07. 상속-중첩 Class

강사 강태광

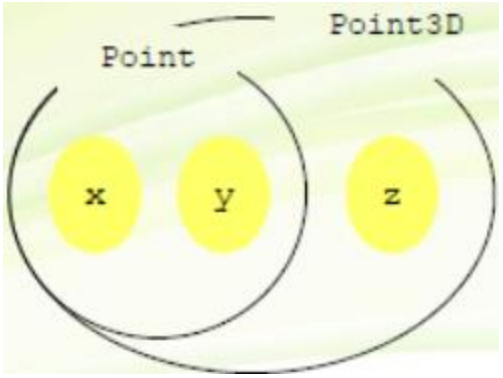
I] 상속(inheritance)의 정의와 장점

[1] 상속 정의

- 기존의 클래스를 재사용해서 새로운 클래스를 작성
- 두 클래스를 조상과 자손으로 관계를 맺어주는 것
- 자손은 조상의 모든 멤버를 상속받는다.
  (생성자, 초기화블럭 제외)
- 자손의 멤버개수는 조상보다 적을 수 없다.
  (같거나 많다.)

```
class Point {
    int x;
    int y;
}
```

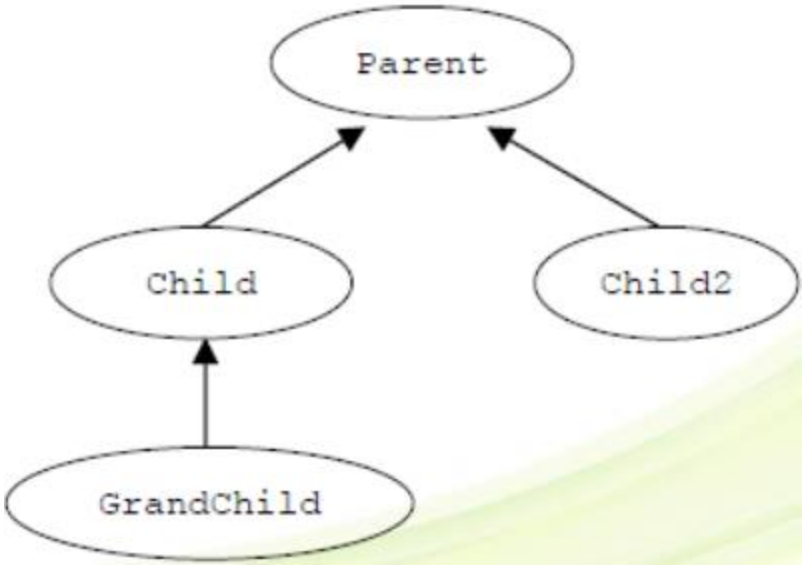
```
class Point3D extends Point {
    int z;
}
```



2] 클래스간의 관계 – 상속관계(inheritance)

- 공통부분은 조상에서 관리하고 개별부분은 자손에서 관리
- 조상의 변경은 자손에 영향을 미치지만, 자손의 변경은 조상에 아무런 영향을 미치지 않는다

```
class Parent {}
class Child extends Parent {}
class Child2 extends Parent {}
class GrandChild extends Child {}
```



3] 클래스간의 관계 – 상속예시문

```
class A{
	int k=5;
	A(){
		System.out.println("A class");
	}
}
class B extends A{
	B(){
		System.out.println("B Class");
	}
	public static void main(String[] a){
		B b = new B();
		System.out.println("k=>"+b.k);
	}
}
```

4] 상속(클래스의 관계) I

부모클래스(객체)의 멤버들을 자식클래스(객체)가 물려받는것
1. - 상속을 사용하는 이유
기존에 만들어놓은 클래스의 재사용, 확장을 위해 사용.
2. 자바에서는 단일상속만이 가능하다(부모클래스가 한 개 만 가능)
3. 부모클래스(super)와 자식클래스(sub)가 존재한다.
4. 자바에서 제공되어지는 모든 클래스들은 Object 라고 하는
최상위 클래스로부터 상속되어진다.
5. 사용자정의 클래스들도 Object 클래스라는 최상위클래스를
상속 받음

```
public class Parent {
	public String member1="난 부모에서 정의한 멤버변수";
	public void print(){
		System.out.println("난 부모에서 정의한 멤버 메쏘드 member1:"+member1);
	}
}
```

