

## 06. 메소드와 캡슐화

# I] 메소드(method) 개념

## [1] 메소드 정의

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

## [2] 메소드의 장점과 작성지침

- 반복적으로 수행되는 여러 문장을 메소드로 작성
- 하나의 메소드는 한 가지 기능만 수행하도록 작성하는 것 권장

## 2] 메소드를 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] 메소드를 정의 예시문

```
class AAA {  
    public void printName() {  
        System.out.println("AAA");  
    }  
}  
  
class BBB {  
    public void printName() {  
        System.out.println("BBB");  
    }  
}  
  
class ClassPath {  
    public static void main(String args[]) {  
        AAA aaa=new AAA();  
        aaa.printName();  
        BBB bbb=new BBB();  
        bbb.printName();  
    }  
}
```

## 4] 메소드 Return 문

메소드가 정상적으로 종료되는 경우

- 메소드의 블록{}의 끝에 도달했을 때
- 메소드의 블록{}을 수행 도중 return문을 만났을 때

1. 반환값이 없는 경우 - return문만 써주면 된다.

```
return;
```

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

```
return 반환값;
```

## 4] return 문 - 주의사항

- ▶ 반환값이 있는 메소드는 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;  
}
```

# 5] 메소드의 호출

## ▶ 메소드의 호출방법

참조변수.메소드 이름 ();                      // 메소드에 선언된 매개변수가 없는 경우  
참조변수.메소드 이름(값1, 값2, ... );        // 메소드에 선언된 매개변수가 있는 경우

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

## 5] 메소드의 호출 예시 |

```
class MethodReturns {
    public static void main(String[] args) {
        int result=addder(4, 5);
        System.out.println("4와 5의 합: " + result);
        System.out.println("3.5의 제곱: " + square(3.5));
    }
    public static int addder(int num1, int num2) {
        int addResult=num1+num2;
        return addResult;
    }
    public static double square(double num) {
        return num*num;
    }
}

class ReculFactorial {
    public static void main(String[] args) {
        System.out.println("7 factorial: " + factorial(7));
        System.out.println("12 factorial: " + factorial(12));
    }

    public static int factorial(int n) {
        if(n==1) return 1;
        else return n*factorial(n-1);
    }
}
```



## 5] 메소드의 호출 예시2

[연습문제1] Account 클래스에 잔액 조회 메소드 로 잔액조회하기

- 1) 속성 계좌번호, 예금주, 잔액
- 2) 기능 예금하다, 인출하다, 조회하다(추가)

Account
accountNo:String ownerName:String balance:int
deposit():void withdraw():void getBal():int



홍길동: 300000

이순신: 400000

실행결과

클래스 구성도

## 6] 메소드의 호출 예시2(계속)

