Java의 정석

제 7 장

객체지향개념 II-3

2009. 10. 27

남궁성 강의

castello@naver.com

1. 상속
2. 오버라이딩
3. package와 import

→ 객체지향개념 II-1

4. 제어자
5. 다형성

→ 객체지향개념 II-2

6. 추상클래스
7. 인터페이스
8. 내부 클래스

→ 객체지향개념 II-3

6. 추상클래스(abstract class)

6.1 추상클래스(abstract class)란?

- 클래스가 설계도라면 추상클래스는 '미완성 설계도'
- 추상메서드(미완성 메서드)를 포함하고 있는 클래스

  * 추상메서드 : 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
abstract class Player {
     int currentPos;          // 현재 Play되고 있는 위치를 저장하기 위한 변수

     Player() {               // 추상클래스도 생성자가 있어야 한다.
          currentPos = 0;
     }

     abstract void play(int pos);      // 추상메서드
     abstract void stop();             // 추상메서드

     void play() {
          play(currentPos);            // 추상메서드를 사용할 수 있다.
     }
     ...
}
```

- 일반메서드가 추상메서드를 호출할 수 있다.(호출할 때 필요한 건 선언부)
- 완성된 설계도가 아니므로 인스턴스를 생성할 수 없다.
- 다른 클래스를 작성하는 데 도움을 줄 목적으로 작성된다.

6.2 추상메서드(abstract method)란?

- 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
/* 주석을 통해 어떤 기능을 수행할 목적으로 작성하였는지 설명한다. */
abstract 리턴타입 메서드이름();

Ex)
/* 지정된 위치(pos)에서 재생을 시작하는 기능이 수행되도록 작성한다.*/
abstract void play(int pos);
```

- 꼭 필요하지만 자손마다 다르게 구현될 것으로 예상되는 경우에 사용
- 추상클래스를 상속받는 자손클래스에서 추상메서드의 구현부를 완성해야 한다.

```
abstract class Player {
    ...
     abstract void play(int pos);      // 추상메서드
     abstract void stop();             // 추상메서드
    ...
}

class AudioPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
    void stop() { /* 내용 생략 */ }
}

abstract class AbstractPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
}
```

6.3 추상클래스의 작성

- 여러 클래스에 공통적으로 사용될 수 있는 추상클래스를 바로 작성하거나 기존클래스의 공통 부분을 뽑아서 추상클래스를 만든다.

```
class Marine {    // 보병
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void stimPack()         { /* 스팀팩을 사용한다.*/}
}

class Tank {      // 탱크
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void changeMode()       { /* 공격모드를 변환한다. */}
}

class Dropship {  // 수송선
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
}
```

```
abstract class Unit {
    int x, y;
    abstract void move(int x, int y);
    void stop() {      /* 현재 위치에 정지 */ }
}

class Marine extends Unit {  // 보병
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stimPack()         { /* 스팀팩을 사용한다.*/}
}

class Tank extends Unit {    // 탱크
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode()       { /* 공격모드를 변환한다. */}
}

class Dropship extends Unit { // 수송선
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
}
```

```
Unit[] group = new Unit[4];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();

for(int i=0;i< group.length;i++) {
    group[i].move(100, 200);
}
```

추상메서드가 호출되는 것이 아니라 각 자손들에 실제로 구현된 move(int x, int y)가 호출된다.

7. 인터페이스(interface)

7.1 인터페이스(interface)란?

- 일종의 추상클래스. 추상클래스(미완성 설계도)보다 추상화 정도가 높다.
- 실제 구현된 것이 전혀 없는 기본 설계도.(알맹이 없는 껍데기)
- 추상메서드와 상수만을 멤버로 가질 수 있다.
- 인스턴스를 생성할 수 없고, 클래스 작성에 도움을 줄 목적으로 사용된다.
- 미리 정해진 규칙에 맞게 구현하도록 표준을 제시하는 데 사용된다.

