.

그런데 보통 커스텀 생성자는 매개변수를 갖고 있다. 당연하게도 매개변수를 갖고 있는 생성자를 호출할 때(=인스턴스를 만들 때)에는 인스턴스를 생성할 때 인자를 넣어서 생성자의 매개변수를 채워주어야 한다.

그러면 이러한 상황에서 상속을 했다고 가정하자.

자바에서는 상속한 자식 클래스가 인스턴스화될 때, 부모 클래스의 생성자를 불러온다. 이것은 룰이다.

그런데 만약 부모 클래스에 커스텀 생성자가 없다면 당연히 자동으로 기본 생성자를 불러오는 행위를 한다.

그래서 부모 클래스에 생성자를 따로 정의해 주지 않았다면 자식 클래스도 생성자를 재정의해줄 필요가 없다.

그런데 부모 클래스에 커스텀 생성자만 있다면(매개변수를 받는 생성자만 있다면) 자식 클래스의 인스턴스를 만들 때 부모 클래스 생성자를 실행하려는데 매개변수를 줘야하므로(왜냐면 부모 클래스 생성자가 매개변수를 받으니까) 자식 클래스도 매개변수를 받는 생성자를 만들어줘야 한다는 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

참고용. 부모에 생성자가 있지만 매개변수가 없는 기본 생성자나 다름 없으므로 자식 클래스에서 생성자를 따로 정의 안해줘도 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명반대로 위와 같은 경우는? 사실 이해한다면 되게 쉬운건데..

그니까 요약하자면

1. 인스턴스를 만들 때 매개변수를 주고 만들 수도 있고 안 주고 만들 수도 있다.
2. 만약 커스텀 생성자만 있고 기본 생성자가 없다면 무조건 매개변수를 줘야한다.(왜냐면 커스텀 생성자만 있으니까. 자바는 이런 상황에서 기본 생성자를 자동으로 만들어주지 않는다.)
3. 부모가 커스텀 생성자만 있고, 자식은 기본 생성자만 있거나 아무런 생성자가 없다면(그니까 자바가 자동으로 기본 생성자를 만들어줘야 하는 상황이라면) 당연히 에러가 난다. 왜냐하면 자식 클래스의 인스턴스를 생성할 때 어쩔 수 없이 매개변수를 주지 못하고 생성할 수밖에 없는데(자식은 기본 생성자만 있으니까. 당연한 얘기다. 매개변수를 받을 커스텀 생성자가 없으니까) 부모 클래스의 생성자를 호출해야 한다는 자바의 룰에 의해서 부모의 생성자를 호출해야 하는데 봤더니 부모는 커스텀 생성자밖에 없다(매개변수가 꼭 필요한 상태)면 에러를 뿜을 수밖에 없다는 것이다.

즉 1. 둘 다 생성자를 아예 안쓰거나

2. 기본 생성자(매개변수가 없는 생성자)만 자식, 부모에 아무렇게나 쓰거나(있어도 되고 없어도 되고)

3. 커스텀 생성자를 쓸 생각이면 두 클래스 모두에 매개변수를 전달할 수 있게 하거나. 이런거다.

텍스트이(가) 표시된 사진

자동 생성된 설명이런 것도 가능하다.

자식을 생성할 땐 인자 없이 생성했지만, 인자 없는 기본 생성자에서는 부모 생성자에 인자를 주면서 호출하고 있다.(ASDF를 super에 주고 있음)

BMWSports bms = new BMWSports(); 의 결과는?

BMW 100 Online. (부모 생성자를 먼저 호출함)

BMW Sports Online.

이 될 것이다.

[왜 overwriting이 아니고overriding인가?]

영미권 사전을 뒤져봐야 하지만, stack overflow에 의하면

overwriting은 원래 있던 것을 지우고(파괴하고) 다른 것으로 대체하는 것인데 반해,

overriding은 원래 있던 것 위에 무엇인가 틀을 올리는 개념이라고 이해하면 될 듯하다.

마치 금니 씌우듯이 말이다.

