

자바 프로그래밍 기초

임성국 (eventia@gmail.com)

자바 : 클래스와 객체



각체지향개념 II-1

- 1. 상속
- 2. 오버라이딩
- 3. package와 import

객체지향개념 II-1

- 4. 제어자
- 5. 다형성

객체지향개념 II-2

- 6. 추상클래스
- 7. 인터페이스

객체지향개념 II-3

1. 상속(inheritance)

- 1.1 상속의 정의와 장점
- 1.2 클래스간의 관계
- 1.3 클래스간의 관계결정하기
- 1.4 단일 상속(single inheritance)
- 1.5 Object클래스

2. 오버라이딩(overriding)

- 2.1 오버라이딩이란?
- 2.2 오버라이딩의 조건
- 2.3 오버로딩 vs. 오버라이딩
- 2.4 super
- 2.5 super()

3. package와 import

- 3.1 패키지(package)
- 3.2 패키지의 선언
- 3.3 클래스패스 설정
- 3.4 import문
- 3.5 import문의 선언

1. 상속(inheritance)

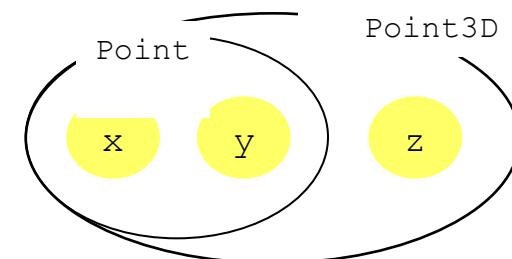
1.1 상속(inheritance)의 정의와 장점

▶ 상속이란?

- 기존의 클래스를 재사용해서 새로운 클래스를 작성하는 것.
- 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 {  
    int x;  
    int y;  
    int z;  
}  
  
class 자손클래스 extends 조상클래스 {  
    // ...  
}
```

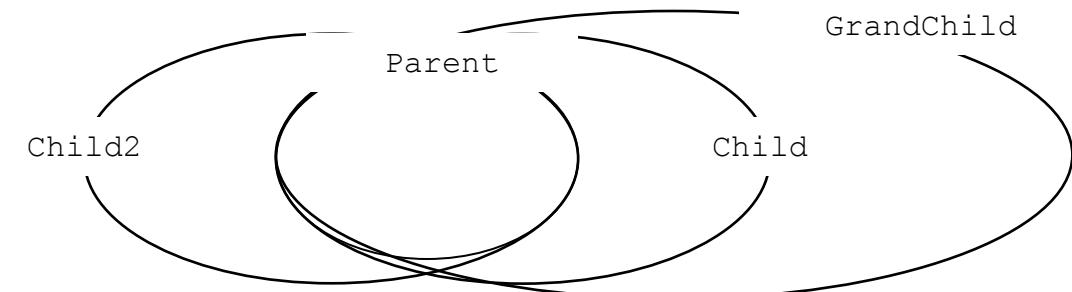
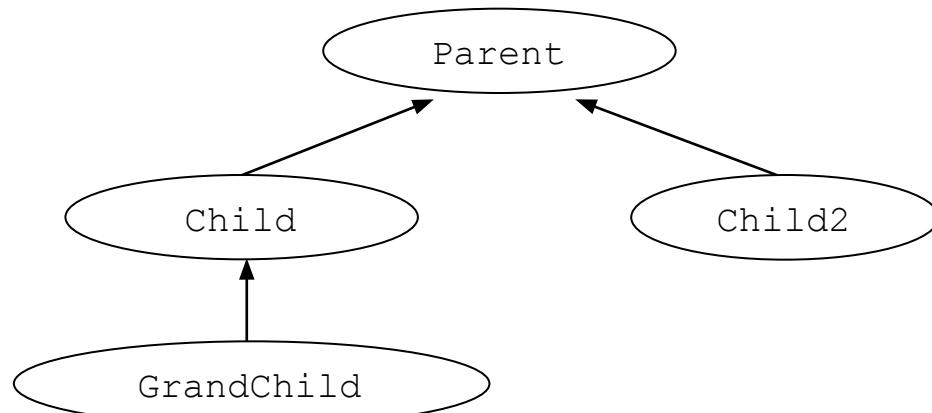
```
class Point3D extends Point {  
    int z;  
}
```



1.2 클래스간의 관계 – 상속관계(inheritance)

- 공통부분은 조상에서 관리하고 개별부분은 자손에서 관리한다.
- 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 {}
```



1.2 클래스간의 관계 – 포함관계(composite)

▶ 포함(composite)이란?

- 한 클래스의 멤버변수로 다른 클래스를 선언하는 것
- 작은 단위의 클래스를 먼저 만들고, 이들을 조합해서 하나의 커다란 클래스를 만든다.

```
class Circle {  
    int x; // 원점의 x좌표  
    int y; // 원점의 y좌표  
    int r; // 반지름(radius)  
}
```

```
class Circle {  
    Point c = new Point(); // 원점  
    int r; // 반지름(radius)  
}
```

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

원점

```
class Car {  
    Engine e = new Engine(); // 엔진  
    Door[] d = new Door[4]; // 문, 문의 개수를 넷으로 가정하고 배열로 처리했다.  
    //...  
}
```

1.3 클래스간의 관계결정하기 – 상속 vs. 포함

- 가능한 한 많은 관계를 맺어주어 재사용성을 높이고 관리하기 쉽게 한다.
- '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.

상속관계 – '~은 ~이다.(is-a)'

포함관계 – '~은 ~을 가지고 있다.(has-a)'

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

```
class Circle extends Point{  
    int r; // 반지름(radius)  
}
```

```
class Circle {  
    Point c = new Point(); // 원점  
    int r; // 반지름(radius)  
}
```

1.3 클래스간의 관계결정하기 – 예제설명

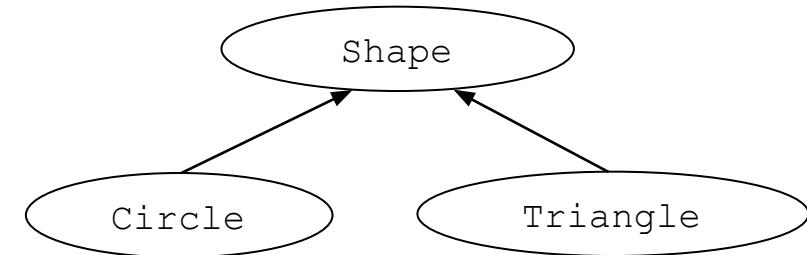
- 원(Circle)은 도형(Shape)이다.(A Circle is a Shape.) : 상속관계
- 원(Circle)은 점(Point)을 가지고 있다.(A Circle has a Point.) : 포함관계

```
class Shape {  
    String color = "blue";  
    void draw() {  
        // 도형을 그린다.  
    }  
}
```

```
class Point {  
    int x;  
    int y;  
  
    Point() {  
        this(0,0);  
    }  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
class Circle extends Shape {  
    Point center;  
    int r;  
  
    Circle() {  
        this(new Point(0,0),100);  
    }  
  
    Circle(Point center, int r) {  
        this.center = center;  
        this.r = r;  
    }  
  
    class Triangle extends Shape {  
        Point[] p;  
  
        Triangle(Point[] p) {  
            this.p = p;  
        }  
  
        Triangle(Point p1, Point p2, Point p3) {  
            p = new Point[]{p1,p2,p3};  
        }  
    }  
}
```

```
Circle c1 = new Circle();  
Circle c2 = new Circle(new Point(150,150),50);  
  
Point[] p = {new Point(100,100),  
            new Point(140,50),  
            new Point(200,100)};  
Triangle t1 = new Triangle(p);
```



1.3 클래스간의 관계결정하기 – 예제설명2

```
class Deck {  
    final int CARD_NUM = 52;      // 카드의 개수  
    Card c[] = new Card[CARD_NUM];  
  
    Deck () { // Deck의 카드를 초기화한다.  
        int i=0;  
  
        for(int k=Card.KIND_MAX; k > 0; k--) {  
            for(int n=1; n < Card.NUM_MAX + 1 ; n++) {  
                c[i++] = new Card(k, n);  
            }  
        }  
    }  
  
    Card pick(int index) { // 지정된 위치(index)에 있는 카드 하나를 선택한다.  
        return c[index%CARD_NUM];  
    }  
  
    Card pick() { // Deck에서 카드 하나를 선택한다.  
        int index = (int)(Math.random() * CARD_NUM);  
        return pick(index);  
    }  
  
    void shuffle() { // 카드의 순서를 섞는다.  
        for(int n=0; n < 1000; n++) {  
            int i = (int)(Math.random() * CARD_NUM);  
            Card temp = c[0];  
            c[0] = c[i];  
            c[i] = temp;  
        }  
    }  
} // Deck클래스의 끝
```

```
public static void main(String[] args) {  
    Deck d = new Deck();  
    Card c = d.pick();  
  
    d.shuffle();  
    Card c2 = d.pick(55);  
}
```

1.4 단일상속(single inheritance)

