● **다형성(polymorphism): 조상 타입 참조변수로 자손 타입 객체(인스턴스)를 다루는 것(참조하는 것).** 여러 가지 형태를 가질 수 있는 능력을 의미하며, 한 타입의 참조변수로 여러 타입의 객체를 참조할 수 있으므로써 다형성을 프로그램적으로 구현하였다. 즉, **조상클래스 타입의 참조변수로 자손클래스의 인스턴스를 참조**할 수 있도록 하였다. 인스턴스 타입과 참조변수의 타입이 일치하는 것이 보통이지만, 두 클래스가 서로 상속관계에 있을 경우, 조상 클래스 타입의 참조변수로 자손클래스의 인스턴스를 참조하도록 하는 것도 가능하다. 이때, **조상 클래스 타입의 참조변수로는 자손클래스 인스턴스의 모든 멤버를 사용할 수는 없고, 그 중에서 조상클래스의 멤버들(상속받은 멤버들)만 사용할 수 있다. 따라서, 조상클래스의 참조변수로는 자손클래스 인스턴스의 멤버 중에서 조상클래스에 정의되지 않는 멤버를 사용할 수 없다.** 그러므로 둘 다 같은 타입의 인스턴스지만 **참조변수의 타입에 따라 참조변수로 사용할 수 있는 멤버의 개수가 달라진다.** 단, 자손타입의 참조변수로 조상타입의 인스턴스를 참조하는 것은 존재하지 않는 멤버를 사용하고자 할 가능성이 있으므로 허용하지 않는다. **참조변수가 사용할 수 있는 멤버의 개수는 인스턴스의 멤버 개수보다 같거나 적어야 한다** (클래스는 상속을 통해서 확장될 수는 있어도 축소될 수는 없어서, 조상 인스턴스의 멤버 개수는 자손 인스턴스의 멤버 개수보다 항상 적거나 같다). 참조변수의 타입이 참조변수가 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 개수를 결정한다 (모든 차참마조변수는 null 또는 4 byte의 주소값이 저장되며, 참조변수의 타입은 ㅊ마조할 수 있는 객체의 종듀와 사용할 수 있는 멤버의 수를 결정한다).

```
class Tv {
   boolean power;
   int channel;

   public void channelUp() {
       ++channel;
    }

   public void channelDown() {
       --channel;
    }
}

class CaptionTv extends Tv {
    String text;

    protected void caption() {
    }
}

public class Ex {
    public static void main(String[] args) {
        CaptionTv c = new CaptionTv();
        // 참조변수의 타입이 인스턴스의 타입과 일치하므로 c는 인스턴스의
        // 모든 멤버 변수를 사용가능하다.
        System.out.println(c.power);
        System.out.println(c.channel);
        System.out.println(c.text);
        c.caption();

        Tv t = new CaptionTv();
        // 조상클래스가 참조변수의 타입으로 인스턴스의 타입과 다르므로 t로는
        // 인스턴스의 조상클래스 멤버들만 사용가능하다.
        System.out.println(t.power);
        System.out.println(t.channel);
//        System.out.println(t.text);  // 에러
//        t.caption();  // 에러


    }
}
```

● 참조변수의 형변환: 사용할 수 있는 멤버의 갯수를 조절하는 것이다. 상속관계에 있는 클래스들의 참조변수를 서로 형변환 할 수 있다. 바로 윗 조상이나 자손이 아닌, 조상의 조상으로도 형변환이 가능하다. (따라서 모든 참조변수는 모든 클래스의 조상인 Object클래스 타입으로 형변환이 가능하다). 캐스트 연산자를 사용하며, 괄호()안에 변환하고자 하는 타입의 이름(클래스명)을 적어주면 된다.

