

# 메서드와 캡슐화

**강사 : 강병준**

# 메서드란 무엇인가?

메서드를 설명하기 전에 믹서를 생각해 보자. 우리는 믹서에 과일을 넣는다. 그리고 믹서를 사용해서 과일을 갈아 과일 주스를 만든다. 우리가 믹서에 넣는 과일은 "입력"이 되고 과일 주스는 "출력(결과값)"이 된다.

그렇다면 믹서는 무엇인가?



(믹서는 과일을 입력 받아 주스를 출력하는 메소드와 같다.)

우리가 배우려는 함수가 바로 믹서와 비슷하다. 입력 값을 가지고 어떤 일을 수행한 다음에 그 결과물을 내어놓는 것, 이것이 바로 메서드가 하는 일이다. 우리는 어려서부터 메서드에 대해 공부했지만 메서드에 관해 깊이 생각해 본 적은 별로 없다. 예를 들어  $y = 2x + 3$ 도 메서드이다. 하지만 이를 수학 시간에 배운 직선 그래프로만 알고 있지  $x$ 에 어떤 값을 넣었을 때 어떤 변화에 의해서  $y$  값이 나오는지 그 과정에 대해서는 별로 관심을 두지 않았을 것이다.

이제 우리는 메서드에 대해 조금 더 생각해 보는 시간을 가져야 한다. 프로그래밍에서 메서드는 정말 중요하기 때문이다.

## 메소드(method)

### ▶ 메소드란?

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

### ▶ 메소드의 장점과 작성지침

- 반복적으로 수행되는 여러 문장을 메소드로 작성한다.
- 하나의 메소드는 한 가지 기능만 수행하도록 작성하는 것이 좋다.

## 메소드(method)

- ▶ 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;

}

```
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();  
    }  
}
```

```
class LocalVariable {
    public static void main(String[] args) {
        boolean scope=true;
        if(scope) {
            int num=1;
            num++;
            System.out.println(num);
        } else {
            int num=2;
            System.out.println(num);
        }

        simple();
    }
    public static void simple() {
        int num=3;
        System.out.println(num);
    }
}
```

```
class OnlyExitReturn {
    public static void main(String[] args)    {
        divide(4, 2);
        divide(6, 2);
        divide(9, 0);
    }

    public static void divide(int num1, int num2)    {
        if(num2==0)    {
            System.out.println("0으로는 값을 나눌 수 없습니다.");
            return;
        }
        System.out.println("나눗셈 결과: " + (num1/num2));
    }
}
```

```
class MethodDefAdd {
    public static void main(String[] args)    {
        System.out.println("프로그램의 시작");
        hiEveryone(12);
        hiEveryone(13);
        System.out.println("프로그램의 끝");
    }

    public static void hiEveryone(int age)
    {
        System.out.println("좋은 아침입니다.");
        System.out.println("제 나이는 "+ age+"세입니다.");
    }
}

class RightRecul {
    public static void main(String[] args)    {
        showHi(3);
    }
    public static void showHi(int cnt)        {
        System.out.println("Hi~ ");
        if(cnt==1)        return;
        showHi(--cnt);
    }
}
```



## return문

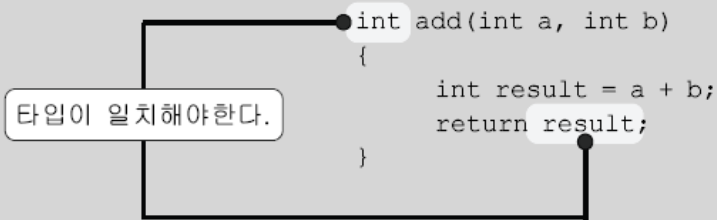
- ▶ 메소드가 정상적으로 종료되는 경우
  - 메소드의 블록 {}의 끝에 도달했을 때
  - 메소드의 블록 {}을 수행 도중 return문을 만났을 때

- ▶ 1. 반환값이 없는 경우 - return문만 써주면 된다.

retu

- 2. 반환값이 있는

retu



# 메소드

## return문 - 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;  
}
```

# 메소드

## 메소드의 호출

### ▶ 메소드의 호출방법

참조변수.메소드 이름 ();

// 메소드에 선언된 매개변수가 없는 경우

참조변수.메소드 이름(값1, 값2, ... );

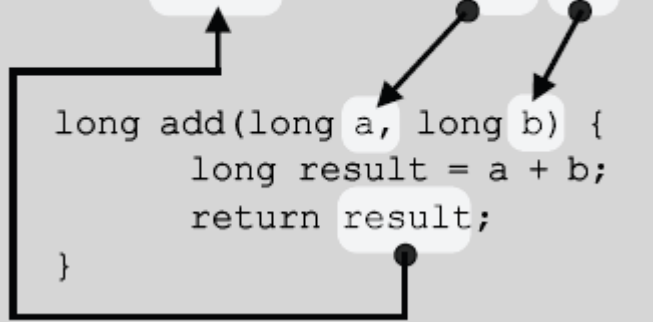
// 메소드에 선언된 매개변수가 있는 경우

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

```
MyMath mm = new MyMath();
```

```
long value = mm.add(1L, 2L);
```

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



```

class MethodReturns {
    public static void main(String[] args) {
        int result=add(4, 5);
        System.out.println("4와 5의 합: " + result);
        System.out.println("3.5의 제곱: " + square(3.5));
    }
    public static int adder(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);
    }
}

```

# 메소드

## [연습문제1] Account 클래스에 잔액 조회 메소드로 잔액조회하기

구분

내용

속성

계좌번호, 예금주, 잔액

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

# 메소드

```
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) {  
    Account obj1 = new Account();        // 객체1  
    obj1.accountNo = "520-152-1234";  
    obj1.owerName = "홍길동";  
    obj1.balance = 200000;  
  
    Account obj2 = new Account();        // 객체2  
    obj2.accountNo = "425-143-1414";  
    obj2.owerName = "이순신";  
    obj2.balance = 500000;  
  
    System.out.print(obj1.owerName);  
    for(int i=1; i<=10; i++)              // 예금하기  
        obj1.deposit();  
    System.out.println(": "+ obj1.getBal());  
    System.out.print(obj2.owerName);  
    for(int i=1; i<=10; i++)              // 출금하기  
        obj2.withdraw();  
    System.out.println(": "+ obj2.getBal());  
} // main()  
} // Account
```

# 메소드

## [연습문제2] Account 클래스에 매개변수를 이용하여 예금 및 인출하기

구분

내용

속성

계좌번호, 예금주, 잔액

기능

예금하다(수정), 인출하다(수정), 조회하다.

홍길동: 300000

이순신: 400000

실행결과

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

클래스 구성도



Account (obj1)
"520-152-1234" "홍길동" 200,000원
deposit(10만원):void withdraw(10만원):void

객체

# 메소드

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