```
public class Account {  
    String accountNo;  
    String owerName;  
    int balance;  
    void deposit(){  
        balance += 10000;  
    }  
    void withdraw(){  
        balance -= 10000;  
    }  
    int getBal() {  
        return balance;  
    }  
}
```

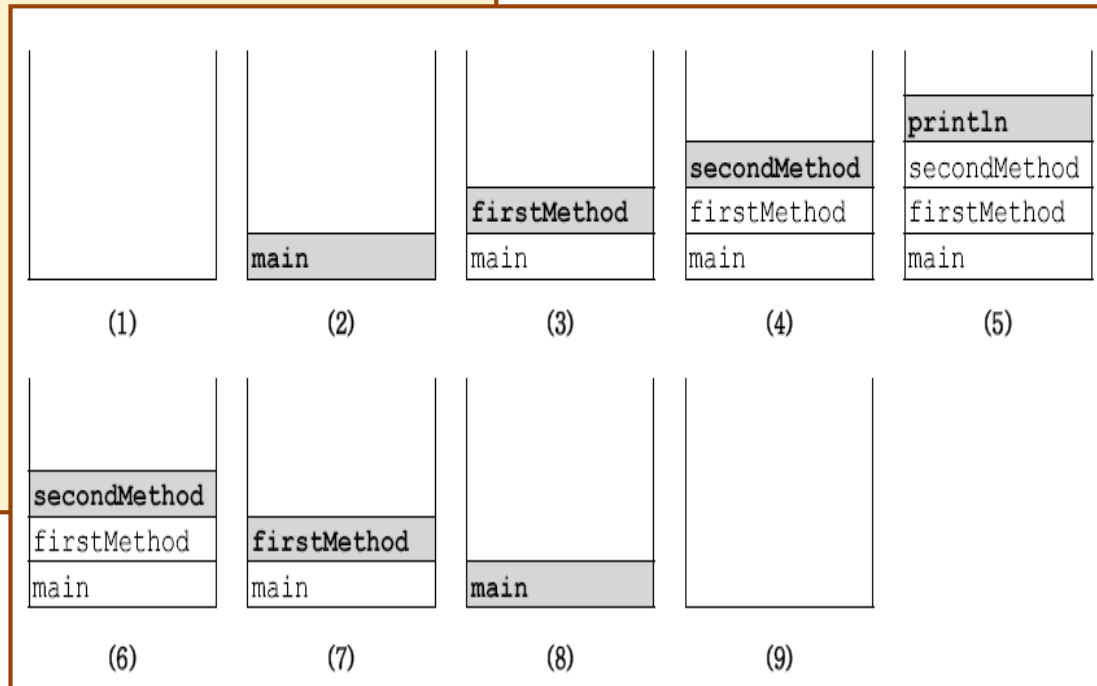
```
public static void main(String[] args) {  
    // 객체1  
    Account acc1 = new Account();  
    acc1.accountNo = "520-152-1234";  
    acc1.owerName = "홍길동";  
    acc1.balance = 200000;  
    // 객체2  
    Account acc2 = new Account();  
    acc2.accountNo = "425-143-1414";  
    acc2.owerName = "이순신";  
    acc2.balance = 500000;  
    System.out.print(acc1.owerName);  
    // 예금하기  
    for(int i=1; i<=10; i++) acc1.deposit();  
    System.out.println(": " + acc1.getBal());  
    System.out.print(acc2.owerName);  
    // 출금하기  
    for(int i=1; i<=10; i++) acc2.withdraw();  
