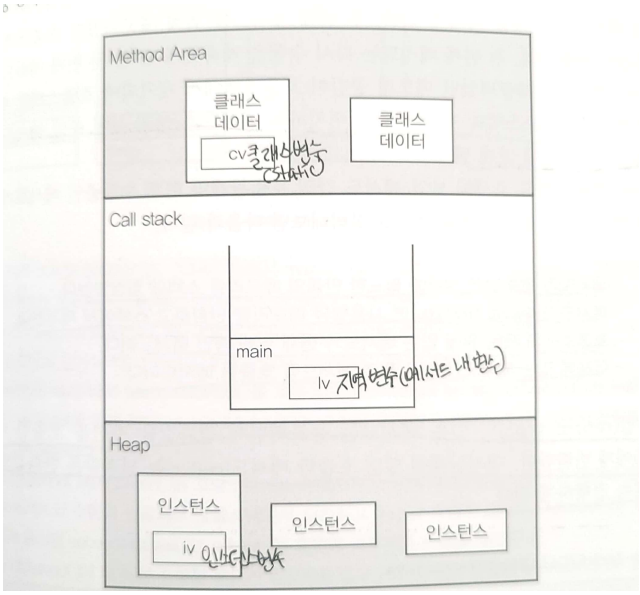
● **JVM의 메모리 구조**: 응용프로그램이 실행되면, JVM은 시스템으로부터 프로그램을 수행하는데 필요한 메모리를 할당받고 이 메모리를 용도에 따라 여려 영역으로 나누어 관리한다.

▶ 메서드 영역(method area): 어떤 클래스가 사용되면, JVM은 해당 클래스의 클래스파일(.class)을 읽어서 분석하여 클래스에 대한 정보(클래스 데이터)를 이곳에 저장한다. 이때, 그 클래스의 클래스 변수(class variable; cv)도 함께 생성된다.

▶ 힙(heap): 인스턴스가 생성되는 공간. 프로그램 실행 중 생성되는 인스턴스는 모두 이곳에 생성된다. 인스턴스 변수(instance variable; iv)들이 생성되는 공간이다.

▶ 호출스택(call stack/execution stack): 메서드의 작업에 필요한 메모리 공간을 제공한다. 메서드가 호출되면, 호출스택에 호출된 메서드를 위한 메모리가 할당되며, 이 메모리는 메서드가 작업을 수행하는 동안 지역변수(매개변수 포함)들과 연산의 중간결과 등을 저장하는데 사용된다. 그리고 메서드가 작업을 마치면 할당되었던 메모리공간은 반환되어 비워진다. 첫번째로 호출된 메서드(메인 메서드)를 위한 작업공간이 호출 스택의 맨 밑에 마련되고, 첫번째 메서드 수행 중에 다든 메서드를 호출하면, 첫번째 메서드의 바로 위에 두번째 호출된 메서드를 위한 공간이 마련된다. 두번째로 호출된 메서드가 수행을 마치게 되면, 두번째 메서드를 위해 제공되었던 호출스택의 메모리 공간이 반환되며, 첫번째 메서드는 다시 수행을 계속하게 되고, 수행을 마치면 역시 제공되었던 메모리 공간이 호출스택에서 제거되며 호출스택은 완전히 비워지게 된다. 호출스택의 제일 상위에 위치하는 메서드가 현재 실행 중인 메서드이며, 나머지는 대기 상태에 있게 된다.



- 메서드가 호출되면 수행에 필요한 만큼의 메모리를 스택에 할당받는다.
- 메서드가 수행을 마치고 나면 사용했던 메모리를 반환하고 스택에서 제거된다.
- 호출스택의 제일 위에 있는 메서드가 현재 실행 중인 메서드이다.
- 아래에 있는 메서드가 바로 위에 메서드를 호출한 메서드이다.

반환타입(return type)이 있는 메서드는 종료되면서 결과값을 자신을 호출한 메서드(caller)에게 반환한다. 대기상태에 있던 호출한 메서드(caller)은 넘겨받은 반환값으로 수행을 계속 진행하게 된다.