```
public static void main(String[] args) {  
    Account obj1 = new Account(); // 객체1  
    obj1.accountNo = "520-152-1234";  
    obj1.owerName = "홍길동";  
    obj1.balance = 200000;  
  
    Account obj2 = new Account(); // 객체2  
    obj2.accountNo = "425-143-1414";  
    obj2.owerName = "이순신";  
    obj2.balance = 500000;  
  
    System.out.print(obj1.owerName);  
    obj1.deposit(100000); // 예금하기  
    System.out.println(": " + obj1.getBal());  
    System.out.print(obj2.owerName);  
    obj2.withdraw(100000); // 출금하기  
    System.out.println(": " + obj2.getBal());  
} // main()  
} // Account
```



# 메소드

**[연습문제3]** 인출 금액이 잔액보다 많은 경우에는 0을 리턴한다.  
또한 리턴 값을 비교 판단하여 "잔액이 부족합니다"라는 메시지를 출력한다.

구분

내용

속성

계좌번호, 예금주, 잔액

기능

예금하다, 인출하다, 조회하다.

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

클래스 구성도



Account (obj1)
"520-152-1234" "홍길동" 200,000원
deposit(10만원):void withdraw(10만원):void

객체

"잔액이 부족  
합니다."

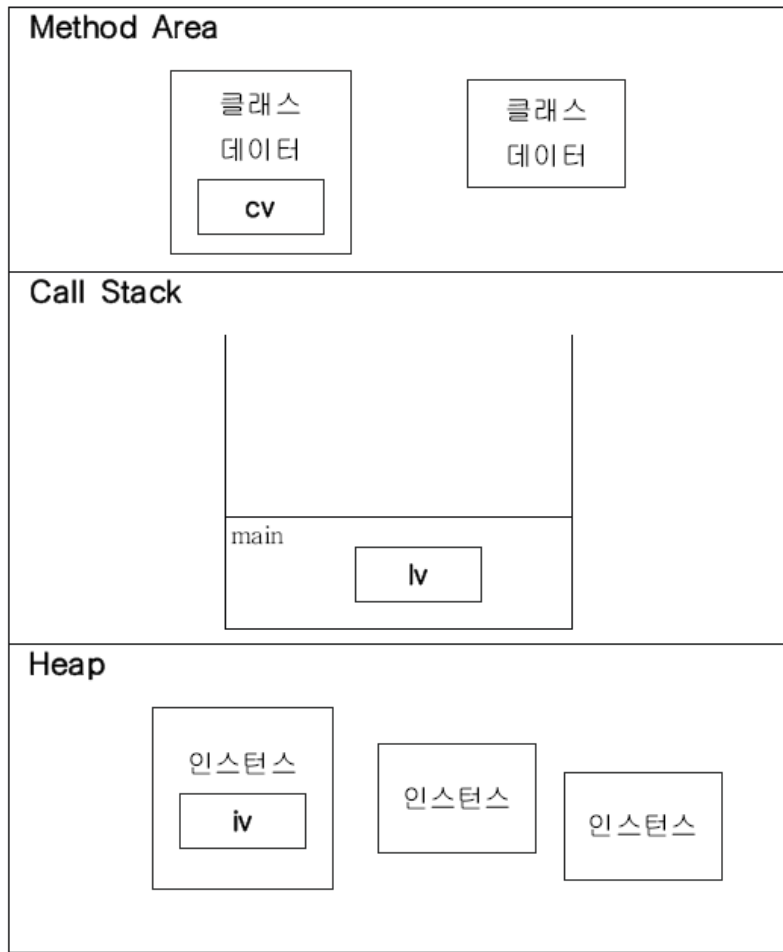
# 메소드

```
public class Account {  
    String accountNo;  
    String owerName;  
    int balance;  
  
    void deposit(int b){  
        balance += b;  
    }  
  
    int withdraw(int w){  
        if(balance < w)  
            return 0;  
        else  
            return balance -= w;  
    }  
  
    int getBal() {  
        return balance;  
    }  
}
```

```
public static void main(String[] args) {  
    Account obj1 = new Account(); // 객체1  
    obj1.accountNo = "520-152-1234";  
    obj1.owerName = "홍길동";  
    obj1.balance = 200000;  
  
    Account obj2 = new Account(); // 객체2  
  
    obj2.accountNo = "425-143-1414";  
    obj2.owerName = "이순신";  
    obj2.balance = 500000;  
  