    System.out.println(": " + acc2.getBal());  
} // main()  
} // Account
```

# 6] JVM의 메모리 구조 - 호출스택

## ▶ 호출스택의 특징

- 메소드가 호출되면 수행에 필요한 메모리를 스택에 할당.
- 메소드가 수행을 마치면 사용했던 메모리를 반환.
- 호출스택의 제일 위에 있는 메소드가 현재 실행중인 메소드.
- 아래에 있는 메소드가 바로 위의 메소드를 호출한 메소드

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

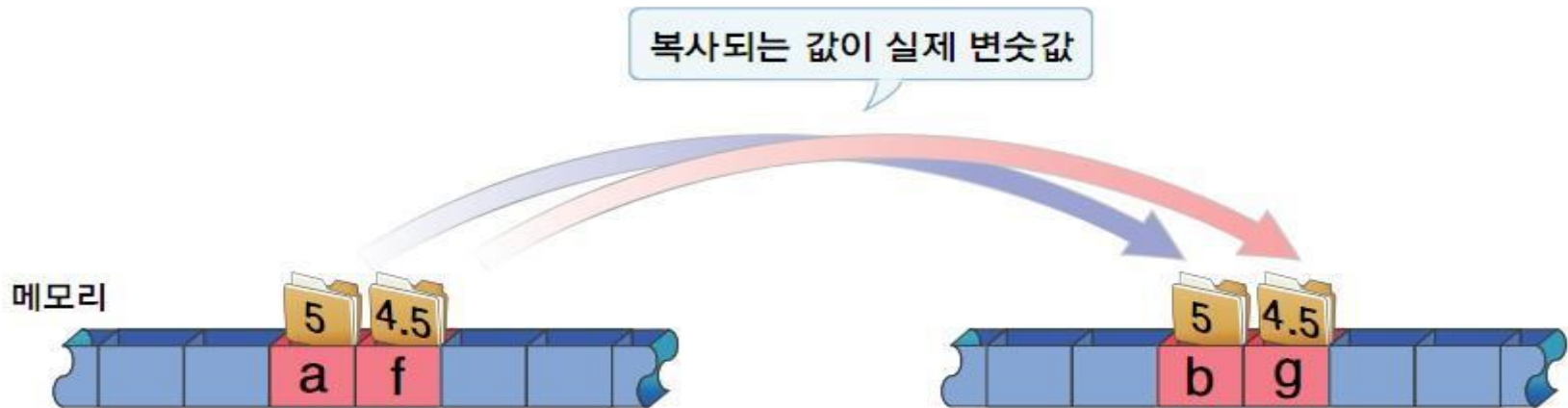


## 7] 메소드 - 기본형 매개변수와 참조형 매개변수

- ▶ 기본형 매개변수 - 변수의 값을 읽기만 할 수 있다.(read only)
- ▶ 참조형 매개변수 - 변수의 값을 읽고 변경할 수 있다.  
(read & write)
- 기본형 매개변수 예제 : `void add(int a, int b)`  
call by value 방식(값을 전달)
- 참조형 매개변수 예제 : `void add(Integer a, Integer b)`  
call by reference 방식(객체 참조 주소 전달)

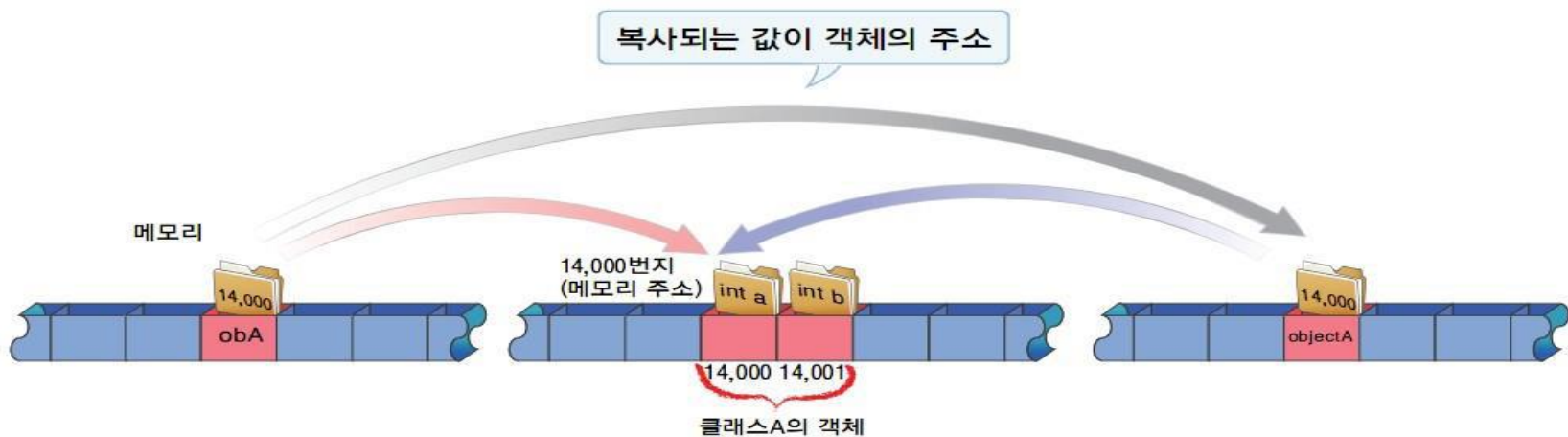
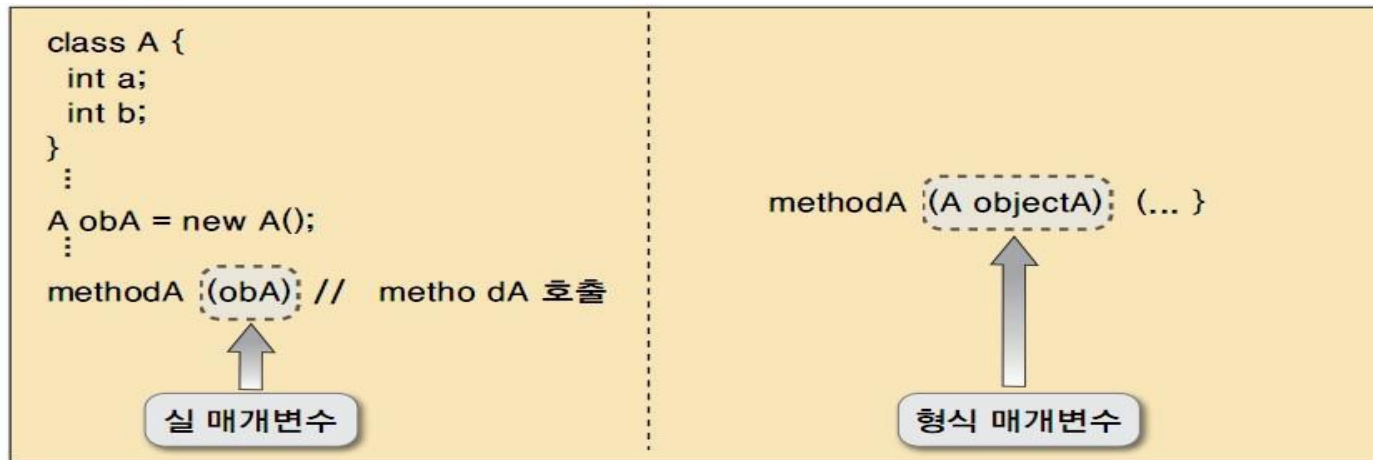
## 7] 메소드 - 기본형 매개변수와 참조형 매개변수 예시

실 매개변수와 형식 매개변수로 기본 자료형이 사용되는 경우



<매개변수 전달방법> 값 전달(Call by value)

# 7] 메소드 - Call by reference



매개변수 전달방법 > 값 전달(Call by reference)

# 7] Call by reference 예제 사용예시