● **기본형 매개변수와 참조형 매개변수**

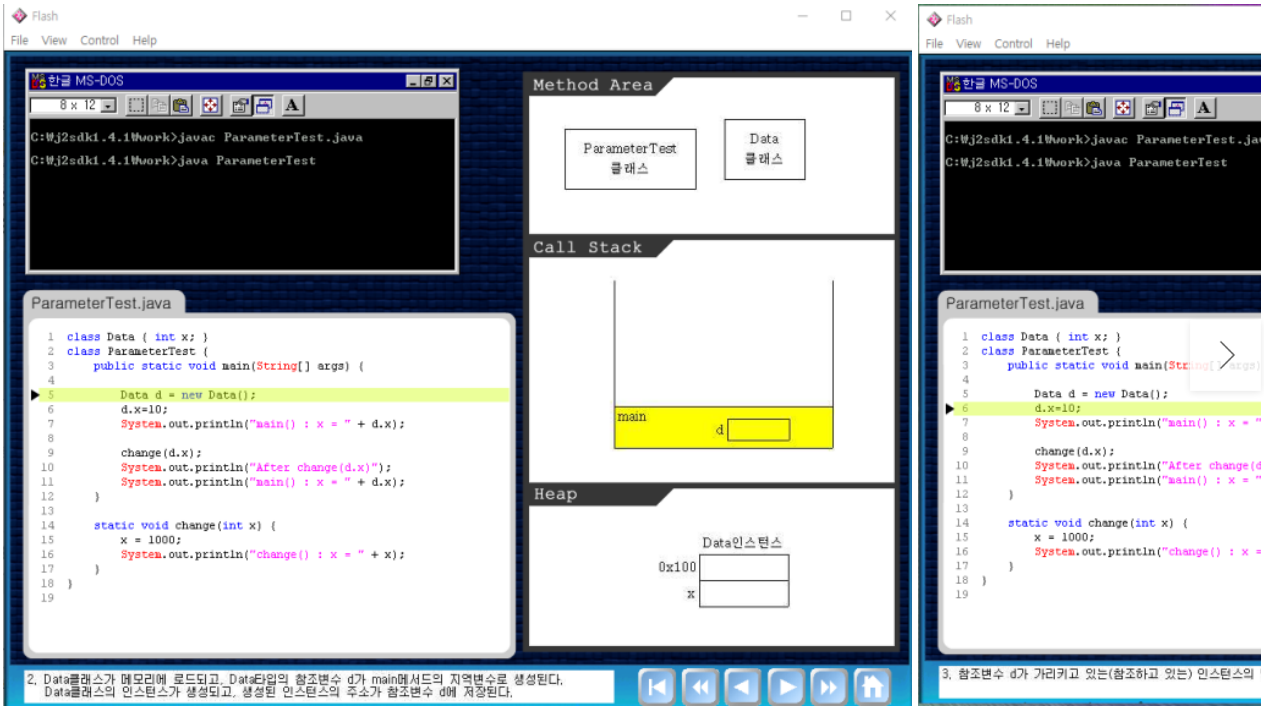
매개변수의 타입이 기본형(primitive type)일 때는 기본형 값이 복사되겠지만, 참조형(reference type)이면 인스턴스의 주소가 복사된다. 때문에 메서드의 매개변수를 기본형으로 선언하면 단순히 저장된 값을 얻어서 변수의 값을 읽기만 할 수 있지만(read only), 참조형으로 선언하면 값이 저장된 곳의 주소를 알 수 있기 때문에 값을 읽고 변경하는 것도 가능하다(read & write).

1) 기본형 매게변수 예시

```
class Data {
    int x;
}

class PrimitiveParamEx {
    public static void main(String[] args) {
    Data d = new Data();
    d.x = 10;
    System.out.println("main() : x = " + d.x);      // 1. main() : x = 10

    change(d.x);
    System.out.println("After change(d.x)");        // 3. After change(d.x)
    System.out.println("main() : x = " + d.x);      // 4. main() : x = 10
    }

    static void change(int x) {     // 기본형 매게변수
        x = 1000;
        System.out.println("change() : x = " + x);      // 2. change() : x = 1000
    }
}
```

1) change메서드가 호출되면서 d.x가 change메서드의 매게변수 x에 복사된다
2) change메서드에서 x의 값을 1000으로 변경된다
3) change메서드가 종료되면서 매개변수 x는 스택에서 제거된다

d.x의 값이 변경된 것이 아니라, change메서드의 매개변수 x의 값이 변경된 것이다. 즉, 복사본이 변경된 것이라 원본에는 영향을 미치지 못한다. 이처럼 기본형 매개변수는 변수에 저장된 값을 읽을 수만 있을 뿐, 변경할 수는 없다.

