

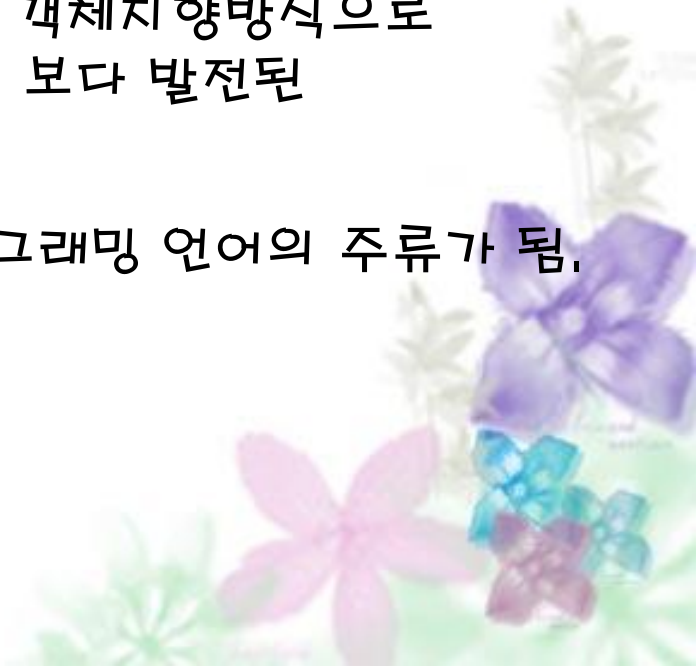
# Java



# 객체지향언어란?

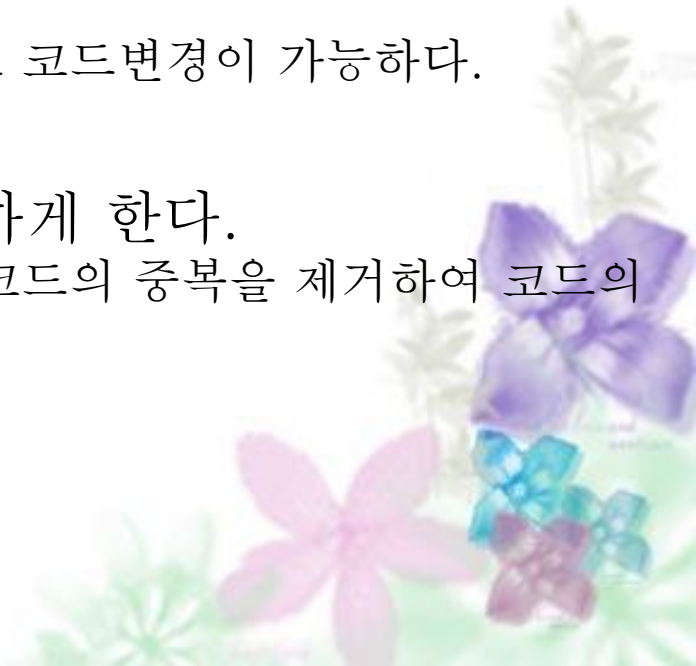
## 1.1 객체지향언어의 역사

- 과학, 군사적 모의실험(simulation)을 위해 컴퓨터를 이용한 가상세계를 구현하려는 노력으로부터 객체지향이론이 시작됨
- 1960년대 최초의 객체지향언어 Simula 탄생
- 1980년대 절차방식의 프로그래밍의 한계를 객체지향방식으로 극복하려고 노력함. (C++, Smalltalk과 같은 보다 발전된 객체지향언어가 탄생)
- 1995년 말 Java 탄생. 객체지향언어가 프로그래밍 언어의 주류가 됨.



## 1.2 객체지향언어의 특징

- ▶ 기존의 프로그래밍언어와 크게 다르지 않다.
  - 기존의 프로그래밍 언어에 몇가지 규칙을 추가한 것일 뿐이다.
- ▶ 코드의 재사용성이 높다.
  - 새로운 코드를 작성할 때 기존의 코드를 이용해서 쉽게 작성할 수 있다.
- ▶ 코드의 관리가 쉬워졌다.
  - 코드간의 관계를 맷어줌으로써 보다 적은 노력으로 코드변경이 가능하다.
- ▶ 신뢰성이 높은 프로그램의 개발을 가능하게 한다.
  - 제어자와 메서드를 이용해서 데이터를 보호하고, 코드의 중복을 제거하여 코드의 불일치로 인한 오류를 방지할 수 있다.



# 클래스와 객체

## 2.1 클래스와 객체의 정의와 용도

- ▶ 클래스의 정의 - 클래스란 객체를 정의해 놓은 것이다.
- ▶ 클래스의 용도 - 클래스는 객체를 생성하는데 사용된다.
- ▶ 객체의 정의 - 실제로 존재하는 것. 사물 또는 개념.
- ▶ 객체의 용도 - 객체의 속성과 기능에 따라 다름.

클래스	객체
제품 설계도	제품
TV설계도	TV
붕어빵기계	붕어빵



## 2.2 객체와 인스턴스

### ▶ 객체 ≡ 인스턴스

- 객체(object)는 인스턴스(instance)를 포함하는 일반적인 의미

책상은 인스턴스다.  
책상은 객체다.

책상은 책상 클래스의 객체다.  
책상은 책상 클래스의 인스턴스다.

### ▶ 인스턴스화(instantiate, 인스턴스化)

- 클래스로부터 인스턴스를 생성하는 것.



## 2.3 객체의 구성요소 - 속성과 기능

- ▶ 객체는 속성과 기능으로 이루어져 있다.
  - 객체는 속성과 기능의 집합이며, 속성과 기능을 객체의 멤버(member, 구성요소)라고 한다.
- ▶ 속성은 변수로, 기능은 메서드로 정의한다.
  - 클래스를 정의할 때 객체의 속성은 변수로, 기능은 메서드로 정의한다.

속성	크기, 길이, 높이, 색상, 볼륨, 채널 등
기능	켜기, 끄기, 볼륨 높이기, 볼륨 낮추기, 채널 높이기 등

변수

메서드

```
class Tv {
```

```
String color; // 색깔  
boolean power; // 전원상태(on/off)  
int channel; // 채널
```

```
}
```

```
void power() { power = !power; } // 전원on/off  
void channelUp( channel++;) // 채널 높이기  
void channelDown {channel--;} // 채널 낮추기
```



## 2.4 인스턴스의 생성과 사용(2/4)

```
Tv t;
```

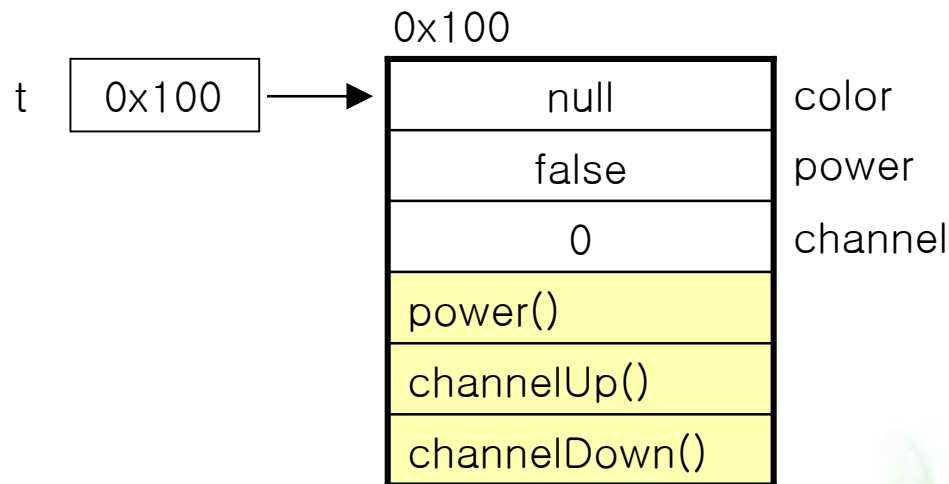
```
t = new Tv();
```

```
t.channel = 7;
```

```
t.channelDown();
```

```
System.out.println(t.channel);
```

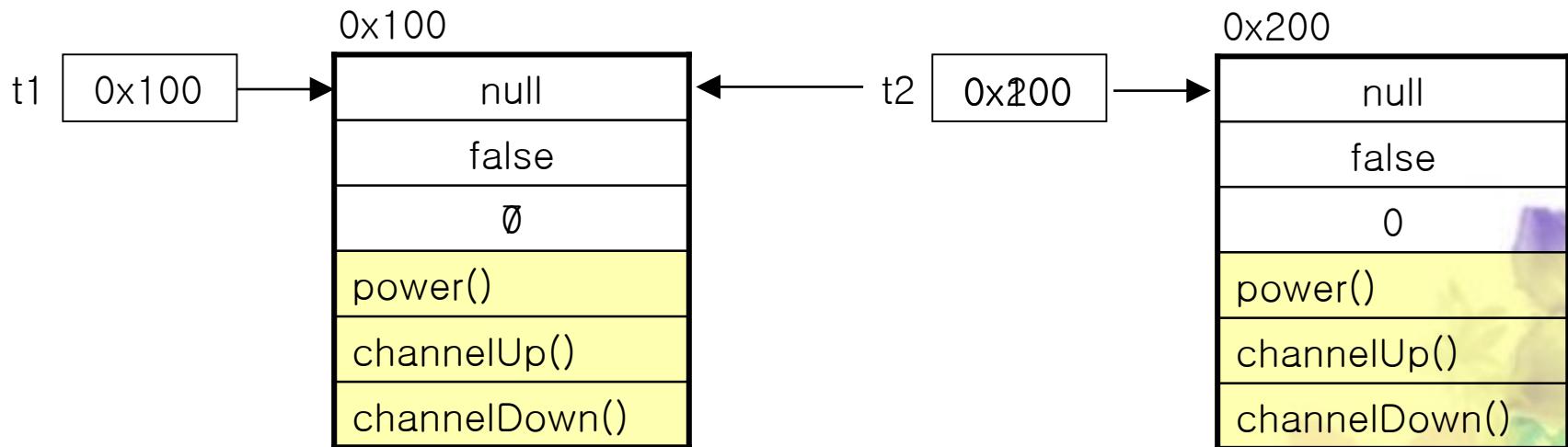
```
class Tv {  
    String color; // 색깔  
    boolean power; // 전원상태(on/off)  
    int channel; // 채널  
    void power() { power = !power; } // 전원on/off  
    void channelUp( channel++;) // 채널 높이기  
    void channelDown {channel--;} // 채널 낮추기  
}
```



