

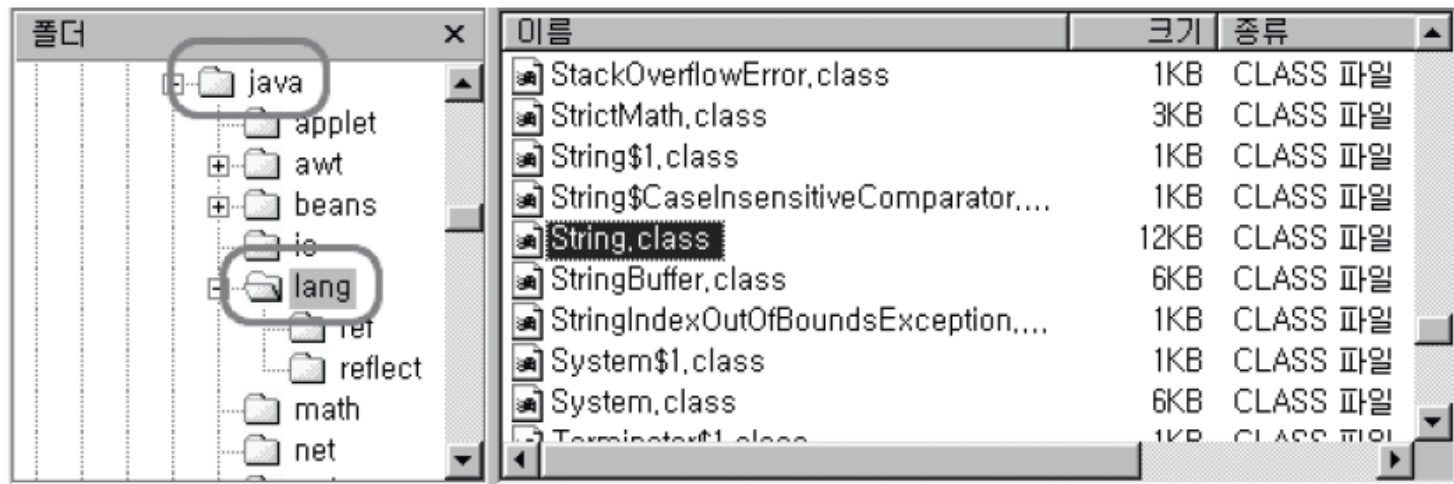
JAVA

- 제어 지시자와 정보은닉

package 와 import

패키지(package)

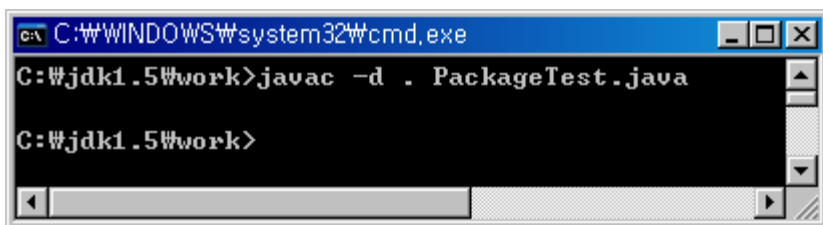
- 서로 관련된 클래스와 인터페이스의 묶음.
- 클래스가 물리적으로 클래스파일(*.class)인 것처럼, 패키지는 물리적으로 폴더이다. 패키지는 서브패키지를 가질 수 있으며, '.'으로 구분한다.
- 클래스의 실제 이름(full name)은 패키지명이 포함된 것이다.
(String클래스의 full name은 java.lang.String)
- rt.jar는 Java API의 기본 클래스들을 압축한 파일
(JDK설치경로\jre\lib에 위치)