    System.out.print(obj1.owerName);  
    obj1.deposit(100000); // 예금하기  
    System.out.println(": "+ obj1.getBal());  
    System.out.print(obj2.owerName);  
    int bal = obj2.withdraw(300000);  
    if(bal == 0)  
        System.out.println(" 잔액이 부족합니다");  
    else  
        System.out.println(": "+obj2.getBal());  
} // main()  
} // Account
```

# 메소드

## JVM의 메모리 구조



### ▶ 메소드영역(Method Area)

- 클래스 정보와 클래스변수가 저장되는 곳

### ▶ 호출스택(Call Stack)

- 메소드의 작업공간  
메소드가 호출되면 지역변수를 할당하고  
메소드가 종료되면 반환한다.

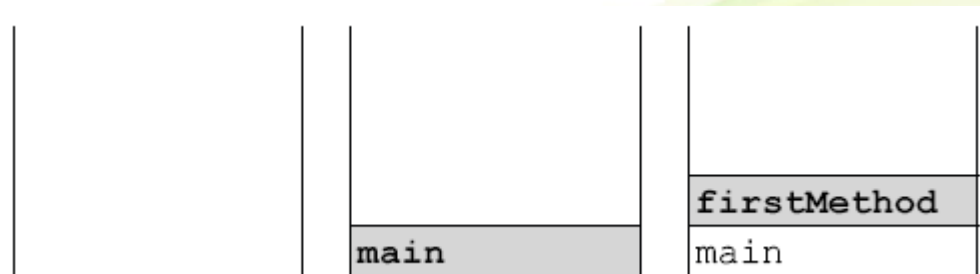
### ▶ 힙(Heap)

- 인스턴스가 생성되는 공간 new연산자  
에 의해서 생성되는 배열과 객체는 모  
두 여기에 생성된다.

## JVM의 메모리 구조 - 호출스택

### ▶ 호출스택의 특징

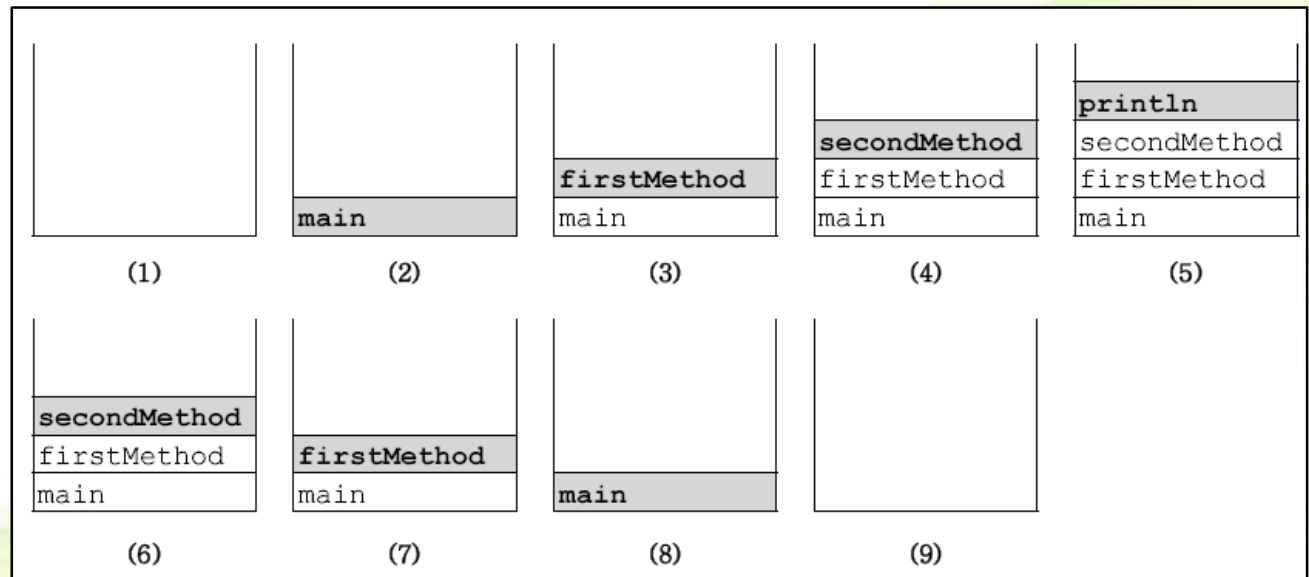
- 메소드가 호출되면 수행에 필요한 메모리를 스택에 할당받는다.
- 메소드가 수행을 마치면 사용했던 메모리를 반환한다.
- 호출스택의 제일 위에 있는 메소드가 현재 실행중인 메소드다.
- 아래에 있는 메소드가 바로 위의 메소드를 호출한 메소드다.



# 메소드

## JVM의 메모리 구조 - 호출스택

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



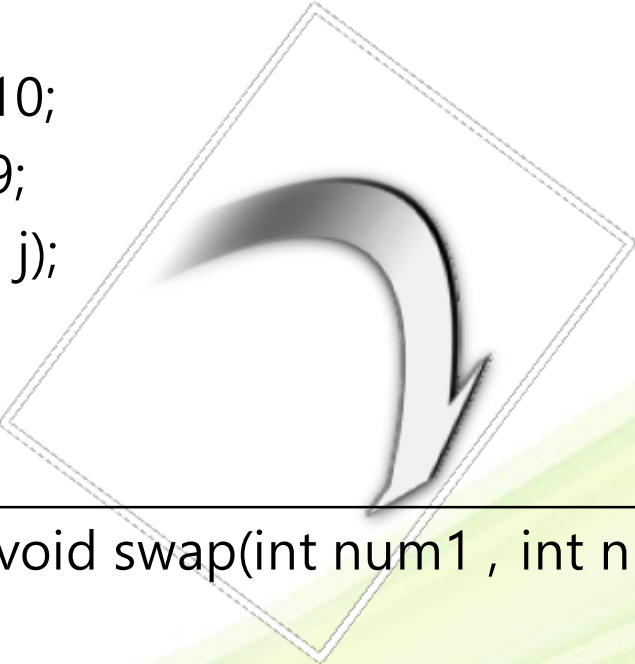
## 기본형 매개변수와 참조형 매개변수

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

## Call by value란?

↳ 값에 의한 호출