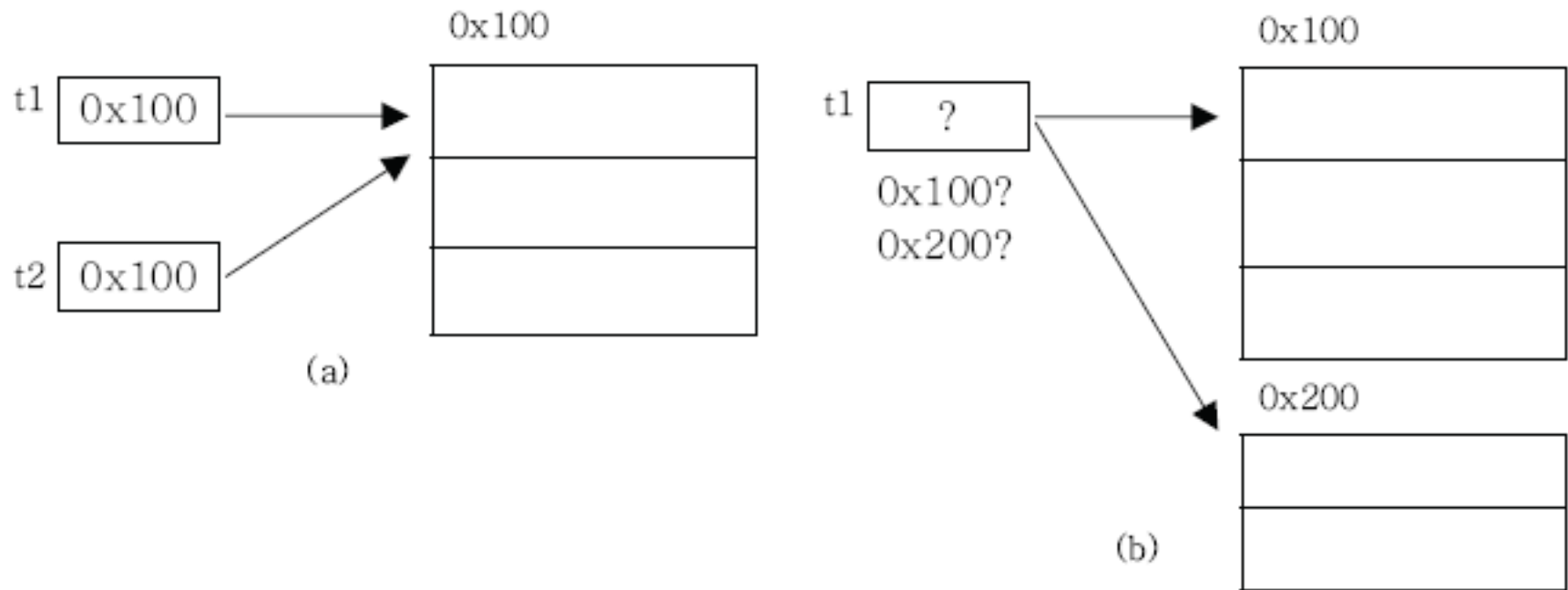


## 2.4 인스턴스의 생성과 사용(3/4)

```
Tv t1 = new Tv();  
Tv t2 = new Tv();  
t2 = t1;  
t1.channel = 7;  
System.out.println(t1.channel);  
System.out.println(t2.channel);
```



## 2.4 인스턴스의 생성과 사용(4/4)



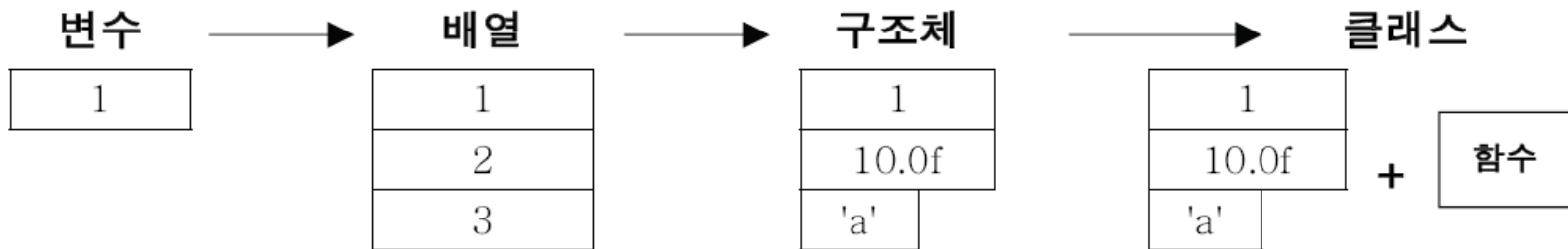
(a) 하나의 인스턴스를 여러 개의 참조변수가 가리키는 경우(가능)

(b) 여러 개의 인스턴스를 하나의 참조변수가 가리키는 경우(불가능)

[그림6-2] 참조변수와 인스턴스의 관계

## 2.5 클래스의 또 다른 정의

### 1. 클래스 - 데이터와 함수의 결합



[그림6-3] 데이터 저장개념의 발전과정

- ▶ 변수 - 하나의 데이터를 저장할 수 있는 공간
- ▶ 배열 - 같은 타입의 여러 데이터를 저장할 수 있는 공간
- ▶ 구조체 - 타입에 관계없이 서로 관련된 데이터들을 저장할 수 있는 공간
- ▶ 클래스 - 데이터와 함수의 결합(구조체+함수)

## 3.1 선언위치에 따른 변수의 종류

“변수의 선언위치가 변수의 종류와 범위(scope)을 결정한다.”

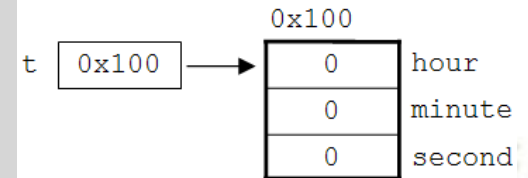
```
class Variables
```

```
{  
    int iv;           // 인스턴스변수  
    static int cv;    // 클래스변수 (static변수, 공유변수)  
  
    void method()  
    {  
        int lv = 0;   // 지역변수  
    }  
}
```

클래스영역

메서드영역

```
class Time {  
    int hour;  
    int minute;  
    int second;  
}
```



변수의 종류	선언위치	생성시기
클래스변수	클래스 영역	클래스가 메모리에 올라갈 때
인스턴스변수		인스턴스 생성시
지역변수	메서드 영역	변수 선언문 수행시

## 3.1 선언위치에 따른 변수의 종류

### ▶ 인스턴스변수(instance variable)

- 각 인스턴스의 개별적인 저장공간. 인스턴스마다 다른 값 저장가능
- 인스턴스 생성 후, '참조변수.인스턴스변수명'으로 접근
- 인스턴스를 생성할 때 생성되고, 참조변수가 없을 때 가비지컬렉터에 의해 자동제거됨

### ▶ 클래스변수(class variable)

- 같은 클래스의 모든 인스턴스들이 공유하는 변수
- 인스턴스 생성없이 '클래스이름.클래스변수명'으로 접근
- 클래스가 로딩될 때 생성되고 프로그램이 종료될 때 소멸

### ▶ 지역변수(local variable)

- 메서드 내에 선언되며, 메서드의 종료와 함께 소멸
- 조건문, 반복문의 블록{} 내에 선언된 지역변수는 블록을 벗어나면 소멸

## 3.2 클래스변수와 인스턴스변수

“인스턴스변수는 인스턴스가 생성될 때마다 생성되므로 인스턴스마다 각기 다른 값을 유지할 수 있지만, 클래스변수는 모든 인스턴스가 하나의 저장공간을 공유하므로 항상 공통된 값을 갖는다.”



속성	무늬 숫자
	폭 높이
기능	...

인스턴스변수

클래스변수

```
class Card {
```

```
    String kind;    // 무늬  
    int number;    // 숫자
```

```
}
```

```
    static int width = 100; // 폭  
    static int height = 250; // 높이
```

## 3.3 메서드(method)

### ▶ 메서드란?

- 작업을 수행하기 위한 명령문의 집합
- 어떤 값을 입력받아서 처리하고 그 결과를 돌려준다.  
(입력받는 값이 없을 수도 있고 결과를 돌려주지 않을 수도 있다.)

### ▶ 메서드의 장점과 작성지침

- 반복적인 코드를 줄이고 코드의 관리가 용이하다.
- 반복적으로 수행되는 여러 문장을 메서드로 작성한다.
- 하나의 메서드는 한 가지 기능만 수행하도록 작성하는 것이 좋다.
- 관련된 여러 문장을 메서드로 작성한다.