7.2 인터페이스의 작성

- 'class'대신 'interface'를 사용한다는 것 외에는 클래스 작성과 동일하다.

```
interface 인터페이스이름 {
    public static final 타입 상수이름 = 값;
    public abstract 메서드이름(매개변수목록);
}
```

- 하지만, 구성요소(멤버)는 추상메서드와 상수만 가능하다.

- 모든 멤버변수는 public static final 이어야 하며, 이를 생략할 수 있다.
- 모든 메서드는 public abstract 이어야 하며, 이를 생략할 수 있다.

```
interface PlayingCard {
    public static final int SPADE = 4;
    final int DIAMOND = 3;      // public static final int DIAMOND = 3;
    static int HEART = 2;       // public static final int HEART = 2;
    int CLOVER = 1;             // public static final int CLOVER = 1;

    public abstract String getCardNumber();
    String getCardKind();  // public abstract String getCardKind();
}
```

7.3 인터페이스의 상속

- 인터페이스도 클래스처럼 상속이 가능하다.(클래스와 달리 다중상속 허용)

```
interface Movable {
     /** 지정된 위치(x, y)로 이동하는 기능의 메서드 */
     void move(int x, int y);
}

interface Attackable {
     /** 지정된 대상(u)을 공격하는 기능의 메서드 */
     void attack(Unit u);
}

interface Fightable extends Movable, Attackable { }
```

- 인터페이스는 Object클래스와 같은 최고 조상이 없다.

7.4 인터페이스의 구현

- 인터페이스를 구현하는 것은 클래스를 상속받는 것과 같다.
  다만, 'extends' 대신 'implements'를 사용한다.

```
class 클래스이름 implements 인터페이스이름 {
     // 인터페이스에 정의된 추상메서드를 구현해야한다.
}
```

- 인터페이스에 정의된 추상메서드를 완성해야 한다.

```
class Fighter implements Fightable {
     public void move() { /* 내용 생략*/ }
     public void attack() { /* 내용 생략*/ }
}

interface Fightable {
     void move(int x, int y);
     void attack(Unit u);
}
```

```
abstract class Fighter implements Fightable {
     public void move() { /* 내용 생략*/ }
}
```

- 상속과 구현이 동시에 가능하다.

```
class Fighter extends Unit implements Fightable {
     public void move(int x, int y) { /* 내용 생략 */}
     public void attack(Unit u) { /* 내용 생략 */}
}
```

7.5 인터페이스를 이용한 다형성

- 인터페이스 타입의 변수로 인터페이스를 구현한 클래스의 인스턴스를 참조할 수 있다.

```
class Fighter extends Unit implements Fightable {
     public void move(int x, int y)  { /* 내용 생략 */ }
     public void attack(Fightable f) { /* 내용 생략 */ }
}
```

```
Fighter   f = new Fighter();
Fightable f = new Fighter();
```

- 인터페이스를 메서드의 매개변수 타입으로 지정할 수 있다.

```
void attack(Fightable f) {   // Fightable인터페이스를 구현한 클래스의 인스턴스를
    //...                    // 매개변수로 받는 메서드
}
```

- 인터페이스를 메서드의 리턴타입으로 지정할 수 있다.

```
Fightable method() { // Fightable인터페이스를 구현한 클래스의 인스턴스를 반환
    // ...
    return new Fighter();
}
```

7.6 인터페이스의 장점

1. 개발시간을 단축시킬 수 있다.

일단 인터페이스가 작성되면, 이를 사용해서 프로그램을 작성하는 것이 가능하다. 메서드를 호출하는 쪽에서는 메서드의 내용에 관계없이 선언부만 알면 되기 때문이다.

그리고 동시에 다른 한 쪽에서는 인터페이스를 구현하는 클래스를 작성하도록 하여, 인터페이스를 구현하는 클래스가 작성될 때까지 기다리지 않고도 양쪽에서 동시에 개발을 진행할 수 있다.

2. 표준화가 가능하다.

프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음, 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 함으로써 보다 일관되고 정형화된 프로그램의 개발이 가능하다.