- Java는 단일상속만을 허용한다.(C++은 다중상속 허용)

```
class TVCR extends TV, VCR {      // 이와 같은 표현은 허용하지 않는다.  
    //...  
}
```

- 비중이 높은 클래스 하나만 상속관계로, 나머지는 포함관계로 한다.

```
class TV {  
    boolean power; // 전원상태 (on/off)  
    int channel; // 채널  
  
    void power() { power = !power; }  
    void channelUp() { ++channel; }  
    void channelDown() { --channel; }  
}
```

```
class VCR {  
    boolean power; // 전원상태 (on/off)  
    int counter = 0;  
    void power() { power = !power; }  
    void play() { /* 내용생략 */ }  
    void stop() { /* 내용생략 */ }  
    void rew() { /* 내용생략 */ }  
    void ff() { /* 내용생략 */ }  
}
```

상속

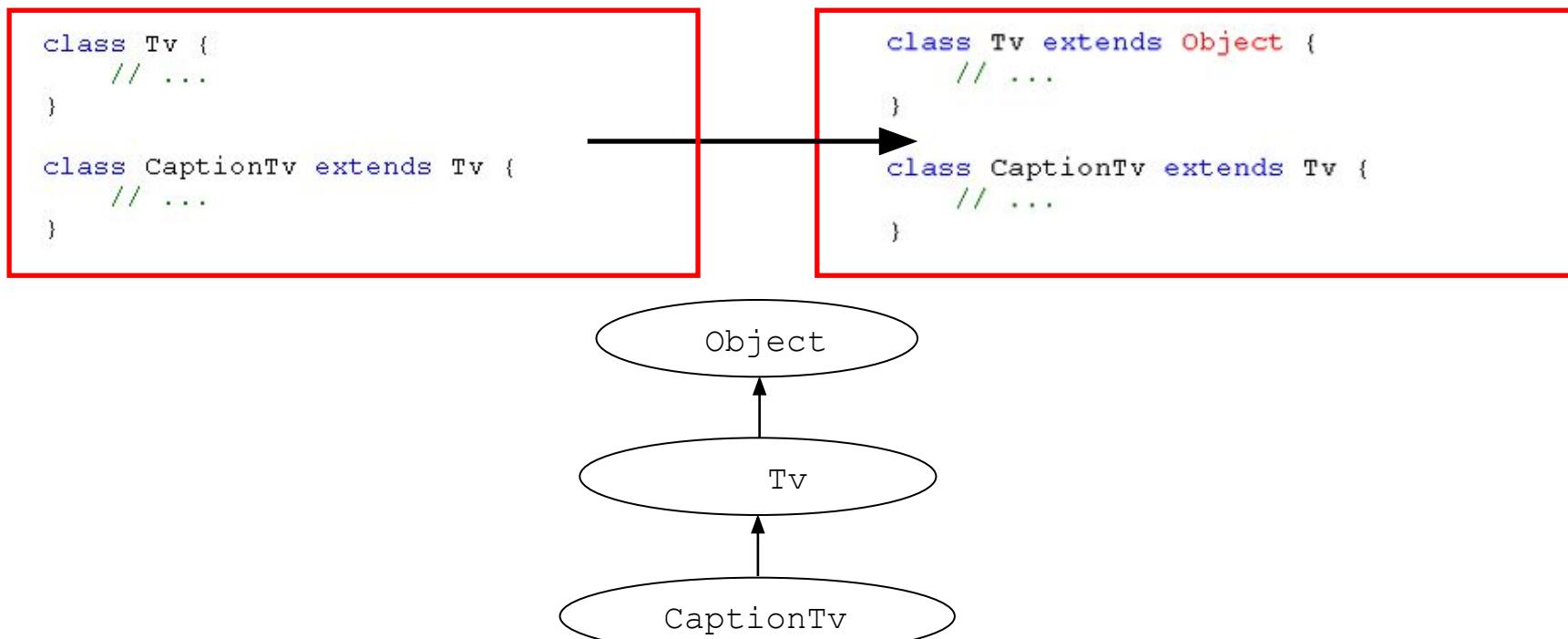
포함

```
class TVCR extends TV {  
    VCR vcr = new VCR();  
    int counter = vcr.counter;  
  
    void play() {  
        vcr.play();  
    }  
  
    void stop() {  
        vcr.stop();  
    }  
  
    void rew() {  
        vcr.rew();  
    }  
  
    void ff() {  
        vcr.ff();  
    }  
}
```

1.5 Object클래스 – 모든 클래스의 최고조상

- 조상이 없는 클래스는 자동적으로 Object클래스를 상속받게 된다.
- 상속계층도의 최상위에는 Object클래스가 위치한다.
- 모든 클래스는 Object클래스에 정의된 11개의 메서드를 상속받는다.

toString(), equals(Object obj), hashCode(), ...



2. 오버라이딩(overriding)

2.1 오버라이딩(overriding)이란?

“조상클래스로부터 상속받은 메서드의 내용을 상속받는 클래스에 맞게 변경하는 것을 오버라이딩이라고 한다.”

* override - vt. ‘~위에 덮어쓰다(overwrite).’, ‘~에 우선하다.’

```
class Point {  
    int x;  
    int y;  
  
    String getLocation() {  
        return "x :" + x + ", y :" + y;  
    }  
}  
  
class Point3D extends Point {  
    int z;  
    String getLocation() {      // 오버라이딩  
        return "x :" + x + ", y :" + y + ", z :" + z;  
    }  
}
```

2.2 오버라이딩의 조건

1. 선언부가 같아야 한다.(이름, 매개변수, 리턴타입)
2. 접근제어자를 좁은 범위로 변경할 수 없다.
 - 조상의 메서드가 **protected**라면, 범위가 같거나 넓은 **protected**나 **public**으로만 변경할 수 있다.
3. 조상클래스의 메서드보다 많은 수의 예외를 선언할 수 없다.

```
class Parent {  
    void parentMethod() throws IOException, SQLException {  
        // ...  
    }  
}  
  
class Child extends Parent {  
    void parentMethod() throws IOException {  
        //..  
    }  
}  
  
class Child2 extends Parent {  
    void parentMethod() throws Exception {  
        //..  
    }  
}
```

오류

2.3 오버로딩 vs. 오버라이딩

오버로딩(over loading) - 기존에 없는 새로운 메서드를 정의하는 것(new)

오버라이딩(over riding) - 상속받은 메서드의 내용을 변경하는 것(change, modify)

```
class Parent {  
    void parentMethod() {}  
}  
  
class Child extends Parent {  
    void parentMethod() {}      // 오버라이딩  
    void parentMethod(int i) {} // 오버로딩  
  
    void childMethod() {}  
    void childMethod(int i) {} // 오버로딩  
    void childMethod() {}     // 에러!!! 중복정의임  
}
```

2.4 super – 참조변수(1/2)

- ▶ **this** – 인스턴스 자신을 가리키는 참조변수. 인스턴스의 주소가 저장되어있음
모든 인스턴스 메서드에 지역변수로 숨겨진 채로 존재
- ▶ **super – this**와 같음. 조상의 멤버와 자신의 멤버를 구별하는 데 사용.

```
class Parent {  
    int x=10;  
}  
  
class Child extends Parent {  
    int x=20;  
    void method() {  
        System.out.println("x=" + x);  
        System.out.println("this.x=" + this.x);  
        System.out.println("super.x=" + super.x);  
    }  
}  
  
class Parent {  
    int x=10;  
}  
  
class Child extends Parent {  
    void method() {  
        System.out.println("x=" + x);  
        System.out.println("this.x=" + this.x);  
        System.out.println("super.x=" + super.x);  
    }  
}  
  
public static void main(String args[]) {  
    Child c = new Child();  
    c.method();  
}
```

2.4 super – 참조변수(2/2)

- ▶ **this** – 인스턴스 자신을 가리키는 참조변수. 인스턴스의 주소가 저장되어있음
모든 인스턴스 메서드에 지역변수로 숨겨진 채로 존재
- ▶ **super – this**와 같음. 조상의 멤버와 자신의 멤버를 구별하는 데 사용.

```
class Point {  
    int x;  
    int y;  
  
    String getLocation() {  
        return "x :" + x + ", y :" + y;  
    }  
}  
  
class Point3D extends Point {  
    int z;  
    String getLocation() { // 오버라이딩  
        // return "x :" + x + ", y :" + y + ", z :" + z;  
        return super.getLocation() + ", z :" + z; // 조상의 메서드 호출  
    }  
}
```

2.5 super() – 조상의 생성자(1/3)

- 자손클래스의 인스턴스를 생성하면, 자손의 멤버와 조상의 멤버가 합쳐진 하나의 인스턴스가 생성된다.
- 조상의 멤버들도 초기화되어야 하기 때문에 자손의 생성자의 첫 문장에서 조상의 생성자를 호출해야 한다.

Object클래스를 제외한 모든 클래스의 생성자 첫 줄에는 생성자(같은 클래스의 다른 생성자 또는 조상의 생성자)를 호출해야 한다.

그렇지 않으면 컴파일러가 자동적으로 'super();'를 생성자의 첫 줄에 삽입한다.