## 3.3 메서드(method)

- ▶ 메서드를 정의하는 방법 - 클래스 영역에만 정의할 수 있음

```
리턴타입 메서드이름 (타입 변수명, 타입 변수명, ... )
```

선언부

```
{
```

```
    // 메서드 호출시 수행될 코드
```

구현부

```
}
```

```
int add(int a, int b)
```

선언부

```
{
```

```
    int result = a + b;
```

```
    return result;    // 호출한 메서드로 결과를 반환한다.
```

구현부

```
}
```

```
void power() {        // 반환값이 없는 경우 리턴타입 대신 void를 사용한다.
```

```
    power = !power;
```

```
}
```



## 3.4 return문

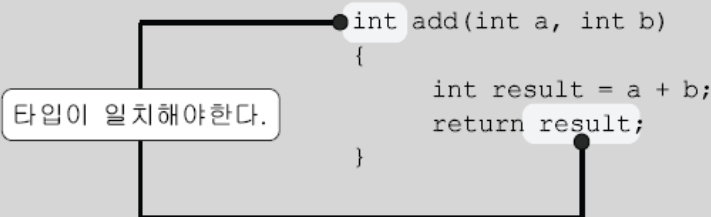
- ▶ 메서드가 정상적으로 종료되는 경우
  - 메서드의 블록{}의 끝에 도달했을 때
  - 메서드의 블록{}을 수행 도중 return문을 만났을 때
- ▶ return문
  - 현재 실행 중인 메서드를 종료하고 호출한 메서드로 되돌아간다.

1. 반환값이 없는 경우 - return문만 써주면 된다.

```
return;
```

2. 반환값이 있는 경우 - return문 뒤에 반환값을 지정해 주어야 한다.

```
return 반환값;
```



## 3.4 return문 - 주의사항

- ▶ 반환값이 있는 메서드는 모든 경우에 return문이 있어야 한다.

```
int max(int a, int b) {  
    if(a > b)  
        return a;  
}
```

```
int max(int a, int b) {  
    if(a > b)  
        return a;  
    else  
        return b;  
}
```

- ▶ return문의 개수는 최소화하는 것이 좋다.

```
int max(int a, int b) {  
    if(a > b)  
        return a;  
    else  
        return b;  
}
```

```
int max(int a, int b) {  
    int result = 0;  
    if(a > b)  
        result = a;  
    else  
        result = b;  
    return result;  
}
```

## 3.5 메서드의 호출

### ▶ 메서드의 호출방법

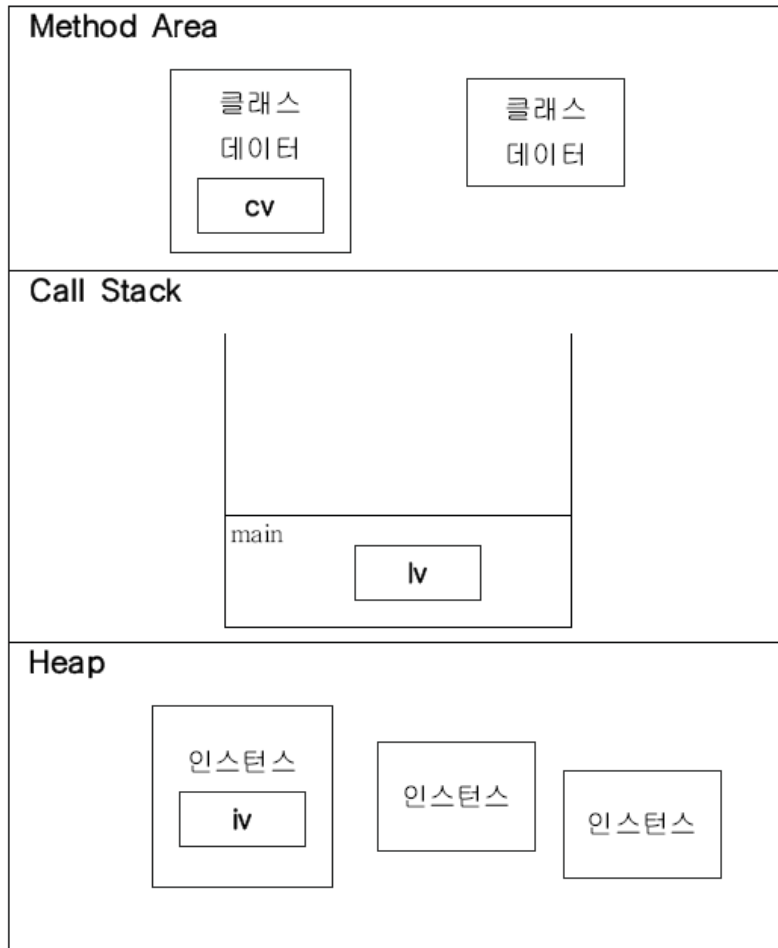
참조변수.메서드 이름 ();  
참조변수.메서드 이름(값1, 값2, ... );

// 메서드에 선언된 매개변수가 없는 경우  
// 메서드에 선언된 매개변수가 있는 경우

```
class MyMath {  
    long add(long a, long b) {  
        long result = a + b;  
        return result;  
    }  
    //  
    ...  
}
```

```
MyMath mm = new MyMath();  
  
long value = mm.add(1L, 2L);  
  
long add(long a, long b) {  
    long result = a + b;  
    return result;  
}
```

## 3.6 JVM의 메모리 구조



### ▶ 메서드영역(Method Area)

- 클래스 정보와 클래스변수가 저장되는 곳

### ▶ 호출스택(Call Stack)

- 메서드의 작업공간. 메서드가 호출되면 메서드 수행에 필요한 메모리공간을 할당받고 메서드가 종료되면 사용하던 메모리를 반환한다.

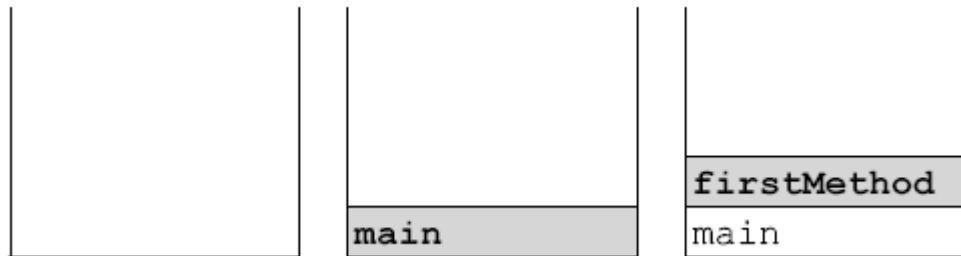
### ▶ 힙(Heap)

- 인스턴스가 생성되는 공간. new연산자에 의해서 생성되는 배열과 객체는 모두 여기에 생성된다.

## 3.6 JVM의 메모리 구조 - 호출스택

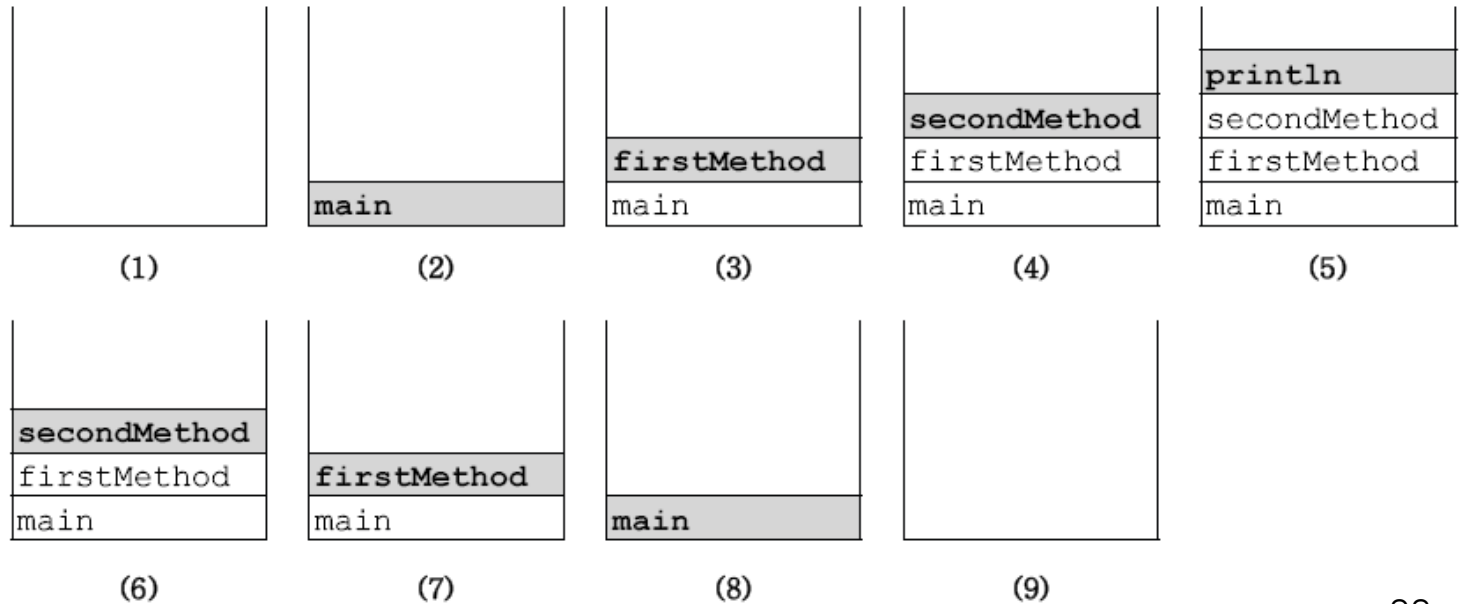
### ▶ 호출스택의 특징

- 메서드가 호출되면 수행에 필요한 메모리를 스택에 할당받는다.
- 메서드가 수행을 마치면 사용했던 메모리를 반환한다.
- 호출스택의 제일 위에 있는 메서드가 현재 실행중인 메서드다.
- 아래에 있는 메서드가 바로 위의 메서드를 호출한 메서드다.