```
int i = 10;  
int j = 9;  
swap(i, j);
```



```
Public void swap(int num1 , int num2)
```

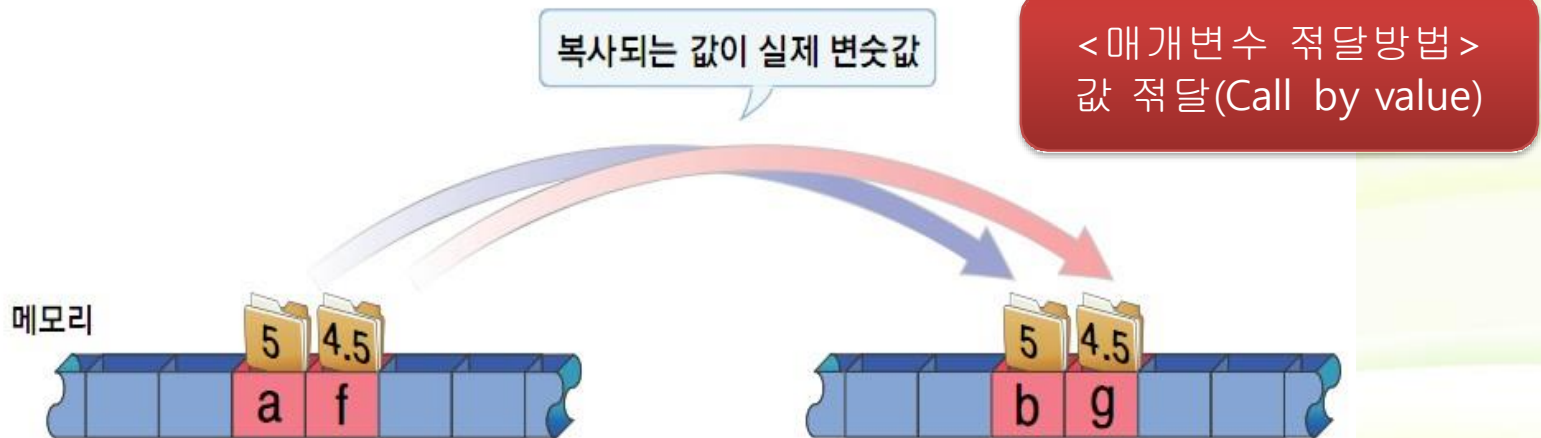
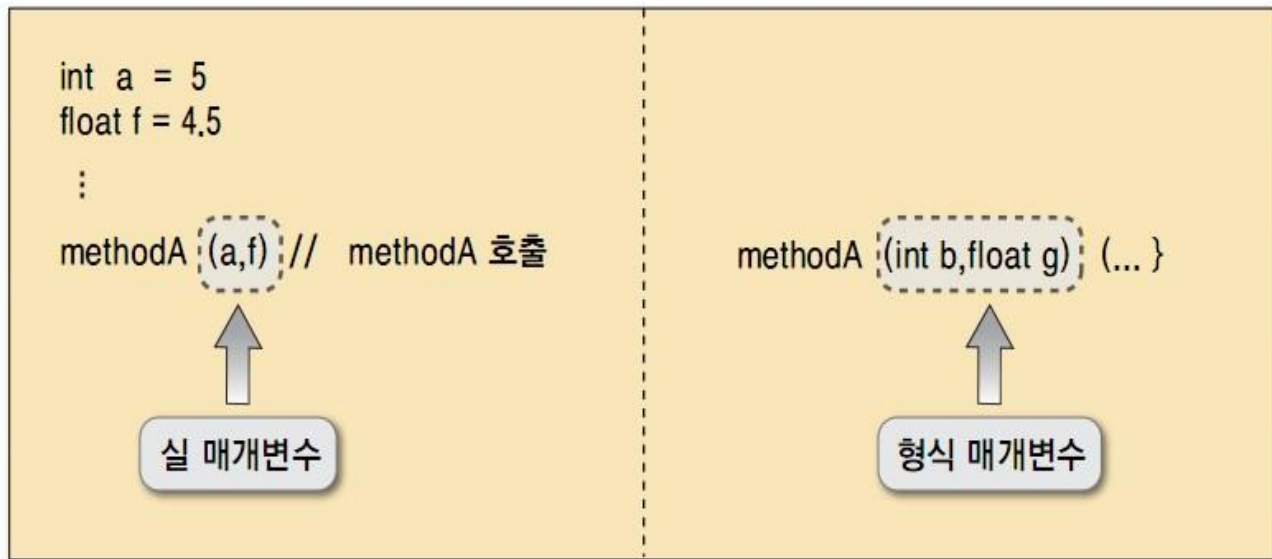
```
{
```

```
.....
```

```
}
```

# 메소드

- 실 매개변수와 형식 매개변수로 기본 자료형이 사용되는 경우





# 메소드

```
class B1 {  
    void add(int x, int y) {  
        x += y; y+=2;  
        System.out.println("연산결과");  
        System.out.println("x = " + x);  
        System.out.println("y = " + y);  
    }  
}  
  
public class CallValue {  
    public static void main(String[] args) {  
        B1 b = new B1();  
        int x = 10; int y = 8;  
        b.add(x, y); // call by value  
        System.out.println("메인메이서");  
        System.out.println("x = " + x);  
        System.out.println("y = " + y);  
    }  
}
```

## Call by value 예제

```
public void swap(int i, int j)
{
    int tmp;
    tmp = i;
    i = j;
    j = tmp;
}
```

```
public static void main (String[] args)
{
    int num1 = 10;
    int num2 = 9;
    Swap sp = new Swap();
    sp.swap(num1, num2);
    System.out.println(num1 + num2 );
}
```

## 결과

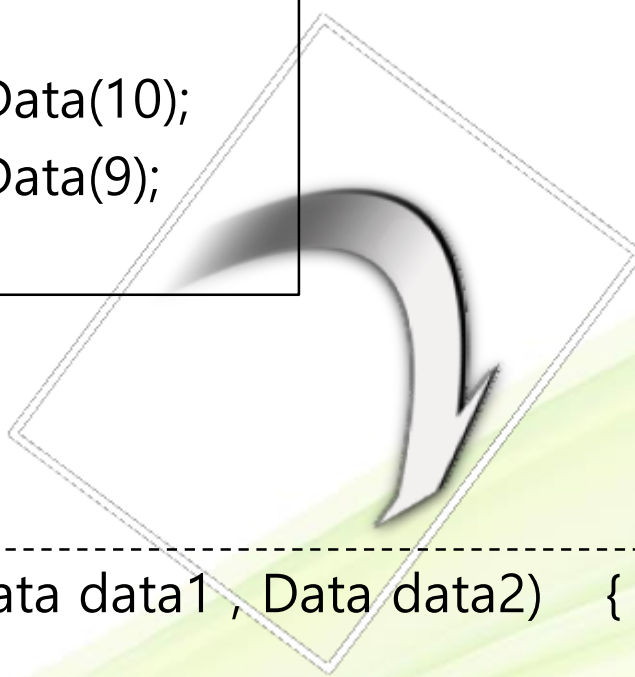
↻ 바뀌지 않는다.

↻ **Swap함수의 결과 : num1=10, num2=9**

## Call by reference?

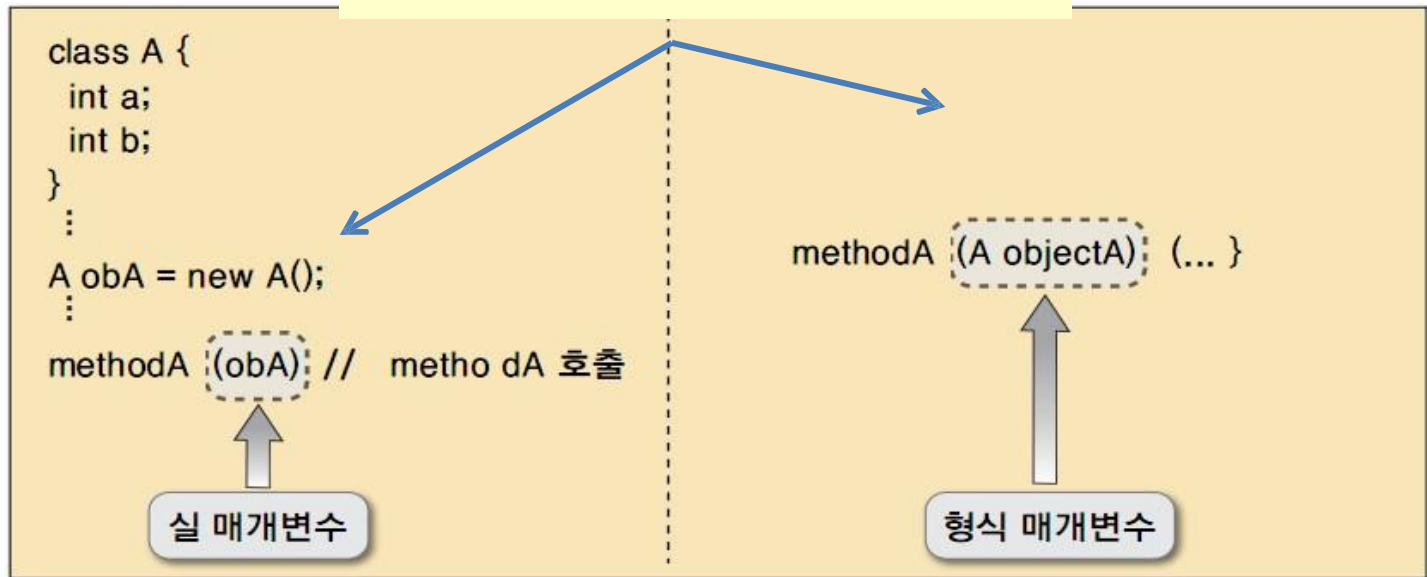
↳ 참조에 의한 호출