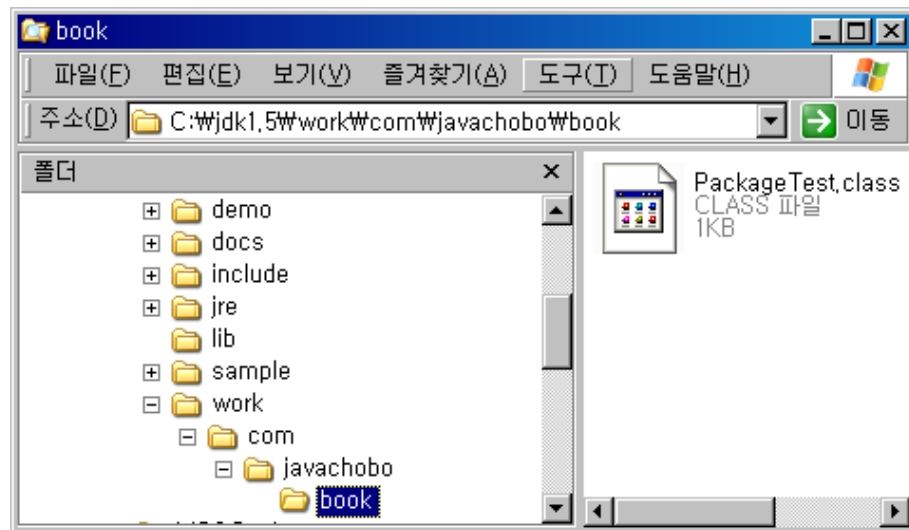
패키지의 선언

- 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:\j\jdk1.5\work>javac -d . PackageTest.java
C:\j\jdk1.5\work>
```




package 와 import

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!");  
}
```

import문의 선언

- import문은 패키지문과 클래스선언의 사이에 선언한다.

일반적인 소스파일(*.java)의 구성은 다음의 순서로 되어 있다.

1. package문
2. import 문
3. 클래스선언

- import문을 선언하는 방법은 다음과 같다.

```
import 패키지명.클래스명;  
또는  
Import 패키지명.*;
```

package 와 import

```
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 }
```

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();  
    }  
}
```


package 와 import

```
import java.text.SimpleDateFormat;  
import java.util.Date;
```

```
class ImportTest
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Date today = new Date();
```

```
        SimpleDateFormat date = new SimpleDateFormat("yyyy/MM/dd");
```

```
        SimpleDateFormat time = new SimpleDateFormat("hh:mm:ss a");
```

```
        System.out.println("오늘 날짜는 " + date.format(today));
```

```
        System.out.println("현재 시간은 " + time.format(today));
```

```
    }
```

```
}
```

접근제어 지시자와 정보은닉, 그리고 캡슐화

제어자

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.

접근제어자 – public, protected, default, private
그 외 – static, final, abstract, native, transient, synchronized, volatile, scriptfp

static – 클래스의, 공통적인

static이 사용될 수 있는 곳 – 멤버 변수, 메서드, 초기화 블록

| 제어자 | 대상 | 의미 |
|--------|-------|---|
| static | 멤버 변수 | 모든 인스턴스에 공통적으로 사용되는 클래스변수가 된다. 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다. 클래스가 메모리에 로드될 때 생성된다. |
| | 메서드 | 인스턴스를 생성하지 않고도 호출이 가능한 static메서드가 된다. Static메서드 내에서는 인스턴스 멤버들을 직접 사용할 수 없다. |

```
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;
    }
}
```

final – 마지막의, 변경될 수 없는

final 이 사용될 수 있는 곳 – 클래스, 매서드, 멤버변수, 지역변수

| 제어자 | 대상 | 의미 |
|--------|--------------|---|
| static | 클래스 | 변경될 수 없는 클래스, 확장될 수 없다는 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. |

```
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  
}
```

abstract – 추상의, 미완성의

abstract가 사용될 수 있는 속 – 클래스, 메서드

| 제어자 | 대상 | 의미 |
|----------|-----|-------------------------------------|
| abstract | 클래스 | 클래스 내에 추상메서드가 선언되어 있음을 의미한다. |
| | 메서드 | 선언부만 작성하고 구현부는 작성하지 않은 추상메서드임을 알린다. |

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

접근 제어지시자(access modifier)

- 멤버 또는 클래스에 사용되어, 외부로부터의 접근을 제한한다.

접근 제어지시자가 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 | 자손 클래스 | 전 체 |
|-----------|--------|--------|--------|-----|
| public | | | | |
| protected | | | | |
| default | | | | |
| private | | | | |

제어자의 조합

| 대상 | 사용 가능한 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.

정보은닉(Information Hiding)

```
class FruitSeller
{
    int numOfApple;
    int myMoney;
    final int APPLE_PRICE;
    public FruitSeller(int money, int appleNum, int price)
    {
        myMoney=money;
        numOfApple=appleNum;
        APPLE_PRICE=price;
    }
    public int saleApple(int money)
    {
        int num=money/APPLE_PRICE;
        numOfApple-=num;
        myMoney+=money;
        return num;
    }
    public void showSaleResult()
    {
        System.out.println("남은 사과: " + numOfApple);
        System.out.println("판매 수익: " + myMoney);
    }
}
```