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

```
Point() {  
    this(0, 0);  
}
```

```
Point(int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

```
class Point extends Object {  
    int x;  
    int y;
```

```
Point() {  
    this(0, 0);  
}
```

```
Point(int x, int y) {  
    super(); // Object()  
    this.x = x;  
    this.y = y;  
}
```



2.5 super() – 조상의 생성자(2/3)

```
class Point {  
    int x;  
    int y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    String getLocation() {  
        return "x :" + x + ", y :" + y;  
    }  
}  
  
class Point3D extends Point {  
    int z;  
  
    Point3D(int x, int y, int z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
  
    String getLocation() { // 오버라이딩  
        return "x :" + x + ", y :" + y + ", z :" + z;  
    }  
}
```

```
class PointTest {  
    public static void main(String args[]) {  
        Point3D p3 = new Point3D(1, 2, 3);  
    }  
}
```

```
----- javac -----  
PointTest.java:24: cannot find symbol  
symbol : constructor Point()  
location: class Point  
    Point3D(int x, int y, int z) {  
           ^  
1 error
```

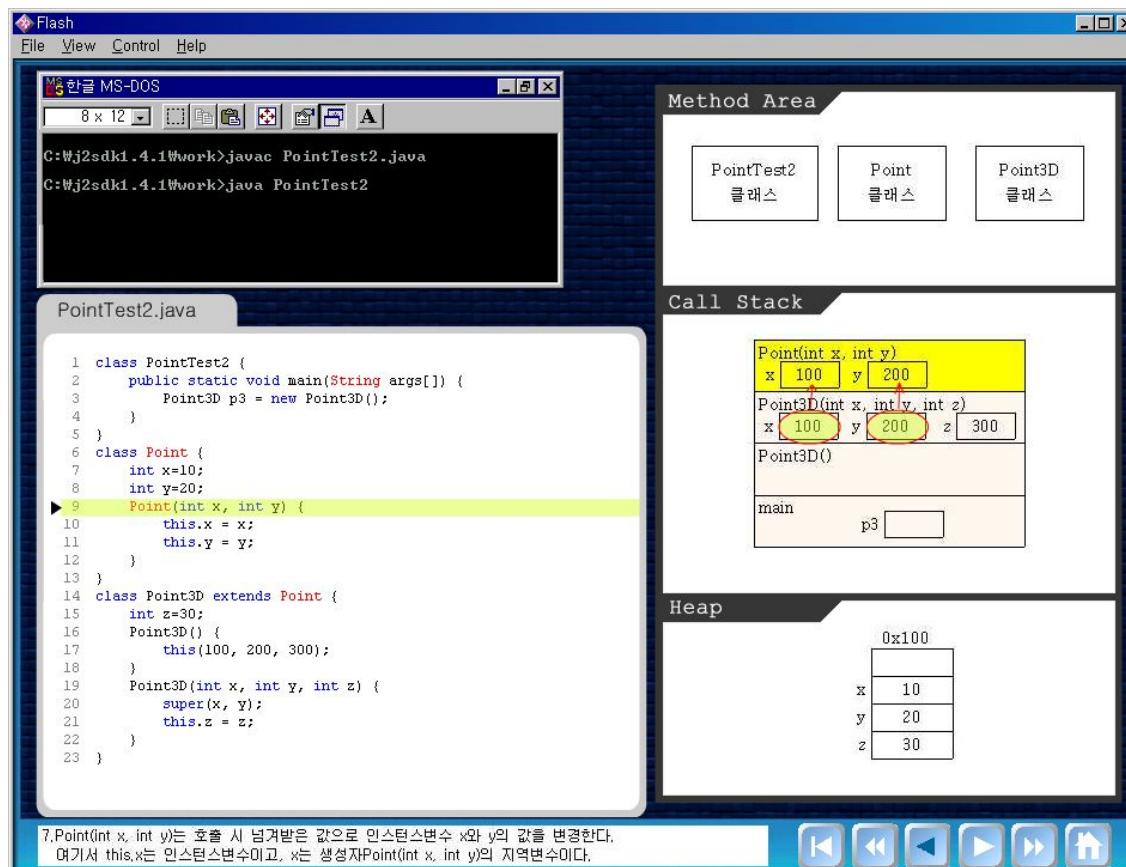
```
Point3D(int x, int y, int z) {  
    super(); // Point() 를 호출  
    this.x = x;  
    this.y = y;  
    this.z = z;  
}
```

```
Point3D(int x, int y, int z) {  
    // 조상의 생성자 Point(int x, int y) 를 호출  
    super(x, y);  
    this.z = z;  
}
```

2.5 super() – 조상의 생성자(3/3)

* 플래시 동영상 : Super.exe 또는 Super.swf

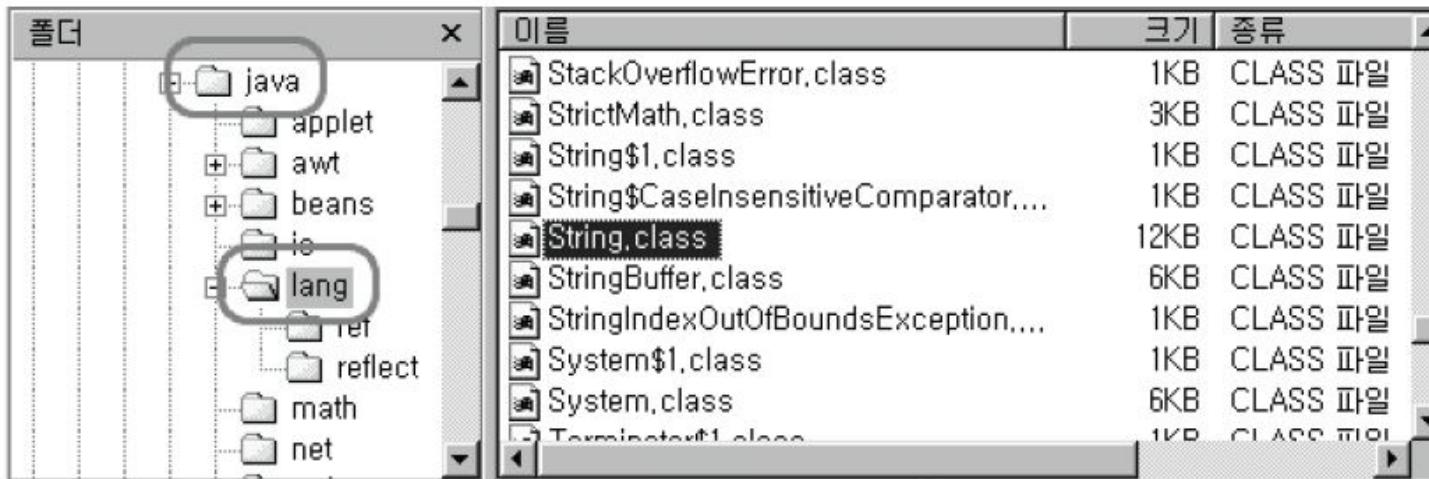
(java_jungsuk_src.zip의 flash폴더에 위치)



3. package와 import

3.1 패키지(package)

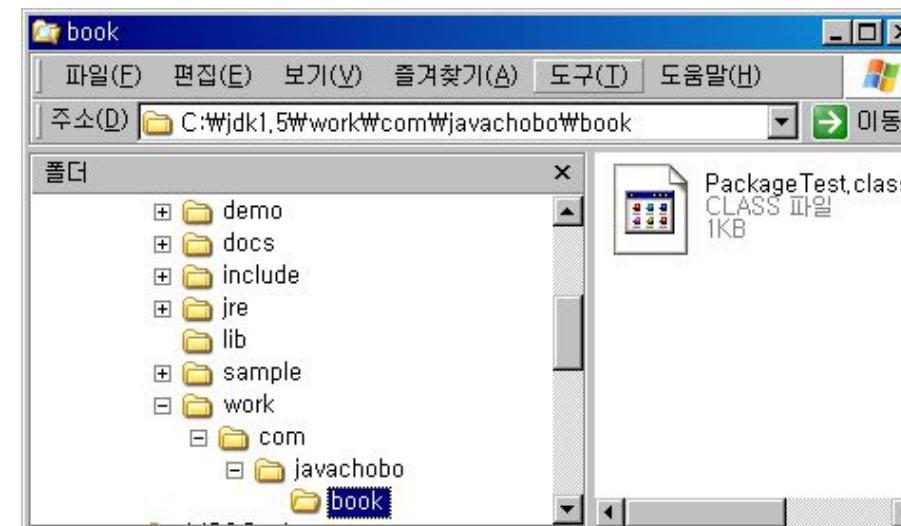
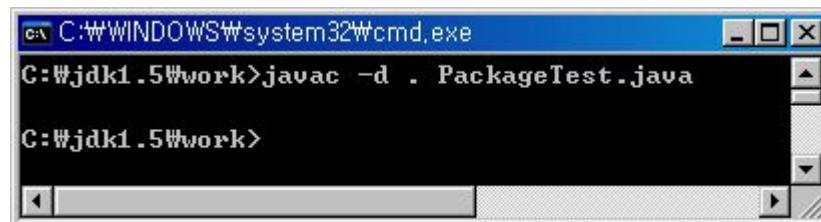
- 서로 관련된 클래스와 인터페이스의 묶음.
- 클래스가 물리적으로 클래스파일(*.class)인 것처럼, 패키지는 물리적으로 폴더이다. 패키지는 서브패키지를 가질 수 있으며, '.'으로 구분한다.
- 클래스의 실제 이름(full name)은 패키지명이 포함된 것이다.
(String 클래스의 full name은 java.lang.String)
- rt.jar는 Java API의 기본 클래스들을 압축한 파일
(JDK 설치 경로\jre\lib에 위치)



3.2 패키지의 선언

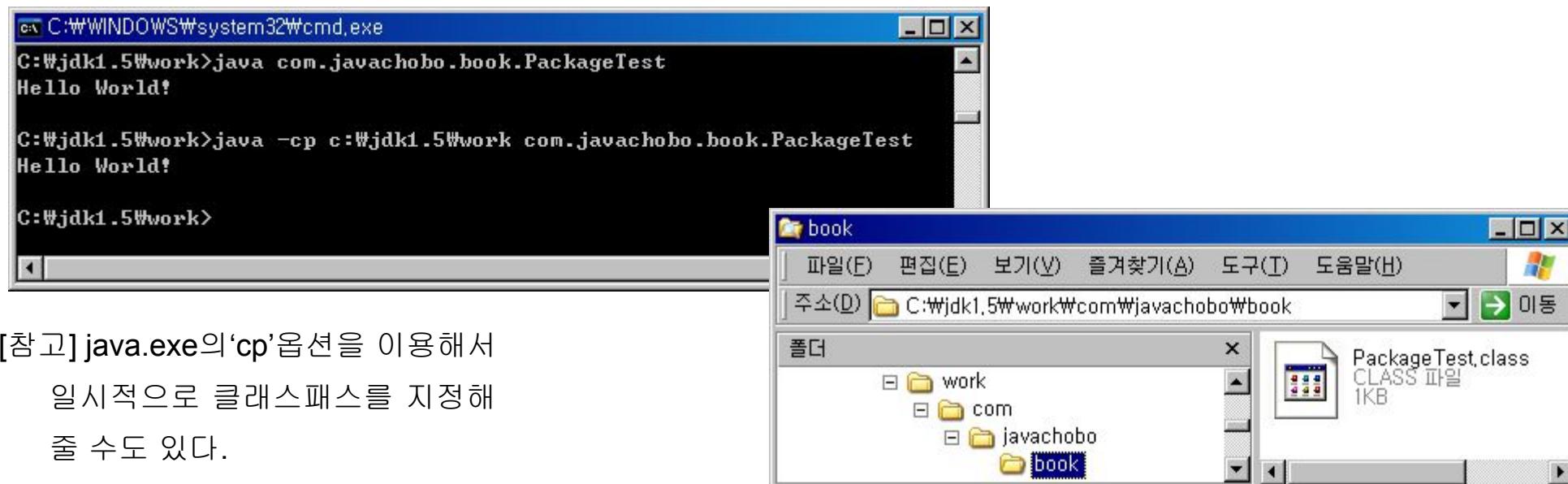
- 패키지는 소스파일에 첫 번째 문장(주석 제외)으로 단 한번 선언한다.
- 하나의 소스파일에 둘 이상의 클래스가 포함된 경우, 모두 같은 패키지에 속하게 된다.(하나의 소스파일에 단 하나의 **public**클래스만 허용한다.)
- 모든 클래스는 하나의 패키지에 속하며, 패키지가 선언되지 않은 클래스는 자동적으로 이름없는(unnamed) 패키지에 속하게 된다.

