

# JAVA 9강 상속과super

강사 박주병

## 1. 상속

클래스를 개발하다보면 공통된 멤버들을 가지는 클래스들이 있을 것이다. 그림1처럼 게임을 개발한다고 하면 유닛들은 기본적으로 체력, 공격력 등이 모두 있어야 한다. 유닛의 개수가 많아질수록 이러한 코드들은 중복이 되며 차후 수정이 될 때 모든 클래스들의 코드를 변경해야 하는 불편함이 따른다.



그림 1

만약 공통적으로 필요한 멤버변수나 메서드들을 어느 한 클래스에서 물려 줄 수만 있다면 위와 같은 문제를 쉽게 해결하고 확장성있게 개발 할 수 있지 않을까?

그래서 나온 기능이 바로 상속이다.

부모

```
public class Unit {  
    int hp=40;  
    static int power=4;  
    static int armor=0;  
}
```

자식

```
public class Marine extends Unit {  
    String name;  
    Marine()  
    {  
    }  
}
```

```
public class Marine extends Unit {  
    String name;  
    Marine()  
    {  
        hp = 50;  
    }  
}
```

위의 그림과 같이 Marine은 클래스명 옆에 extends 키워드를 사용하여 부모 클래스를 지정하여 상속을 받고 있다. Marine 클래스에는 hp 멤버변수를 선언하지 않았지만 부모로부터 물려받아 마치 선언해놓은 것처럼 사용 할 수 있다.(단 초기화 불력은 상속되지 않는다)

부모로부터 상속받는다 해도 부모와 데이터를 공유 하는 것은 아니다. 아래 그림처럼 부모의 멤버변수를 생성자를

```
public static void main(String[] args) {  
    Parent parent = new Parent("부모1",45);  
    parent.showState();  
    Child child = new Child();  
    child.showState();  
}
```

```
<terminated> main [Java Application]  
이름: 부모 나이: 45  
이름: null 나이: 0
```

통해 초기화 한후 자식클래스를 생성하여 값을 출력하면 서로 다른 값을 지니고 있는걸 알 수 있다.  
이는 생각해보면 당연한 것이다. 애초에 둘은 서로 다른 객체이고 상속이란 것은 부모가 가지고 있는 멤버변수와 메서드를 자식 또한 가지고 있는 것이다. 코드 자체를 복사 해오는 것 이라 생각하면 된다.

### 1.1 생성자의 상속

부모의 생성자는 과연 자식에서 상속될까? 결론부터 말하면 상속되지 않는다. 하지만 처음 자바를 배울때는 개념적으로 상속된다고 봐야 한다.(이와 관련하여 정확히 공부하고 싶다면 오라클 홈페이지의 자바 설명서를 참고하면 된다)

기본적으로 상속이란 부모의 소스코드를 그대로 복사하여 자식에게 갖다 놓는거라고 보면 된다. 그렇다면 부모의 생성자 역시도 코드를 그대로 자식으로 가져와 붙여 놓는다면 그것이 자식의 생성자인가? 그렇지 않다. 왜냐하면 생성자는 클래스의 이름과 동일해야 하는데 부모의 이름 그대로 가져왔기 때문이다.

그림6처럼 부모에 2개의 매개변수를 받는 생성자를 만들고 자식은 그대로 상속하였다 해도 자식이 2개의 매개변수를 가지는 생성자가 있는 것이 아니다.

```
public class Parent {  
    String name;  
    int age;  
  
    Parent()  
    {  
    }  
  
    Parent(String name, int age)  
    {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Child child = new Child("자식", 10);  
  
}
```

그림 6

### 1.2 디폴트 생성자가 없는 부모 클래스

잠시 뒤에 살펴볼 super에서 다시 언급을 하겠지만 자식은 항상 부모의 생성자를 호출 하도록 되어 있다. 명시적으로 되어 사용하지 않는다면 부모의 디폴트 생성자를 호출하는 코드를 컴파일 단계에서 기본적으로 삽입을 한다.

따라서 부모가 만약 디폴트 생성자가 없다면 자식은 디폴트 생성자 이외에 매개변수가 있는 부모의 생성자를 명시적으로 사용해주지 않는 이상 컴파일 오류가 발생한다.