중요한 것은 메소드 오버라이딩이 메소드 오버라이’팅’이 아닌 이유는 기존의 메소드를 해하지 않고 메소드를 재정의 하는 것이기 때문이다.

[오버라이딩 주의점]

리턴 타입도 똑같아야 한다.

매개 변수의 타입, 매개 변수의 개수, 매개 변수의 순서조차 똑같아야 한다.

이름도 당연히 똑같아야 한다.

그 후 메소드의 내용이 다르면 그것을 오버라이딩이라 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 사진에 잘 설명해놨다.

단순히 더 하위 스코프(하위 스코프이자 더 상세한 스코프이므로 더 우선순위가 높은 스코프)에 메소드를 또 하나 추가로 정의하는 것과 오버라이딩은 구분되어야 한다.

[메소드의 서명(signature)]

메소드의 주민등록증을 뜻한다. 즉 어떤 메소드를 다른 메소드와 구분지을 수 있게 하는 요소를 말한다.

그 요소로는 위 [오버라이딩 주의점] 에서 다룬 바와 같이

1. 메소드의 이름
2. 메소드의 매개변수의 개수
3. 메소드의 매개변수의 데이터 타입
4. 메소드의 매개변수의 입력 받는 순서
5. 메소드의 리턴 데이터 타입

위 5종류를 말한다.

[메소드 오버로딩method overloading]

상기했던 [오버라이딩 주의점] 항목에서 코드를 보면 자식 클래스의 method1은 부모 클래스의 method1과 이름은 같아도 리턴 타입, 매개변수 개수가 달랐다.

즉 둘이 아예 관계가 없이 이름만 같은 서로 다른 메소드라는 것이다.

이렇게 이름이 같지만 다른 리턴 타입, 다른 매개변수로 정의된 메소드를 만드는 행위를 메소드 오버로딩이라 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드를 보면 method1이 3번이나 정의되어 있다.

이를 모두 메소드 오버로딩이라 한다.

이름은 같은데 매개변수 관련된 메소드의 서명만 다르다.

즉 리턴 값과 매개변수의 데이터 타입, 매개변수의 개수, 매개변수의 입력 순서만 다르고 새로 정의한다면 그것은 메소드 오버로딩이라 한다.

[메소드 오버로딩 주의점]

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드를 보자. 아직 실행하기 전임에도 불구하고 이클립스에서 벌써 에러라고 밑줄을 긋고 있다.

왜 에러가 날까? 리턴 타입을 바꿔준 “메소드 오버로딩” 이라고 생각하는데 말이다.

이유는 바로 이건 메소드 오버로딩이 아니라 그냥 오류이기 때문이다.

상식적으로 메소드 오버로딩이 등장한 이유, 유래를 정확히는 모르지만 상상만이라도 해본다면 어떨까.

우리는 똑같거나 비슷한 메소드를 단지 매개변수에 값이 한 두개 바뀐다는 이유 만으로 완전히 새로운 메소드를 정의 해야하는 귀찮은 일을 기존의 메소드를 살짝 수정해서 정의함(메소드 오버로딩)으로써 해결했다.

즉 매개변수가 다른가? 가 굉장히 주요한 관전 포인트라는 것이다.

그런데 위 코드를 보면 매개변수는 전혀 다르지 않다. int A로 똑같다.

단지 리턴 타입만 다른데, 물론 리턴 타입이 다른 것도 메소드 오버로딩이다. 오버로딩인데 전제 조건이 틀렸다.

리턴 타입이 다르다고 메소드 오버로딩이 아니라 매개변수가 다를 경우(매개변수 타입이 다르거나, 매개변수 순서가 다르거나, 매개변수 타입이 다르거나)에 리턴 타입이 달라도 그것을 메소드 오버로딩이라 부를 수 있는 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드는 전혀 문제 없이 동작함을 보면, 메소드 오버로딩은 “매개변수”가 다름에 따라 정의됨을 알 수 있다.