```
Data data1 = new Data(10);  
Data data2 = new Data(9);  
swap(data1 , data2);
```



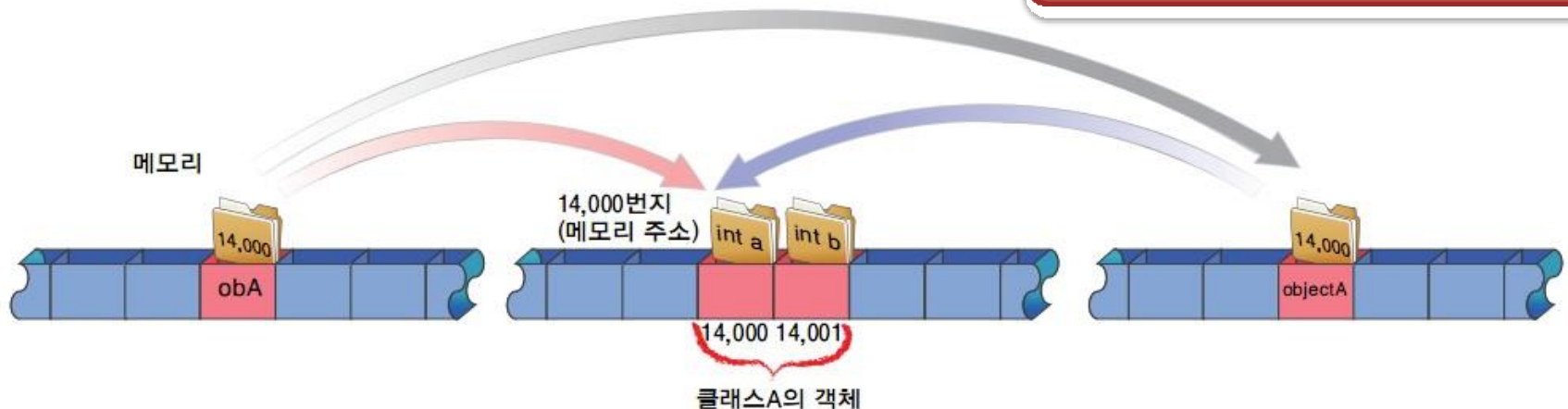
```
Public void swap(Data data1 , Data data2) {  
    .....  
}
```

## A클래스형 변수일치



복사되는 값이 객체의 주소

<매개변수 전달방법>  
값 전달(Call by reference)



```
class B2 {  
    void aa(int[] x) {  
        x[0] += x[1];  
        x[1] += 2;  
        System.out.println("연산결과");  
        System.out.println("x[0] = "+x[0]);  
        System.out.println("x[1] = "+x[1]);  
    }  
}  
  
public class CallRef1 {  
    public static void main(String[] args) {  
        B2 b = new B2();  
        int[] x = {10,8};  
        b.aa(x); // call by reference  
        System.out.println("매인");  
        System.out.println("x[0] = "+x[0]);  
        System.out.println("x[1] = "+x[1]);  
    }  
}
```

# 객체

```
class B3 {  
    int x, y;  
    void bb(B3 b) {  
        b.x +=10;b.y +=2;  
        System.out.println("연산결과");  
        System.out.println("x = "+b.x);  
        System.out.println("y = "+b.y);  
    }  
}  
  
public class Callref2 {  
    public static void main(String[] args) {  
        B3 b = new B3();  
        b.x = 8;b.y = 10;b.bb(b);  
        System.out.println("대입");  
        System.out.println("x = "+b.x);  
        System.out.println("y = "+b.y);  
    }  
}
```

# Swap

```
class C1 {  
    int a, b;  
    void swap(C1 c) {  
        int t = a; c.a = b;    c.b = t;  
        System.out.println(" 변경 a = "+c.a);  
        System.out.println(" 변경 b = "+c.b);  
    }  
}
```

```
public class CallRef3 {  
    public static void main(String[] args) {  
        C1 c = new C1();  
        c.a = 7;    c.b = 15;    c.swap(c);  
        System.out.println(" 호출 a = "+c.a);  
        System.out.println(" 호출 b = "+c.b);  
    }  
}
```



## 결론

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.

# 메소드

## 인스턴스메소드와 클래스메소드(static메소드)

### ▶ 인스턴스메소드

- 인스턴스 생성 후, „참조변수.메소드이름0”으로 호출
- 인스턴스변수나 인스턴스메소드와 관련된 작업을 하는 메소드