이러한 이유로 자식 또한 디폴트 생성자를 만들 수 없다. 왜냐하면 디폴트 생성자가 만들어지면 자동으로 생성된 생성자이기 때문에 내부 또한 자동으로 생성되며 부모의 디폴트 생성자 호출 코드를 삽입할 것이다. 이 과정에서 부모의 디폴트 생성자가 없기에 에러가 발생한다.

```
public class Child extends Parent{  
  
    void isChildMethod()  
    {  
    }  
  
}
```

**부모가 디폴트생성자가 없다면 자식또한 디폴트 생성자를 만들어주지 않는다.**

이러한 상황에서 자식이 디폴트 생성자를 쓰고 싶다면 뒤에서 배울 super를 통해 디폴트 생성자 내부에서 명시적으로 부모의 매개변수가 있는 생성자를 호출해야 한다.

1.3 상속은 계속 내려가면서 무한히 내려 갈 수 있다.

```
public class GrandParent {
    String name;
    int age;
    void showState()
    {
        System.out.println("이름: " + name + " 나이: " + age);
    }
}

public class Parent extends GrandParent{
    Parent()
    {
    }
    Parent(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
}

public class Child extends Parent{
    void ageUp()
    {
        age++;
    }
}
```

위의 그림을 보면 손자까지 이어서 상속이 된 걸 볼 수 있다. 이런식으로 상속은 무한히 내려갈수 있다.

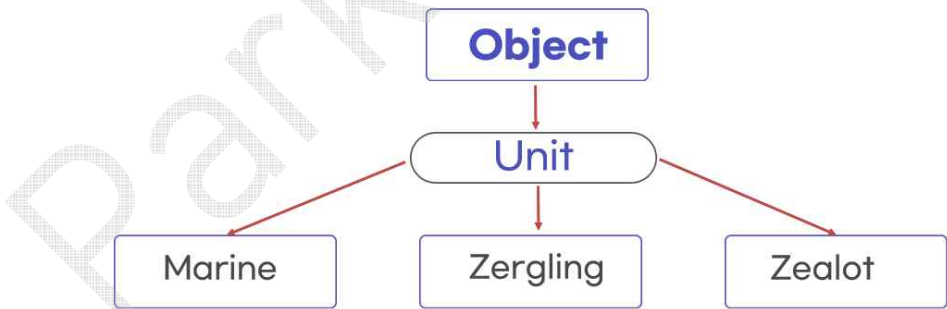
### 1.4 Object 클래스

Object 클래스는 모든 클래스의 부모이다. 우리가 클래스를 만들 때 부모클래스를 지정하지 않아도 기본적으로 부모가 되며 심지어 자바는 다중 상속을 허용하지 않는데도 불구하고 부모 클래스를 지정하여도 Object 클래스를 추가로 상속받도록 되어 있다.

Object 클래스는 toString, equal 과 같은 유용한 메서드들을 가지고 있고 이는 모든 클래스에서 필요로 하는 기본적인 기능들이다. 나중에 다형성을 배우게 된다면 이러한 Object의 특성으로 인해 많은 활용이 가능해진다.

### Object 클래스

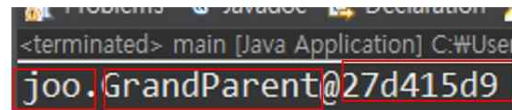
- 1. 모든 클래스의 부모
- 2. toString, equal과 같이 클래스에 기본적으로 필요한 메서드의 틀을 가지고 있다.
- 3. 모든 객체는 Object로 형변환이 가능하다.



#### 1.4.1 Object.toString()

toString은 기본적으로 객체의 상태를 String 형태로 변환하여 반환하는 메서드이다. 이 메서드는 앞서 말한것처럼 Object 클래스에 의해 모든 클래스가 기본적으로 상속을 받은 상태이다. 별도의 오버라이드를 하지 않으면 그림11 처럼 패키지, 클래스명, 객체의 주소등을 String 형태로 반환한다.

```
GrandParent gp = new GrandParent();  
System.out.println(gp.toString());
```



패키지      클래스명      객체주소

그림 11

일반적으로 해당 메서드는 멤버변수의 값을 보여주는 형태로 오버라이딩(잠시 뒤 배울 것이다) 한다.