```
1 // PackageTest.java
2 package com.javachobo.book;
3
4 public class PackageTest {
5     public static void main(String[] args) {
6         System.out.println("Hello World!");
7     }
8 }
9
10 public class PackageTest2 {}
```



3.3 클래스패스(classpath) 설정(1/2)

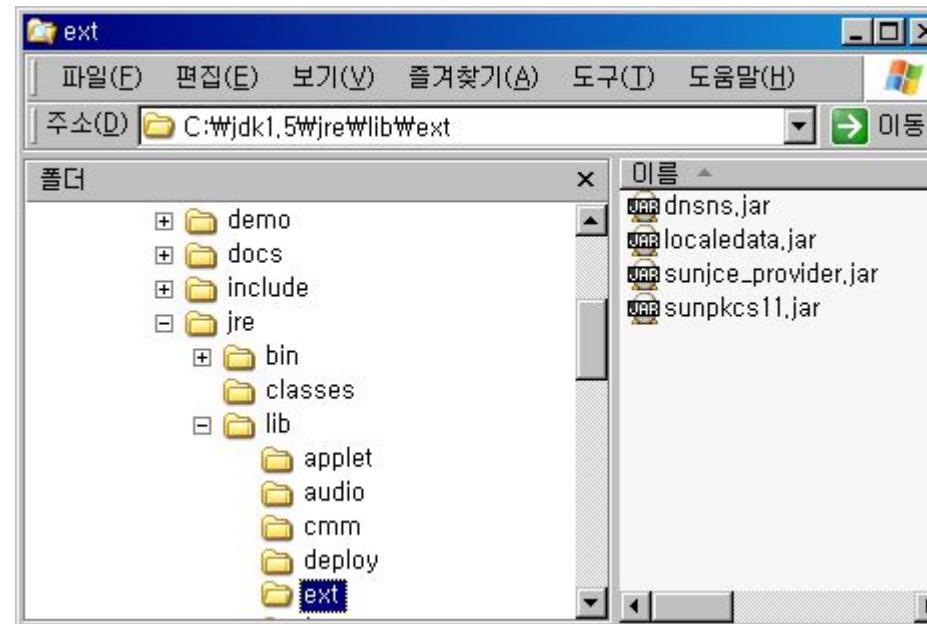
- 클래스패스(classpath)는 클래스파일(*.class)를 찾는 경로. 구분자는 ‘,’
- 클래스패스에 패키지가 포함된 폴더나 jar파일을(*.jar) 나열한다.
- 클래스패스가 없으면 자동적으로 현재 폴더가 포함되지만
클래스패스를 지정할 때는 현재 폴더(.)도 함께 추가해주어야 한다.



[참고] java.exe의 'cp' 옵션을 이용해서
일시적으로 클래스패스를 지정해
줄 수도 있다.

3.3 클래스패스(classpath) 설정(2/2)

- ▶ 클래스패스로 자동 포함된 폴더 for 클래스파일(*.class) : 수동생성 해야함.
 - JDK설치경로\jre\classes
- ▶ 클래스패스로 자동 포함된 폴더 for jar파일(*.jar) : JDK설치시 자동생성됨.
 - JDK설치경로\jre\lib\ext



3.4 import문

- 사용할 클래스가 속한 패키지를 지정하는데 사용.
- import문을 사용하면 클래스를 사용할 때 패키지명을 생략할 수 있다.

```
class ImportTest {  
    java.util.Date today = new java.util.Date();  
    // ...  
}
```

```
import java.util.*;  
  
class ImportTest {  
    Date today = new Date();  
}
```

- java.lang 패키지의 클래스는 import하지 않고도 사용할 수 있다.

String, Object, System, Thread ...

```
import java.lang.*;  
  
class ImportTest2  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World!");  
    }  
}
```

```
public static void main(java.lang.String[] args)  
{  
    java.lang.System.out.println("Hello World!");  
}
```

3.5 import문의 선언

- import문은 패키지문과 클래스선언의 사이에 선언한다.

일반적인 소스파일 (*.java)의 구성은 다음의 순서로 되어 있다.

- ① package문
- ② import문
- ③ 클래스 선언

- import문을 선언하는 방법은 다음과 같다.

import 패키지명.클래스명;
또는
import 패키지명.*;

```
1 package com.javachobo.book;
2
3 import java.text.SimpleDateFormat;
4 import java.util.*;
5
6 public class PackageTest {
7     public static void main(String[] args) {
8         // java.util.Date today = new java.util.Date();
9         Date today = new Date();
10        SimpleDateFormat date = new SimpleDateFormat("yyyy/MM/dd");
11    }
12 }
```

3.5 import문의 선언 - 선언 예

- import문은 컴파일 시에 처리되므로 프로그램의 성능에 아무런 영향을 미치지 않는다.

```
import java.util.Calendar;  
import java.util.Date;  
import java.util.ArrayList;
```

```
import java.util.*;
```

- 다음의 두 코드는 서로 의미가 다르다.

```
import java.util.*;  
import java.text.*;
```

```
import java.*;
```

- 이름이 같은 클래스가 속한 두 패키지를 import할 때는 클래스 앞에 패키지명을 붙여줘야 한다.