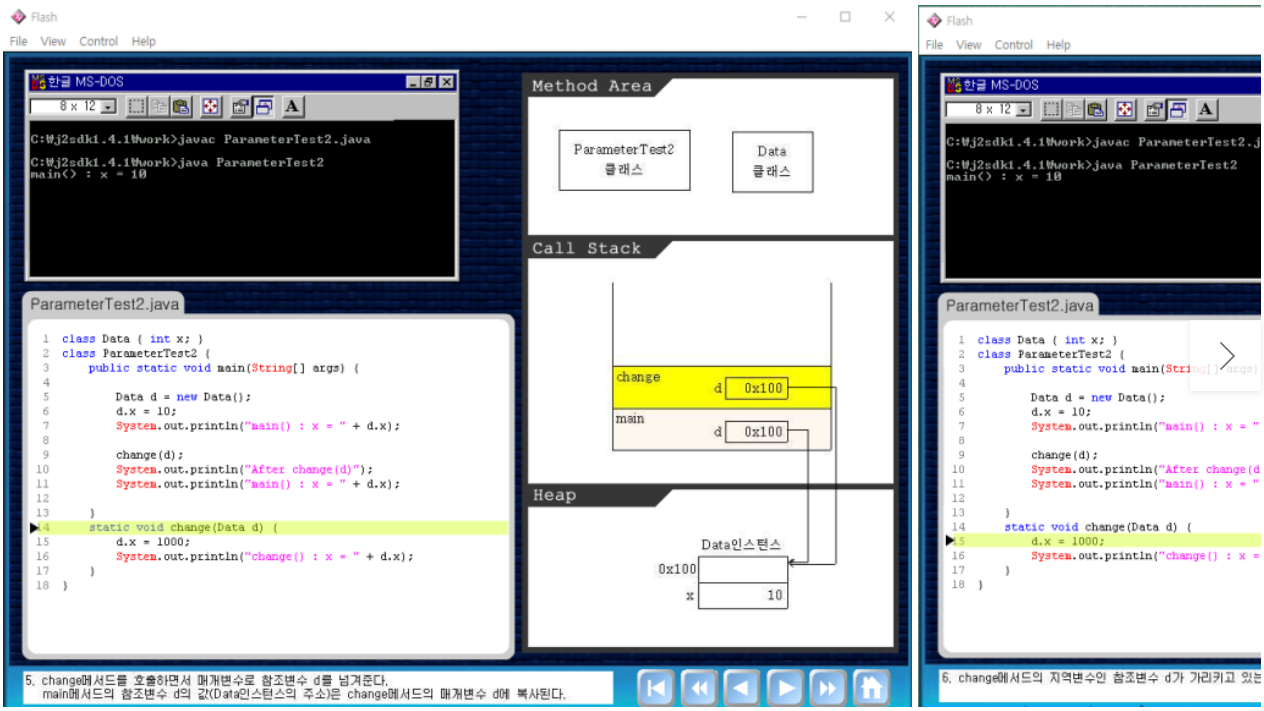


2) 참조형 매개변수 예시

```
class Data {
    int x;
}

class PrimitiveParamEx {
    public static void main(String[] args) {
    Data d = new Data();
    d.x = 10;
    System.out.println("main() : x = " + d.x);      // 1. main() : x = 10

    change(d);
    System.out.println("After change(d)");         // 3. After change(d.x)
    System.out.println("main() : x = " + d.x);      // 4. main() : x = 1000
    }

    static void change(Data d) {      // 참조형 매개변수
        d.x = 1000;
        System.out.println("change() : x = " + d.x);      // 2. change() : x = 1000
    }
}
```

1. change메서드가 호출되면서 참조변수 d의 값(주소)가 매개변수 d에 복사됨. 이제 매개변수d에 저장된 주소값으로 x에 접근이 가능.
2. change메서드에서 매개변수 d로 x의 값을 1000으로 변경
3. change메서드가 종료되면서 매개변수 d는 스택에서 제거됨.

main메서드의 참조변수 d와 change메서드의 참조변수 d는 같은 객체를 가르키게 된다. 그래서 매개변수 d로 x의 값을 읽는 것과 변경하는 것이 모두 가능한 것이다. 즉, change메서드의 매개변수가 참조형이라서 값이 저장된 주소를 change메서드에게 넘겨주었기 때문에 값을 읽어오는 것뿐만이 아니라 변경하는 것도 가능하다. 만약 change의 매개변수가 배열이라고 해도, 배열도 객체와 같이 참조변수를 통해 데이터가 저장된 공간에 접근하기 때문에 같은 결과를 얻는다.