사실 단순하게 생각하면 메소드 A1과 오버로딩된 메소드 A2가 있다고 하고 A1과 A2가 같은 이름이라고 할 때 똑같은 이름을 가진 메소드 두개를 구분해서 호출할 수 있는지 여부에 따라서 메소드 오버로딩 여부가 결정된다고 이해하면 된다. 구분해서 호출가능하면 오버로딩이고, 구분할 수 없다면 오류다.

[클래스 패스]

자바가 클래스를 찾아야 하는 영역을 말한다. 다른 문서에도 적어두긴 했지만.

[환경변수와 클래스 패스]

운영체제에 환경변수를 설정하는 것을 알고 있을 것이다.

환경변수에 클래스패스를 설정할 수 있다. 즉 자동적으로 클래스 패스를 설정할 수 있다는 말이다.

그런데 이러면 다른 시스템에 파일을 옮겼을 때 실행되지 않을 수 있다.

이런 문제를 해결하기 위해서 윈도우 같은 경우에는 bat 파일을 통해 스크립트를 넣어서 환경변수를 자동으로 세팅하기도 한다.

[패키지명]

패키지명은 중복되지 않는 것이 베스트이므로, 보통 도메인 주소(naver.somebody.else.something 이런 식)로 패키지 명을 정한다.

[수동 컴파일 – 패키지 경로가 따로 지정된 경우]

텍스트이(가) 표시된 사진

자동 생성된 설명

우리가 하고 싶은 일은 java 소스 파일은 A라는 경로에, class 파일은 B라는 경로에 두는 것이다.

위 명령어를 보면 javac로 컴파일 하는 것은 동일한데, 패키지 경로, 즉 .java파일의 경로를 명시하고 있다.

그런데 .java 파일의 경로만 명시해주었다고 해서 다른 경로에 class 파일을 만들어주진 않는다.

위 그림의 아래쪽 명령어를 보면 컴파일 할 파일의 경로를 명시하고(패키지 경로를 명시하고) -d 옵션을 주었는데 저것이 바로 핵심이다.

-d 하고 경로를 입력하면 해당 경로 위에 패키지 경로를 쫙 따서 class 파일을 만들어준다.

[import 했는데 같은 클래스가 있는 경우]

텍스트, 모니터, 전자기기, 화면이(가) 표시된 사진

자동 생성된 설명

오른쪽 패키지 구성도를 보면 example1, example2 패키지에 모두 Test 라는 이름의 클래스가 있다. 즉 example1과 example2를 import하면 클래스 커버리지에 Test라는 클래스가 2개 이상 있게 된다.

이 경우 Test 의 인스턴스를 만들면 어떻게 될까?

사진에서도 보이듯이 빨간 줄이 그어지면서 인스턴스가 생성되지 않는다.

내용은 Test가 Ambiguous하다 라는 것이다.

그러면 명시를 해줘야 한다. 클래스명을 바꾸던지.

텍스트, 모니터, 스크린샷, 닫기이(가) 표시된 사진

자동 생성된 설명

이렇게 줄줄이 명시를 해줘야한다.. ㅋㅋㅋ

[sysout]

System.out.println("Test");

지금까지 아무 생각없이 그런가보다 하고 써오긴 했는데, 이것들은 다 무엇일까?

System은 대문자로 시작하니까 클래스 같고. out은 그러면 메소드 내지는 클래스 변수여야 하는데 괄호가 없으니 클래스 변수인 듯 하고. println() 은 문법을 보니 메소드다.

그런데 변수에 메소드가 있다는 것은? 변수에 객체가 저장되어 있다는 것.

즉 클래스.변수(객체가 담긴변수).메소드(인자) 구조라고 볼 수 있겠다.

한 가지 궁금한 것은, System 이라는 것을 import 한 적도 없고, 만든 적도 없는데 사용할 수 있는 이유는 무엇일까?

텍스트이(가) 표시된 사진

자동 생성된 설명

사실 우리는 지금까지 java.lang.\*을 임포트하고 있었다.

왜 그렇게 만들었냐면 저 java.lang 아래에 있는 클래스들은 자바 프로그래밍에 있어서 너무도 중요하고 필수적인 요소들이라서 귀찮게 매번 임포트하는 수고를 덜어주기 위해서 자바에서 암시적으로 매번 임포트하고 있었던 것이다.