```
import java.sql.*; // java.sql.Date  
import java.util.*; // java.util.Date  
  
public class ImportTest {  
    public static void main(String[] args) {  
        java.util.Date today = new java.util.Date();  
    }  
}
```

액체지향개념 II-2

- 1. 상속
- 2. 오버라이딩
- 3. package와 import

객체지향개념 II-1

- 4. 제어자
- 5. 다형성

객체지향개념 II-2

- 6. 추상클래스
- 7. 인터페이스

객체지향개념 II-3

4. 제어자(modifiers)

- 4.1 제어자란?
- 4.2 static
- 4.3 final
- 4.4 생성자를 이용한 **final** 멤버변수 초기화
- 4.5 abstract
- 4.6 접근 제어자
- 4.7 접근 제어자를 이용한 캡슐화
- 4.8 생성자의 접근 제어자
- 4.9 제어자의 조합

5. 다형성(polymorphism)

- 5.1 다형성이란?
- 5.2 참조변수의 형변환
- 5.3 instanceof 연산자
- 5.4 참조변수와 인스턴스변수의 연결
- 5.5 매개변수의 다형성
- 5.6 여러 종류의 객체를 하나의 배열로 다루기

4. 제어자(modifiers)

4.1 제어자(modifier)란?

- 클래스, 변수, 메서드의 선언부에 사용되어 부가적인 의미를 부여한다.
- 제어자는 크게 접근 제어자와 그 외의 제어자로 나뉜다.
- 하나의 대상에 여러 개의 제어자를 조합해서 사용할 수 있으나,
접근제어자는 단 하나만 사용할 수 있다.

접근 제어자 - public, protected, default, private

그 외 - static, final, abstract, native, transient, synchronized,
volatile, strictfp

4.2 static – 클래스의, 공통적인

static이 사용될 수 있는 곳 - 멤버변수, 메서드, 초기화 블럭

제어자	대상	의 미
static	멤버변수	<ul style="list-style-type: none">- 모든 인스턴스에 공통적으로 사용되는 클래스변수가 된다.- 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다.- 클래스가 메모리에 로드될 때 생성된다.
	메서드	<ul style="list-style-type: none">- 인스턴스를 생성하지 않고도 호출이 가능한 static 메서드가 된다.- static메서드 내에서는 인스턴스멤버들을 직접 사용할 수 없다.

```
class StaticTest {
    static int width = 200;
    static int height = 120;

    static { // 클래스 초기화 블럭
        // static변수의 복잡한 초기화 수행
    }

    static int max(int a, int b) {
        return a > b ? a : b;
    }
}
```

4.3 final – 마지막의, 변경될 수 없는

final이 사용될 수 있는 곳 - 클래스, 메서드, 멤버변수, 지역변수

제어자	대상	의 미
final	클래스	변경될 수 없는 클래스, 확장될 수 없는 클래스가 된다. 그래서 final로 지정된 클래스는 다른 클래스의 조상이 될 수 없다.
	메서드	변경될 수 없는 메서드, final로 지정된 메서드는 오버라이딩을 통해 재정의 될 수 없다.
	멤버변수	
	지역변수	변수 앞에 final이 붙으면, 값을 변경할 수 없는 상수가 된다.

[참고] 대표적인 final클래스로는 String과 Math가 있다.

```
final class FinalTest {  
    final int MAX_SIZE = 10; // 멤버변수  
  
    final void getMaxSize() {  
        final LV = MAX_SIZE; // 지역변수  
        return MAX_SIZE;  
    }  
}  
  
class Child extends FinalTest {  
    void getMaxSize() {} // 오버라이딩  
}
```

4.4 생성자를 이용한 final 멤버변수 초기화

- `final`이 붙은 변수는 상수이므로 보통은 선언과 초기화를 동시에 하지만, 인스턴스변수의 경우 생성자에서 초기화 할 수 있다.

```
class Card {  
    final int NUMBER;          // 상수지만 선언과 함께 초기화 하지 않고  
    final String KIND;         // 생성자에서 단 한번만 초기화할 수 있다.  
    static int width = 100;  
    static int height = 250;  
  
    Card(String kind, int num) {  
        KIND = kind;  
        NUMBER = num;  
    }  
    Card() {  
        this("HEART", 1);  
    }  
  
    public String toString() {  
        return "" + KIND + " " + NUMBER;  
    }  
}  
  
public static void main(String args[]) {  
    Card c = new Card("HEART", 10);  
    // c.NUMBER = 5;      에러!!!  
    System.out.println(c.KIND);  
    System.out.println(c.NUMBER);  
}
```

4.5 abstract – 추상의, 미완성의

abstract가 사용될 수 있는 곳 – 클래스, 메서드

제어자	대상	의 미
abstract	클래스	클래스 내에 추상메서드가 선언되어 있음을 의미한다.
	메서드	선언부만 작성하고 구현부는 작성하지 않은 추상메서드임을 알린다.

[참고] 추상메서드가 없는 클래스도 abstract를 붙여서 추상클래스로 선언하는 것이 가능하기는 하지만 그렇게 해야 할 이유는 없다.

```
abstract class AbstractTest { // 추상클래스
    abstract void move(); // 추상메서드
}
```

4.6 접근 제어자(access modifier)

- 멤버 또는 클래스에 사용되어, 외부로부터의 접근을 제한한다.

접근 제어자가 사용될 수 있는 곳 - 클래스, 멤버변수, 메서드, 생성자

private - 같은 클래스 내에서만 접근이 가능하다.

default - 같은 패키지 내에서만 접근이 가능하다.

protected - 같은 패키지 내에서, 그리고 다른 패키지의 자손클래스에서
접근이 가능하다.

public - 접근 제한이 전혀 없다.

제어자	같은 클래스	같은 패키지	자손클래스	전체
public				
protected				
default				
private				

public
(default)

public
protected
(default)
private

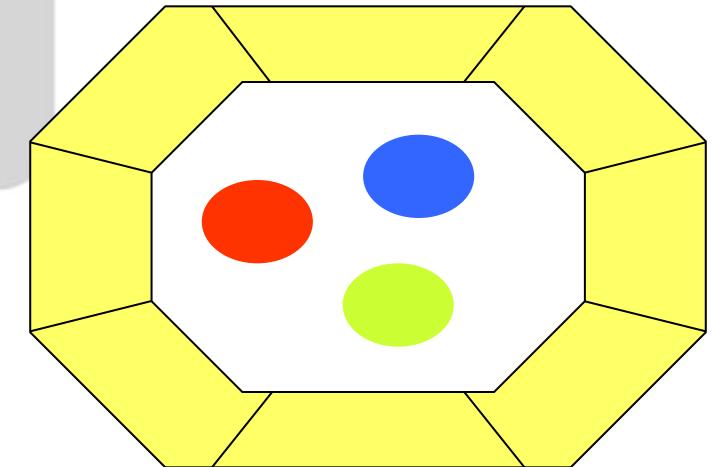
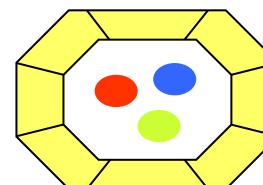
```
class AccessModifierTest {  
    int iv;      // 멤버변수(인스턴스변수)  
    static int cv; // 멤버변수(클래스변수)  
  
    void method() {}  
}
```

4.7 접근 제어자를 이용한 캡슐화

접근 제어자를 사용하는 이유

- 외부로부터 데이터를 보호하기 위해서
- 외부에는 불필요한, 내부적으로만 사용되는, 부분을 감추기 위해서

```
class Time {  
    private int hour;  
    private int minute;  
    private int second;  
  
    Time(int hour, int minute, int second) {  
        setHour(hour);  
        setMinute(minute);  
        setSecond(second);  
    }  
  
    public int getHour() { return hour; }  
  
    public void setHour(int hour) {  
        if (hour < 0 || hour > 23) return;  
        this.hour = hour;  
    }  
    ... 중간 생략 ...  
  
    public String toString() {  
        return hour + ":" + minute + ":" + second;  
    }  
}
```



```
public static void main(String[] args) {  
    Time t = new Time(12, 35, 30);  
    // System.out.println(t.toString());  
    System.out.println(t);  
    // t.hour = 13; 에러!!!  
  
    // 현재시간보다 1시간 후로 변경한다.  
    t.setHour(t.getHour() + 1);  
    System.out.println(t);  
}
```

```
----- java -----  
12:35:30  
13:35:30
```

출력 완료 (0초 경과)

4.8 생성자의 접근 제어자

- 일반적으로 생성자의 접근 제어자는 클래스의 접근 제어자와 일치한다.
- 생성자에 접근 제어자를 사용함으로써 인스턴스의 생성을 제한할 수 있다.

```
final class Singleton {  
    private static Singleton s = new Singleton();  
  
    private Singleton() { // 생성자  
        //...  
    }  
  
    public static Singleton getInstance() {  
        if(s==null) {  
            s = new Singleton();  
        }  
        return s;  
    }  
    //...  
}
```

getInstance()에서 사용될 수 있도록 인스턴스가 미리 생성되어야 하므로 static이어야 한다.