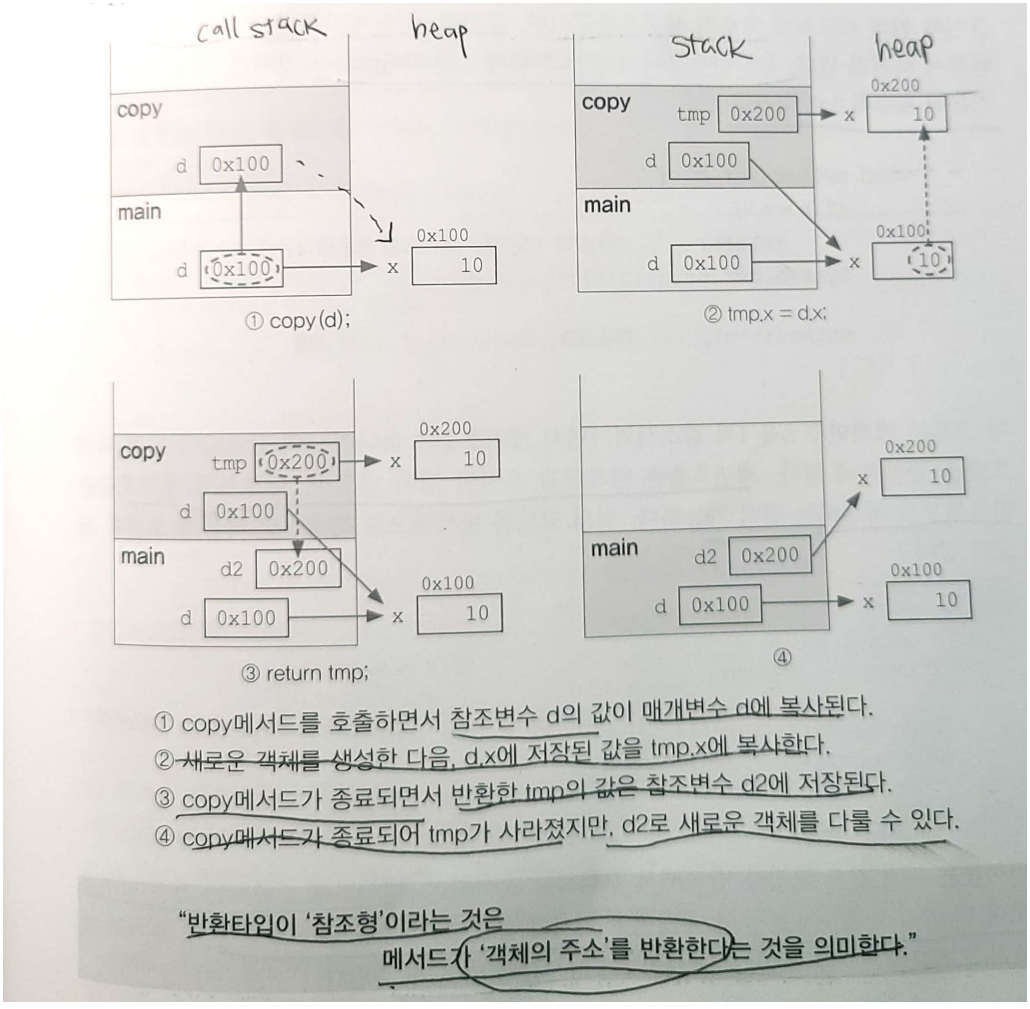
▶ 참조형 매개변수를 활용하면 반환값이 있는 메서드를 반환값이 없는 메서드로 바꾸어 반환값이 없어도 메서드의 실행결과는 얻어올 수 있다. 메서드는 단 하나의 값만을 반환할 수 있지만, 이것을 응용하면 여러 개의 값을 반환받는 것과 같은 효과를 얻을 수 있다.

```
class ReturnTest {
    public static void main(String[] args) {
        ReturnTest r = new ReturnTest();

        int result = r.add(3, 5);
        System.out.println(result);     // 8

        int[] result2 = {0};    // 배열을 생성하고 result2[0]의 값을 0으로 초기화
        r.add(3, 5, result2);    // 배열을 add메서드의 매개변수로 전달
        System.out.println(result2[0]);
    }

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

    void add(int a, int b, int[] result) {
        result[0] = a + b;
    }
}
```

● **참조형 반환타입**: 반환하는 값의 타입이 참조형일 수 있다. 모든 참조형 타입의 값은 '객체의 주소'이므로 정수값이 반환되는 것이다.

```
class Data {
    int x;
}

Class ReferneceReturnEx {
    public static void main(String[] args) {
    Data d = new Data();
    d.x = 10;

    Data d2 = copy(d);
    System.out.println("d.x = "+d.x);          // 10
    System.oout.println("d2.x = "+d2.x);          // 10
    }

    static Data copy(Data d) {
    Data tmp = new Data();          // 새로운 객체 tmp을 생성한다
    tmp.x = d.x;          // d.x(10)값을 tmp.x에 복사한다

    return tmp;          // 복사한 객체의 주소를 반환한다.
}
}
```

copy메서드 내에서 생성한 객체를 main메서드에서 사용할 수 있으려면, 새로운 객체의 주소를 반환해 줘야 한다. 그렇지 않으면, copy메서드가 종료되면서 새로운 객체의 참조가 사라지기 때문에 더이상 이 객체를 사용할 방법이 없다.



● **재귀호출(recursive call)**: 메서드 내부에서 메서드 자신을 다시 호출하는 것을 재귀호출이라고 하며, 재귀호출을 하는 메서드를 **제귀 메서드(recursive method)**라고 한다. 호출된 메서드는 값에 의한 호출(call by value)를 통해, 원래의 값이 아닌 복사된 값으로 작업하기 때문에 호출한 메서드와 관계없이 독립적인 작업수행이 가능하다. 오로지 재귀호출 뿐이면, 무한히 자기 자신을 호출하기 때문에 무한 반복문에 빠지기 때문에, 조건문이 필수적으로 따라다닌다 (특정 값일 때 reutrn;을 적어서 메서드를 종료하거나 오듀문구 출력). 재귀호출은 반복문과 유사한 점이 많으며, 대부분의 재귀호출을 반복문으로 작성하는 것이 가능하다.