System이라는 클래스는 바로 java.lang.System 인 것이다.

[API]

Application Programming Interface 의 약자로 귀에 고름이 앉도록 많이 들어봤지만 그 개념이 모호하다.

인터페이스라는 것은 접점이라고 할 수 있다.

System.out.println("asdf");

위와 같은 코드도 API의 일종이라고 생각할 수 있다.

콘솔에 한 줄 그어주는 이 명령이 그다지 대단해보이지 않을 수도 있겠지만, 사실 내부적으로는 굉장히 복잡하고 다양한 무언가 과정을 거쳐서 화면에 찍어주는 것이다.

자바 애플리케이션, 즉 자바 프로그램은 자바 환경을 사용하는 일종의 사용자라고 볼 수 있는데, Application은 일반 User와는 다르게 코드를 사용해서 자바 시스템을 사용할 수 있는 사용자라는 것이다.

좀 감이 오고 있는 것 같다. 살짝 소름이 돋으면서.

API는 생각보다 넓은 범위라고 생각이 든다.

별게 다 API가 될 수 있다. 코드로 코드와 소통한다면 말이다.

유저는 버튼으로 프로그램과 소통하지만

개발자는 코드로 프로그램과 소통할 것이다. 버튼과 코드는 인터페이스이다.

버튼은 UI요, 코드는 API가 되는 것이다.

우리는 다른 사람이 만든 API를 이용해 다른 사용자에게 API를 제공할 수 있는 것이다.

[API문서]

즉 자바로 자바 애플리케이션을 개발한다는 것은 자바 API를 사용해서 UI를 만들어 사용자는 UI로 우리 프로그램과 소통하고, 우리 프로그램은 코드로, API를 통해 자바 시스템과 소통하는 구조가 될 것이다.

텍스트, 스크린샷, 노트북이(가) 표시된 사진

자동 생성된 설명

java api documentation에 가면 이런 문서를 볼 수 있다.

왼쪽 위는 package들이고

왼쪽 아래는 그 package들에 속한 class 들이다.

그리고 오른쪽 영역은 해당 class에 속한 멤버들이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

와.. System 클래스에 out이라는 멤버, 변수 즉 Field가 있는 걸 확인할 수 있다!

그것도 정적 필드인 것을 확인할 수 있다!

그런데 데이터 타입이 PrintStream이다. 그러면 out이라는 변수의 타입은 PrintStream이라는 것이겠네.

그러면 PrintStream 클래스를 한번 봐보자.

테이블이(가) 표시된 사진

자동 생성된 설명

PrintStream Class에 가면 멤버 메소드로 println이 있는 것을 확인할 수 있다.

그리고 메소드 오버로딩이 되어있는 것을 확인할 수 있다.

[멤버 접근제어자 private, public]

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드가 private와 public을 구분하는 예제다.

잘 보면 27번 줄에 A 클래스의 객체에선 y를 부를 수 없다. private로 선언됐기 때문이다.

그런데 private y를 호출하는 public z는 또 28번 줄에서 호출할 수 있다.

왜일까?

private는 접근제한자로, 객체 외부에선 부를 수 없고, 객체 내부에서만 호출할 수 있도록 엄격하게 제한하는 역할을 하기 때문이다.

객체또한 같다. 29줄에서 test.a의 값을 직접 변경할 수 없다. 심지어 30줄에선 호출도 안되는 것을 볼 수 있다.

그 대신 9번줄을 보면 객체의 a를 내부에서는 세팅할 수 있다. private로 보호되기 때문이다.

잘 생각해보면 이렇게 구성될 경우 클래스의 사용자는 private로 구성된 멤버들은 신경을 전혀 안써도 된다는 것을 알 수 있다. 왜냐면 만질 수가 없으니까 말이다.

[그 외 멤버 접근제어자들]

테이블이(가) 표시된 사진

자동 생성된 설명

위 표가 모든 것을 설명해준다. java를 책으로 배울 때도 자주 등장했던 표다.