```
public class Data {  
    int data;  
    public Data() {}  
    public Data(int a) {  
        this.data = a;  
    }  
}
```

```
public void swap(Data d1, Data d2){  
    int tmp;  
    tmp = d1.data;  
    d1.data=d2.data;  
    d2.data = tmp;  
}
```

```
public static void main(String[] args){  
    Data data1 = new Data(10);  
    Data data2 = new Data(9);  
    Data sp = new Data();  
    System.out.println  
        (data1.data + " " + data2.data);  
    sp.swap(data1, data2);  
    System.out.println  
        (data1.data + " " + data2.data);  
}
```

결과     두 변수값이 바뀌었다.

Swap함수 호출 전 결과 : num1=10 , num2=9

Swap함수 호출 후 결과 : num1=9 , num2=10

## 7] Call by reference / call by value

Call by value	Call by reference
data의 값에 의한 호출	data의 메모리 주소에 의한 호출
data에 직접적으로 영향을 줄 수 없다.	data에 직접적으로 영향을 줄 수 있다.



## 8] 인스턴메소드와 클래스메소드(static메소드)

### ▶ 인스턴스메소드

- 인스턴스 생성 후, "참조변수.메소드이름()"으로 호출
- 인스턴스변수나 인스턴스메소드와 관련된 작업을 하는 메소드

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