# JVM의 메모리 구조 - 호출스택

```
class CallStackTest {  
    public static void main(String[] args) {  
        firstMethod();  
    }  
    static void firstMethod() {  
        secondMethod();  
    }  
    static void secondMethod() {  
        System.out.println("secondMethod()");  
    }  
}
```



# 기본형 매개변수와 참조형 매개변수

- ▶ 기본형 매개변수 – 변수의 값을 읽기만 할 수 있다.(read only)
- ▶ 참조형 매개변수 – 변수의 값을 읽고 변경할 수 있다.(read & write)

## 3.8 재귀호출(recursive call)

### ▶ 재귀호출이란?

- 메서드 내에서 자기자신을 반복적으로 호출하는 것
- 재귀호출은 반복문으로 바꿀 수 있으며 반복문보다 성능이 나쁨
- 이해하기 쉽고 간결한 코드를 작성할 수 있다

### ▶ 재귀호출의 예(例)

- 팩토리얼, 제곱, 트리은행, 폴더목록표시 등

\*팩토리얼 (factorial)

$5! = 5 * 4 * 3 * 2 * 1$

$f(n) = n * f(n-1)$  단,  $f(1) = 1$

```
long factorial(int n) {  
    long result = 0;  
    if(n==1) {  
        result = 1;  
    } else {  
        result = n * factorial(n-1);  
    }  
    return result;  
}
```



## 3.9 클래스메서드(static메서드)와 인스턴스메서드

### ▶ 인스턴스메서드

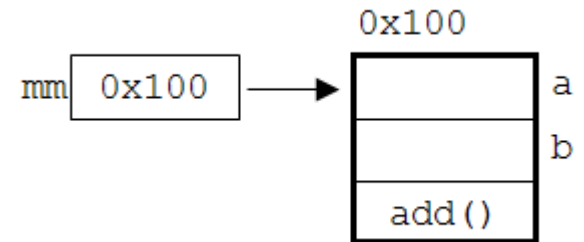
- 인스턴스 생성 후, '참조변수.메서드이름()'으로 호출
- 인스턴스변수나 인스턴스메서드와 관련된 작업을 하는 메서드
- 메서드 내에서 인스턴스변수 사용가능

### ▶ 클래스메서드(static메서드)

- 객체생성없이 '클래스이름.메서드이름()'으로 호출
- 인스턴스변수나 인스턴스메서드와 관련없는 작업을 하는 메서드
- 메서드 내에서 인스턴스변수 사용불가
- 메서드 내에서 인스턴스변수를 사용하지 않는다면 static을 붙이는 것을 고려한다.

## 3.9 클래스메서드(static메서드)와 인스턴스메서드

```
class MyMath2 {  
    long a, b;  
  
    long add() {    // 인스턴스메서드  
        return a + b;  
    }  
  
    static long add(long a, long b) { // 클래스메서드 (static)  
        return a + b;  
    }  
}
```



```
class MyMathTest2 {  
    public static void main(String args[]) {  
        System.out.println(MyMath2.add(200L,100L)); // 클래스메서드 호출  
        MyMath2 mm = new MyMath2(); // 인스턴스 생성  
        mm.a = 200L;  
        mm.b = 100L;  
        System.out.println(mm.add()); // 인스턴스메서드 호출  
    }  
}
```

## 3.10 멤버간의 참조와 호출(1/2) – 메서드의 호출

“같은 클래스의 멤버간에는 객체생성이나 참조변수 없이 참조할 수 있다. 그러나 static멤버들은 인스턴스멤버들을 참조할 수 없다.”

```
class TestClass {  
    void instanceMothod() {}          // 인스턴스메서드  
    static void staticMethod() {}    // static메서드  
  
    void instanceMothod2() {          // 인스턴스메서드  
        instanceMethod();            // 다른 인스턴스메서드를 호출한다.  
        staticMethod();              // static메서드를 호출한다.  
    }  
  
    static void staticMethod2() {     // static메서드  
        instanceMethod();            // 에러!!! 인스턴스메서드를 호출할 수 없다.  
        staticMethod();              // static메서드는 호출 할 수 있다.  
    }  
} // end of class
```

## 3.10 멤버간의 참조와 호출(2/2) – 변수의 접근

“같은 클래스의 멤버간에는 객체생성이나 참조변수 없이 참조할 수 있다. 그러나 static멤버들은 인스턴스멤버들을 참조할 수 없다.”

```
class TestClass2 {  
    int iv;           // 인스턴스변수  
    static int cv;    // 클래스변수  
  
    void instanceMethod() {           // 인스턴스에서  
        System.out.println(iv);      // 인스턴스변수를 사용할 수 있다.  
        System.out.println(cv);      // 클래스변수를 사용할 수 있다.  
    }  
  
    static void staticMethod() {      // static에서  
        System.out.println(iv);      // 에러!!! 인스턴스변수를 사용할 수 없다.  
        System.out.println(cv);      // 클래스변수를 사용할 수 있다.  
    }  
} // end of class
```

## 4. 메서드 오버로딩

### 4.1 메서드 오버로딩(method overloading)이란?

“하나의 클래스에 같은 이름의 메서드를 여러 개 정의하는 것을 메서드 오버로딩, 간단히 오버로딩이라고 한다.”

\* overload - vt. 과적하다. 부담을 많이 지우다.

### 4.2 오버로딩의 조건

- 메서드의 이름이 같아야 한다.
- 매개변수의 개수 또는 타입이 달라야 한다.
- 매개변수는 같고 리턴타입이 다른 경우는 오버로딩이 성립되지 않는다.  
(리턴타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다.)

## 4.3 오버로딩의 예(1/3)

### ▶ System.out.println메서드

- 다양하게 오버로딩된 메서드를 제공함으로써 모든 변수를 출력할 수 있도록 설계

```
void println()  
void println(boolean x)  
void println(char x)  
void println(char[] x)  
void println(double x)  
void println(float x)  
void println(int x)  
void println(long x)  
void println(Object x)  
void println(String x)
```

## 4.3 오버로딩의 예(1/2)

- ▶ 매개변수의 이름이 다른 것은 오버로딩이 아니다.

[보기1]

```
int add(int a, int b) { return a+b; }  
int add(int x, int y) { return x+y; }
```

- ▶ 리턴타입은 오버로딩의 성립조건이 아니다.

[보기2]

```
int add(int a, int b) { return a+b; }  
long add(int a, int b) { return (long) (a + b); }
```

## 4.3 오버로딩의 예(1/3)

- ▶ 매개변수의 타입이 다르므로 오버로딩이 성립한다.

[보기3]

```
long add(int a, long b) { return a+b; }  
long add(long a, int b) { return a+b; }
```

- ▶ 오버로딩의 올바른 예 – 매개변수는 다르지만 같은 의미의 기능수행

[보기4]

```
int add(int a, int b) { return a+b; }  
long add(long a, long b) { return a+b; }  
int add(int[] a) {  
    int result =0;  
  
    for(int i=0; i < a.length; i++) {  
        result += a[i];  
    }  
    return result;  
}
```



# 생성자(creator)

## 5.1 생성자(constructor)란?

### ▶ 생성자란?

- 인스턴스가 생성될 때마다 호출되는 ‘인스턴스 초기화 메서드’
- 인스턴스 변수의 초기화 또는 인스턴스 생성시 수행할 작업에 사용
- 몇가지 조건을 제외하고는 메서드와 같다.
- 모든 클래스에는 반드시 하나 이상의 생성자가 있어야 한다.

\* 인스턴스 초기화 - 인스턴스 변수에 적절한 값을 저장하는 것.

```
Card c = new Card();
```

1. 연산자 new에 의해서 메모리(heap)에 Card클래스의 인스턴스가 생성된다.
2. 생성자 Card()가 호출되어 수행된다.
3. 연산자 new의 결과로, 생성된 Card인스턴스의 주소가 반환되어 참조변수 c에 저장된다.

## 5.2 생성자의 조건

### ▶ 생성자의 조건

- 생성자의 이름은 클래스의 이름과 같아야 한다.
- 생성자는 리턴값이 없다. (하지만 void를 쓰지 않는다.)

```
클래스이름 (타입 변수명, 타입 변수명, ... ) {  
    // 인스턴스 생성시 수행될 코드  
    // 주로 인스턴스 변수의 초기화 코드를 적는다.  
}
```

```
class Card {  
    ...  
  
    Card() { // 매개변수가 없는 생성자.  
        // 인스턴스 초기화 작업  
    }  
  
    Card(String kind, int number) { // 매개변수가 있는 생성자  
        // 인스턴스 초기화 작업  
    }  
}
```

## 5.3 기본 생성자(default constructor)

### ▶ 기본 생성자란?

- 매개변수가 없는 생성자
- 클래스에 생성자가 하나도 없으면 컴파일러가 기본 생성자를 추가한다.  
(생성자가 하나라도 있으면 컴파일러는 기본 생성자를 추가하지 않는다.)

```
클래스이름 () { }
```

```
Card() { } // 컴파일러에 의해 추가된 Card클래스의 기본 생성자. 내용이 없다.
```

“모든 클래스에는 반드시  
하나 이상의 생성자가 있어야 한다.”

## 5.3 기본 생성자(default constructor)

“모든 클래스에는 반드시 하나 이상의 생성자가 있어야 한다.”