4] 상속(클래스의 관계) 2

```
public class Child extends Parent {
	public String member2="난 자식에서 정의한 멤버변수";
	public void childPrint(){
		System.out.println("난 자식에서 정의한 메쏘드");
		System.out.println("member1:"+this.member1);
		System.out.println("member2:"+this.member2);
	}
}
public class ParentChildMain {
	public static void main(String[] args) {
		Child c1=new Child();
		System.out.println("c1.member1:"+c1.member1);
		c1.print();
		System.out.println("c1.member2:"+c1.member2);
		c1.childPrint();
	}
}
```

4] Object클래스 – 모든 클래스의 최고조상

- 조상이 없는 클래스는 자동적으로 Object클래스를 상속.
- 상속계층도의 최상위에는 **Object클래스가 위치한다.**
- 모든 클래스는 Object클래스에 정의된 11개의 메서드를 상속.
  toString(), equals(Object obj), hashCode(), ...

5] 상속(inheritance)- private

- private 객체변수는 상속 안됨(**InheritanceTest2.java**)

```
class AA {
        int i;
        private int j;   // 객체 변수 j를 private로 선언
        void setij(int x, int y) {
                i = x;
                j = y;
        }
}
class BB extends AA {
         int total;
        void sum() {   // 오류 발생 BB
                total = i + j;
        }
}
public class InheritanceTest2 {
        public static void main(String  args[]) {
            BB subOb = new BB();
            subOb.sum();
        }
}
```

5] 상속(inheritance)-super

1) 하위 클래스에 의해 가려진 상위 클래스의 멤버 변수나 메소드에 접근할 때

- ▷ **super.객체변수**
- ▷ **super.메소드이름(매개변수)**

2) 상위 클래스의 생성자를 호출할 때
   ▷ **super(매개변수)**

3) **super문장은 반드시 첫 번째 라인에 와야 한다**

4) this( )가 자신의 생성자를 호출하기 위해 제공되는 예약어라면 수퍼 클래스의 생성자를 호출하기 위해서는 super( ) 예약어가 제공.

5) 만일 매개 변수가 있는 수퍼 클래스의 생성자를 호출하고 싶은 경우 에는 super(매개 변수)라고 호출.

6) 주의할 점은 수퍼 클래스의 생성자를 호출하는 위치이다.
서브 클래스의 생성자에서 무엇보다도 수퍼 클래스의 생성자를 제일 먼저 호출해주어야 함

5] 상속(inheritance)-super 예시

```
class D1 {
	int x = 1000;
	void display() {
		System.out.println("상위클래스 D1의 display() 메소드 입니다");
	}
}
class D2 extends D1 {
	int x = 2000;
	void display() {
		System.out.println("하위클래스 D2의 display() 메소드 입니다");
	}
	void write() {
		this.display();
		super.display();
		System.out.println("D2 클래스 객체의 x 값은 : " + x);
		System.out.println("D1 클래스 객체의 x 값은 : " + super.x);
	}
}
class InheritenceSuper {
	public static void main(String args[]) {
		D2 d = new D2();
		d.write();
	}
}
```

5]상속 관계에서의 생성자 문제와 해결책

1) 디폴트 생성자는 JVM이 제공해주지만, 클래스 내의 매개변수가 있는 생성자가 하나라도 존재하게 되면 JVM은 더 이상 디폴트 생성자를 제공해 주지 않게 됨

2) 만일 수퍼 클래스에 매개 변수가 있는 생성자를 정의하면서 매개 변수 없는 디폴트 생성자를 정의하지 않으면 수퍼 클래스에는 매개 변수 없는 생성자가 존재하지 않게 됨

3) 이러한 상태에서 서브 클래스의 생성자는 수퍼 클래스의 매개 변수 없는 디폴트 생성자를 여전히 호출하고 있기에 존재하지 않는 생성자를 호출하는 셈이 되어 문제가 발생

6] 내부 클래스(inner class)

1) 클래스 안에 선언된 클래스
- 특정 클래스 내에서만 주로 사용되는 클래스를 내부 클래스로 선언
- GUI어플리케이션(AWT, Swing)의 이벤트처리에 주로 사용된다.