```
public class MyMethodTest1 {  
    public static void main(String args[]) {  
        // 인스턴스 생성  
        MyMethod mm = new MyMethod();  
        mm.a = 200L;  
        mm.b = 100L;  
        // 인스턴스 메소드 호출  
        System.out.println(mm.add());  
    }  
}
```

# 8] 인스턴메소드와 클래스메소드(static메소드)

## ▶ 클래스메소드(static메소드)

- 객체생성없이 "클래스이름.메소드이름()" 으로 호출
- 인스턴스변수나 인스턴스메소드와 관련 없는 작업을 하는 메소드
- 메소드 내에서 인스턴스변수 사용불가

```
class MyMethod {  
    long a, b;  
    // 인스턴스메소드  
    long add() {  
        return a + b;  
    }  
    // 클래스 메소드(static 메소드)  
    static long add(long a, long b) {  
        return a + b;  
    }  
}
```

```
class MyMethodTest1 {  
    public static void main(String args[]) {  
        // 클래스 메소드 호출  
        System.out.println(  
            MyMethod.add(200L,100L);  
        // 인스턴스 생성  
        MyMethod mm = new MyMethod();  
        mm.a = 200L;  
        mm.b = 100L;  
        // 인스턴스메소드 호출  
        System.out.println(mm.add());  
    }  
}
```

## 9] 멤버간의 참조와 호출-변수의 접근

- 같은 클래스의 멤버간에는 객체생성이나 참조변수 없이 참조가능
- 그러나 static멤버들은 인스턴스멤버들을 참조할 수 없음

```
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
```

# 10] 메소드 오버로딩

1) 메소드 오버로딩(method overloading)이란?

- 하나의 클래스에 같은 이름의 메소드를 여러 개 정의하는 것을 메소드 오버로딩
- 오버로딩이라고 함

2) 오버로딩의 조건

- 메소드의 이름이 같아야 한다.
- 매개변수의 개수 또는 타입이 달라야 한다.
- 매개변수는 같고 리턴타입이 다른 경우는 오버로딩 성립 않된다.  
(리턴타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다.)

3) 오버로딩의 예시)

▶ System.out.println메소드

- 다양하게 오버로딩된 메소드를 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)
```

## 10] 메소드 오버로딩 2

1] 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; }
```

2] 리턴Type 은 오버로딩의 성립조건이 아니다.

[보기2]

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

3] 오버로딩의 올바른 예 - 매개변수는 다르지만 같은 의미의 기능수행

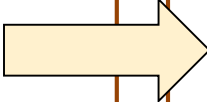
[보기4]

```
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;  
}
```

# 11] 참조변수 this

- 인스턴스 자신을 가리키는 참조변수.
- 인스턴스의 주소가 저장되어있음
- 인스턴스변수와 지역변수를 구별하기 위해 참조변수 this사용
- 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
```



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

# 11] 참조변수 this — 예시

- 1) 생성자나 메소드의 매개변수와 객체 변수가 같은 이름을 가질 때 사용
- 2) 같은 클래스내의 다른 생성자를 호출할 때 사용

this.kor는 현재 객체의 객체 속성 변수 kor를 의미.  
this를 사용 함으로서 같은 이름을 객체변수와 생성자 매개변수의 이름으로 사용

```
class Test {  
    int kor;  
    int eng;  
    int mat;  
    public void sum(int kor, int eng, int mat) {  
        // 같은 변수(생성자 매개변수) 값을 배정  
        this.kor=kor;  
        this.eng=eng;  
        this.mat=mat;  
    }  
}
```

## 12] 접근 제어자 (access modifier)

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.