```
class MyMethod {  
    long a, b;           // 인스턴스 변수  
    long add() {         // 인스턴스 메소드  
        return a + b;  
    }  
}
```

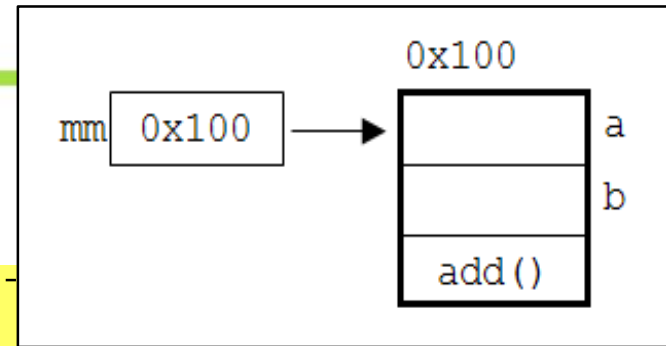
```
public class MyMethodTest1 {  
    public static void main(String args[]) {  
        MyMethod mm = new MyMethod(); // 인스턴스 생성  
        mm.a = 200L;  
        mm.b = 100L;  
        System.out.println(mm.add()); // 인스턴스 메소드 호출  
    }  
}
```

## 클래스메소드(static메소드)와 인스턴스메소드

### ▶ 클래스메소드(static메소드)

- 객체생성없이 ‘**클래스이름.메소드이름0**’으로 호출
- 인스턴스변수나 인스턴스메소드와 관련 없는 작업을 하는 메소드
- 메소드 내에서 인스턴스변수 사용불가

# 메소드



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

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

# 메소드

## 멤버간의 참조와 호출(1/2) – 메소드의 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
```

## 멤버간의 참조와 호출(2/2) – 변수의 접근

“같은 클래스의 멤버간에는 객체생성이나 참조변수 없이 참조할 수 있다. 그러나 static 멤버들은 인스턴스 멤버들을 참조할 수 없다.”

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

# 메소드 오버로딩

## 메소드 오버로딩(method overloading)이란?

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

## 오버로딩의 조건

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

# 메소드 오버로딩

## 오버로딩의 예(1/3)

### ▶ 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)
```



# 메소드 오버로딩

## 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); }
```

# 메소드 오버로딩

## 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;  
}
```

# 메소드 오버로딩

## ● 메소드 오버 로딩

**Overloading**(중복정의) : 하나의 클래스 내에서 같은 이름을 가지는 메서드(멤버함수)를 여러 개 정의되는 것. 이것은 컴파일 시 컴파일러에 의해 각 메소드들이 구별되며 기준은 인자가 된다

교체가능      반드시 동일      반드시 다르게

[ 접근제한 ] [ 반환형 ] [ 메서드명 ] (자료형 인자1, 자료형 인자2, ...){ }

● 메소드는 인자로 구분된다.

# 메소드 오버로딩

```
class Box {
```

```
    int i_volume;
```

```
    double d_volume;
```

```
    public Box(int w, int h, int d){
```

```
        volume(w, h, d);
```

```
    }
```

```
    private void volume(int w, int h, int d) {
```

```
        i_volume = w * h * d;
```

```
    }
```

```
    private void volume(double w, double h, double d) {
```

```
        d_volume = w * h * d;
```

```
    }
```

```
}
```

정수 박스 부피 : 6000

실수 박스 부피 : 6565.125

정수와 실수 박스 부피 : 6457.5

```
public class BoxTest {
```

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

```
        Box mybox1 = new Box(10,20,30);
```

```
        Box mybox2 = new Box(10.5,20.5,30.5);
```

```
        Box mybox3 = new Box(10.5,20.5,30);
```

```
        System.out.println("정수 박스 부피 : " + mybox1.i_volume);
```

```
        System.out.println("실수 박스 부피 : " + mybox2.d_volume);
```

```
        System.out.println("정수와실수박스부피: “ +mybox3.d_volume);
```

```
    }
```

```
}
```

## 1. 다음과 같은 멤버변수를 갖는 Student클래스를 정의

타입	변수명	설명
String	name	학생이름
int	ban	반
int	no	번호
int	kor	국어점수
int	eng	영어점수
int	math	수학점수

getTotal()과 getAverage()를 추가하시오.

### 1. 메서드명 : getTotal

기능 : 국어(kor), 영어(eng), 수학(math)의 점수를 모두 더해서 반환한다.

반환타입 : int

매개변수 : 없음

### 2. 메서드명 : getAverage

기능 : 총점(국어점수+영어점수+수학점수)을 과목수로 나눈 평균을 구한다.

반환타입 : float

매개변수 : 없음

## 2. 문제

이름 : 홍길동, 국어 : 100, 영어 ; 60, 수학 : 76

위의 값을 할당하여 이름, 총점, 평균을 출력

- 1) 디폴트생성자를 생성하고 값을 대입하여 처리
- 2) 생성자에 위 값을 매개변수로하여 대입하여 처리

1) main에서 출력

2) 출력하는 method 작성하여 처리

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

# this

“this” 는 현재의 객체를 의미

☞ 생성자나 메소드의 매개변수와 객체 변수가 같은 이름을 가질 때  
사용

☞ 같은 클래스내의 다른 생성자를 호출할 때 사용

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



# this - 예

(같은 이름을 객체 속성 변수와 생성자 매개변수의 이름으로 사용)

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

this.kor는 현재 객체의 객체 속성 변수 kor를 의미.

this를 사용 함으로서 같은 이름을 객체변수와 생성자 매개변수의 이름으로 사용할 수 있다

# 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);  
}
```

# this - 예

(같은 클래스 내의 다른 생성자를 호출)

```
class Test {  
    int kor;  int eng;  
    int mat;  
    public Test() {  
        this(1,1,1);  
    }  
    public Test(int k) {  
        this(k,1,1);  
    }  
    public Test(int k, int e) {  
        this(k,e,1);  
    }  
    public Test(int k, int e, int m) {  
        kor=k;  
        eng=e;  
        mat=m;  
    }  
}
```

이 클래스는 어떠한 형태로 객체가 생성되어도  
결국 마지막 생성자가 호출된다.  
여기서 사용된 this는 같은 클래스 내의  
다른 생성자를 호출하는 역할을 한다

# this 예제

```
class ThisTest{
    int my;
    ThisTest(){
        System.out.println("인자가 없는 생성자 함수..default래요..");
    }
    ThisTest(int i){
        this();
        my=i;
        System.out.println("인자가 있는 생성자 함수..인자가 1개 있네요..");
    }
    public void print(int my){
        this.my=my;
        System.out.print("my="+this.my+" ");
    }
    public static void main(String[] arg){
        ThisTest t = new ThisTest(3);
        System.out.println("my="+t.my);
        t.print(5);
        System.out.println("my="+t.my);
    }
}
```