3. 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.

서로 상속관계에 있지도 않고, 같은 조상클래스를 가지고 있지 않은 서로 아무런 관계도 없는 클래스들에게 하나의 인터페이스를 공통적으로 구현하도록 함으로써 관계를 맺어 줄 수 있다.

4. 독립적인 프로그래밍이 가능하다.

인터페이스를 이용하면 클래스의 선언과 구현을 분리시킬 수 있기 때문에 실제구현에 독립적인 프로그램을 작성하는 것이 가능하다.

클래스와 클래스간의 직접적인 관계를 인터페이스를 이용해서 간접적인 관계로 변경하면, 한 클래스의 변경이 관련된 다른 클래스에 영향을 미치지 않는 독립적인 프로그래밍이 가능하다.

7.6 인터페이스의 장점 – 예제

```
interface Repairable {}

class GroundUnit extends Unit {
    GroundUnit(int hp) {
        super(hp);
    }
}

class AirUnit extends Unit {
    AirUnit(int hp) {
        super(hp);
    }
}

class Unit {
    int hitPoint;
    final int MAX_HP;
    Unit(int hp) {
        MAX_HP = hp;
    }
}
```

```
class Tank extends GroundUnit implements Repairable {
    Tank() {
        super(150);      // Tank의 HP는 150이다.
        hitPoint = MAX_HP;
    }

    public String toS
        return "Tank"
    }
}
```

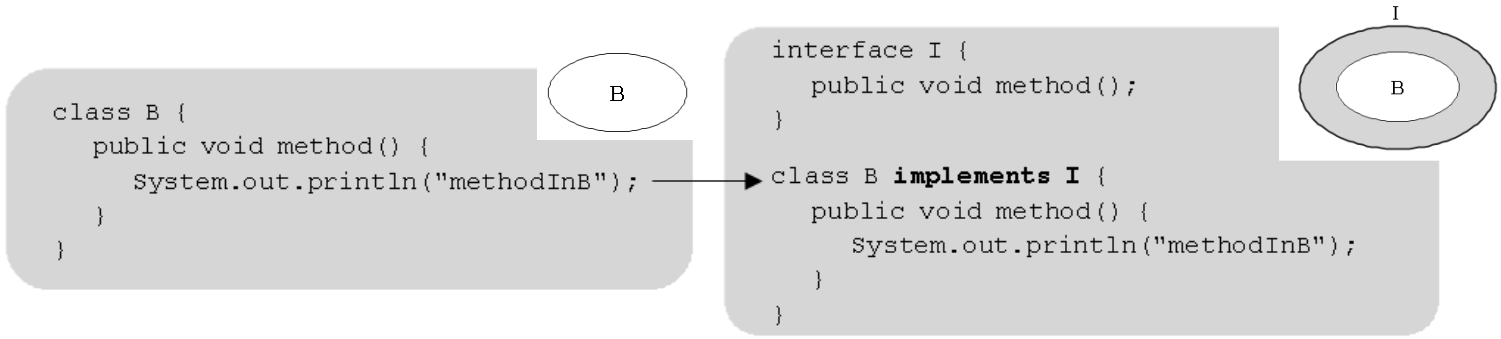
```
class Marine extends GroundUnit {
    Marine() {
        super(40);
        hitPoint = MAX_HP;
    }
}
```

```
class SCV extends GroundUnit implements Repairable{
    SCV() {
        super(60);
        hitPoint = MAX_HP;
    }

    void repair(Repairable r) {
        if (r instanceof Unit) {
            Unit u = (Unit)r;
            while(u.hitPoint!=u.MAX_HP) {
                u.hitPoint++;  // Unit의 HP를 증가시킨다.
            }
        }
    } // repair(Repairable r) {
}
```

```
public static void main(String[] a
    Tank tank = new Tank();
    Marine marine = new Marine();
    SCV scv = new SCV();

    scv.repair(tank);   // SCV가 Tank를 수리한다.
 // scv.repair(marine); // 에러!!!
}
```

7.7 인터페이스의 이해(1/3)

- 인터페이스는...
  - 두 대상(객체) 간의 '연결, 대화, 소통'을 돕는 '중간 역할'을 한다.
  - 선언(설계)와 구현을 분리시키는 것을 가능하게 한다.