**[예제6-18]**/ch6/ConstructorTest.java

```
class Data1 {
    int value;
}

class Data2 {
    int value;
    Data2(int x) {    // 매개변수가 있는 생성자.
        value = x;
    }
}

class ConstructorTest {
    public static void main(String[] args) {
        Data1 d1 = new Data1();
        Data2 d2 = new Data2();    // compile error발생
    }
}
```

```
class Data1 {
    int value;
    Data1() {} // 기본생성자
}
```

## 5.4 매개변수가 있는 생성자

```
class Car {  
    String color;           // 색상  
    String gearType;       // 변속기 종류 - auto(자동), manual(수동)  
    int door;              // 문의 개수  
  
    Car() {} // 생성자  
    Car(String c, String g, int d) { // 생성자  
        color = c;  
        gearType = g;  
        door = d;  
    }  
}
```

```
Car c = new Car();  
c.color = "white";  
c.gearType = "auto";  
c.door = 4;
```



```
Car c = new Car("white", "auto", 4);
```

## 5.5 생성자에서 다른 생성자 호출하기 - this()

- ▶ this() - 생성자, 같은 클래스의 다른 생성자를 호출할 때 사용  
다른 생성자 호출은 생성자의 첫 문장에서만 가능

```
1 class Car {  
2     String color;  
3     String gearType;  
4     int door;  
5  
6     Car() {  
7         color = "white";  
8         gearType = "auto";  
9         door = 4;  
10    }  
11  
12    Car(String c, String g, int d) {  
13        color = c;  
14        gearType = g;  
15        door = d;  
16    }  
17  
18 }  
19
```

\* 코드의 재사용성을 높인 코드

```
Car() {  
    //Car("white","auto",4);  
    this("white","auto",4);  
}
```

```
Car() {  
    door = 5;  
    this("white","auto",4);  
}
```

## 5.6 참조변수 this

- ▶ this - 인스턴스 자신을 가리키는 참조변수. 인스턴스의 주소가 저장되어있음  
모든 인스턴스 메서드에 지역변수로 숨겨진 채로 존재

```
1 class Car {  
2     String color;  
3     String gearType;  
4     int door;  
5  
6     Car() {  
7         //Card("white", "auto", 4);  
8         this("white", "auto", 4);  
9     }  
10  
11     Car(String c, String g, int d) {  
12         color = c;  
13         gearType = g;  
14         door = d;  
15     }  
16 }  
17
```

\* 인스턴스변수와 지역변수를 구별하기  
위해 참조변수 this사용

```
Car(String c, String g, int d) {  
    color = c;  
    gearType = g;  
    door = d;  
}
```

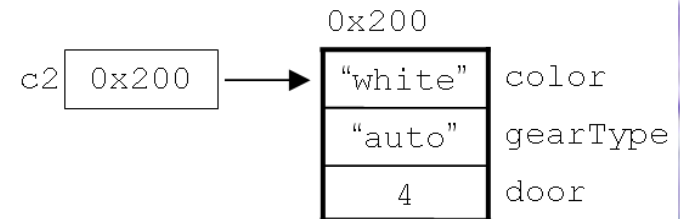
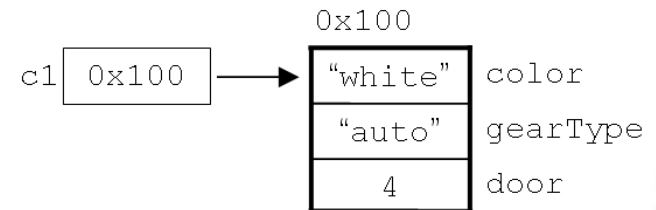


```
Car(String color, String gearType, int door) {  
    this.color = color;  
    this.gearType = gearType;  
    this.door = door;  
}
```

## 5.7 생성자를 이용한 인스턴스의 복사

- 인스턴스간의 차이는 인스턴스변수의 값 뿐 나머지는 동일하다.
- 생성자에서 참조변수를 매개변수로 받아서 인스턴스변수들의 값을 복사한다.
- 똑같은 속성값을 갖는 독립적인 인스턴스가 하나 더 만들어진다.

```
1 class Car {
2     String color;        // 색상
3     String gearType;     // 변속기 종류 - auto(자동), manual(수동)
4     int door;            // 문의 개수
5
6     Car() {
7         this("white", "auto", 4);
8     }
9
10    Car(String color, String gearType, int door) {
11        this.color = color;
12        this.gearType = gearType;
13        this.door = door;
14    }
15
16    Car(Car c) {           // 인스턴스의 복사를 위한 생성자.
17        color = c.color;
18        gearType = c.gearType;
19        door = c.door;
20    }
21 }
22
23 class CarTest3 {
24     public static void main(String[] args) {
25         Car c1 = new Car();
26         Car c2 = new Car(c1); // Car(Car c)를 호출
27     }
28 }
```



```
Car(Car c) {
    this(c.color, c.gearType, c.door);
}
```



# 6. 변수의 초기화

## 6.1 변수의 초기화

- 변수를 선언하고 처음으로 값을 저장하는 것
- 멤버변수(인스턴스변수, 클래스변수)와 배열은 각 타입의 기본값으로 자동초기화되므로 초기화를 생략할 수 있다.
- 지역변수는 사용전에 꼭!!! 초기화를 해주어야한다.

자료형	기본값
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

```
class InitTest {  
    int x;           // 인스턴스변수  
    int y = x;       // 인스턴스변수  
  
    void method1() {  
        int i;       // 지역변수  
        int j = i;    // 컴파일 에러!!! 지역변수를 초기화하지 않고 사용했음.  
    }  
}
```

## 6.1 변수의 초기화 - 예시(examples)

선언예	설 명
<pre>int i=10; int j=10;</pre>	int형 변수 i를 선언하고 10으로 초기화 한다. int형 변수 j를 선언하고 10으로 초기화 한다.
<pre>int i=10, j=10;</pre>	같은 타입의 변수는 콤마(,)를 사용해서 함께 선언하거나 초기화 할 수 있다.
<pre>int i=10, long j=0;</pre>	타입이 다른 변수는 함께 선언하거나 초기화할 수 없다.
<pre>int i=10; int j=i;</pre>	변수 i에 저장된 값으로 변수 j를 초기화 한다. 변수 j는 i의 값인 10으로 초기화 된다.
<pre>int j=i; int i=10;</pre>	변수 i가 선언되기 전에 i를 사용할 수 없다.

```
class Test
{
    int j = i;
    int i = 10; // 에러!!!
}
```

```
class Test
{
    int i = 10;
    int j = i; // OK
}
```

## 6.2 멤버변수의 초기화

### ▶ 멤버변수의 초기화 방법

#### 1. 명시적 초기화(explicit initialization)

```
class Car {  
    int door = 4;           // 기본형 (primitive type) 변수의 초기화  
    Engine e = new Engine(); // 참조형 (reference type) 변수의 초기화  
  
    //...  
}
```

#### 2. 생성자(constructor)

```
Car(String color, String gearType, int door){  
    this.color = color;  
    this.gearType = gearType;  
    this.door = door;  
}
```

#### 3. 초기화 블록(initialization block)

- 인스턴스 초기화 블록 : { }
- 클래스 초기화 블록 : static { }

## 6.3 초기화 블록(initialization block)

- ▶ 클래스 초기화 블록 - 클래스변수의 복잡한 초기화에 사용되며 클래스가 로딩될 때 실행된다.
- ▶ 인스턴스 초기화 블록 - 생성자에서 공통적으로 수행되는 작업에 사용되며 인스턴스가 생성될 때 마다 (생성자보다 먼저) 실행된다.

```
class InitBlock {  
    static { /* 클래스 초기화블럭 입니다. */ }  
  
    { /* 인스턴스 초기화블럭 입니다. */ }  
  
    // ...  
}
```

```
1 class StaticBlockTest {  
2     static int[] arr = new int[10]; // 명시적 초기화  
3  
4     static { // 배열 arr을 1~10사이의 값으로 채운다.  
5         for(int i=0;i<arr.length;i++) {  
6             arr[i] = (int) (Math.random()*10) + 1;  
7         }  
8     }  
9     //...  
10 }
```

## 6.4 멤버변수의 초기화 시기와 순서

- ▶ 클래스변수 초기화 시점 : 클래스가 처음 로딩될 때 단 한번
- ▶ 인스턴스변수 초기화 시점 : 인스턴스가 생성될 때 마다

```
1 class InitTest {  
2     static int cv = 1; // 명시적 초기화  
3     int iv = 1;        // 명시적 초기화  
4  
5     static { cv = 2; } // 클래스 초기화 블록  
6     { iv = 2; }        // 인스턴스 초기화 블록  
7  
8     InitTest() { // 생성자  
9         iv = 3;  
10    }  
11 }
```

```
InitTest it = new InitTest();
```

클래스 초기화			인스턴스 초기화			
기본값	명시적 초기화	클래스 초기화블록	기본값	명시적 초기화	인스턴스 초기화블록	생성자
cv 0	cv 1	cv 2	cv 2 iv 0	cv 2 iv 1	cv 2 iv 2	cv 2 iv 3
1	2	3	4	5	6	7

## 6.4 멤버변수의 초기화 시기와 순서

```
1 class Product {
2     static int count = 0;    // 생성된 인스턴스의 수를 저장하기 위한 변수
3     int serialNo;           // 인스턴스 고유의 번호
4
5     { // 인스턴스 초기화 블록 : 모든 생성자에서 공통적으로 수행될 코드
6         ++count;
7         serialNo = count;
8     }
9
10    public Product() {}
11 }
12
13 class ProductTest {
14     public static void main(String args[]) {
15         Product p1 = new Product();
16         Product p2 = new Product();
17         Product p3 = new Product();
18
19         System.out.println("p1의 제품번호(serial no)는 " + p1.serialNo);
20         System.out.println("p2의 제품번호(serial no)는 " + p2.serialNo);
21         System.out.println("p3의 제품번호(serial no)는 " + p3.serialNo);
22         System.out.println("생산된 제품의 수는 모두 "+Product.count+"개 입니다.");
23     }
24 }
```

Diagram illustrating the state of the `Product` class and its instances during execution:

- Static Variable:** `count` (represented by a box).
- Instance 1 (p1):** Address `0x100`. Its `serialNo` (represented by a box) is `0x100`.
- Instance 2 (p2):** Address `0x200`. Its `serialNo` (represented by a box) is `0x200`.
- Instance 3 (p3):** Address `0x300`. Its `serialNo` (represented by a box) is `0x300`.

# 상속(inheritance)

## 상속(inheritance)의 정의와 장점

### ▶ 상속이란?

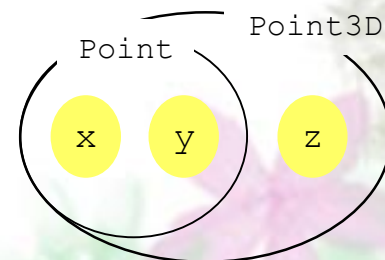
- 기존의 클래스를 재사용해서 새로운 클래스를 작성하는 것.
- 두 클래스를 조상과 자손으로 관계를 맺어주는 것.
- 자손은 조상의 모든 멤버를 상속받는다.(생성자, 초기화블럭 제외)
- 자손의 멤버개수는 조상보다 적을 수 없다.(같거나 많다.)