```
class FruitBuyer
{
    int myMoney;
    int numOfApple;

    public FruitBuyer(int money)
    {
        myMoney=money;
        numOfApple=0;
    }

    public void buyApple(FruitSeller seller, int money)
    {
        numOfApple+=seller.saleApple(money);
        myMoney-=money;
    }

    public void showBuyResult()
    {
        System.out.println("현재 잔액: " + myMoney);
        System.out.println("사과 개수: " + numOfApple);
    }
}
```

```
class FruitSalesMain4
{
    public static void main(String[] args)
    {
        FruitSeller seller = new FruitSeller(0, 30, 1500);
        FruitBuyer buyer = new FruitBuyer(10000);

        seller.myMoney += 500; // 돈 500원 내고!
        buyer.myMoney -= 500;

        seller.numOfApple -= 20;
        buyer.numOfApple += 20; // 사과 스무 개 가져가는 꼴이네!

        System.out.println("과일 판매자의 현재 상황");
        seller.showSaleResult();

        System.out.println("과일 구매자의 현재 상황");
        buyer.showBuyResult();
    }
}
```

객체지향 관점에서 넌 빵점이야!

메소드를 통해서 정의한 판매의 규칙이 완전히 무너졌다!

```
class FruitSalesMain4
{
    public static void main(String[] args)
    {
        FruitSeller seller = new FruitSeller(0, 30, 1500);
        FruitBuyer buyer = new FruitBuyer(10000);

        seller.myMoney += 500;
        buyer.myMoney -= 500;

        seller.numOfApple -= 20;
        buyer.numOfApple += 20;

        System.out.println("과일 판매자의 현재 상황");
        seller.showSaleResult();

        System.out.println("과일 구매자의 현재 상황");
        buyer.showBuyResult();
    }
}
```

```
class FruitSeller {  
    private int numOfApple;  
    private int myMoney;  
    private final int APPLE_PRICE;  
  
    public FruitSeller(int money, int appleNum, int price) {  
        myMoney=money;  
        numOfApple=appleNum;  
        APPLE_PRICE=price;  
    }  
    public int saleApple(int money) {  
        int num=money/APPLE_PRICE;  
        numOfApple-=num;  
        myMoney+=money;  
        return num;  
    }  
    public void showSaleResult() {  
        System.out.println("남은 사과: " + numOfApple);  
        System.out.println("판매 수익: " + myMoney);  
    }  
}
```

```
class FruitBuyer {
```

```
    private int myMoney;  
    private int numOfApple;
```

```
    public FruitBuyer(int money) {  
        myMoney=money;  
        numOfApple=0;  
    }
```

```
    public void buyApple(FruitSeller seller, int money) {  
        numOfApple+=seller.saleApple(money);  
        myMoney-=money;  
    }
```

```
    public void showBuyResult() {  
        System.out.println("현재 잔액: " + myMoney);  
        System.out.println("사과 개수: " + numOfApple);  
    }
```

```
}
```



```
class FruitSalesMain5 {  
  
    public static void main(String[] args) {  
  
        FruitSeller seller = new FruitSeller(0, 30, 1500);  
        FruitBuyer buyer = new FruitBuyer(10000);  
  
        buyer.buyApple(seller, 4500); // 유일한 과일 구매 방법  
  
        System.out.println("과일 판매자의 현재 상황");  
        seller.showSaleResult();  
  
        System.out.println("과일 구매자의 현재 상황");  
        buyer.showBuyResult();  
    }  
}
```

```
class FruitSeller
```

```
{
```

```
    private int numOfApple;
```

```
    private int myMoney;
```

```
    private final int APPLE_PRICE;
```

외부에서의 접근 금지!

```
class FruitBuyer
```

```
{
```

```
    private int myMoney;
```

```
    private int numOfApple;
```

외부에서의 접근 금지!

private과 같은 키워드를 가리켜 접근제어 지시자라 한다.

인스턴스 변수의 private 선언으로 인해서 메소드가 유일한 접근수단이 되었다.

접근제어 지시자

Private과 public

private – 클래스 내부(메소드)에서만 접근 가능.

public – 어디서든 접근 가능(접근을 제안하지 않는다).

