Java의 정석

제 7 장

객체지향개념 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 instanceof연산자
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	- 모든 인스턴스에 공통적으로 사용되는 클래스변수가 된다. - 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다. - 클래스가 메모리에 로드될 때 생성된다.
	메서드	- 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	변수 앞에 final이 붙으면, 값을 변경할 수 없는 상수가 된다.
	지역변수	

[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

- final이 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	전 체
public				
protected				
default				
private				

public
(default)

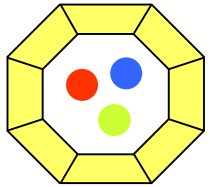
public
protected
(default)
private

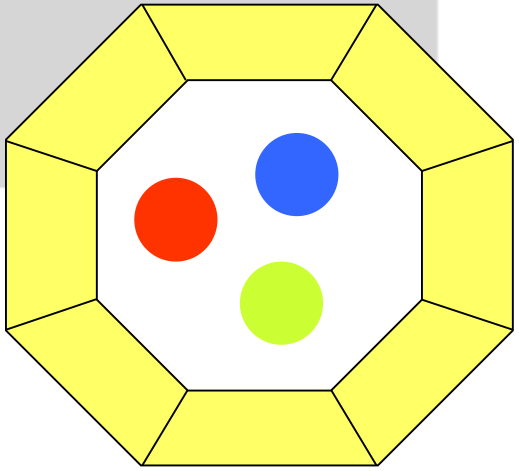
```
class AccessModifierTest {
    int iv;         // 멤버변수(인스턴스변수)
    static int cv; // 멤버변수(클래스변수)

    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();  에러!!!
        Singleton s1 = Singleton.getInstance();
    }
}
```

4.9 제어자의 조합

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

1. 메서드에 static과 abstract를 함께 사용할 수 없다.
   - static메서드는 몸통(구현부)이 있는 메서드에만 사용할 수 있기 때문이다.
2. 클래스에 abstract와 final을 동시에 사용할 수 없다.
   - 클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미이고, abstract는 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.
3. abstract메서드의 접근제어자가 private일 수 없다.
   - abstract메서드는 자손클래스에서 구현해주어야 하는데 접근 제어자가 private이면, 자손클래스에서 접근할 수 없기 때문이다.
4. 메서드에 private과 final을 같이 사용할 필요는 없다.
   - 접근 제어자가 private인 메서드는 오버라이딩될 수 없기 때문이다. 이 둘 중 하나만 사용해도 의미가 충분하다.

5. 다형성(polymorphism)

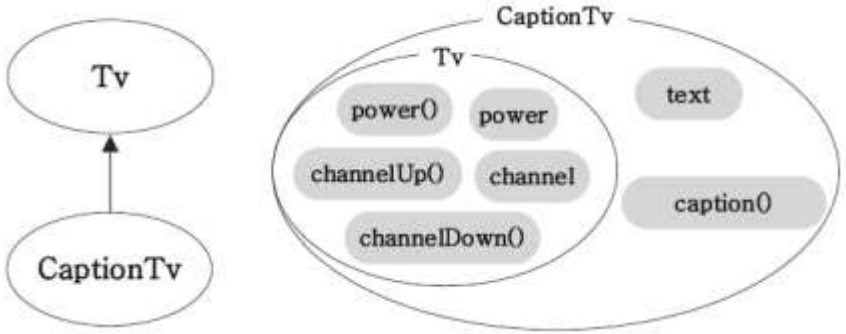
5.1 다형성(polymorphism)이란?(1/3)