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

```
class 자손클래스 extends 조상클래스 {  
    // ...  
}
```

```
class Point3D {  
    int x;  
    int y;  
    int z;  
}
```

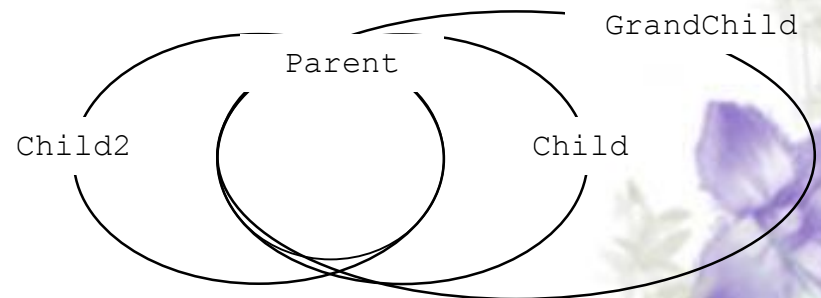
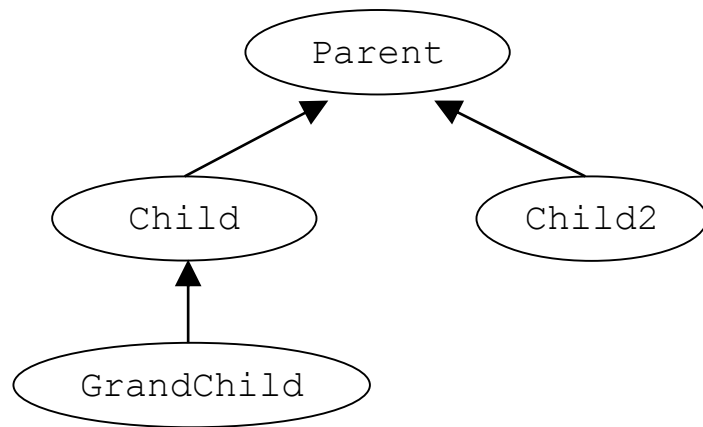
```
class Point3D extends Point {  
    int z;  
}
```



## 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, 이 들을 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;  
}
```

```
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

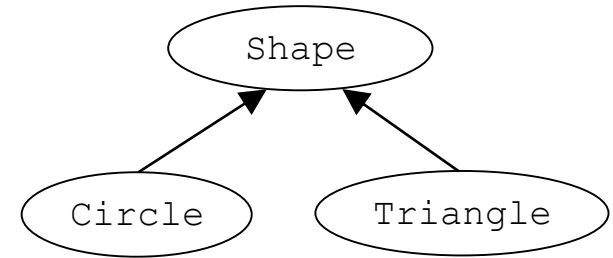
- 원(Circle)은 도형(Shape)이다.(A Circle is a Shape.) : 상속관계
- 원(Circle)은 점(Point)를 가지고 있다.(A Circle has a Point.) : 포함관계

```
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 한다.(C++은 다중상속 지원함)

```
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; }
}
```

상속

```
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();
    }
}
```

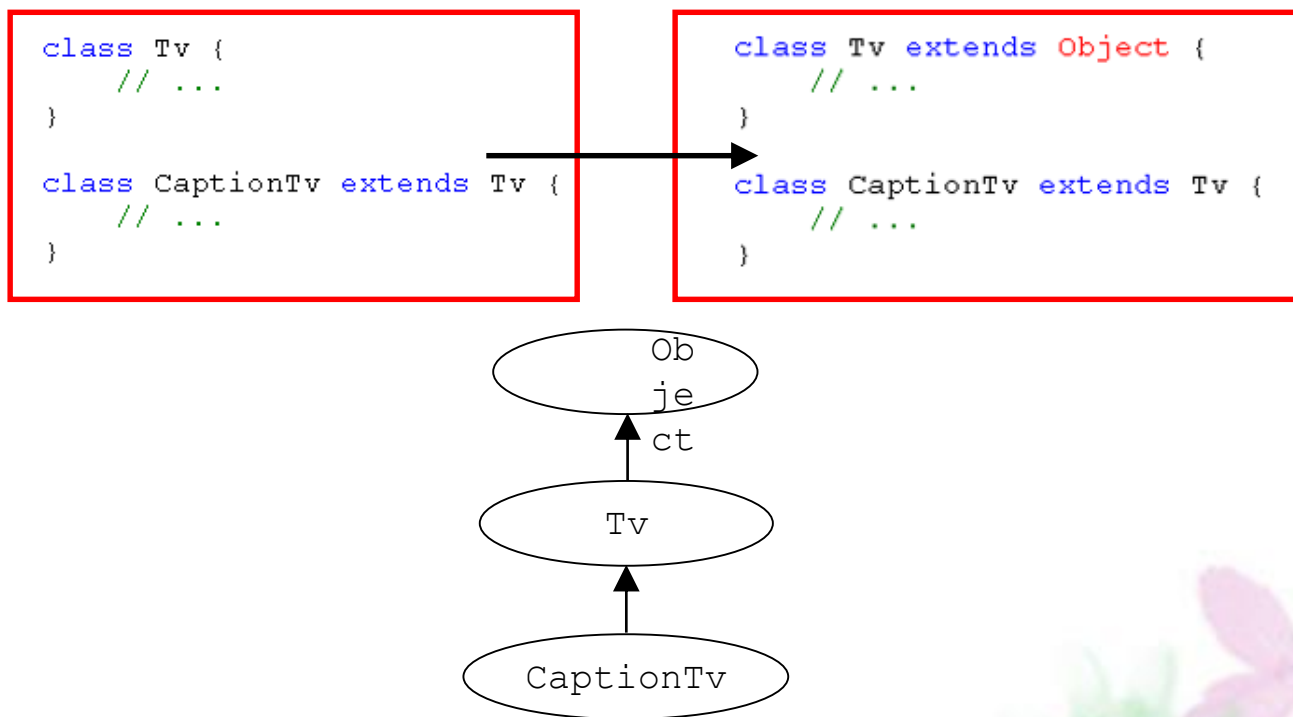
```
class VCR {
    boolean power; // 전원상태(on/off)
    int counter = 0;

    void power() { power = !power; }
    void play() { /* 내용생략*/ }
    void stop() { /* 내용생략*/ }
    void rew() { /* 내용생략*/ }
    void ff() { /* 내용생략*/ }
}
```

포함

## 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 protected라면, 범위가 같거나 넓은 protected나 public으로만 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 {  
        //..  
    }  
}
```



## 2.3 오버로딩 vs. 오버라이딩

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

오버라이딩(overriding) - 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 – this와 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 – this와 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;  
    }
```

```
    Point(int x, int y) {  
        super(); // Object();  
        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
```

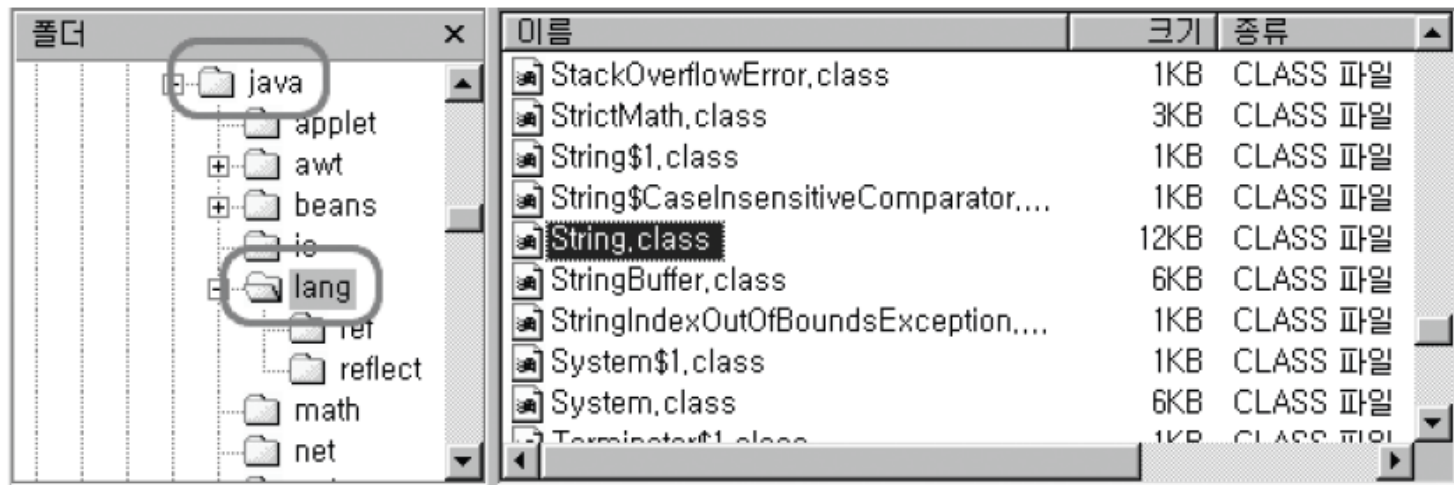
```
    Point3D(int x, int y, int z) {  
        super(); // Point()를 호출  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }
```

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

# 3. package와 import

## 3.1 패키지(package)

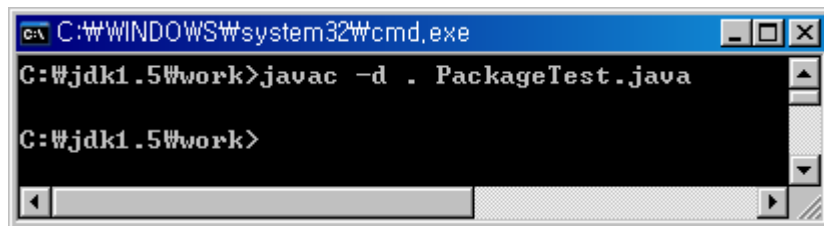
- 서로 관련된 클래스와 인터페이스의 묶음.
- 클래스가 물리적으로 클래스파일(\*.class)인 것처럼, 패키지는 물리적으로 폴더이다. 패키지는 서브패키지를 가질 수 있으며, '.'으로 구분한다.
- 클래스의 실제 이름(full name)은 패키지명이 포함된 것이다.  
(String클래스의 full name은 java.lang.String)
- rt.jar는 Java API의 기본 클래스들을 압축한 파일  
(JDK설치경로\jre\lib에 위치)



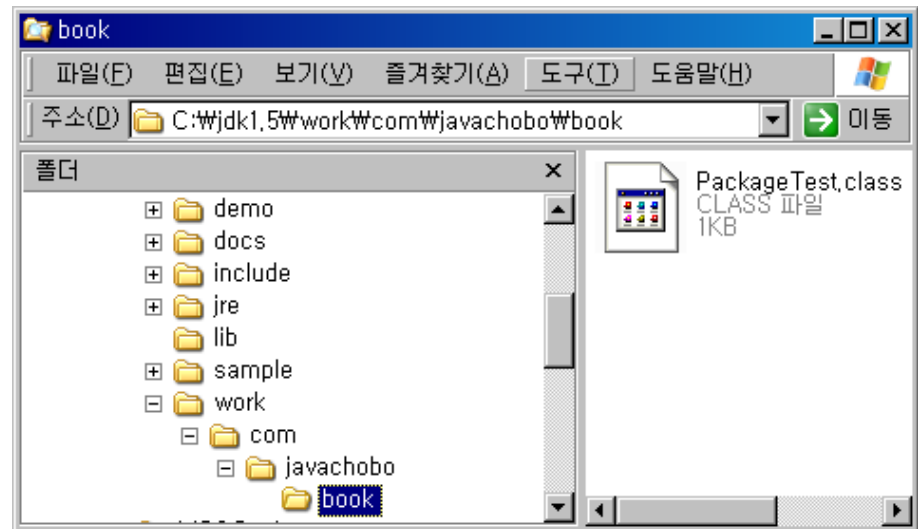
## 3.2 패키지의 선언

- 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 public클래스만 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 {}
```