```
class AAA
{
    private int num;
    public void setNum(int n) { num=n; }
    public int getNum() { return num; }
    . . . .
}
```

```
class BBB
{
    public accessAAA(AAA inst)
    {
        inst.num=20;
        inst.setNum(20);
        System.out.println(inst.getNum());
    }
    . . . .
}
```

num은 private 멤버이므로

컴파일 불가!

setNum, getNum은 public이므로

호출 가능!

접근제어 지시자를 선언하지 않는 경우(default)

접근제어 지시자 선언을 하지 않은 경우 – default 선언이 된다.
default 선언은 동일 패키지 내에서의 접근 허용이 가능하다.

```
package orange;

class AAA    // package orange로 묶인다.
{
    int num; default 선언!
    . . . .
}

class BBB    // package orange로 묶인다.
{
    public init(AAA a) { a.num=20; }
    . . . .
}
```

BBB는 AAA와 동일 패키지로 선언
되었으므로 접근 가능!

protected

protected – 상속관계에 놓여 있어도 접근을 허용한다.

default 선언으로 접근 가능한 영역 접근가능 하고 상속관계에서도 가능하다.

```
class AAA
{
    protected int num;
    . . . .
}

class BBB extends AAA
{
    protected int num;          // 상속된 인스턴스 변수
    public init(int n) { num=n; } // 상속된 변수 num의 접근!
    . . . .
}
```

상속을 의미함.

```
package valtest;
```

```
public class QQQ {  
    protected int pnum;  
    int inum;  
}
```

```
package qqq;
```

```
import valtest.QQQ;
```

```
public class QQQSub extends QQQ {  
    void printTest() {  
        System.out.println(pnum);  
        //System.out.println(inum); // 변수 참조 불가  
    }  
}
```

접근제어 지시자의 관계

| 지시자 | 클래스 내부 | 동일 패키지 | 상속받은 클래스 | 이외의 영역 |
|-----------|--------|--------|----------|--------|
| private | ● | X | X | X |
| default | ● | ● | X | X |
| protected | ● | ● | ● | |
| public | ● | ● | ● | ● |

public > protected > default > private

Public 클래스와 default 클래스

default 클래스

default 클래스

- 동일한 패키지 내에 정의된 클래스에 의해서만 인스턴스 생성이 가능하다.

```
package apple;  
class AAA    // default 클래스 선언  
{  
    . . . .  
}
```

```
package peal;  
class BBB    // default 클래스 선언  
{  
    public void make()  
    {  
        apple.AAA inst=new apple.AAA();  
        . . . .  
    }  
    . . . .  
}
```

인스턴스 생성 불가! AAA와 BBB의
패키지가 다르므로!

파일을 대표할 정도로 외부에
의미가 있는 클래스 파일을
public으로 선언한다.

Public 클래스

public 클래스

- 동일한 패키지 내에 정의된 클래스에 의해서만 인스턴스 생성이 가능하다.
- 하나의 소스파일에 하나의 클래스만 public으로 선언이 가능하다.
- public 클래스 이름과 소스파일 이름은 일치해야 한다.

```
package apple;
public class AAA    // public 클래스 선언
{
    . . . . .
}
```

```
package peal;
public class BBB    // public 클래스 선언
{
    public void make()
    {
        apple.AAA inst=new apple.AAA();
        . . . . . AAA는 public 클래스이므로 어디서
        든 인스턴스 생성 가능!
    }
    . . . . .
}
```

생성자는 public인데, 클래스는 default?

```
public class AAA
{
    AAA(){...}
    . . . .
}
```

클래스는 public으로 선언되어서 파일을 대표하는 상황!
그럼에도 불구하고 생성자가 default로 선언되어서 동
일 패키지 내에서만 인스턴스 생성을 허용하는 상황!

```
class BBB
{
    public BBB(){...}
    . . . .
}
```

생성자가 public임에도 클래스가 default로 선언되어
서 동일 패키지 내에서만 인스턴스 생성이 허용되는 상
황!

디폴트 생성자

디폴트 생성자의 접근제어 지시자는 클래스의 선언형태에 따라서 결정 된다.