- 인터페이스를 이해하려면 먼저 두 가지를 기억하자.
  - 클래스를 사용하는 쪽(User)과 클래스를 제공하는 쪽(Provider)이 있다.
  - 메서드를 사용(호출)하는 쪽(User)에서는 사용하려는 메서드(Provider)의 선언부만 알면 된다.

7.7 인터페이스의 이해(2/3)

▶ 직접적인 관계의 두 클래스(A-B)

```
class A {
	public void methodA(B b) {
		b.methodB();
	}
}
```

```
class B {
	public void methodB() {
		System.out.println("methodB()");
	}
}
```

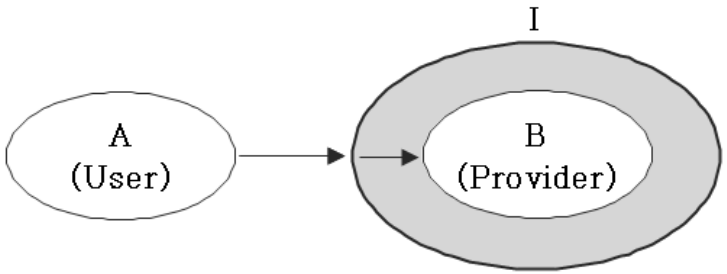
```
class InterfaceTest {
	public static void main(String args[]) {
		A a = new A();
		a.methodA(new B());
	}
}
```

▶ 간접적인 관계의 두 클래스(A-I-B)

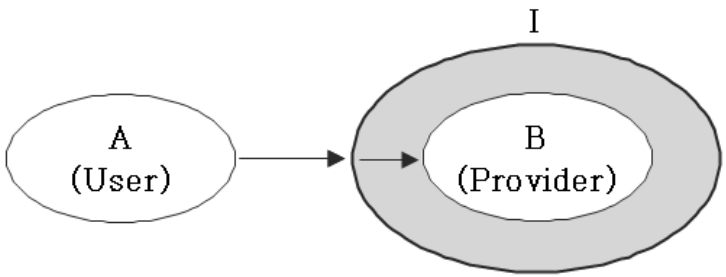
```
class A {
	public void methodA(I i) {
		i.methodB();
	}
}
```

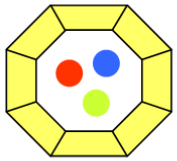
```
interface I { void methodB(); }

class B implements I {
	public void methodB() {
		System.out.println("methodB()");
	}
}
```

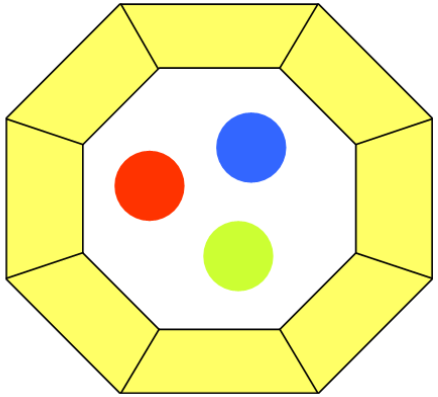
```
class C implements I {
	public void methodB() {
		System.out.println("methodB() in C");
	}
}
```



7.7 인터페이스의 이해(3/3)







```
public class Time {
      private int hour;
      private int minute;
      private int second;

      public int getHour() { return hour; }
      public void setHour(int h) {
            if (h < 0 || h > 23) return;
            hour=h;
      }
      public int getMinute() { return minute; }
      public void setMinute(int m) {
            if (m < 0 || m > 59) return;
            minute=m;
      }
      public int getSecond() { return second; }
      public void setSecond(int s) {
            if (s < 0 || s > 59) return;
            second=s;
      }
}
```

```
public interface TimeIntf {
      public int  getHour();
      public void setHour(int h);

      public int  getMinute();
      public void setMinute(int m);

      public int  getSecond();
      public void setSecond(int s);
}
```