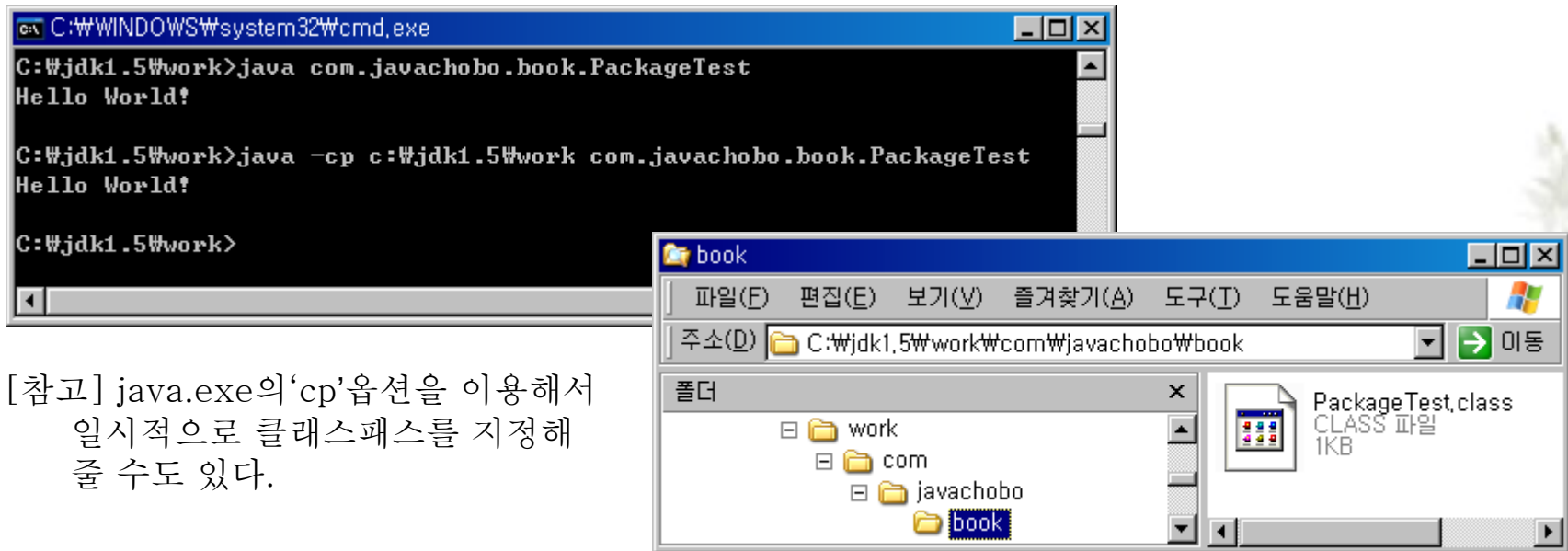
```
C:\WINDOWS\system32\cmd.exe
C:\jdk1.5\work>javac -d . PackageTest.java
C:\jdk1.5\work>
```





## 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.

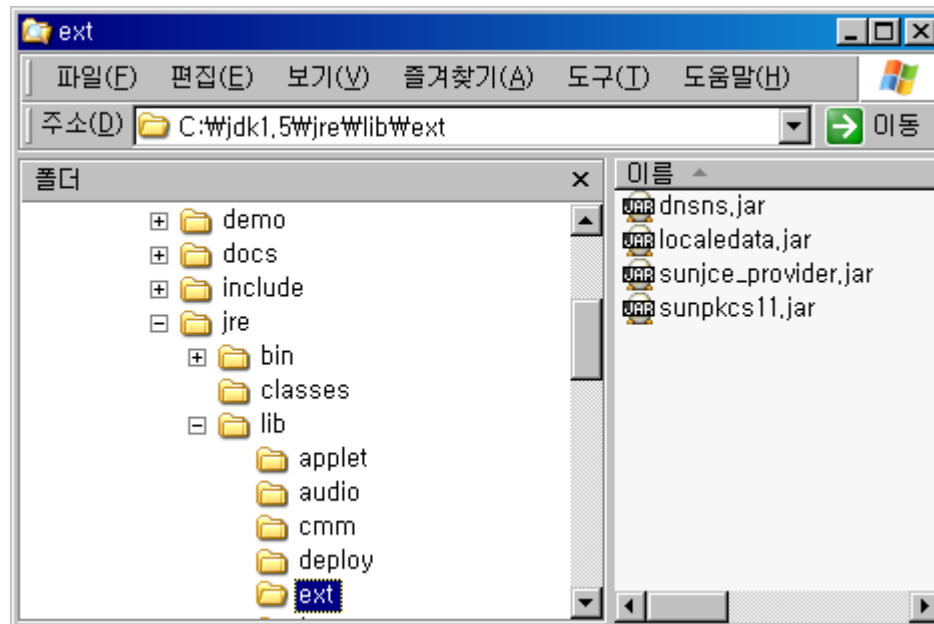


[참고] java.exe의 'cp' 옵션을 이용해서  
일시적으로 클래스패스를 지정해  
줄 수도 있다.



## 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.lang패키지의 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 - 선언예

- 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();  
    }  
}
```

## 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.

## 4.2 static – 클래스의, 공통적인

static이 사용될 수 있는 곳 – 멤버변수, 메서드, 초기화 블록

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

```
class StaticTest {  
    static int width = 200;  
    static int height = 120;  
  
    static { // 클래스 초기화 블록  
        // 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

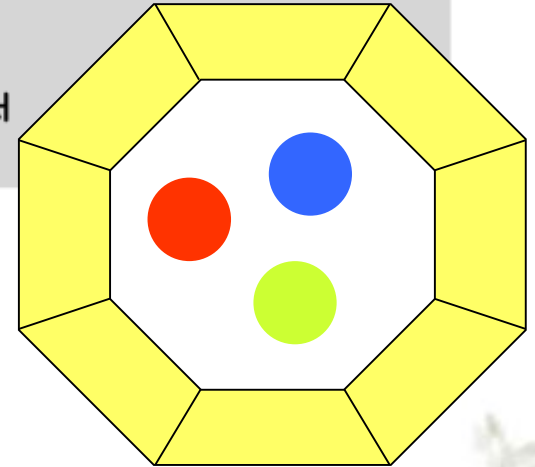
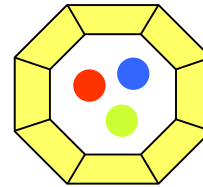
```
class AccessModifierTest {  
    int iv;           // 멤버변수 (인스턴스변수)  
    static int cv;    // 멤버변수 (클래스변수)  
  
    void method() {}  
}
```