이젠 패키지와 클래스의 개념에 대해서 알고 있으므로 이것을 이해하기 한결 나아졌다. 물론 어렵고 복잡함에는 변함은 없지만.

자주 사용하는 것은 public과 private이다. 둘은 속성이 극단적인 것을 알 수 있다.

극단적이라는 말은 public은 모든 장소에서 접근이 가능하고, private는 오로지 같은 클래스에서만 접근이 가능하다는 것을 말한다.

문제는 protected와 default(어떠한 접근제어자 선언도 안한 것)이다.

요점은 default와 protected의 차이는 “패키지가 다른 곳에서 참조할 수 있는가?” 이다.

default로 선언된 멤버는, 다른 패키지에 있다면 참조할 수 없다.

protected는 그럼 다른 패키지에 있다면 무조건 참조할 것 같지만? 또 그건 아니다.

이게 어려운 점인데, 화가 날 수도 있지만 심호흡 한 번, 진정하고 들어보자. 어려우면 다른 자료를 찾아봐도 좋으니 급하지 말자.

protected는 default와 다르게 다른 패키지에 있는 것을 “조건부로” 참조할 수 있다.

무슨 말이냐면 위의 표를 잘 보면 서브클래스라는 말이 있다. 자식 클래스를 말하는 것이다.

즉 다른 패키지의 클래스를 상속받았을 경우(자신이 서브 클래스=자식 클래스가 되었을 경우), 부모를 호출할 때 부모의 멤버 중에 protected가 있으면 그 protected로 선언된 멤버는 다른 패키지라도(다른 패키지에 있는 클래스를 상속 받았으므로 부모 클래스는 다른 패키지에 있는 클래스가 된다) 호출이 가능하다 라는 말이다.

그런 의미에서 다른 패키지 서브 클래스라는 것이 표의 스키마로 들어가있다.

요점은 default, private를 한 쌍으로 묶고, public, protected를 한 쌍으로 묶어서 생각한다면

그 두 그룹을 나누는 근본적인 차이는 바로 “다른 패키지에서 참조할 수 있는가” 이다.

protected는 단순히 “다른 패키지에서 참조할 수 있다. 그런데 네가 바로 그 다른 패키지에 있는 클래스의 자식 클래스이고, 너의 내부여야 참조할 수 있다” 라는 말을 해주는 것이다.

default는 “같은 패키지라면 어찌됐건 허용한다.” 라는 말을 하는 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이해를 돕기 위해 예제도 만들어봤다.

위는 javatutorials.accesmodifier라는 패키지에 있는 test2라는 클래스다.

2번줄을 보면 다른 패키지에 TT라는 클래스를 import한 것을 알 수 있다.

TT라는 클래스는 그냥 내부에 protected로 된 void go()라는 메소드를 하나 가지고 있는 클래스다.

4번줄에서 tester라는 내부 클래스를 만들어 그 TT를 상속했다.

17번 줄에서 tester 인스턴스를 만들었다.

19번 줄에서는 상속받은 TT의 go() 메소드를 호출하고자 한다.

그런데 에러가 났다. 왜냐하면 go 메소드는 protected로 선언되어있는데, 외부에서, 즉 서브클래스인 tester 내부가 아닌 서브클래스의 인스턴스(이자 외부)에서 호출하려고 했기 때문이다.

그런데 7번줄을 보면 this.go를 통해서 내부적으로는 TT 클래스의 go 메소드를 호출할 수 있다.

[클래스 접근 제어자]

지금까지 멤버 접근 제어자에 대해서 배운 것이다.

다행히, 클래스에 접근제어자를 붙이는 경우는 2개밖에 없다. public과, default.

사실 여기까지 오면서 이미 대강 감을 잡은 상태일지도 모르겠다.

결론부터 말하자면

default : 같은 패키지에서만 사용 가능한 클래스

public : 다른 패키지에서 “import” 하고 사용 가능한 클래스

여기서 중요한 점은

public 클래스와 소스코드 파일명은 무조건 같아야 한다! 라는 것이다.

즉, 소스코드 한 개에선 한 개의 public 클래스밖에 생성할 수 없다.