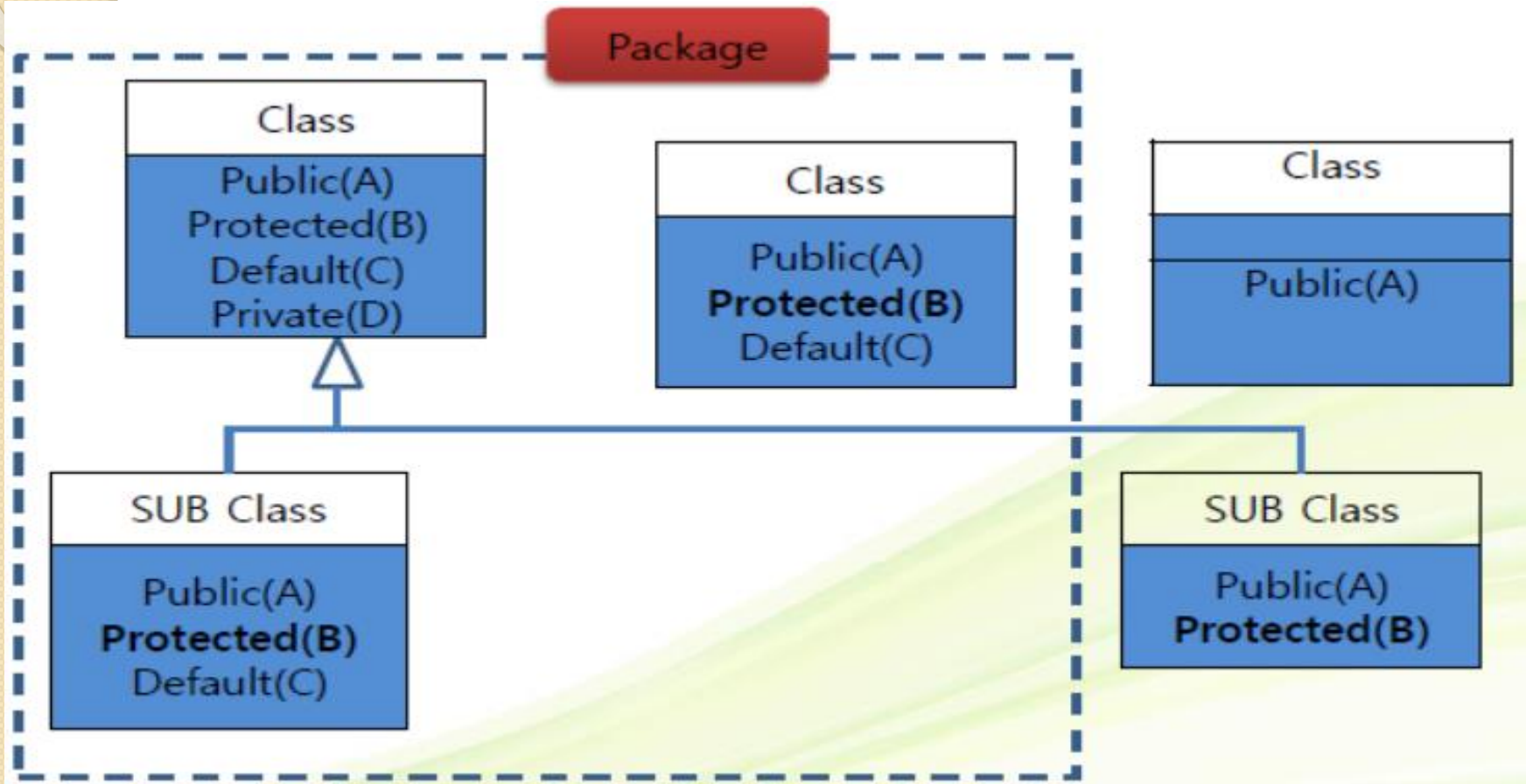
제어자	같은 클래스	같은 패키지	자손클래스	전 체
public				
protected				
default				
private				



## 12] 접근 제어자 (access modifier)

protected 접근 한정자

- 같은 패키지 내의 클래스와 같은 패키지는 아니지만 상속된 클래스에서 사용 가능한 접근 한정자



## 12] 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; 예러!!!  
  
    // 현재시간보다 1시간 후로 변경한다.  
    t.setHour(t.getHour()+1);  
    System.out.println(t);  
}
```

## [2] 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;  
    }  
  
    //...  
}
```