- "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();
```

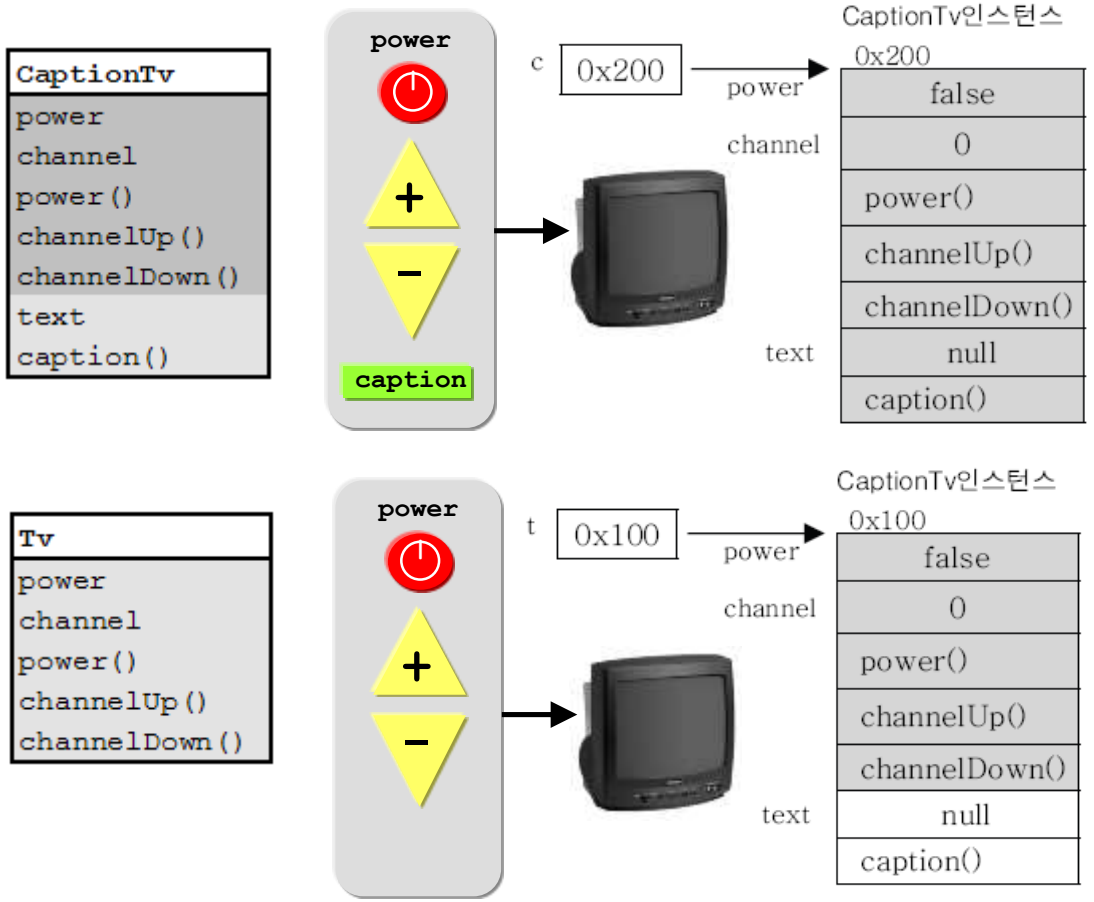
5.1 다형성(polymorphism)이란?(2/3)

"하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것"

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

```
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 */}
}
```

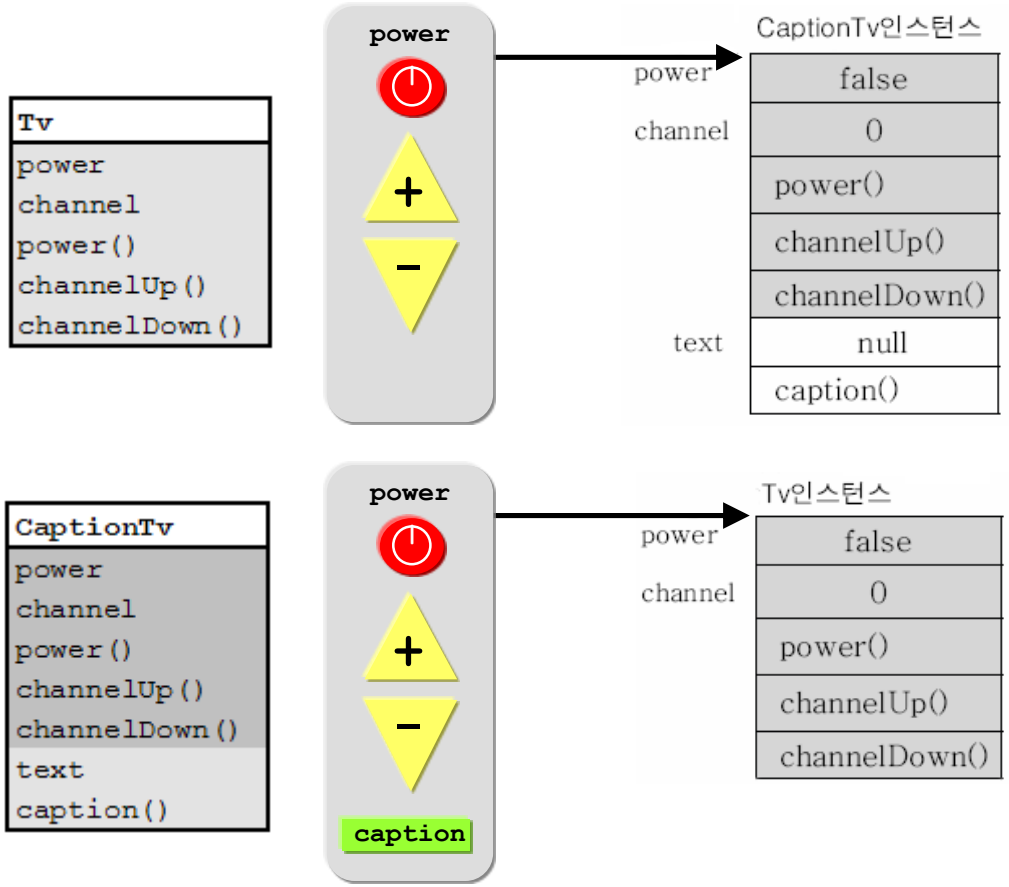


5.1 다형성(polymorphism)이란?(3/3)

"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 */}
}
```