## 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) 75
```

## 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 **static**이어야 한다.

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

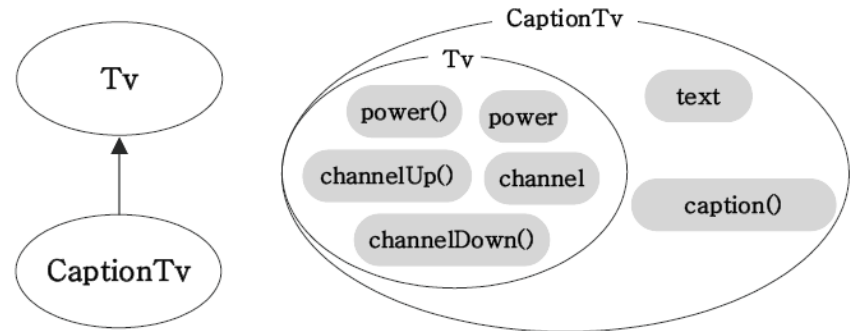
1. 메서드에 static과 abstract를 함께 사용할 수 없다.
  - static메서드는 몸통(구현부)이 있는 메서드에만 사용할 수 있기 때문이다.
2. 클래스에 abstract와 final을 동시에 사용할 수 없다.
  - 클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미이고, abstract는 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.
3. abstract메서드의 접근제어자가 private일 수 없다.
  - abstract메서드는 자손클래스에서 구현해주어야 하는데 접근 제어자가 private이면, 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)

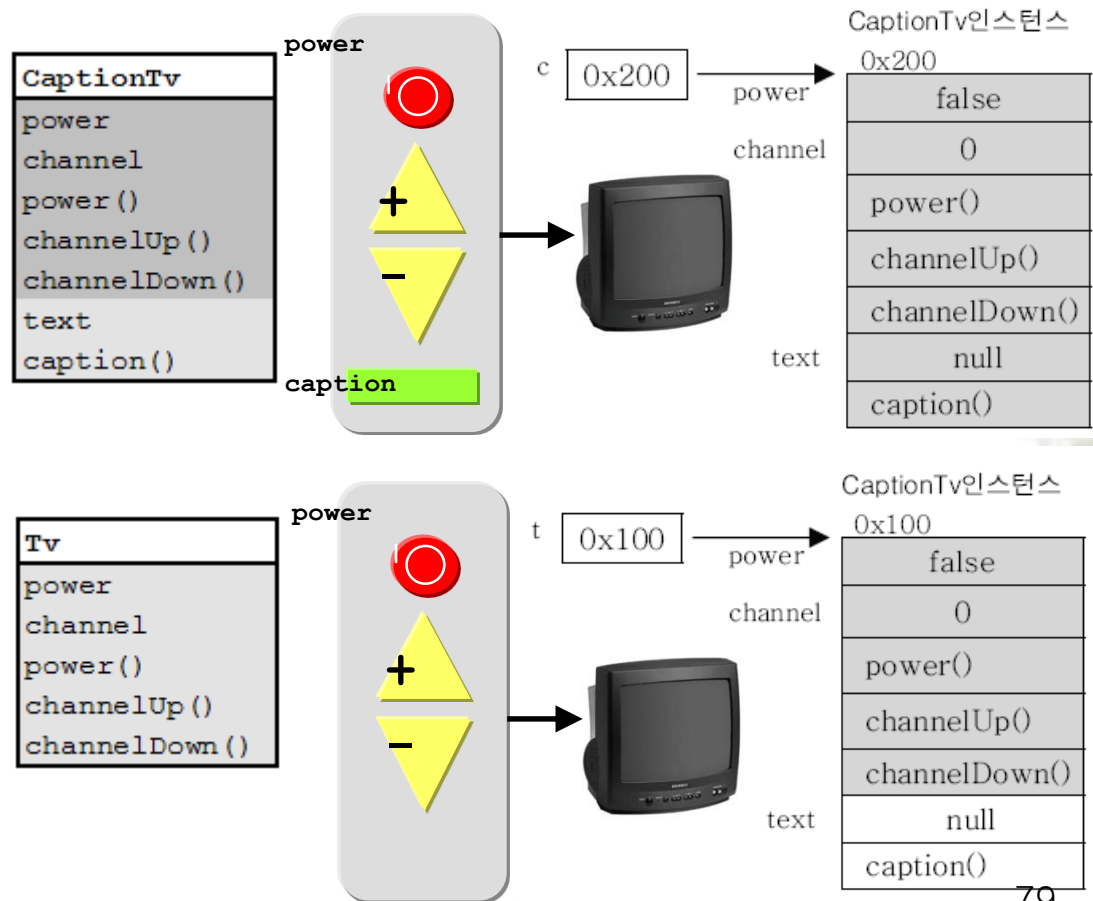
“하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것”

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 */}  
}
```



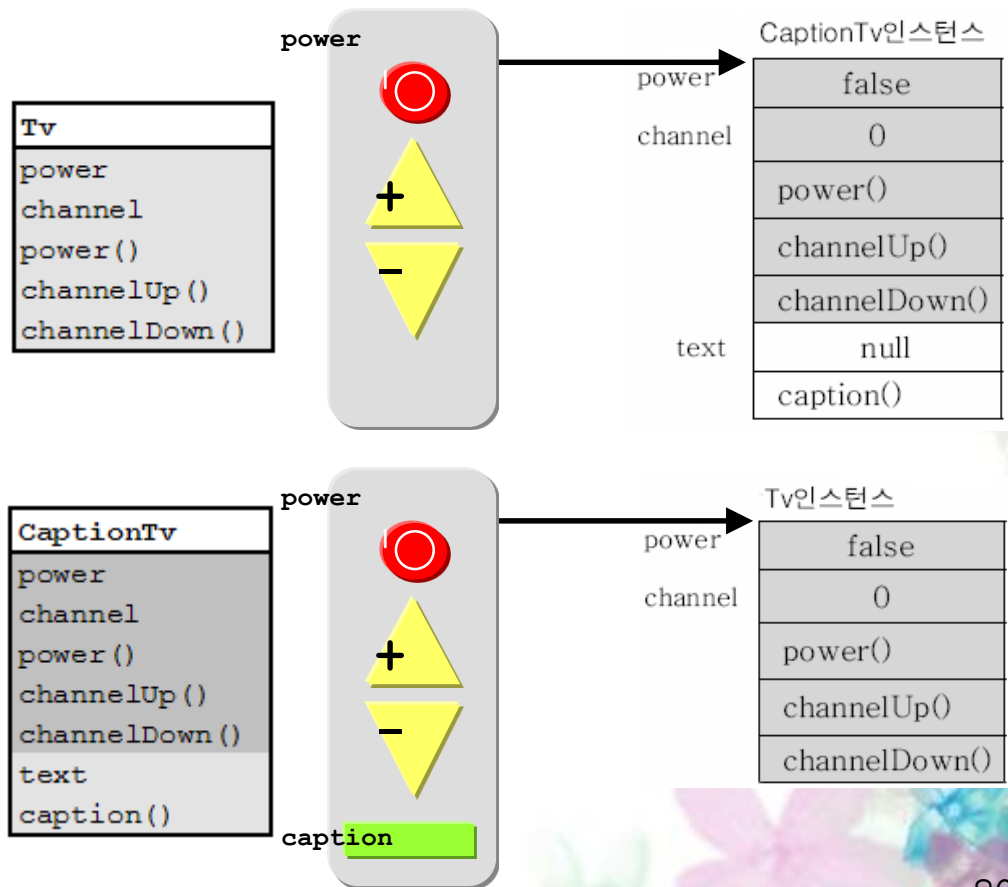


## 5.1 다형성(polymorphism)이란?(3/3)

“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 */ }  
}
```



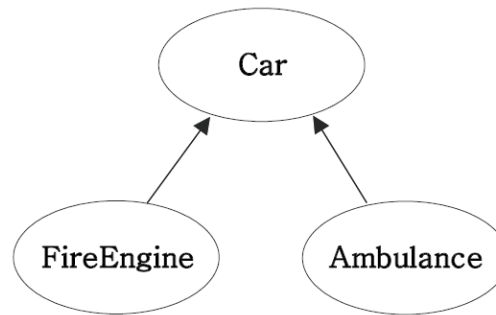


## 5.2 참조변수의 형변환

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

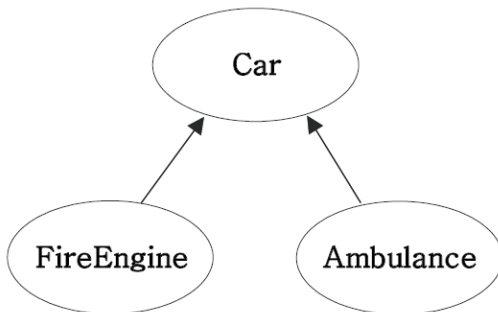


```
FireEngine f;  
Ambulance a;  
  
a = (Ambulance)f;  
f = (FireEngine)a;
```

```
Car car = null;  
FireEngine fe = new FireEngine();  
FireEngine fe2 = null;  
  
car = fe; // car = (Car)fe;  
fe2 = (FireEngine)car;
```

## 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

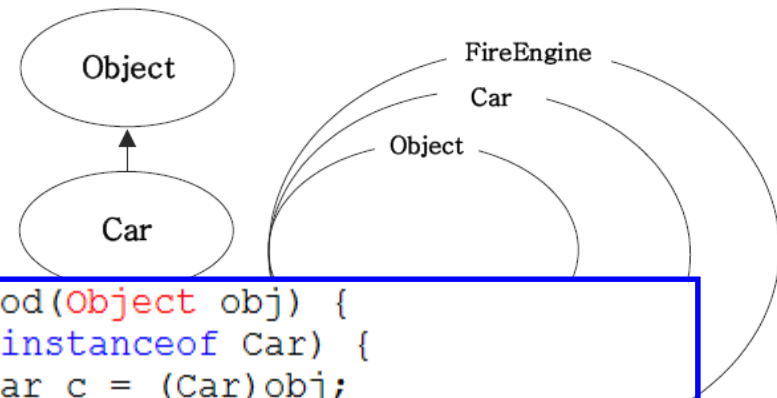
## 5.3 instanceof 연산자

- 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 instance.");  
        }  
    }  
}
```

```
----- java -----  
This is a FireEngine instance.  
This is a Car instance.  
This is an Object instance.
```

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 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  
}
```

```
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);
}
```

```
void summary() {
    int sum = 0;
    String cartList = "";
```

// 구매한 물품에 대한 정보를 요약해서 보여준다.  
// 구입한 물품의 가격합계  
// 구입한 물품목록

```
if(cart.isEmpty()) {
    System.out.println("구입한 물품이 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;
}
```

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

```
System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
System.out.println("구입하신 제품은 " + cartList + "입니다.");
```

```
}
```

### 메서드 / 생성자

Vector()

boolean add(Object o)

boolean remove(Object o)

boolean isEmpty()

Object get(int index)

int size()



어제

많은

일을

이루고

싶습니다~