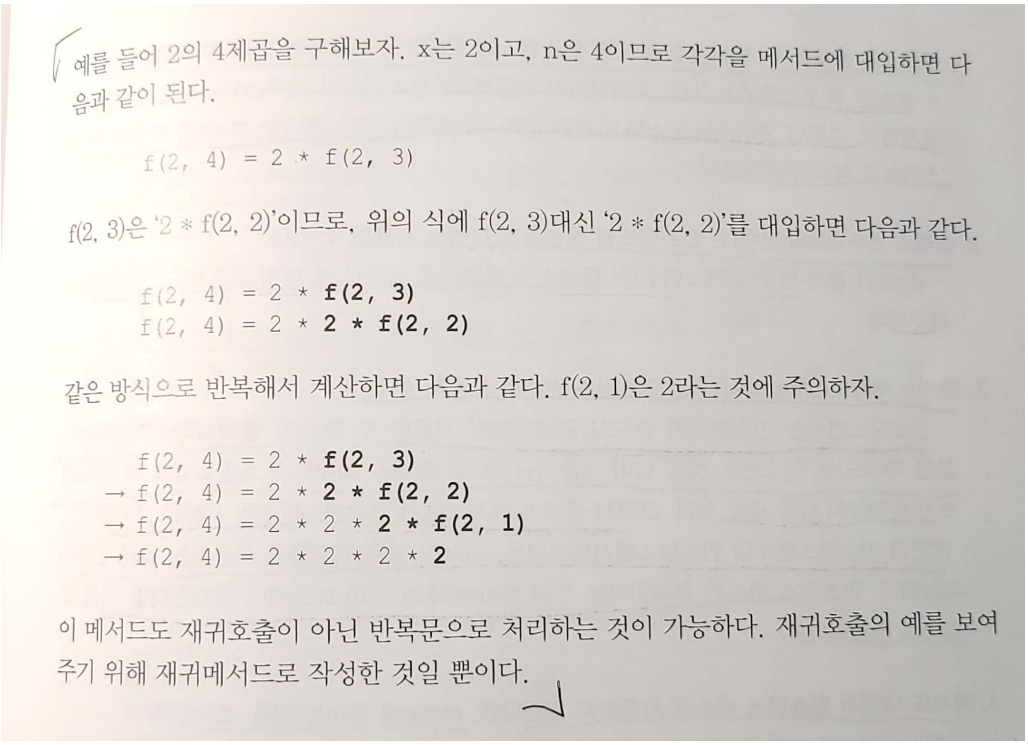
```
// 재귀 메서드 예시
void method(int n) {
    if (n == 0) {
        return;
    }

    System.out.println(n);
    method(--n);          // 재귀호출
}

// 위의 재귀 메서드를 while반복문으로 작성
void method(int n) {
    while (n!= 0) {
        System.out.println(n--);
    }
}
```

메서드를 호출하는 것은 매개변수 복사와 종료 후 복귀할 주소 저장 등 반복문보다 몇 가지 과정이 추가로 필요하기 때문에 반복문보다 재귀호출의 수행시간이 오래 걸린다. 그러나 재귀호출이 주는 논리적 간결함 때문에 다소 비효율적이더라도 알아보기 쉽게 작성한는 것이 논리적 오듀가 발생할 확률도 줄어들고 나중에 수정하기도 좋다. 재귀호출은 비효율적이므로 재귀호출에 드는 비용보다 재귀호출의 간결함이 주는 이득이 충분히 큰 경우에만 사용해야 한다. 대표적인 재귀효출의 예는 factorial(팩토리얼)을 구하는 것이다.

```java
class FactorialTest {
    public static void main(String args[]) {
        int n = 13;
        int result = 0;

        for (int i = 1; i <= n; i++) {
            result = factorial(i);
            if (result == -1) {
                System.out.printf("유효하지 않은 값입니다.(0<n<=12):%d%n", n);
                break;
            }
        }
            System.out.printf("%2d!=%20d\n", i, result);
    }

// 만일 factorial메서드의 매개변수 n의 값이 0이나 100,000와 같이 큰 경우에는,
// 재귀호출이 무한반복 일어나고 스택에 계속 데이터가 쌓여가기 때문에, 스택의 저장한계를
// 넘어서서 스택오버플로우 에러(Stack Overflow Error)가 발생한다.
// 때문에 메게변수의 유효성검사가 있어야 한다.

    static int factorial(int n) {
        if (n <= 0 || n > 12) {     // 매개변수의 유효성 검사
        // 상한값을 12로 정한 이유는 13!의 값이 반환타입인 int의 최대값을 넘어서기 때문이다.
            return -1;
        }
        if (n == 1) {       // factorial(1)의 결과값은 1로, 1이 return된다.
            return 1;
        }
        return n * factorial(n - 1);    // 재귀호출
    }
}

// 위의 재귀호출은 이렇게 반복문으로 적을 수도 있다.
// int factorial(int n) {
//     int result = 1;
//     while (n! = 0) {
//         result *= n--;
//     }
//     return result;
// }
// 위의 반복문은 재귀호출과 달리 많은 수의 반복에도 스택오버플로우 에러 같은 메모리 부족문제를
// 겪지 않을 뿐만이 아니라 속도도 빠르다.
// 재귀호출은 비효율적이므로 재귀호출에 드는 비용보다 재귀호출의 간결함이 주는 이득이 충분히
// 큰 경우에만 사용해야 한다.
```

```
 1!=                   1
 2!=                   2
 3!=                   6
 4!=                  24
 5!=                 120
 6!=                 720
 7!=                5040
 8!=               40320
 9!=              362880
10!=             3628800
11!=            39916800
12!=           479001600
유효하지 않은 값입니다.(0<n<=12):13
```

```java
// x^1~x^n의 합을 구하는 예제

class PowerTest {
    public static void main(String[] args) {
        int x = 2;
        int n = 4;
        long result = 0;

        for (int i = 1; i <= n; i++) {
            result += power(x, i);
        }
        System.out.println(result);
    }

    static long power(int x, int n) {
        if (n == 1) {
            return x;
        }
        return x * power(x, n-1);
    }
}
```