업캐스팅(Up-casting)	자손타입 → 조상타입 (자손타입의 참조변수를 조상타입의 참조변수에 할당할 경우)	형변환 생략 가능 (참조변수가 다를 수 없는 멤버의 개수가 실제 인스턴스가 갖고 있는 멤버의 개수보다 적을 것이 분명하므로 문제가 되지 않음)
다운캐스팅(Down-casting)	조상타입 → 자손타입	형변환 생략 불가

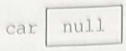
	(조상타입의 참조변수를 자손타입의 참조변수에 저장할 경우)	

형변환은 참조변수의 타입을 변환하는 것이지 인스턴스를 변환하는 것은 아니기 때문에, 참조변수의 형변환은 인스턴스에 아무런 영향을 미치지 않는다. 단지 참조변수의 형변환을 통해서, 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 범위(개수)를 조절하는 것뿐이다. 조상타입의 참조변수로 형변환 될 경우 조상 클래스에 해당하는 인스턴스의 멤버들만 사용할 수 있다. 단, 조상 타입의 인스턴스를 자손 타입의 참조변수로 참조하는 것은 허용되지 않는다.

```
class CastingTest1 {
    public static void main(String args[]) {
        Car car = null;
        FireEngine fe = new FireEngine();
        FireEngine fe2 = null;

        fe.water();
        car = fe;    // car =(Car)fe;에서 형변환이 생략된 형태. 업 캐스팅.
//      car.water(); 컴파일 에러. Car타입의 참조변수로는 water()을 호출할 수 없다.
        fe2 = (FireEngine)car; // 자손타입 ← 조상타입. 다운 캐스팅.
        fe2.water();
    }
}

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() {      // 물을 뿌리는 기능
        System.out.println("water!!!");
    }
}
```

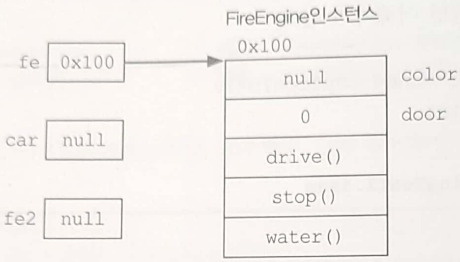
1. `Car car = null;`

Car타입의 참조변수 car를 선언하고 null로 초기화한다.



2. `FireEngine fe = new FireEngine();`

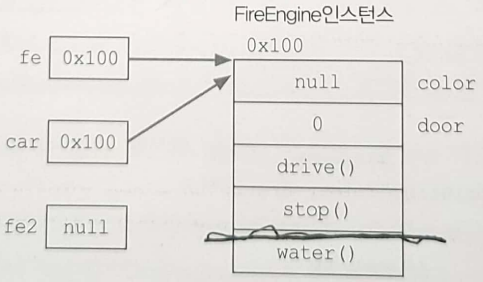
FireEngine인스턴스를 생성하고 FireEngine타입의 참조변수가 참조하도록 한다.



3. `car = fe;` // 조상 타입 ← 자손 타입

참조변수 fe가 참조하고 있는 인스턴스를 참조변수 car가 참조하도록 한다. fe의 값(fe가 참조하고 있는 인스턴스의 주소)이 car에 저장된다. 이 때 두 참조변수의 타입이 다르므로 참조변수 fe가 형변환되어야 하지만 생략되었다.

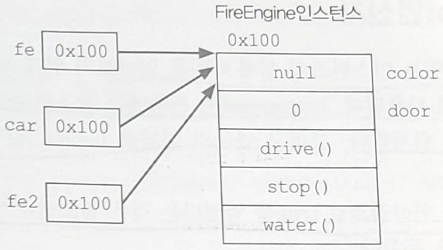
이제는 참조변수 car를 통해서도 FireEngine인스턴스를 사용할 수 있지만, fe와는 달리 car는 Car타입이므로 Car클래스의 멤버가 아닌 water( )는 사용할 수 없다.



4. `fe2 = (FireEngine)car;` // 자손 타입 ← 조상 타입

참조변수 car가 참조하고 있는 인스턴스를 참조변수 fe2가 참조하도록 한다. 이 때 두 참조변수의 타입이 다르므로 참조변수 car를 형변환하였다. car에는 FireEngine인스턴스의 주소가 저장되어 있으므로 fe2에도 FireEngine인스턴스의 주소가 저장된다.

이제는 참조변수 fe2를 통해서도 FireEngine인스턴스를 사용할 수 있지만, car와는 달리 fe2는 FireEngine타입이므로 FireEngine인스턴스의 모든 멤버들을 사용할 수 있다.