```
public class AAA
{
    public AAA() {...}
    . . . .
}
```

public 클래스에 디폴트로
삽입되는 생성자

```
class BBB
{
    BBB() {...}
    . . . .
}
```

default 클래스에 디폴트로
삽입되는 생성자

생성자는 public인데, 클래스는 default?

생성자의 접근 제어지시자

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

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

생성자는 public인데, 클래스는 default?

생성자를 이용한 final 멤버변수 초기화

- final이 붙은 변수는 상수이므로 보통은 선언과 초기화를 동시에 하지만, 인스턴스 변수의 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);  
}
```

제어자

```
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;  
    }  
}
```


제어자

```
class FinalCardTest {  
    public static void main(String args[]) {  
        Card c = new Card("HEART", 10);  
  
        //                c.NUMBER = 5;  
        // 에러!!! cannot assign a value to final variable NUMBER  
  
        System.out.println(c.KIND);  
        System.out.println(c.NUMBER);  
    }  
}
```

어떤 클래스를 public으로 선언 할까요?

```
package orange.cal;
public class Calculator
{
    private Adder adder;
    private Subtractor subtractor;
    public Calculator()
    {
        adder = new Adder();
        subtractor = new Subtractor();
    }
    public int addTwoNumber(int num1, int num2)
    {
        return adder.addTwoNumber(num1, num2);
    }
    public int subTwoNumber(int num1, int num2)
    {
        return subtractor.subTwoNumber(num1, num2);
    }
    public void showOperatingTimes()
    {
        System.out.println("덧셈 횟수: " + adder.getCntAdd());
        System.out.println("뺄셈 횟수: " + subtractor.getCntSub());
    }
}
```

```
class Adder
{
    private int cntAdd;

    Adder() { cntAdd = 0; }
    int getCntAdd() { return cntAdd; }
    int addTwoNumber(int num1, int num2)
    {
        cntAdd++;
        return num1 + num2;
    }
}
```

```
class Subtractor
{
    private int cntSub;

    Subtractor() { cntSub = 0; }
    int getCntSub() { return cntSub; }
    int subTwoNumber(int num1, int num2)
    {
        cntSub++;
        return num1 - num2;
    }
}
```

```
import orange.cal.Calculator;
```

```
class CalculatorMain
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Calculator cal = new Calculator();
```

```
        System.out.println("1+2=" + cal.addTwoNumber(1, 2));
```

```
        System.out.println("2+4=" + cal.addTwoNumber(2, 4));
```

```
        System.out.println("5-1=" + cal.subTwoNumber(5, 1));
```

```
        cal.showOperatingTimes();
```

```
    }
```

```
}
```

파일당 하나의 외부 제공 클래스 정의하기

외부에서는 Calculaor 클래스의 존재만 알면 된다. Adder와 Subtractor 클래스의 존재는 알 필요 없다.
그리고 이렇게 외부에 노출시킬 클래스를 public으로 선언한다.

```
public class Calculator
{
    private Adder adder;
    private Subtractor subtractor;

    public Calculator()
    {
        adder = new Adder();
        subtractor = new Subtractor();
    }

    public int addTwoNumber(int num1, int num2)
    {
        return adder.addTwoNumber(num1, num2);
    }

    public int subTwoNumber(int num1, int num2)
    {
        return subtractor.subTwoNumber(num1, num2);
    }

    public void showOperatingTimes()
    {
        System.out.println("덧셈 횟수 : " + adder.getCntAdd());
        System.out.println("뺄셈 횟수 : " + subtractor.getCntSub());
    }
}
```

```
class Adder
{
    private int cntAdd;

    Adder() { cntAdd=0; }
    int getCntAdd() { return cntAdd; }
    int addTwoNumber(int num1, int num2)
    {
        cntAdd++;
        return num1 + num2;
    }
}
```

```
class Subtractor
{
    private int cntSub;

    Subtractor() { cntSub=0; }
    int getCntSub() { return cntSub; }
    int subTwoNumber(int num1, int num2)
    {
        cntSub++;
        return num1 - num2;
    }
}
```

위의 세 클래스는 하나의 소스파일 Calculator.java에 정의되어 있다.

클래스를 하나로 묶으면 안될까요?

계산기 기능의 완성을 위해서 Calculator 클래스 이외에 Adder, Subtractor 클래스를 별도로 구분할 필요가 있는가?



질문에 대한 답변!

- 변경이 있을 때, 변경되는 클래스의 범위를 줄일 수 있다.
- 작은 크기의 클래스를 다른 클래스의 정의에 활용할 수 있다.

객체지향에서는 아주 큰 하나의 클래스보다, 아주 작은
여러 개의 클래스가 더 큰 힘과 위력을 발휘한다!