2) 내부 클래스의 장점
- 내부 클래스에서 외부 클래스의 멤버들을 쉽게 접근할 수 있다.
- 코드의 복잡성을 줄일 수 있다(캡슐화)

```
class A {
    //...
}

class B {
    //...
}
```

```
class A {  // 외부 클래스
    //...
    class B { // 내부 클래스
        //...
    }
    //...
}
```

6] 내부 클래스의 종류와 특징 I

- 내부 클래스의 종류는 변수의 선언위치에 따른 종류와 동일하다.
- 유효범위와 성질도 변수와 유사하므로 비교해보면 이해하기 쉽다

내부 클래스	특 징
인스턴스 클래스 (instance class)	외부 클래스의 멤버변수 선언위치에 선언하며, 외부 클래스의 인스턴스멤버처럼 다루어진다. 주로 외부 클래스의 인스턴스멤버들과 관련된 작업에 사용될 목적으로 선언된다.
스태틱 클래스 (static class)	외부 클래스의 멤버변수 선언위치에 선언하며, 외부 클래스의 static멤버처럼 다루어진다. 주로 외부 클래스의 static멤버, 특히 static메서드에서 사용될 목적으로 선언된다.
지역 클래스 (local class)	외부 클래스의 메서드나 초기화블럭 안에 선언하며, 선언된 영역 내부에서만 사용될 수 있다.
익명 클래스 (anonymous class)	클래스의 선언과 객체의 생성을 동시에 하는 이름없는 클래스(일회용)

```
class Outer {
    int iv=0;
    static int cv=0;

    void myMethod() {
        int lv=0;
    }
}
```

```
class Outer {
    class InstanceInner {}
    static class StaticInner {}

    void myMethod() {
        class LocalInner {}
    }
}
```

7] 내부 클래스의 종류와 특징 2

1) 중첩(내부) 클래스 (Inner Class)
- 클래스 내부에 또 다른 클래스를 가짐으로 클래스 관리의 효율을 높인 것(static 포함불가)

2) 중첩 클래스의 형식과 생성파일
- 형식) class Outer { class Inner { … } }
- 생성파일) Outer.class, Outer$Inner.class

3) 중첩 클래스 객체 생성
- Outer.Inner oi = new Outer().new Inner();

7] 내부 클래스의 종류와 특징 3(예시)

```
class Outer {
    private int height;
    private int width;
    public Outer(int h, int w) {
        height = h;
        width = w;
    }
    public Inner getInner() {
        return new Inner();
    }

    class Inner {
        private float rate = 0.5f;
        public float capacity() {
            return rate * height
            * width;
        }
    }
}
```

```
class TestOuter {
   public static void main(String args[]) {
            Outer outRef = new Outer(100,200);
            Outer.Inner myInner = outRef.getInner();
            // Inner myInner = outRef.getInner();
            System.out.println("Inner value is " +
            myInner.capacity());

            Outer.Inner yourInner = outRef.new Inner();
            System.out.println("Inner value is " +
            yourInner.capacity());


   }
}
```

7] 정적 중첩 클래스

1) 정적 중첩 클래스(Static Inner Class)
- 중첩 클래스 내부에 static 멤버를 포함할 수 있는 형태
  (Outer의 non-static 멤버 포함 불가)
2) 정적 중첩 클래스의 형식과 생성파일
- 형식) class Outer { static class Inner {…} }
- 생성파일) Outer.class, Outer$Inner.class
3) 정적 중첩 클래스 객체 생성

```
Outer.Inner oi = new Outer.Inner();
```

7] 내부 클래스의 제어자와 접근성(1/5)

1) 내부 클래스의 접근제어자는 변수에 사용할 수 있는 접근제어자와 동일

```
class Outer {
    private int iv=0;
    protected static int cv=0;

    void myMethod() {
        int lv=0;
    }
}
```

```
class Outer {
    private class InstanceInner {}
    protected static class StaticInner {}

    void myMethod() {
        class LocalInner {}
    }
}
```

2) static클래스만 static멤버를 정의할 수 있다.

```
class InnerEx1 {
      class InstanceInner {
            int iv = 100;
//          static int cv = 100;            // 에러! static변수를 선언할 수 없다.
            final static int CONST = 100;   // final static은 상수이므로 허용한다.
      }

      static class StaticInner {
            int iv = 200;
            static int cv = 200;      // static클래스만 static멤버를 정의할 수 있다.
      }

      void myMethod() {
            class LocalInner {
                  int iv = 300;
//                static int cv = 300;          // 에러! static변수를 선언할 수 없다.
                  final static int CONST = 300; // final static은 상수이므로 허용
            }
      } // void myMethod() {
}
```

7] 내부 클래스 예제 사용예시

```
class Outer1 {
	private int x = 100;
	 private static int y = 200;
	public Outer1() {}
		public void disp() {
		System.out.println("x  = " + x);
		System.out.println("y  = " + y);
	}
	static class Inner1 {
		private int a = 10;
		static int b = 20;
		public Inner1() {}
		public void disp_in() {
			System.out.println("a  = " + a);
			System.out.println("b  = " + b);
			//System.out.println("x  = " + x);
			System.out.println("y  = " + y);
			//disp();
		}
	}
}
```

```
public class Exam_02 {
	public static void main(String[] ar) {
	Outer1.Inner1 oi = new Outer1.Inner1();
	oi.disp_in();
	System.out.println("b = " + Outer1.Inner1.b);
	}
}
```

7] 익명 중첩 클래스

1) 익명 중첩 클래스(Anonymous Inner Class)
- 기존 클래스의 특정 메서드를 오버라이딩 하여 원하는 형태로 재정의 하여 사용하는 방식
- 외부 멤버 중 final만 포함할 수 있다.