7.8 디폴트 메서드

- 인터페이스에 디폴트 메서드, static메서드를 추가 가능하게 바뀜.(JDK1.8)
- 클래스와 달리 인터페이스에 새로운 메서드(추상메서드)를 추가하기 어려움.
  (해당 인터페이스를 구현한 클래스가 추가된 메서드를 구현하도록 변경필요)
- 이러한 문제점을 해결하기 위해 디폴트 메서드(default method)를 고안
- 디폴트 메서드는 인터페이스에 추가된 일반 메서드(인터페이스 원칙 위반)

```
interface MyInterface {
    void method();
    void newMethod(); // 추상 메서드
}
```

```
interface MyInterface {
    void method();
    default void newMethod(){}
}
```

- 디폴트 메서드가 기존의 메서드와 충돌하는 경우 아래와 같이 해결

1. 여러 인터페이스의 디폴트 메서드 간의 충돌

– 인터페이스를 구현한 클래스에서 디폴트 메서드를 오버라이딩해야 한다.

2. 디폴트 메서드와 조상 클래스의 메서드 간의 충돌

– 조상 클래스의 메서드가 상속되고, 디폴트 메서드는 무시된다.

8. 내부클래스 (inner class)

8.1 내부 클래스(inner class)란?

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

```
class A {
    ...
}

class B {
    ...
}
```



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

▶ 내부 클래스의 장점

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

8.2 내부 클래스의 종류와 특징

- 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 {}
   }
}
```

8.3 내부 클래스의 제어자와 접근성(1/5)

- 내부 클래스의 접근제어자는 변수에 사용할 수 있는 접근제어자와 동일하다.

```
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 {}
    }
}
```

- 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;
    }

    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() {
}
```

```
class InnerTest {
    public static void main(String args[]) {
        System.out.println(InnerEx1.InstanceInner.CONST);
        System.out.println(InnerEx1.StaticInner.cv);
    }
}
```

8.3 내부 클래스의 제어자와 접근성(2/5)

- 내부 클래스도 외부 클래스의 멤버로 간주되며, 동일한 접근성을 갖는다.

```
class InnerEx2 {
   class InstanceInner {}
   static class StaticInner {}

   InstanceInner iv = new InstanceInner();    // 인스턴스멤버 간에는 서로 직접 접근이 가능하다.
   static StaticInner cv = new StaticInner(); // static 멤버 간에는 서로 직접 접근이 가능하다.

   static void staticMethod() {
//     InstanceInner obj1 = new InstanceInner(); // static멤버는 인스턴스멤버에 직접 접근할 수 없다.
       StaticInner obj2 = new StaticInner();

       // 굳이 접근하려면 아래와 같이 객체를 생성해야한다.
       InnerEx2 outer = new InnerEx2();
       InstanceInner obj1 = outer.new InstanceInner();
   }

   void instanceMethod() {
       InstanceInner obj1 = new InstanceInner();
       StaticInner obj2 = new StaticInner();
//     LocalInner lv = new LocalInner();
   }

   void myMethod() {
       class LocalInner {}
       LocalInner lv = new LocalInner();
   }
}
```

인스턴스클래스는 외부 클래스를 먼저 생성해야만 생성할 수 있다.

인스턴스메서드에서는 인스턴스멤버와 static멤버 모두 접근 가능하다.

메서드 내에 지역적으로 선언된 내부 클래스는 외부에서 접근할 수 없다.