#### 1.4.2 Object.equal()

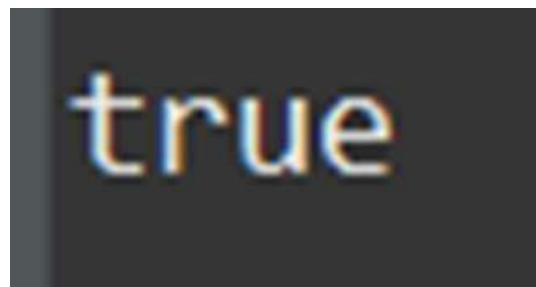
equal 메서드는 객체가 서로 같은 객체인지 비교하여 같다면 true 다르다면 false를 반환한다. 객체가 같다 다르다를 판단 하는 기준은 객체의 주소의 주소이다. 따라서 그림12와 같이 비교를 한다면 서로 다른 주소를 가지기 때문에 멤버변수의 값은 같더라도 false를 반환한다.

```
public static void main(String[] args) {  
  
    GrandParent gp = new GrandParent();  
    GrandParent gp2 = new GrandParent();  
  
    gp.name = "할아버지";  
    gp2.name = "할아버지";  
  
    System.out.println(gp.equals(gp2));  
  
}
```

그림 12

그렇다면 true 가 나오게 할려면 어떻게 해야 할까? 참조형 변수 역시 변수이기에 서로 값을 대입 할 수 있다. 즉 서로 같은 객체를 가리키게 하면 된다.

```
public static void main(String[] args) {  
  
    GrandParent gp = new GrandParent();  
    GrandParent gp2 = new GrandParent();  
  
    gp.name = "할아버지";  
    gp2.name = "할아버지2";  
  
    gp2 = gp;  
    System.out.println(gp.equals(gp2));  
  
}
```



## 2. 포함관계

지금까지 상속관계에 대해서 다뤘는데 클래스들 간의 관계에는 상속만 있는 것이 아니다. 다른 클래스를 멤버변수로 가지고 있는 관계가 있을 수 있는데 이를 포함 관계라고 부른다.

상속을 하던 포함을 하던 사실 기능적으로는 둘 다 동일하게 작동 될 수 있다. 접근하는 방식이 약간 다를 뿐이다 하지만 그렇다고 두 개를 아무 기준도 없이 아무렇게나 써도 된다는 것은 아니다. 현실세계에서 객체간의 관계를 잘 고려하여 설계하여야 된다.

예를 들어 지금까지 예시로 보여줬던 parent와 child의 관계는 상속관계이다. 반면 Car와 Door 혹은 People과 Eye와 같은 클래스는 어떤가? 이는 자동차가 문을 소유하고 있는 관계로 봐야한다. 물론 상속을 받아도 Door의 멤버들을 모두 가져오기에 기능적인 면에선 포함관계와 차이가 없다. 하지만 이는 현실세계를 코드로 그대로 녹여내어 단순하고 직관적인 코드를 만들고자 하는 객체지향언어와는 거리가 먼 코드가 된다.

```
public abstract class Car {  
  
    int speed;  
  
    public Door doors[] = new Door[4];  
}
```

상속관계인지 포함관계인지 잘 구분이 안된다면 is a, has a로 생각을 해보면 명확해진다.  
is a 는 결국 서로 같다는 것을 말하며 has a는 한쪽이 다른 한쪽을 소유한다고 보면 된다.

is a

has a

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>• 상속으로 표현한다.</li><li>• 같은 범주에 속한다.</li><li>• 차, 전기차 와의 관계</li></ul> | <ul style="list-style-type: none"><li>• 멤버변수로 표현한다.</li><li>• 소유나 일부분을 나타낸다.</li><li>• 차, 핸들,문 과의 관계</li></ul> |
|---|--|

## 3. 오버라이딩

```
public class Parent extends GrandParent{  
  
    String name = "부모";  
  
    void parentMethod()  
    {  
        System.out.println("부모 메서드");  
    }  
}
```



```
public class Child extends Parent{  
  
    String name = "자식";  
  
    void parentMethod()  
    {  
        System.out.println("자식 메서드");  
    }  
}
```

그림 17

그림17을 보면 Child는 부모로부터 parentMethod()를 상속받아 이미 가지고 있는 상태이다. 그런데 메서드의 시그니처(메서드의 이름과 매개변수의갯수,타입)가 동일하고 심지어 반환타입까지 완전히 동일한 메서드를 다시 정의하였다.이렇게 될 경우 부모로부터 상속받은 메서드는 가려지며 자식이 새롭게 정의한 메서드로 덮히게 된다. 이를 오버라이딩 이라고 한다.