예를 들어 2의 4제곱을 구해보자. x는 2이고, n은 4이므로 각각을 메서드에 대입하면 다음과 같이 된다.

f(2, 4) = 2 * f(2, 3)

f(2, 3)은 '2 * f(2, 2)'이므로, 위의 식에 f(2, 3)대신 '2 * f(2, 2)'를 대입하면 다음과 같다.

f(2, 4) = 2 * **f(2, 3)**
f(2, 4) = 2 * **2 * f(2, 2)**

같은 방식으로 반복해서 계산하면 다음과 같다. f(2, 1)은 2라는 것에 주의하자.

f(2, 4) = 2 * **f(2, 3)**
→ f(2, 4) = 2 * **2 * f(2, 2)**
→ f(2, 4) = 2 * 2 * **2 * f(2, 1)**
→ f(2, 4) = 2 * 2 * 2 * **2**

이 메서드도 재귀호출이 아닌 반복문으로 처리하는 것이 가능하다. 재귀호출의 예를 보여주기 위해 재귀메서드로 작성한 것일 뿐이다.

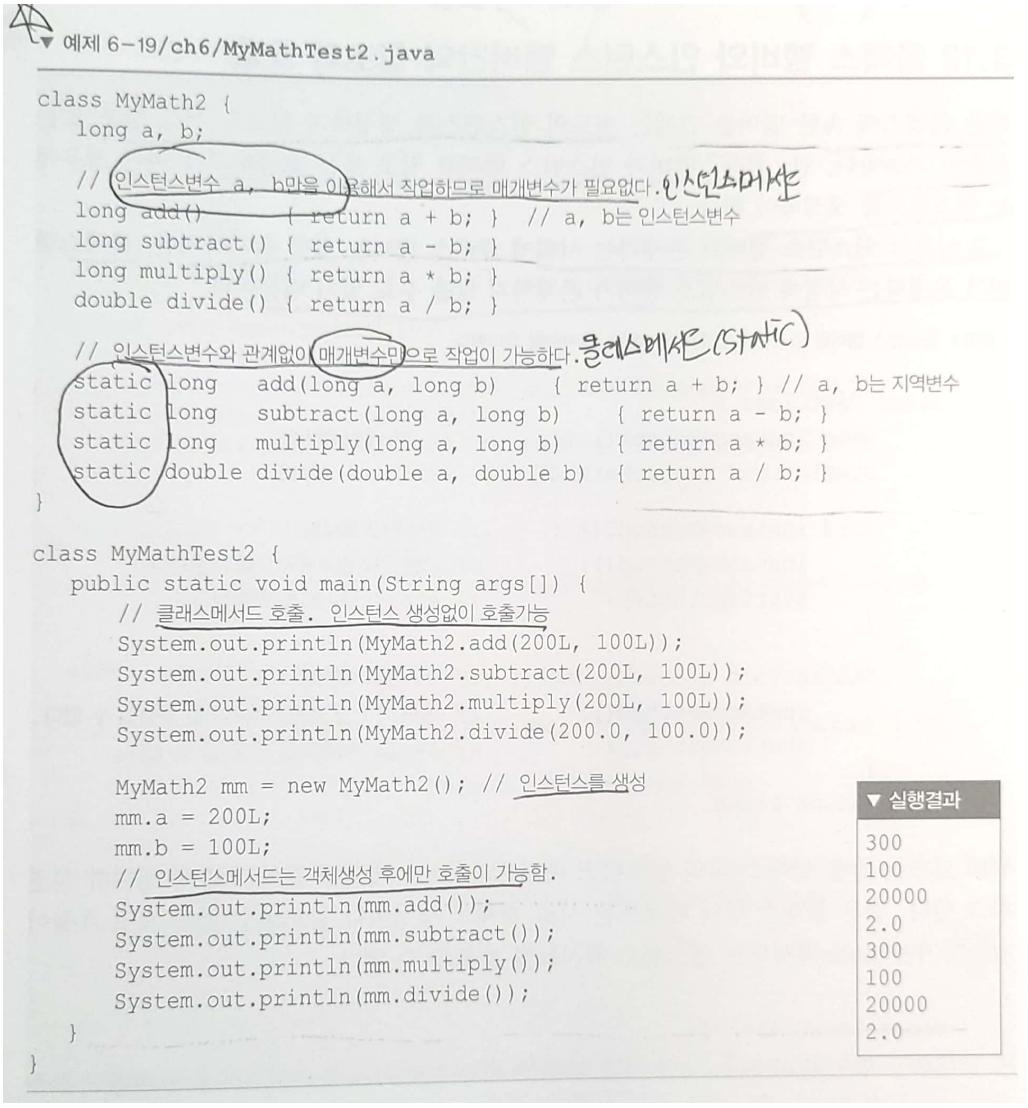
● **클래스 메서드(static 메서드)와 인스턴스 메서드:** 작성된 메서드 중에서 인스턴스 변수나 인스턴스 메서드를 사용하지 않는 메서드에 static을 붙일 것을 고려한다.