```
class SingletonTest {  
    public static void main(String args[]) {  
        // Singleton s = new Singleton(); 예러!!!  
        Singleton s1 = Singleton.getInstance();  
    }  
}
```

4.9 제어자의 조합

대상	사용가능한 제어자
클래스	public, (default), final, abstract
메서드	모든 접근 제어자, final, abstract, static
멤버변수	모든 접근 제어자, final, static
지역변수	final

1. 메서드에 **static**과 **abstract**를 함께 사용할 수 없다.

- static메서드는 봄통(구현부)이 있는 메서드에만 사용할 수 있기 때문이다.

2. 클래스에 **abstract**와 **final**을 동시에 사용할 수 없다.

- 클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미이고, abstract는 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.

3. **abstract**메서드의 접근제어자가 **private**일 수 없다.

- abstract메서드는 자손클래스에서 구현해주어야 하는데 접근 제어자가 private이면, 자손클래스에서 접근할 수 없기 때문이다.

4. 메서드에 **private**과 **final**을 같이 사용할 필요는 없다.

- 접근 제어자가 private인 메서드는 오버라이딩될 수 없기 때문이다. 이 둘 중 하나만 사용해도 의미가 충분하다.

5. 다형성(polymorphism)

5.1 다형성(polymorphism)이란?(1/3)

-“여러 가지 형태를 가질 수 있는 능력”

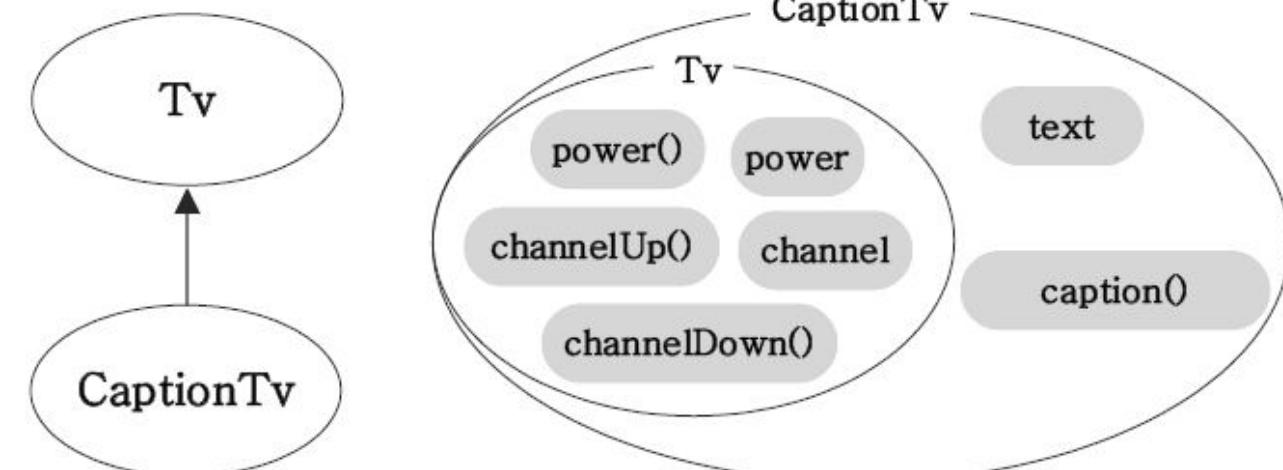
-“하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것”

즉, 조상타입의 참조변수로 자손타입의 객체를 다룰 수 있는 것이 다형성.

```
class Tv {  
    boolean power; // 전원상태(on/off)  
    int channel; // 채널  
  
    void power(){ power = !power; }  
    void channelUp(){ ++channel; }  
    void channelDown(){ --channel; }  
}  
  
class CaptionTv extends Tv {  
    String text; // 캡션내용  
    void caption() /* 내용생략 */  
}
```

```
Tv t = new Tv();  
CaptionTv c = new CaptionTv();
```

```
Tv t = new CaptionTv();
```



```
CaptionTv c = new CaptionTv();
```

```
Tv t = new CaptionTv();
```

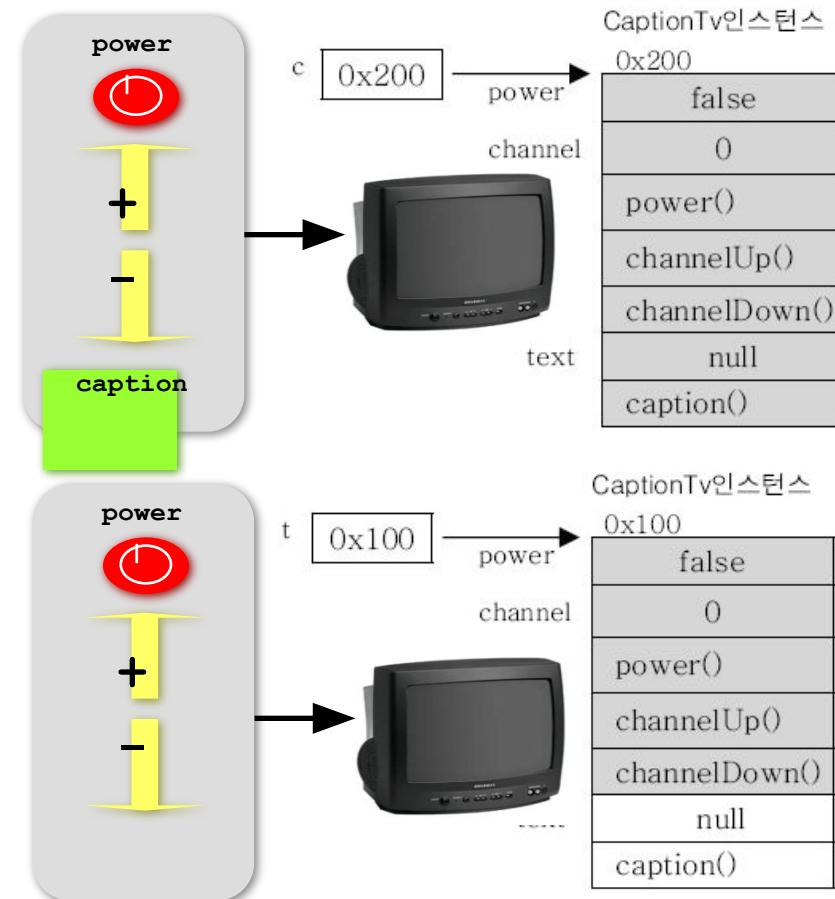
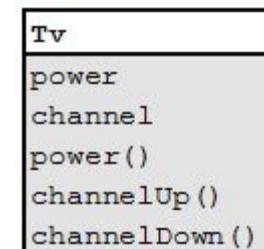
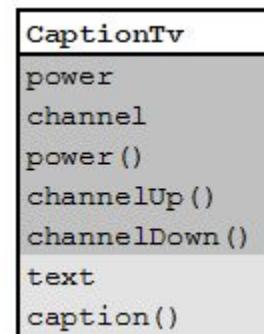
5.1 다형성(polymorphism)이란?(2/3)

“하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것”

즉, 조상타입의 참조변수로 자손타입의 객체를 다룰 수 있는 것

```
CaptionTv c = new CaptionTv();  
TV t = new CaptionTv();
```

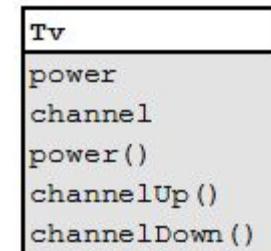
```
class TV {  
    boolean power; // 전원상태(on/off)  
    int channel; // 채널  
  
    void power(){ power = !power; }  
    void channelUp(){ ++channel; }  
    void channelDown(){ --channel; }  
}  
  
class CaptionTv extends TV {  
    String text; // 캡션내용  
    void caption() /* 내용생략 */  
}
```



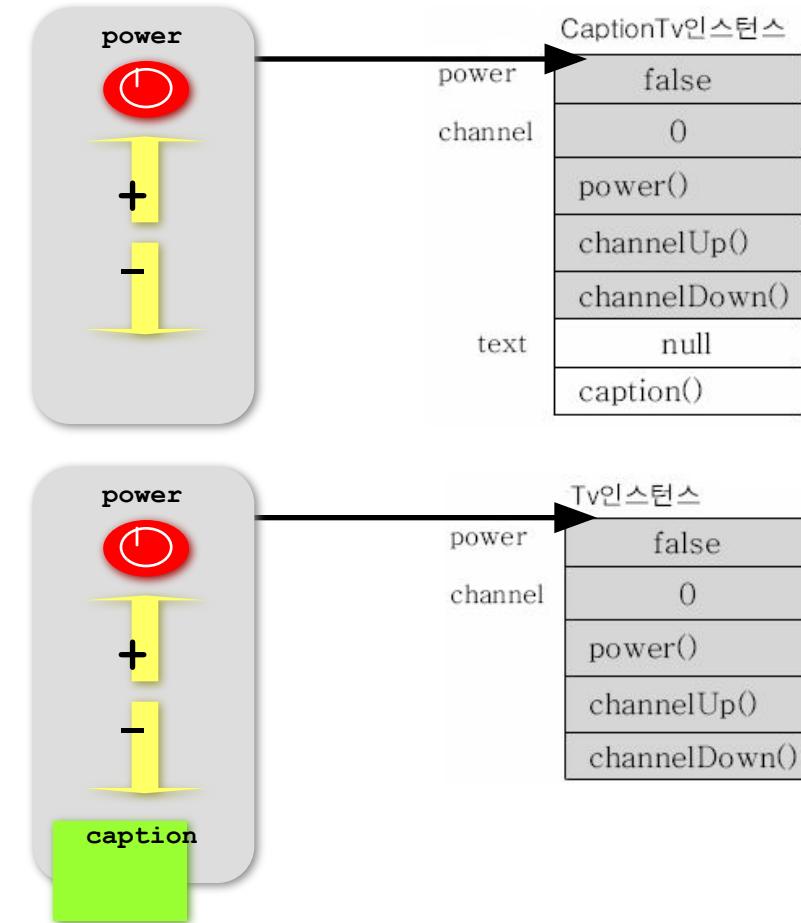
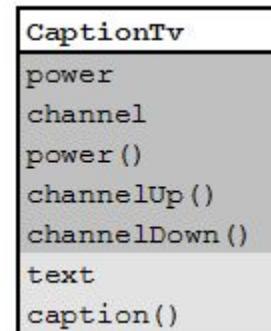
5.1 다형성(polymorphism)이란?(3/3)

“조상타입의 참조변수로 자손타입의 인스턴스를 참조할 수 있지만,
반대로 자손타입의 참조변수로 조상타입의 인스턴스를 참조할 수는 없다.”