자식 객체를 생성하고 parentMethod() 메서드를 호출하면 “자식 메서드”가 출력될 것이다.

이와 더불어 멤버변수 또한 오버라이딩이 가능하다.

```
public class GrandParent {  
  
    String name;  
    int age;  
  
    void showState()  
    {  
        System.out.println("이름: " + name+" 나이: "+age);  
    }  
  
    @Override  
    public String toString() {  
  
        return "name: " + name +"age: "+age ;  
    }  
}
```

그림 18

```
public class Parent extends GrandParent{  
  
    String name;  
    Parent()  
    {  
    }  
  
    Parent(String name, int age)  
    {  
        this.name = name;  
        this.age = age;  
    }  
}
```

그림 19

그림19를 보면 부모로부터 물려받은 name 멤버변수를 다시 선언하였다. 하지만 잘 생각해보면 메서드와 다르게 멤버변수를 오버라이딩 한다는건 사용하는것에 있어 큰 차이가 없다. 변수는 값을 담아두기 위한용도이기에 새롭게 정의 한다 해도 그 기능이 달라지는 것이 아니기 때문이다.

하지만 뒤에서 배울 super를 배운다면 멤버변수를 오버라이딩 한다는건 의미가 있게된다. 현재는 멤버변수도 오버라이딩이 가능하다는것만 알아두자.

### 3.1 공변반환타입

앞서 오버라이딩은 부모로부터 상속받은 메서드를 시그니처가 동일하고 반환타입까지 모두 동일하게 재정의하는것이라고 하였다. 만약 시그니처가 다르다면 그것은 부모로부터 상속받은 메서드에다가 오버로딩을 적용하여 메서드를 추가하는 것이다.

그런데 이러한 오버라이딩의 규칙에 예외가 한가지 있다. 그림20을 보면 메서드의 리턴타입이 자기 자신이다.

```
public class GrandParent {  
  
    String name;  
    int age;  
  
    void showState()  
    {  
        System.out.println("이름: " + name+" 나이: "+age);  
    }  
  
    GrandParent getInstance()  
    {  
        return this;  
    }  
}
```

그림 20

```
Parent getInstance() {  
    // TODO Auto-generated method stub  
    return this;  
}
```

그림 21

그림21을 보면 GrandParent 클래스를 상속받은 Parent 클래스 내부에서는 리턴타입을 Parent로 변경하여 재정의 하였다. 원래 대로라면 리턴타입이 달라졌기에 오버라이딩에 해당되지 않고 시그니처역시 동일하여 메서드 중복으로 에러가 발생해야 한다. 하지만 공변 반환타입이라는 예외조건으로 인해 이 또한 오버라이딩으로 인정된다. 이 외에는 예외없이 시그니처+리턴타입까지 모두 동일해야 오버라이딩으로 인정된다.

### 3.2 static <-> 인스턴스 메서드 간의 변환

부모에서 static으로 정의된 메서드를 자식클래스에서 인스턴스 메서드로 오버라이딩 할수 없다. 이 반대의 경우도 마찬가지이다.

```
public class GrandParent {  
  
    String name;  
    int age;  
  
    void showState()  
    {  
        System.out.println("이름: " + name+" 나이: "+age);  
    }  
  
    static void print()  
    {  
        System.out.println("test");  
    }  
  
    @Override  
    public String toString() {  
        return "name: " + name +"age: "+age ;  
    }  
}
```

```
public class Parent extends GrandParent{  
  
    String name;  
  
    Parent()  
    {  
    }  
  
    Parent(String name, int age)  
    {  
        this.name = name;  
        this.age = age;  
    }  
  
    void print()  
    {  
        System.out.println("test");  
    }  
}
```

### 3.3 toString() 메서드의 오버라이딩

상속에서 모든 클래스는 Object 클래스를 상속받기에 toString() 메서드를 가지고 있다고 하였다. 일반적으로 해당 클래스를 오버라이딩하여 멤버변수의 값을 리턴해주기도 한다.

```
2  
3 public class GrandParent {  
4  
5  
6     String name;  
7     int age;  
8  
9     void showState()  
10    {  
11        System.out.println("이름: " + name+" 나이: "+age);  
12    }  
13  
14  
15    @Override  
16    public String toString() {  
17  
18        return "name: " + name +"age: "+age ;  
19    }  
20 }
```

### 3.4 오버라이딩 VS 오버로딩

용어가 비슷하고 이름이 같은 메서드를 다시금 선언하기에 혼돈하기 쉬운 기능들이다. 하지만 두 개는 명백히 다른 것을 말한다.