조상 타입의 인스턴스를 자손 타입의 참조변수로 참조하는 것은 허용되지 않는다.

```
class CastingTest2 {
    public static void main(String args[]) {
        Car car = new Car();
        Car car2 = null;
        FireEngine fe = null;

        car.drive();
        fe = (FireEngine)car;       // 8번째 줄. 컴파일은 OK. 실행 시 에러가 발생
        // 참조 변수 car이 참조하고 있는 인스턴스가 Car타입의 인스턴스이고,
        // 조상 타입의 인스턴스를 자손 타입의 참조변수로 참조하는 것은 허용되지 않기 때문에,
        // 실행 시 에러(ClassCastException)이 발생한다.
        // (컴파일 시에는 참조변수간의 타입만 체크하기 때문에 컴파일 시에는 문제가 없다)
        fe.drive();
        car2 = fe;
        car2.drive();
    }
}
```

서로 상속관계에 있는 타입간의 형변환은 양방향으로 자유롭게 수행될 수 있지만, 참조변수가 가리키는 인스턴스의 자손타입으로 형변환은 허용되지 않는다. 그래서 참조변수가 가르키는 인스턴스의 타입이 무엇인지 확인하는 것이 중요하다. 형변환을 수행하기 전에 instanceof 연산자를 이용해서 참조변수가 참조하고 있는 실제 인스턴스의 타입을 확인할 수 있다.

● instanceof연산자: 참조변수가 참조하고 있는 인스턴스의 실제 타입을 알아보기 위해 사용된다. 주로 조건문에 사용되며, instanceof 왼쪽에는 참조변수를 오든쪽에는 타입(클래스명)이 피연산자로 위치한다. 연산의 결과로는 boolean값인 true와 false중에 하나를 반환한다. **어떤 타입에 대한 instanceof연산의 결과가 true라는 것은, 참조변수가 검사한 타입으로 형변환이 가능하다는 것을 뜻한다.** (값이 null인 참조변수에 대해 isntanceof연산을 수행하면 false를 결과로 얻는다).

```
// 예를 들면 Car을 조상으로 둔 자손 클래스 FireEngine과 Ambulance가 있다면

void doWork(Car c) {
    if (c instanceof FireEngine) {
        FireEngine fe = (FireEngine)c;
        fe.water();
    } else if (c instanceof Ambulance) {
        Ambulance a = (Ambulance)c;
        a.siren();
    }
}
```

위의 코드는 Car타입의 참조변수 c를 매개변수로 하는 메서드이다. 이 메서드가 호출될 때, 매개변수로 Car클래스 또는 그 자손 클래스의 인스턴스를 넘겨받겠지만, 메서드 내에서는 정확히 어떤 인스턴스인지 알 길이 없다. 그래서 instanceof연선자를 이용해서 참조변수 c가 가리키고 있는 인스턴스의 타입을 체크하고, 적절히 형변환한 다음에 작업을 해야 한다. **조상타입의 참조변수로는 실제 자손 인스턴스의 멤버듣은 모두 사용할 수 없기 때문에, 실제 인스턴스와 같은 타입의 참조변수로 형변환은 해야만 인스턴스의 모든 멤버듣은 사용할 수 있다.**

```java
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.");
        }

        System.out.println(fe.getClass().getName()); // 클래스의 이름을 출력, FireEngine
    }
} // class
class Car {}
class FireEngine extends Car {}

// 위의 세 문장 모두 출력되고 클래스 이름도 출력된다.
```

실제 인스턴스와 같은 타입의 instanceof연산 이외에 조상타입의 instanceof 연산에도 true를 결과로 얻으며, instance연산의 결과가 true라는 것은 검사한 타입으로의 형변환을 해도 아무런 문제가 없다는 뜻이다. (이는 생성된 인스턴스의 클래스가 조상 클래스로부터 멤버들을 상속받았기 때문에, 자손 인스턴스는 조상의 인스턴스들을 모두 포함하고 있는 셈이기 때문이다). **'참조변수.getClass().getName()'은 참조변수가 가리키고 있는 인스턴스의 클래스 이름을 문자열(String)로 변환**한다.