▶ 인스턴스 메서드: 인스턴스 변수와 관련된 작업을 하는, 즉 메서드의 작업을 수행하는데 인스턴스 변수를 필요로 하는 메서드이다. 인스턴스 변수는 인스턴스(객체)를 생성해야만 만들어지므로 인스턴스 메서드는 반드시 인스턴스를 생성해야만 호출할 수 있다.

▶ 클래스 메서드(static 메서드): 인스턴스와 관계없는 (인스턴스 변수나 인스턴스 메서드를 사용하지 않는) 메서드를 클래스 메서드로 정의한다. 클래스 메서드는 객체를 생성하지 않고도 '클래스이름.메서드이름(매개변수)'와 같은

식으로 호출이 가능하다.
Ex) Math.random()과 같은 Math클래스의 메서드는 모두 클래스 메서드이다. Math클래스에는 인스턴스변수가 하나도 없거니와 작업을 수행하는데 필요한 값들을 모두 매개변수로 받아서 처리하기 때문이다.

1) 클래스를 설계할 때, 멤버변수 중 모든 인스턴스에 공통으로 사용하는 것에 static을 붙인다. 생성된 각 인스턴스는 서로 독립적이기 때문에 각 인스턴스 변수는 서로 다른 값을 유지하지만, 모든 인스턴스에서 같은 값이 유지되어야 하는 변수는 static을 붙여서 클래스변수로 정의해야 한다.
2) 클래스 변수(static 변수)는 인스턴스를 생성하지 않아도 사용할 수 있다. 클래스가 메모리에 올라갈 때 이미 method area에 자동적으로 생성되기 때문이다.
3) 클래스 메서드(static method)는 인스턴스 변수를 사용할 수 있다. 인스턴스변수는 인스턴스가 반드시 존재해야만 사용할 수 있는데, 클래스메서드는 인스턴스 생성없이 호출 가능하므로 클래스 메서드가 호출되었을 때 인스턴스가 존재하지 않을 수도 있다. 반면에 인스턴스변수나 인스턴스메서드에서는 static이 붙은 클래스 변수나 클래스 메서드를 사용하는 것이 가능하다. 인스턴스 변수가 존재한다는 것은 static변수가 이미 메모리에 존재한다는 것을 의미하기 때문이다.
4) 메서드 내에서 인스턴스 변수를 사용하지 않는다면, static을 붙이는 것을 고려한다. 메서드 호출시간이 짧아지므로 성능이 향상된다. 인스턴스 메서드는 실행 시 호출되어야 할 메서드를 찾는 과정이 추가적으로 필요하기 때문에 시간이 더 걸린다.