2) 익명 중첩 클래스의 형식 및 생성파일
- 형식) class Inner { ... }
  class Outer { method() { new Inner() {...}}}
- 생성파일) Outer.class, Outer$숫자.class

3) new Inner() 자체가 객체 생성임.

```
import java.awt.*;
import java.awt.event.*;
class InnerEx8 {
	public static void main(String[] args) {
		Button b = new Button("Start");
		b.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				System.out.println("ActionEvent occurred!!!");
			}
		} // 익명 클래스의 끝 );
	} // main메서드의 끝
} // InnerEx8클래스의 끝
```

8] 클래스 간의 관계 결정하기 – 상속 vs. 포함

1) 가능한 한 많은 관계를 맺어주어 재사용성을 높이고 관리하기 쉽게 한다
2) 'is-a'와 'has-a'를 가지고 문장을 만들어 본다

원(Circle)은 점(Point)**이다**. - Circle **is a** Point.
원(Circle)은 점(Point)을 **가지고 있다**. - Circle **has a** Point.

8] 참조변수의 형변환

- 서로 상속관계에 있는 타입간의 형변환만 가능하다.
- 자손 타입에서 조상타입으로 형변환하는 경우, 형변환 생략가능

자손타입 → 조상타입 (Up-casting) : 형변환 생략가능
자손타입 ← 조상타입 (Down-casting) : 형변환 생략불가

```
class Car {
    String color;
    int door;

    void drive() {  // 운전하는 기능
        System.out.println("drive, Brrrr~");
    }

    void stop() {   // 멈추는 기능
        System.out.println("stop!!!");
    }
}

class FireEngine extends Car {  // 소방차
    void water() {  // 물뿌리는 기능
        System.out.println("water!!!");
    }
}

class Ambulance extends Car {   // 구급차
    void siren() {  // 사이렌을 울리는 기능
        System.out.println("siren~~~");
    }
}
```

```
public static void main(String args[]) {
    Car car = null;
    FireEngine fe = new FireEngine();
    FireEngine fe2 = null;

    fe.water();
    car = fe;   // car = (Car)fe; 조상 <- 자손
//  car.water();
    fe2 = (FireEngine)car; // 자손 <- 조상
    fe2.water();
}
```

9] instanceof 연산자

- 참조변수가 참조하는 인스턴스의 실제 타입을 체크하는데 사용.
- 이항연산자이며 피연산자는 참조형 변수와 타입. 연산결과는 true, false.
- instanceof의 연산결과가 true이면, 해당 타입으로 형변환이 가능

```
class InstanceofTest {
    public static void main(String args[]) {
        FireEngine fe = new FireEngine();

        if(fe instanceof FireEngine) {
            System.out.println("This is a FireEngine instance.");
        }

        if(fe instanceof Car) {
            System.out.println("This is a Car instance.");
        }

        if(fe instanceof Object) {
            System.out.println("This is an Object instance.");
        }
    }
}
```

```
void method(Object obj) {
    if(c instanceof Car) {
        Car c = (Car)obj;
        c.drive();
    } else if(c instanceof FireEngine) {
        FireEngine fe = (FireEngine)obj;
        fe.water();
    }
}
```

10] 참조변수와 인스턴스변수의 연결

1) 멤버변수가 중복정의된 경우, 참조변수의 타입에 따라 연결되는 멤버변수가 달라진다.(참조변수타입에 영향받음)
2) 메서드가 중복정의된 경우, 참조변수의 타입에 관계없이 항상 실제 인스턴스의 타입에 정의된 메서드가 호출된다.(참조변수타입에 영향받지 않음)

```
class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    int x = 200;

    void method() {
        System.out.println("Child Method");
    }
}
```

```
class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent { }
```

```
public static void main(String[] args) {
    Parent p = new Child();
    Child c = new Child();

    System.out.println("p.x = " + p.x);
    p.method();

    System.out.println("c.x = " + c.x);
    c.method();
}
```

```
p.x = 100
Child Method
c.x = 200
Child Method
```