● 참조변수와 인스턴스의 연결

조상 클래스에 선언된 멤버변수와 같은 이름의 인스턴스변수를 자손 클래스에 중복으로 정의했을 때, 조상타입의 참조변수로 자손 인스턴스를 참조하는 경우와 자손 타입의 참조변수로 자손 인스턴스를 참조하는 경우는 서로 다른 결과를 얻는다. **멤버변수가 조상 클래스와 자손 클래스에 중복으로 정의된 경우, 조상 타입의 참조변수를 사용했을 때는 조상 클래스에 선언된 멤버변수가 사용되고, 자손타입의 참조변수를 사용했을 때는 자손 클래스에 선언된 멤버변수가 사용된다.** 하지만 중복 정의되지 않았을 경우에 차이는 없다. **메서드의 경우 조상 클래스의 메서드를 자손의 클래스에서 오버라이딩 한 경우에도, 참조변수의 타입에 상관없이 항상 실제 인스턴스의 메서드(인스턴스가 자손클래스일 경우 오버라이딩된 메서드)가 호출되지만, 멤버변수의 경우 참조변수의 타입에 따라 달라진다.** (참고로 static메서드는 static변수처럼 참조변수의 타입에 영향을 받는다. 참조변수의 타입에 영향을 받지 않는 것은 인스턴스메서드 뿐이다. 그래서 static메서드는 반드시 참조변수가 아닌 '클래스이름.메서드()'로 호출해야 한다)

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

        System.out.println("p.x = " + p.x);  // p.x = 100
        p.method(); // Child Method

        System.out.println("c.x = " + c.x);  // c.x = 200;
        c.method(); // child Method
    }
}

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");
    }
}
```

```java
class BindingTest3{
    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();
        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;

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

class Child extends Parent {
    int x = 200;

    void method() {
        System.out.println("x=" + x);  // this.x와 같다.
        System.out.println("super.x=" + super.x);
        System.out.println("this.x="  + this.x);
    }
}

// 실행 결과
//p.x = 100
//x=200
//super.x=100
//this.x=200
//
//c.x = 200
//x=200
//super.x=100
//this.x=200
```

멤버변수들은 주로 private으로 접근을 제한하고, 외부에서는 메서드를 통해서만 멤버변수에 접근하도록 하는 이유 중에 하나도, 이처럼 다른 외부 클래스에서 참조변수를 통해 직접적으로 인스턴스변수에 접근할 수 있게 하면, 참조변수의 타입에 따라 사용되는 인스턴스변수가 달라질 수 있기 때문이다.

※ 다형성의 장점: 1) 다형적 매개변수 2) 하나의 배열로 여러 종듀 객체 다듀기

● 메서드 매개변수의 다형성: 조상 클래스 타입의 참조변수를 메서드의 매개변수로 선언하면, 그 클래스의 자손 타입의 참조변수(혹은 자손 클래스의 인스턴스)면 어느 것이나 매개변수로 받아들일 수 있다. 즉, **참조형 메개변수는 메서드 호춛 시, 자신과 같은 타입 또는 자손타입의 인스턴스(혹은 그 인스턴스의 참조변수)를 넘겨줄 수 있다.**

```java
class Product {
    int price;          // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수

    Product(int price) {
        this.price = price;
        bonusPoint =(int)(price/10.0);  // 보너스점수는 제품가격의 10%
    }
}

class Tv extends Product {
    Tv() {
        // 조상클래스의 생성자 Product(int price)를 호출한다.
        super(100);         // Tv의 가격을 100만원으로 한다.
    }

    public String toString() {  // Object클래스의 toString()을 오버라이딩한다.
        return "Tv";
    }
}

class Computer extends Product {
    Computer() {
        super(200);
    }

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

class Buyer {           // 고객, 물건을 사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
            return;
        }

        money -= p.price;           // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스 점수를 추가한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }
}

class PolyArgumentTest {
    public static void main(String args[]) {
        Buyer b = new Buyer();

        b.buy(new Tv());  // Product p = new Tv(); b.buy(p);를 한 문장으로 줄인 것.
        b.buy(new Computer());

        System.out.println("현재 남은 돈은 " + b.money + "만원입니다.");
        System.out.println("현재 보너스점수는 " + b.bonusPoint + "점입니다.");
    }
}
```

● 여러 종듀의 객체를 배열로 다두기: **조상타입의 참조변수 배열은 사용하면, 공통의 조상은 가진 서로 다른 종류의 객채른 배열로 묶어서 사용할 수 있다**. 또는 묶어서 다두고 싶은 객체들의 상속관계를 따져 가장 가까운 공통조상 클래스 타입의 참조변수 배열을 생성해서 객체들을 저장하면 된다.