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

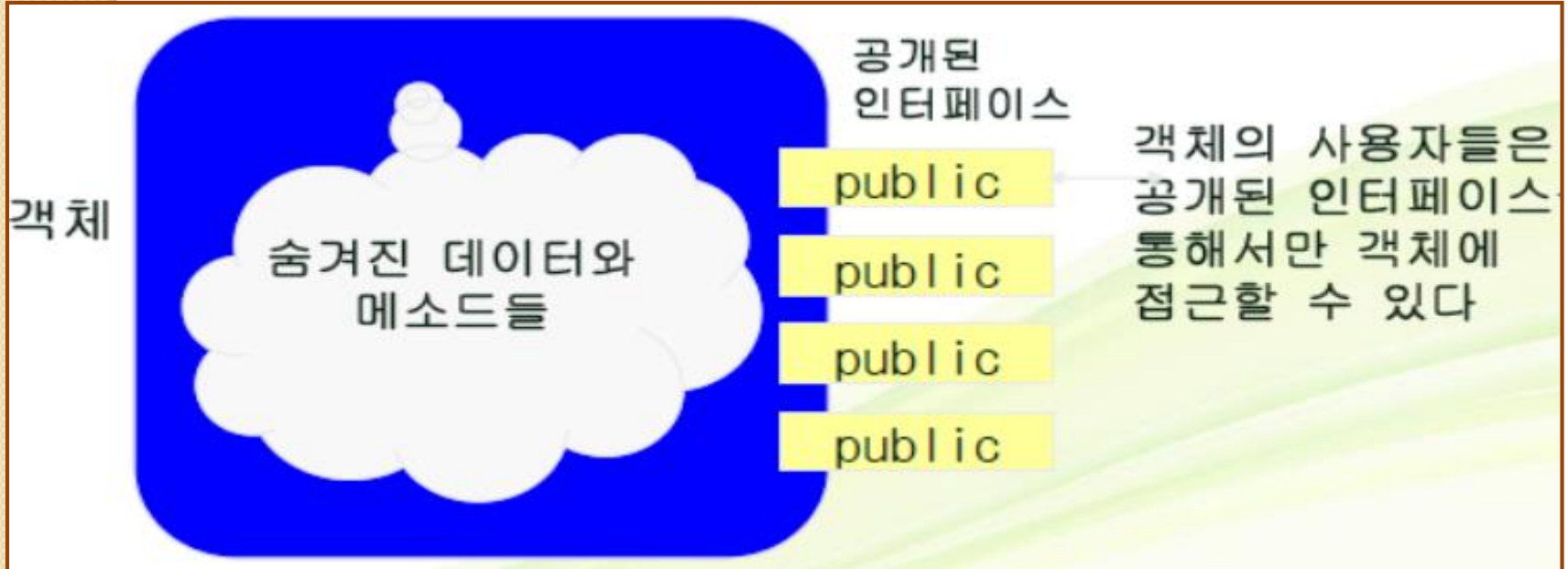
# 13] 제어자의 조합

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

# [4] 캡슐화(encapsulation) I

## 1. 캡슐화 개념

- 다른 객체의 필드(멤버변수)값을 직접 읽거나 수정할 수 없게 하고 반드시 별도의 메소드를 통하도록 속성과 메소드를 결합시키는 행위를 객체 지향 방법론에서는 캡슐화
- 캡슐화의 최대 목적은 정보은닉
- 객체를 캡슐화 하여 자기만 보여주고 user에게는 감춘다.
- 객체를 작성할 때 숨겨야 하는 정보(private)와 공개해야 하는 정보(public)를 구분하여 작성
- 객체의 사용자는 기능만 알고 사용하며 어떻게 처리되는지는 은폐(Information Hiding)
- 은닉화는 속성을, 캡슐화는 기능( 메서드 )를 담당해 중요한 정보를 은닉