오버로딩은 메서드를 확장시키는 것이다. 메서드의 이름은 동일하며 매개변수 부분을 다르게 만들어 해당 메서드를 확장시키는 것이다.

반면에 오버라이딩은 상속관계에서만 가능한 것이다. 부모로부터 상속받은 메서드를 리턴타입까지 완전히 동일하게 재정의하여 덮어쓰는 것이다.

#### 4. super

부모로부터 멤버변수와 메서드를 상속받았다면 그 멤버들은 사실상 부모 클래스와 상관없이 자식 클래스 내부에 선언된거와 같다. 그 상황에서 멤버변수와 메서드를 오버라이딩 한다면 부모로부터 물려받은걸 덮어쓴다고 하였다. 하지만 오버라이딩을 하였어도 부모로부터 물려받은 원본이 필요한 경우가 있다. 예를들어 부모로부터 상속받은 메서드에 추가적인 기능을 덧붙여야 해서 오버라이딩 한다면 메서드 내부에서는 부모의 기능 +@ 이므로 부모가 이미 상속시켜준 메서드를 호출할수만 있다면 거기에 덧붙여 지는 +@만 코드를 짜 놓으면 되는 것이다.

```
2
3 public class Parent {
4
5
6     String name = "부모";
7
8
9     public String toString()
10    {
11        return "이름: " + name;
12    }
13
```

```
public class Child extends Parent{
    int age;
    public String toString()
    {
        return "이름: " + name + " 나이: " + age;
    }
}
```

그림 24

그림24처럼 부모의 toString 메서드는 이름을 출력하고 있다. 자식에서는 age라는 멤버변수가 추가 되었기에 이역 시도 toString에 추가하기 위해 오버라이딩을 하였다. 여기서 이름을 출력하는 기능을 또다시 만들어야만 한다.

이럴 경우 부모와 코드가 중복이며 부모에서 변경이 되어야 하는 경우 자식 클래스 역시도 수정을 해줘야 한다. 지금 예제에서야 코드가 간단하기 쉽게 변경하여도 코드가 길어진다면 혹은 자식 클래스가 많다면 이는 어려운 작업이 될 것이다. 따라서 부모가 상속 시켜준 toString을 오버라이딩 하기 이전의 원본을 호출 할 수만 있다면 이 문제는 해결될 것이다.

```
2
3 public class Parent {
4
5
6     String name = "부모";
7
8
9     public String toString()
10    {
11        return "이름: " + name;
12    }
13
```

```
public class Child extends Parent{
    int age;
    public String toString()
    {
        return super.toString() + " 나이: " + age;
    }
}
```

그림 25

그림25의 Child 클래스를 보면 super.toString()을 사용하고 있다. super 자체는 앞서배운 this와 마찬가지로 자기 자신의 객체주소를 가지고 있다. 그래서 멤버들을 오버라이딩 하지 않았다면 this와는 차이가 없다. 하지만 위의 그림처럼 오버라이딩을 했다면 super는 오버라이딩하기 이전의 상속받은 원본의 멤버변수, 메서드를 실행시킬수 있다.



```
public class Parent extends GrandParent{

    String name = "부모";
```

그림 27

```
public class Child extends Parent{

    String name = "자식";

    void test()
    {

        System.out.println(name);
        System.out.println(this.name);
        System.out.println(super.name);

        System.out.println(System.identityHashCode(name));
        System.out.println(System.identityHashCode(this.name));
        System.out.println(System.identityHashCode(super.name));

        System.out.println(this);
        System.out.println(super.toString());
    }
}
```

그림 26

Parent 클래스에서 name 변수를 선언하였고 그림26에서는 Parent를 상속받아 name 변수를 오버라이딩 하였다. 이럴 경우 name, this.name은 오버라이딩해놓은 변수를 가리키는것이고 super.name은 부모로부터 상속받은 원본을 가리키는 것이다. 아래의 그림은 Child 클래스의 test메서드를 실행한 결과 이다.

```
<terminated> main [Java Application] C:\
자식
자식
부모
838411509
838411509
1434041222
joo.Child@5204062d
joo.Child@5204062d
```

그림 28

super.name 의 경우 name, this.name 과 주소가 다르다는 것을 볼 수 있다. 즉 오버라이딩된 변수와는 다르다는 것이다.