```
class Product {
    int price;          // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수

    Product(int price) {
        this.price = price;
        bonusPoint =(int)(price/10.0);
    }

    Product() {} // 기본 생성자
}

class Tv extends Product {
    Tv() {
        super(100);
    }

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

class Computer extends Product {
    Computer() { super(200); }

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

class Audio extends Product {
    Audio() { super(50); }

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

class Buyer {           // 고객, 물건을 사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수
    Product[] item = new Product[10];   // 구입한 제품을 저장하기 위한 배열
    int i =0;           // Product배열에 사용될 카운터

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
            return;
        }

        money -= p.price;           // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스 점수를 추가한다.
        item[i++] = p;              // 제품을 Product[] item에 저장한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }

    void summary() {            // 구매한 물품에 대한 정보를 요약해서 보여 준다.
        int sum = 0;            // 구입한 물품의 가격합계
```

```
        String itemList ="";    // 구입한 물품목록

        // 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.
        for(int i=0; i<item.length;i++) {
            if(item[i]==null) break;
            sum += item[i].price;
            itemList += item[i] + ", ";
        }
        System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
        System.out.println("구입하신 제품은 " + itemList + "입니다.");
    }
}

class PolyArgumentTest2 {
    public static void main(String args[]) {
        Buyer b = new Buyer();

        b.buy(new Tv());                // Tv을/를 구입하셨습니다.
        b.buy(new Computer());              // Computer을/를 구입하셨습니다.
        b.buy(new Audio());             // Audio을/를 구입하셨습니다.
        b.summary();                // 구입하신 물품의 총금액은 350만원입니다.
                                    // 구입하신 제품은 Tv, Computer, Audio, 입니다.
    }
}
```

위의 예제에서 Product배열로 구입한 제품들을 저장할 수 있지만, 배열의 크기가 10인 탓에 11개 이상의 제품을 구입할 수 없다. 이런 경우, Vector클래스를 사용하면 된다. Vector클래스는 내부적으로 Object타입의 배열을 가지고 있어서, 이 배열에 객체를 추가하거나 제거할 수 있게 작성되어 있다. 그리고 배열의 크기를 알아서 관리해주기 때문에 저장할 인스턴스의 개수에 신경쓰지 않아도 된다. Vector클래스는 동적으로 크기가 관리되는 객체배열이다.

메서드/생성자	설명
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. 비어 있으면 true, 비어 있지 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.

```
import java.util.*;         // Vector클래스를 사용하기 위해서 추가해 주었다.

class Product {
    int price;          // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수

    Product(int price) {
        this.price = price;
```

```
        this.price = price;
        bonusPoint =(int)(price/10.0);
    }

    Product() {
        price = 0;
        bonusPoint = 0;
    }
}

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

class Computer extends Product {
    Computer() { super(200); }
    public String toString() { return "Computer"; }
}

class Audio extends Product {
    Audio() { super(50); }
    public String toString() { return "Audio"; }
}

class Buyer {           // 고객, 물건을 사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수
    Vector item = new Vector(); // 구입한 제품을 저장하는데 사용될 Vector객체

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
            return;
        }
        money -= p.price;           // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스 점수를 추가한다.
        item.add(p);                // 구입한 제품을 Vector에 저장한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }

    void refund(Product p) {    // 구입한 제품을 환불한다.
        if(item.remove(p)) {    // 제품을 Vector에서 제거한다.
            money += p.price;
            bonusPoint -= p.bonusPoint;
            System.out.println(p + "을/를 반품하셨습니다.");
        } else {            //  제거에 실패한 경우
            System.out.println("구입하신 제품 중 해당 제품이 없습니다.");
        }
    }

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

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

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

class PolyArgumentTest3 {
    public static void main(String args[]) {
        Buyer b = new Buyer();
        Tv tv = new Tv();
        Computer com = new Computer();
        Audio audio = new Audio();

        b.buy(tv);                    // Tv을/를 구입하셨습니다.
        b.buy(com);                   // Computer을/를 구입하셨습니다.
        b.buy(audio);                 // Audio을/를 구입하셨습니다.
        b.summary();                  //  구입하신 물품의 총금액은 350만원입니다.
                                      // 구입하신 제품은 Tv, Computer, Audio입니다.
        System.out.println();
        b.refund(com);                // Computer을/를 반품하셨습니다.
        b.summary();                  // 구입하신 물품의 총금액은 150만원입니다.
                                      // 구입하신 제품은 Tv, Audio입니다.
    }
}
```

Resource: 자바의 정석