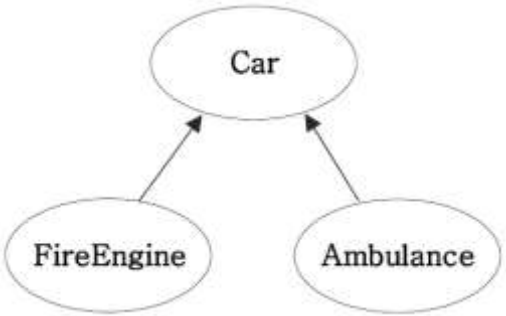


5.2 참조변수의 형변환

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

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

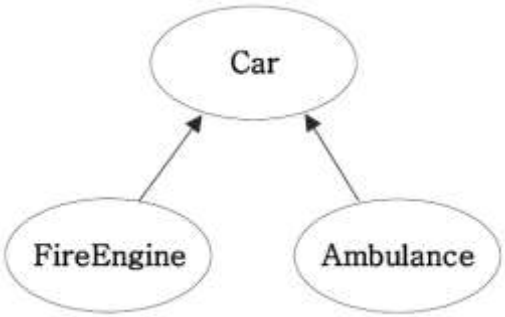
```
class Car {
    String color;
    int door;

    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~~~");
    }
}
```



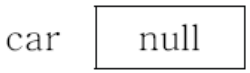

```
FireEngine f
Ambulance a;

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

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~~~");
    }
}
```



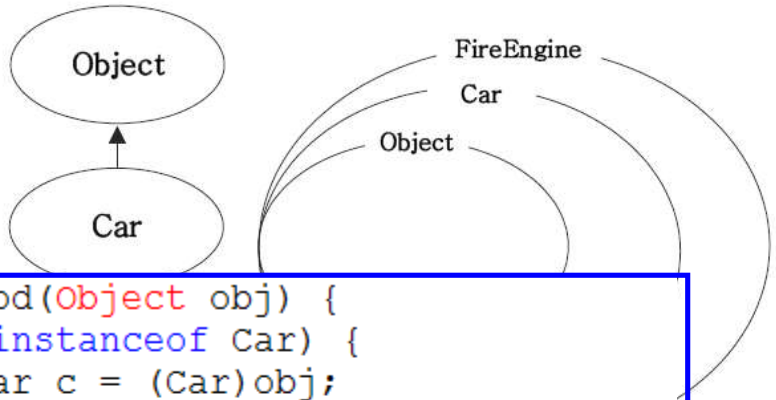

```
public static void main(String args[]) {
    Car car = null;
    FireEngine fe = new FireEngine();
    FireEngine fe2 = null;

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



5.3 instanceof연산자

- 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 instan

출력 완료 (0초 경과)
```

```
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();
    }
}
```

5.4 참조변수와 인스턴스변수의 연결

- 멤버변수가 중복정의된 경우, 참조변수의 타입에 따라 연결되는 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 { }
```

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

```
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();
}
```

5.5 매개변수의 다형성

- 참조형 매개변수는 메서드 호출시, 자신과 같은 타입 또는 자손타입의 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
}
```

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

```
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	설 명
`Vector()`	10개의 객체를 저장할 수 있는 Vector인스턴스를 생성한다. 10개 이상의 인스턴스가 저장되면, 자동적으로 크기가 증가된다.
`boolean add(Object o)`	Vector에 객체를 추가한다. 추가에 성공하면 결과값으로 true, 실패하면 false를 반환한다.
`boolean remove(Object o)`	Vector에 저장되어 있는 객체를 제거한다. 제거에 성공하면 true, 실패하면 false를 반환한다.
`boolean isEmpty()`	Vector가 비어있는지 검사한다. 비어있으면 true, 비어있지 않으면 false를 반환한다.
`Object get(int index)`	지정된 위치(index)의 객체를 반환한다. 반환타입이 Object타입이므로 적절한 타입으로의 형변환이 필요하다.
`int size()`	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);
}
```

메서드 / 생성자
Vector()
boolean add(Object o)
boolean remove(Object o)
boolean isEmpty()
Object get(int index)
int size()

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

    if(cart.isEmpty()) {            //
        System.out.println("구입
        return;
    }

    // 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.");
}
```

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

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