```
Tv t = new CaptionTv();  
  
CaptionTv c = new Tv();
```



```
class Tv {  
    boolean power; // 전원상태(on/off)  
    int channel; // 채널  
  
    void power(){ power = !power; }  
    void channelUp(){ ++channel; }  
    void channelDown(){ --channel; }  
}  
  
class CaptionTv extends Tv {  
    String text; // 캡션내용  
    void caption() /* 내용생략 */  
}
```

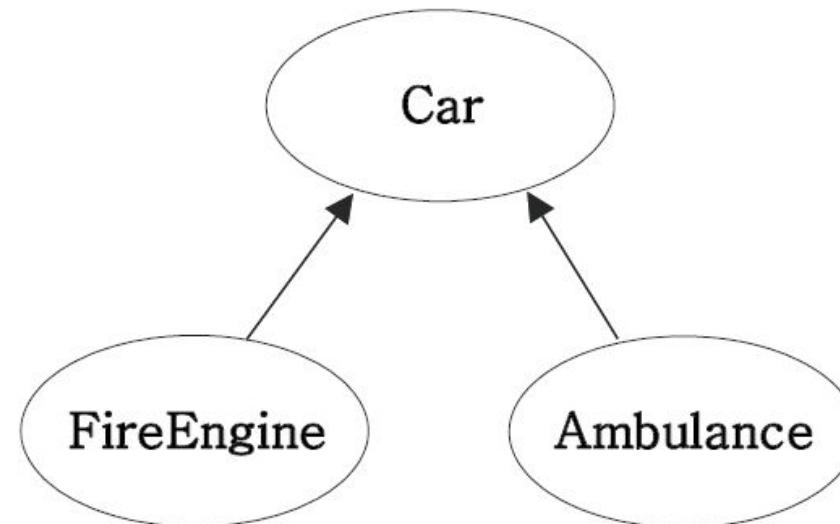


5.2 참조변수의 형변환

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

```
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~~~");  
    }  
}
```

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



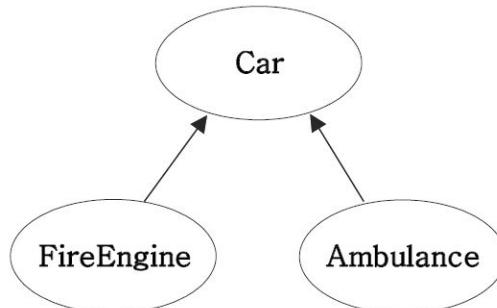
```
FireEngine f  
Ambulance a;
```

```
a = (Ambulance)f;  
f = (FireEngine)a;
```

ERROR

5.2 참조변수의 형변환 - 예제설명

```
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();  
}
```

car null

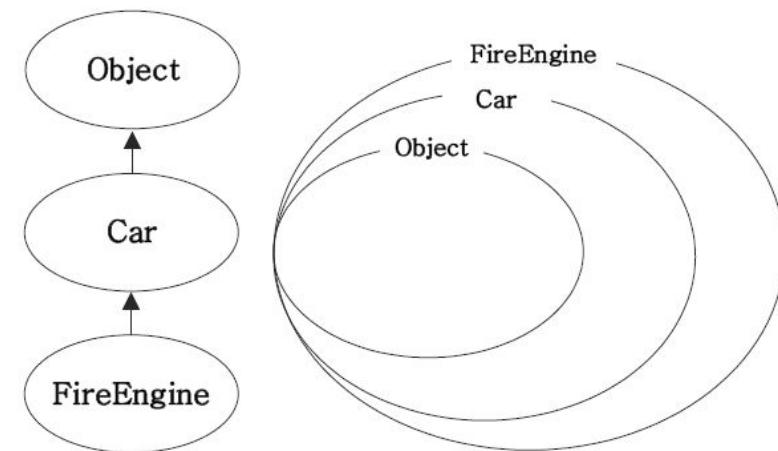
5.3 instanceof 연산자

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

```
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 ins  
        }  
    }  
}  
----- java -----  
this is a fireEngine ins  
This is a Car instance.  
This is an Object instanc  
}
```

```
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();  
    }  
}
```

출력 완료 (0초 경과)



5.4 참조변수와 인스턴스변수의 연결

- 멤버변수가 중복정의된 경우, 참조변수의 타입에 따라 연결 멤버변수가 달라진다. (참조변수타입에 영향받음)
- 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");  
    }  
}
```

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

5.5 매개변수의 다형성

- 참조형 매개변수는 메서드 호출시, **자신과 같은 타입 또는 자손타입**의 인스턴스를 넘겨줄 수 있다.

```
class Product {  
    int price; // 제품가격  
    int bonusPoint; // 보너스점수  
}  
  
class Tv extends Product {}  
class Computer extends Product {}  
class Audio extends Product {}  
  
class Buyer { // 물건사는 사람  
    int money = 1000; // 소유금액  
    int bonusPoint = 0; // 보너스점수  
}
```

```
void buy(Tv t) {  
    money -= t.price;  
    bonusPoint += t.bonusPoint;  
}
```

```
Buyer b = new Buyer();
```

```
Tv tv = new Tv();  
Computer com = new Computer();  
  
b.buy(tv);  
b.buy(com);
```

```
Product p1 = new Tv();  
Product p2 = new Computer();  
Product p3 = new Audio();
```

```
void buy(Product p) {  
    money -= p.price;  
    bonusPoint += p.bonusPoint;  
}
```

5.6 여러 종류의 객체를 하나의 배열로 다루기(1/3)

- 조상타입의 배열에 자손들의 객체를 담을 수 있다.

```
Product p1 = new Tv();
Product p2 = new Computer();
Product p3 = new Audio();
```

```
Product p[] = new Product[3];
p[0] = new Tv();
p[1] = new Computer();
p[2] = new Audio();
```

```
class Buyer { // 물건사는 사람
    int money = 1000; // 소유금액
    int bonusPoint = 0; // 보너스점수

    Product[] cart = new Product[10]; // 구입한 물건을 담을 배열

    int i=0;

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액부족");
            return;
        }

        money -= p.price;
        bonusPoint += p.bonusPoint;
        cart[i++] = p;
    }
}
```

5.6 여러 종류의 객체를 하나의 배열로 다루기(2/3)

- ▶ `java.util.Vector` – 모든 종류의 객체들을 저장할 수 있는 클래스

메서드 / 생성자	설명
<code>Vector()</code>	10개의 객체를 저장할 수 있는 Vector인스턴스를 생성한다. 10개 이상의 인스턴스가 저장되면, 자동적으로 크기가 증가된다.
<code>boolean add(Object o)</code>	Vector에 객체를 추가한다. 추가에 성공하면 결과값으로 true, 실패하면 false를 반환한다.
<code>boolean remove(Object o)</code>	Vector에 저장되어 있는 객체를 제거한다. 제거에 성공하면 true, 실패하면 false를 반환한다.
<code>boolean isEmpty()</code>	Vector가 비어있는지 검사한다. 비어있으면 true, 비어있지 않으면 false를 반환한다.
<code>Object get(int index)</code>	지정된 위치(index)의 객체를 반환한다. 반환타입이 Object타입이므로 적절한 타입으로의 형변환이 필요하다.
<code>int size()</code>	Vector에 저장된 객체의 개수를 반환한다.

```
public class Vector extends AbstractList implements List, Cloneable,  
                      java.io.Serializable {  
    protected Object elementData[];  
    ...  
}
```