# 14] 캡슐화(encapsulation) 2

## 1. 캡슐화의 장점

- 객체에 포함된 정보의 손상과 오용을 막을 수 있다.
- 객체 조작 방법이 바뀌어도 사용방법은 바뀌지 않는다.
- 데이터가 바뀌어도 다른 객체에 영향을 주지 않아 독립성이 유지된다.
- 처리된 결과만 사용하므로 객체의 이식성이 좋다.
- 객체를 부품화 할 수 있어 새로운 시스템의 구성에 부품처럼 사용

```
파일명 = EnCapSul.java
class EnCapSul{
private int age=25;
private String name="Jin Sil";
private String addr="Korea";
public void printAll(){
System.out.println
("Name="+name+"Age="+age+"Addr="+addr);
}
public void setName(String name){
this.name = name; }
public String getName(){ return name; }
```

```
public void setAge(int age){
this.age = age;
}
public int getAge(){
return age;
}
public void setAddr(String addr){
this.addr = addr;
}
public String getAddr(){
return addr;
}
} //class 닫기...
```

# 14] 캡슐화(encapsulation) 3(예제)

파일명 = EnCapTest.java

```
class EnCapTest {  
    public static void main(String[] a) {  
        EnCapSul en = new EnCapSul();  
        en.printAll();  
        en.setName("김민희");  
        en.setAge(19);  
        en.setAddr("서울특별시 여의도");  
        en.printAll();  
    }  
}
```

캡슐화 실습예제(Student.java/Manager.java/Teacher.java)

Student	Teacher	Manager
변 수	변 수	변 수
name age sno	name age subject	name age part
메소드	메소드	메소드
printAll()	printAll()	printAll()

# 15] 캡슐화(encapsulation) 와 정보 은닉

1. 은닉화는 객체에서 속성을 직접 접근하지 못하게 숨기는 것
  - Private 등을 이용
2. 캡슐화는 객체에서 메서드에 대한 은닉
  - 이름과 나이를 가져오는 메서드인 getter, setter에서 메서드 명에 변수 이름을 노출 , 완벽한 캡슐화 이루지 못함

예시

```
public People {  
    // 멤버변수를 private로 선언 해줌과 동시에 은닉화 발생  
    private String name;  
    private int age;  
  
    // 속성 age의 getter와 setter와 다르게 이름에서 어떤 속성에 getter인지 setter인지 명확히  
    // 알 수 있어 완벽한 캡슐화라고 할 수 없다.  
    public String getName() {  
        return this.name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    // 메서드 명을 간단하게 get, set이 아닌 이로 인해 발생하는 일을 생각해 추상적으로 명명하는 것이  
    // 캡슐화에 더 가까운 명명이다.  
    // 이 서비스는 성인인지 아닌지에 따라 다른 서비스를 제공하여 나이를 체크해야하는 서비스라고  
    // 생각했을 때 메서드 명.  
    public int checkAdult() {  
        ...만 19세 이상인가를 체크하는 코드  
        return this.age;  
    }  
    public void changeAge(int age) {  
        this.age = age;  
    }  
}
```



# I 6] Design Pattern

## 1. Design Pattern 개념

- 기존 환경 내에서 반복적으로 일어나는 문제들을 어떻게 풀어나갈 것인가에 대한 해결책 모음 또는 Library

### 생성(Creational) 패턴

Singleton  
Abstract Factory  
Factory Method  
Builder  
Prototype

### 구조(Structural) 패턴

Adapter  
Composite  
Decorator  
Facade  
Flyweight  
Proxy

### 행동(Behavioral) 패턴

Command  
Interpreter  
Iterator  
Mediator  
Memento  
Observer  
State  
Strategy  
Template Method