8.3 내부 클래스의 제어자와 접근성(3/5)

- 외부 클래스의 지역변수는 final이 붙은 변수(상수)만 접근가능하다.
  지역 클래스의 인스턴스가 소멸된 지역변수를 참조할 수 있기 때문이다.

```
class InnerEx3 {
   private int outerIv = 0;
   static int outerCv = 0;

   class InstanceInner {
        int iiv = outerIv;  // 외부 클래스의 private멤버도 접근가능하다.
        int iiv2 = outerCv;
   }

   static class StaticInner {
// 스태틱 클래스는 외부 클래스의 인스턴스멤버에 접근할 수 없다.
//      int siv = outerIv;
        static int scv = outerCv;
   }

   void myMethod() {
        int lv = 0;
        final int LV = 0;

        class LocalInner {
             int liv = outerIv;
             int liv2 = outerCv;
// 외부 클래스의 지역변수는 final이 붙은 변수(상수)만 접근가능하다.
//           int liv3 = lv;    // 에러!!!
             int liv4 = LV;    // OK
        }
   }
}
```

8.3 내부 클래스의 제어자와 접근성(4/5)

```
class Outer {
   class InstanceInner {
        int iv=100;
   }

   static class StaticInner {
        int iv=200;
        static int cv=300;
   }

   void myMethod() {
        class LocalInner {
            int iv=400;
        }
   }
}
```

```
InnerEx4.class
Outer.class
Outer$InstanceInner.class
Outer$StaticInner.class
Outer$1LocalInner.class
```

```
class InnerEx4 {
   public static void main(String[] args) {
        // 인스턴스클래스의 인스턴스를 생성하려면
        // 외부 클래스의 인스턴스를 먼저 생성해야한다.
        Outer oc = new Outer();
        Outer.InstanceInner ii = oc.new InstanceInner();

        System.out.println("ii.iv : "+ ii.iv);
        System.out.println("Outer.StaticInner.cv : "+ Outer.StaticInner.cv);

        // 스태틱 내부 클래스의 인스턴스는 외부 클래스를 먼저 생성하지 않아도 된다.
        Outer.StaticInner si = new Outer.StaticInner();
        System.out.println("si.iv : "+ si.iv);
   }
}
```

8.3 내부 클래스의 제어자와 접근성(5/5)

```
class Outer {
   int value=10;    // Outer.this.value

   class Inner {
        int value=20;    // this.value
        void method1() {
            int value=30;
            System.out.println("           value :" + value);
            System.out.println("      this.value :" + this.value);
            System.out.println("Outer.this.value :" + Outer.this.value);
        }
   } // Inner클래스의 끝
} // Outer클래스의 끝

class InnerEx5 {
   public static void main(String args[]) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.method1();
   }
} // InnerEx5 끝
```

[실행결과]

```
           value :30
      this.value :20
Outer.this.value :10
```

8.4 익명 클래스(anonymous class)

- 이름이 없는 일회용 클래스. 단 하나의 객체만을 생성할 수 있다.

```
new 조상클래스이름() {
   // 멤버 선언
}

   또는

new 구현인터페이스이름() {
   // 멤버 선언
}
```

[예제10-6]/ch10/InnerEx6.java

```
class InnerEx6 {
   Object iv = new Object(){ void method(){} };         // 익명클래스
   static Object cv = new Object(){ void method(){} };// 익명클래스

   void myMethod() {
       Object lv = new Object(){ void method(){} };    // 익명클래스
   }
}
```

```
InnerEx6.class
InnerEx6$1.class ← 익명클래스
InnerEx6$2.class ← 익명클래스
InnerEx6$3.class ← 익명클래스
```

8.4 익명 클래스(anonymous class) - 예제

```
import java.awt.*;
import java.awt.event.*;

class  InnerEx7{
   public static void main(String[] args) {
        Button b = new Button("Start");
        b.addActionListener(new EventHandler());
   }
}

class EventHandler implements ActionListener {
   public void actionPerformed(ActionEvent e) {
        System.out.println("ActionEvent occurred!!!");
   }
}
```

```
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
```

감사합니다.

더 많은 동영상강좌를 아래의 사이트에서 구하실 수 있습니다.

http://www.javachobo.com

관련문의 : 남궁성 castello@naver.com