예제 6-19/ch6/MyMathTest2.java

```
class MyMath2 {
    long a, b;

    // 인스턴스변수 a, b만을 이용해서 작업하므로 매개변수가 필요없다.인스턴스메서드
    long add()      { return a + b; }  // a, b는 인스턴스변수
    long subtract() { return a - b; }
    long multiply() { return a * b; }
    double divide() { return a / b; }

    // 인스턴스변수와 관계없이 매개변수만으로 작업이 가능하다.클래스메서드(static)
    static long   add(long a, long b)     { return a + b; } // a, b는 지역변수
    static long   subtract(long a, long b)    { return a - b; }
    static long   multiply(long a, long b)    { return a * b; }
    static double divide(double a, double b)  { return a / b; }
}

class MyMathTest2 {
    public static void main(String args[]) {
        // 클래스메서드 호출. 인스턴스 생성없이 호출가능
        System.out.println(MyMath2.add(200L, 100L));
        System.out.println(MyMath2.subtract(200L, 100L));
        System.out.println(MyMath2.multiply(200L, 100L));
        System.out.println(MyMath2.divide(200.0, 100.0));

        MyMath2 mm = new MyMath2(); // 인스턴스를 생성
        mm.a = 200L;
        mm.b = 100L;
        // 인스턴스메서드는 객체생성 후에만 호출이 가능함.
        System.out.println(mm.add());
        System.out.println(mm.subtract());
        System.out.println(mm.multiply());
        System.out.println(mm.divide());
    }
}
```

▼ 실행결과

```
300
100
20000
2.0
300
100
20000
2.0
```

● **클래스 멤버와 인스턴스 멤버간의 참조와 호출**: 클래스 멤버로는 클래스 변수와 클래스 메서드가 있으며, 인스턴스 멤버로는 인스턴스 변수와 인스턴스 메서드가 있다. 같은 클래스에 속한 멤버들 간에는 별도의 인스턴스를 생성하지 않고도 서로 참조 또는 호출이 가능하다. 단, 클래스멤버가 인스턴스멤버로 참조 또는 호출하고자 하는 경우에는

인스턴스를 생성해야 한다. 인스턴스 멤버가 존재하는 시점에 클래스 멤버는 항상 존재하지만, 클래스멤버가 존재하는 시점에 인스턴스 멤버가 존재하지 않을 수도 있기 때문이다.

클래스멤버는 언제나 참조 또는 호출이 가능하기 때문에 인스턴스멤버가 클래스멤버를 사용하는 것은 언제나 가능하다. 클래스멤버간의 참조 또는 호출 역시 아무런 문제가 없다. 하지만 인스턴스멤버는 반드시 객체를 생성한 후에만 참조 또는 호출이 가능하기 때문에 클래스멤버가 인스턴스멤버를 참조, 호출하기 위해서는 객체를 생성해야 한다. 하지만 인스턴스멤버간의 호출에는 아무런 문제가 없다. 하나의 인스턴스멤버가 존재한다는 것은 인스턴스가 이미 생성되어있다는 것을 의미하며, 다든 인스턴스멤버들도 모두 존재하기 때문이다. 실제로는 같은 클래스 내에서 클래스멤버가 인스턴스멤버를 참조 또는 호출해야하는 경우는 드물다. 만약 그렇다면 인스턴스메서드로 작성해야할 메서드를 클래스메서드로 한 것은 아닌지 생각해 보아야 한다.

▼ 예제 6-20/ch6/MemberCall.java

```
class MemberCall {
   int iv = 10;
   static int cv = 20;

   int iv2 = cv;
// static int cv2 = iv;         // 에러. 클래스변수는 인스턴스 변수를 사용할 수 없음.
   static int cv2 = new MemberCall().iv;  // 이처럼 객체를 생성해야 사용가능.

   static void staticMethod1() {
       System.out.println(cv);
//     System.out.println(iv); // 에러. 클래스메서드에서 인스턴스변수를 사용불가.
       MemberCall c = new MemberCall();
       System.out.println(c.iv);    // 객체를 생성한 후에야 인스턴스변수의 참조가능.
   }

   void instanceMethod1() {
       System.out.println(cv);
       System.out.println(iv);      // 인스턴스메서드에서는 인스턴스변수를 바로 사용가능.
   }

   static void staticMethod2() {
       staticMethod1();
//     instanceMethod1();       // 에러. 클래스메서드에서는 인스턴스메서드를 호출할 수 없음.
       MemberCall c = new MemberCall();
       c.instanceMethod1(); // 인스턴스를 생성한 후에야 호출할 수 있음.
   }

   void instanceMethod2() {      // 인스턴스메서드에서는 인스턴스메서드와 클래스메서드
       staticMethod1();          // 모두 인스턴스 생성없이 바로 호출이 가능하다.
       instanceMethod1();
   }
}
```

수학에서의 대입법처럼,
c = new MemberCall()이므로
c.instanceMethod1();에서 c대신 new MemberCall()을 대입하여
사용할 수 있다.

MemberCall c = new Membercall();
int result = c.instanceMethod1();

위의 두줄을

int result = new MemberCall().instanceMethod1();

이렇게 적을 수 있다. 대신 참조변수를 선언하지 않았기 때문에 생성된 MemberCall인스턴스는 더이상 사용할 수 없다.

Resource: 자바의 정석