#### 4.1 super 생성자

부모의 생성자 역시 super를 통해 호출 가능하다. 그림29를 보면 자식 클래스에서 name을 초기화 하는코드가 부모의 생성자와 중복이다. 이럴 경우 name 변수를 초기화 하는 방식의 변경되면 모든 자식 클래스들을 수정해줘야 한다.

앞서 배운 toString과 같은 경우이다.

```
public class Parent {  
  
    String name = "부모";  
  
    Parent()  
    {  
  
    }  
  
    Parent(String name)  
    {  
        this.name =name;  
    }  
}  
  
public class Child extends Parent{  
  
    int age;  
  
    Child()  
    {  
  
    }  
  
    Child(String name , int age)  
    {  
        this.name = name;  
        this.age = age;  
    }  
}
```

그림 30

그래서 아래와 같이 super를 통해 부모 생성자의 원본을 호출한다면 이러한 문제는 해결된다.

```
Child(String name , int age)  
{  
    super(name);  
    this.age = age;  
}
```

다만 super 생성자를 이용할 때 주의해야 할것이 있다. 바로 this를 통해 생성자를 사용하는것과 마찬가지로 생성자 내에서 가장 첫줄에 호출을 해야 된다는 것이다.

```
Child(String name , int age)  
{  
    this.age = age;  
    super(name);  
}
```

→ **항상 제일 먼저 수행 되어야 한다.**

이는 부모로부터 상속받은 멤버변수를 초기화하는 것이 만약 다른 초기화 코드보다 뒤에 있다면 초기화 코드들이 의미가 없어질수도 있고 부모클래스가 의도하지 않은 방향으로 흘러갈수 있다.따라서 문법적으로 가장 첫줄에 작성 하도록 되어 있다.

## 4.2 super의 자동삽입

super 생성자의 경우 자식 클래스에서 따로 호출하지 않는다면 컴파일단계에서 super()가 자동으로 삽입된다.

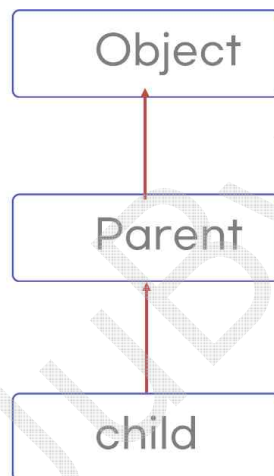
```
Child()
{
}

Child(String name , int age)
{
    this.age = age;
}
```

만약 super 생성자를 사용하지 않으면  
컴파일러가 자동으로 끼워넣는다.

즉 자식 클래스의 생성자가 호출될 때 무조건 부모클래스의 생성자 역시 호출된다는 것이다. 그리고 부모 생성자 역시도 자신의 부모 생성자를 호출하도록 되어 있다.

따라서 모든 클래스는 Object의 생성자를 호출하도록 되어 있는 것이다.



모든 클래스들은 객체 생성시 Object 생성자를 호출한다.

그런데 만약 부모가 디폴트 생성자를 만들어 놓지 않는다면 어떻게 될까?

아래의 그림처럼 자식생성자에서 super를 통해 부모의 생성자를 명시적으로 호출하지 않으니 자동으로 부모의 기본생성자를 삽입하려고 할 것이다. 그러나 부모의 기본 생성자가 없으니 컴파일 에러가 발생한다.

물론 이 상황에서 명시적으로 super("이름") 으로 부모의 생성자를 호출해주면 컴파일에러는 발생하지 않는다.

```
public class Parent {
    String name = "부모";

    Parent(String name)
    {
        this.name = name;
    }
}

public class Child extends Parent {
    int age;

    Child()
    {
    }

    Child(String name , int age)
    {
        super(name);
        this.age = age;
    }
}
```

super()가 자동삽입 되어야 하나 부모의 디폴트 생성자가 없다.