# 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				

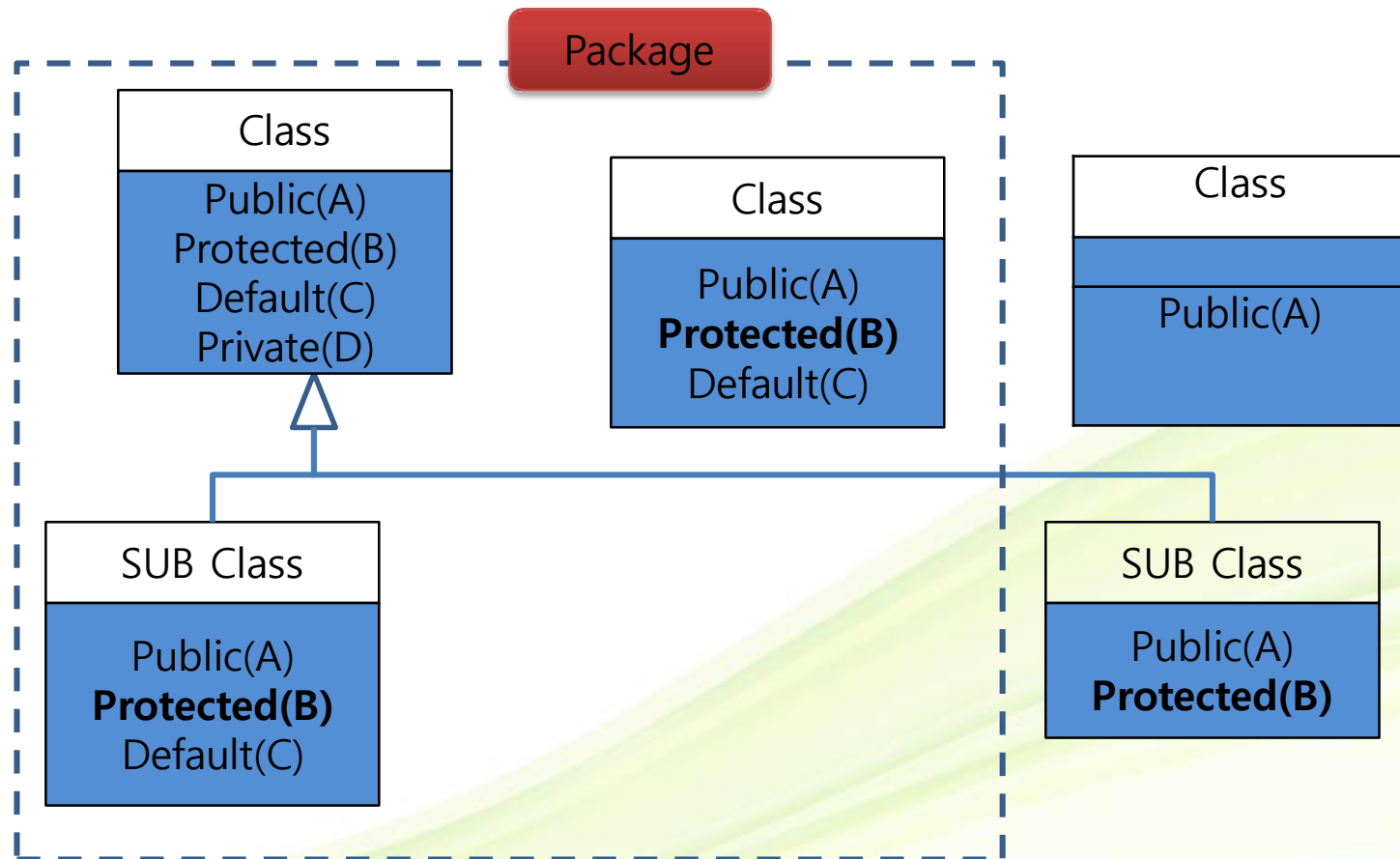
```
public
(default) class AccessModifierTest {
    int iv;          // 멤버변수 (인스턴스변수)
    static int cv;    // 멤버변수 (클래스변수)

    public
    protected
    (default) void method() {}
    private
}
```

# 제어자(modifiers)

- **protected** 접근 한정자

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

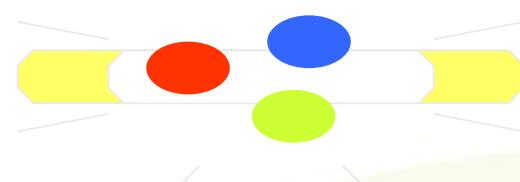
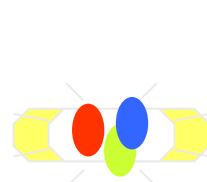