5.6 여러 종류의 객체를 하나의 배열로 다루기(3/3)

```
Product[] cart = new Product[10];
//...
void buy(Product p) {
    //...
    cart[i++] = p;
}
```

```
Vector cart = new Vector();
//...
void buy(Product p) {
    //...
    cart.add(p);
}
```

```
void summary() {           // 구매한 물품에 대한 정보를 요약해서 보여준다.
    int sum = 0;            // 구입한 물품의 가격합계
    String cartList = "";   // 구입한 물품목록
```

```
if(cart.isEmpty()) {      // Vector가 비어있는지 확인함
    System.out.println("구입하신 제품이 없습니다.");
    return;
}
```

```
class Tv extends Product {
    Tv() { super(100); }
    public String toString() { return "Tv"; }
}
```

```
// 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.
for(int i=0; i<cart.size(); i++) {
    Product p = (Product)cart.get(i);
    sum += p.price;
    cartList += (i==0) ? "" + p : ", " + p;
}
System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
System.out.println("구입하신 제품은 " + cartList + "입니다.");
}
```

```
Object obj = cart.get(i);
sum += obj.price; // 예상
```

메서드 / 생성자

Vector()

boolean add(Object o)

boolean remove(Object o)

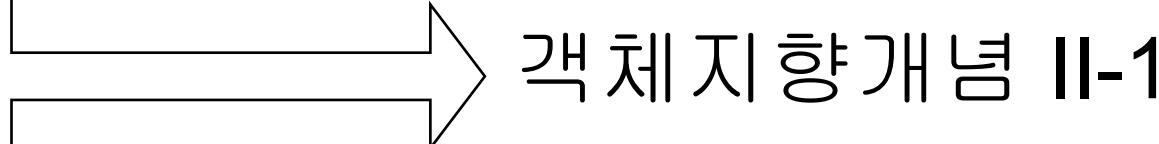
boolean isEmpty()

Object get(int index)

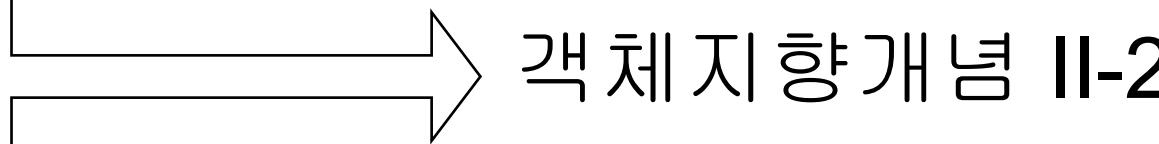
int size()

액체지향개념 II-3

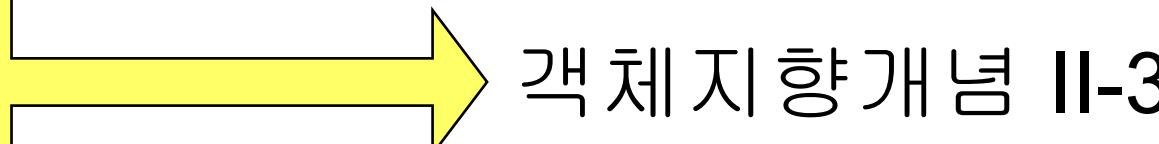
- 1. 상속
- 2. 오버라이딩
- 3. package와 import



- 4. 제어자
- 5. 다형성



- 6. 추상클래스
- 7. 인터페이스
- 8. 내부 클래스



6. 추상클래스(Abstract class)

6.1 추상클래스(Abstract class)란?

6.2 추상메서드(Abstract method)란?

6.3 추상클래스의 작성

7. 인터페이스(interface)

7.1 인터페이스란?

7.2 인터페이스의 작성

7.3 인터페이스의 상속

7.4 인터페이스의 구현

7.5 인터페이스를 이용한 다형성

7.6 인터페이스의 장점

7.7 인터페이스의 이해

7.8 디폴트 메서드

8. 내부 클래스(inner class)

8.1 내부 클래스란?

8.2 내부 클래스의 종류와 특징

8.3 내부 클래스의 제어자

8.4 익명 클래스

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 **Object**클래스와 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);  
}
```

- 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;
    }
}

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

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

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

    public String toString() {
        return "Tank";
    }
}

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

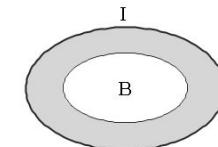
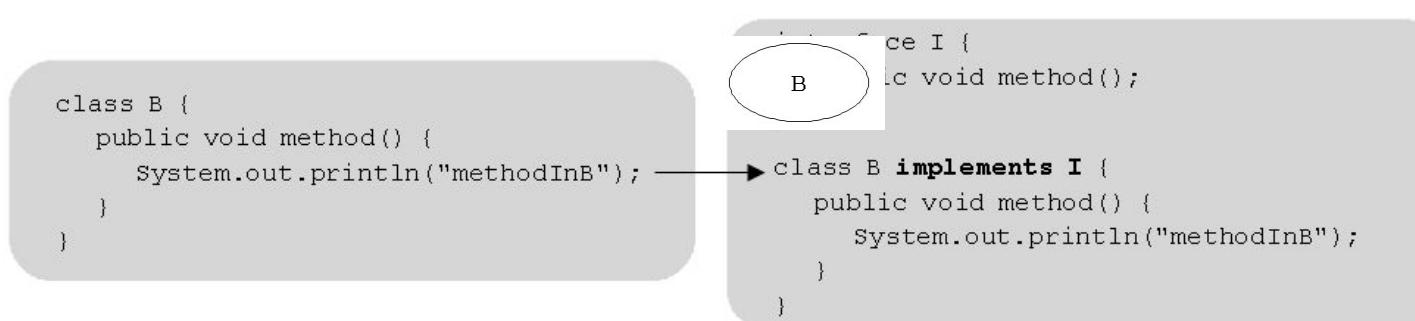
    void repair(Repairable r) {
        if (r instanceof Unit) {
            Unit u = (Unit)r;
            while(u.hitPoint!=u.MAX_HP) {
                u.hitPoint++; // Unit의 HP를 증가시킨다.
            }
        }
    } // repair(Repairable r)
}
```

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

7.7 인터페이스의 이해(1/3)

▶ 인터페이스는...

- 두 대상(객체) 간의 ‘연결, 대화, 소통’을 돋는 ‘중간 역할’을 한다.
- 선언(설계)과 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());  
    }  
}
```

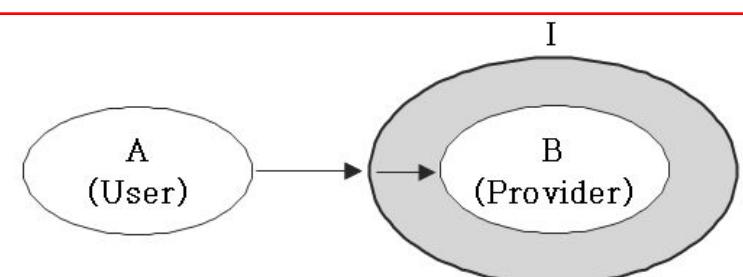


▶ 간접적인 관계의 두 클래스(A-I-B)

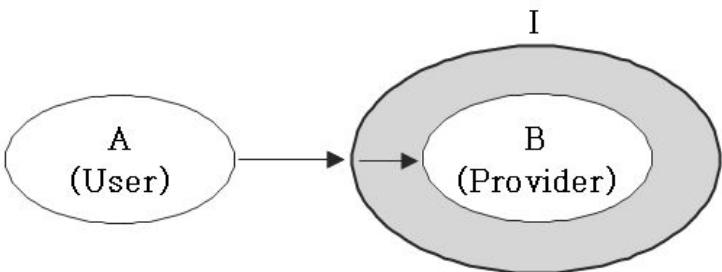
```
class A {  
    public void methodA(I i) {  
        i.methodB();  
    }  
}
```

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

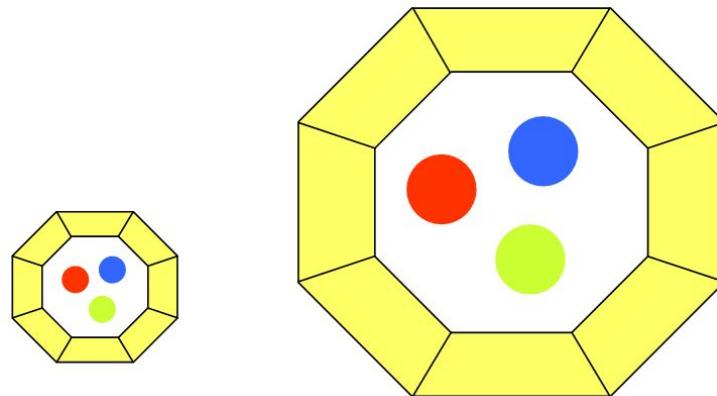
```
class C implements I {  
    public void methodB() {  
        System.out.println("methodB() in C");  
    }  
}
```



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.
- GUI 어플리케이션(AWT, Swing)의 이벤트 처리에 주로 사용된다.



▶ 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;          // static  
    }  
  
    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(); •———— 인스턴스클래스는 외부 클래스를 먼저 생성해야만 생성할 수 있다.  
        InstanceInner obj1 = outer.new InstanceInner();  
    }  
  
    void instanceMethod() {  
        InstanceInner obj1 = new InstanceInner();  
        StaticInner obj2 = new StaticInner(); •———— 인스턴스메서드에서는 인스턴스멤버와 static멤버 모두 접근 가능하다.  
        // 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 **final**이 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 {  
        // 스탠다드 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);  
  
        // 스탠틱 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
```