# 접근 제어자를 이용한 캡슐화

## 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)

# 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();  
    }  
}
```

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



# 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.

# 캡슐화(encapsulation)

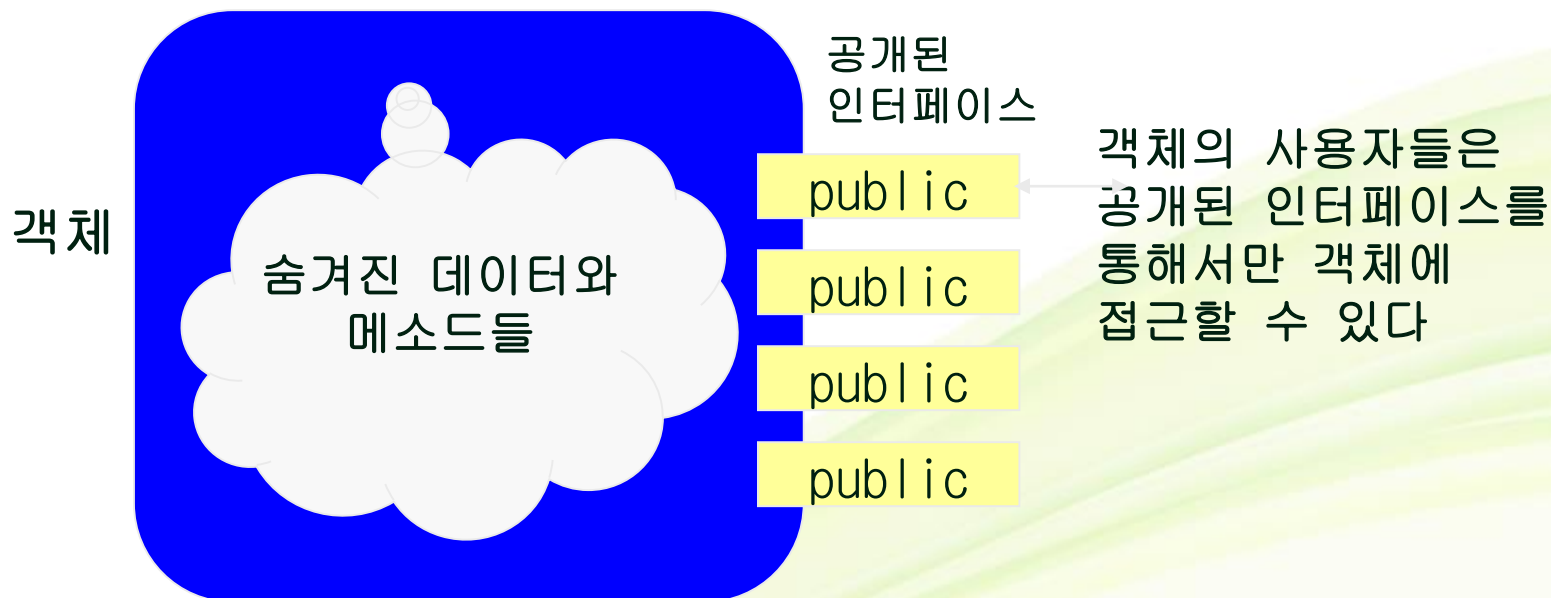
## ▶ 캡슐화란?

다른 객체의 필드(멤버변수)값을 직접 읽거나 수정할 수 없게 하고 반드시 별도의 메소드를 통하도록 속성과 메소드를 결합시키는 행위를 객체 지향 방법론에서는 캡슐화라고 한다.

캡슐화의 최대 목적은 정보은닉이라고 볼 수 있다.

# 캡슐화(encapsulation)

- ☞ 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 은폐된다(Information Hiding)



# 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 사용할 수 있다.

# 캡슐화 예제

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

```
public class EnCapTest {  
    public static void main(String[] a) {  
        EnCapSul en = new EnCapSul();  
        en.printAll();  
        en.age=20;  
        en.name="진실";  
        en.addr="한국";  
        en.printAll();  
    }  
}
```

# 캡슐화하는 방법

1. 메소드나 필드 앞에 private라는 접근제어자를 사용

```
private int age=25;  
private String name="Jin Sil";  
private String addr="Korea";
```

2. 캡슐화된 객체의 필드값을 읽거나 변경하기 위해 사용하는 메소드들을 접근자 메소드 또는 인터페이스 메소드라고 한다. 외부접근자가 사용할수 있는 필드나 메소드 앞에 public사용하여 접근자 메소드를 추가한다.

```
public void setAge(int age){  
    this.age=age;  
}  
public int getAge(){  
    return age;  
}
```

set => 값을 세팅

get => 값을 얻어 온다, 돌려준다

# 캡슐화 예제

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...
```

# 캡슐화 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

변 수
<b>name</b>
<b>age</b>
<b>sno</b>

메소드
<b>printAll()</b>

## Teacher

변 수
<b>name</b>
<b>age</b>
<b>subject</b>

메소드
<b>printAll()</b>

## Manager

변 수
<b>name</b>
<b>age</b>
<b>part</b>

메소드
<b>printAll()</b>

# 예제 Student.java

```
class Student{
    private String name="Jin Sil";
    private int age=20;
    private int sno=123456;
    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 setSno(int sno){
        this.sno = sno;
    }
    public int getSno(){
        return sno;
    }
    public void printAll(){
        System.out.println("Student를 호출했어요...");
        System.out.println("Name="+name+" Age="+age+" Sno="+sno);
    }
}
```

# 예제 Manager.java

```
class Manager{
    private String name="김 씨";
    private int age=50;
    private String part="경비실";
    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 setpart(String part){
        this.part = part;
    }
    public String getpart(){
        return part;
    }
    public void printAll(){
        System.out.println("Manager를 호출했어요...");
        System.out.println("Name="+name+" Age="+age+" part="+part);
    }
}
```

```
class Teacher{
    private String name="Gil Dong";
    private int age=40;
    private String subject="국어";
    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 setSubject(String subject){
        this.subject = subject;
    }
    public String getSubject(){
        return subject;
    }
    public void printAll(){
        System.out.println("Teacher를 호출했어요...");
        System.out.println("Name="+name+" Age="+age+" Subject="+subject);
    }
}
```

# 예제 School.java

```
class School{  
    public static void main(String[] arg){  
        Student s = new Student();  
        Teacher t = new Teacher();  
        Manager m = new Manager();  
  
        s.printAll();  
        s.setName("소녀시대");  
        s.setAge(19);  
        s.setSno(5421563);  
        s.printAll();  
  
        t.printAll();  
        m.printAll();  
    }  
}
```

# 연습문제

1. 섯다카드 20장을 포함하는 섯다카드 한 벌(SutdaDeck클래스)을 정의한 것이다. 섯다카드 20장을 담은 SutdaCard배열을 초기화하시오. 단, 섯다카드는 1부터 10까지의 숫자가 적힌 카드가 한 쌍씩 있고, 숫자가 1, 3, 8인 경우에는 둘 중의 한 장은 광(Kwang)이어야 한다. 즉, SutdaCard의 인스턴스변수 isKwang의 값이 true이어야 한다.

```
class SutdaDeck {
```

```
    final int CARD_NUM = 20;
```

```
    SutdaCard[] cards = new SutdaCard[CARD_NUM];
```

```
    SutdaDeck() {
```

```
        for(int i=0;i < cards.length;i++) {
```

```
            int num = i%10+1;
```

```
            boolean isKwang = (i < 10)&&(num==1||num==3||num==8);
```

```
            cards[i] = new SutdaCard(num,isKwang);
```

```
        }
```

```
    }
```

```
}
```

```
class SutdaCard {
    int num;
    boolean isKwang;
    SutdaCard() { this(1, true); }
    SutdaCard(int num, boolean isKwang) {
        this.num = num;
        this.isKwang = isKwang;
    }
    // info()대신 Object클래스의 toString()을 오버라이딩 했다.
    public String toString() {
        return num + ( isKwang ? "K":"" );
    }
}

class Exercise7_1 {
    public static void main(String args[]) {
        SutdaDeck deck = new SutdaDeck();
        for(int i=0; i < deck.cards.length; i++)
            System.out.print(deck.cards[i]+",");
    }
}
```