메소드, 유효범위

514760
2021년 봄학기
3/24/2021
박경신

메소드

- 특정 작업을 처리할 수 있도록 만들어진 코드의 묶음
- 메소드 구현
  - 여러 가지 명령문들을 조합해서 특정 작업을 처리할 수 있도록 코드를 작성하고 이름을 붙이는 것
  - 프로그램에서 메소드(함수)를 사용한다는 것은 해당 메소드(함수)가 할 수 있는 작업을 의뢰하는 것이며 " 메소드(함수)를 호출한다"라고 말함
  - 자바에서는 클래스 내부에서만 메소드(함수)를 구현할 수 있고 호출할 수 있음
  - 자바에서는 메소드 내부에 중첩 메소드(nested function)는 불가능

메소드

- 메소드는 왜 필요한가?
  - 문제를 작게 나누어서 해결(divide and conquer)
  - 코드의 재사용
  - 코드 수정의 편의성
  - 검증된 코드를 사용
  - 코드의 단순화와 가독성

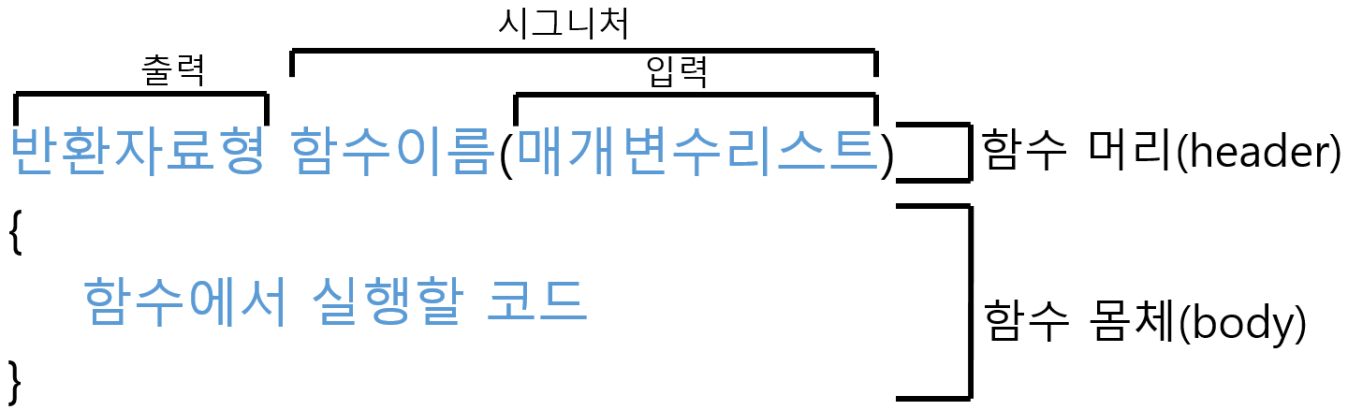
메소드를 이용한 중복코드를 줄이는 예시

```
public class NumberExample {
    public static int sum(int a, int b) {
        if (a < 0) a *= -1;
        if (b < 0) b *= -1;
        return a + b;
    }
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
        int a = Integer.parseInt(scan.nextLine());
        if (a < 0) a *= -1;
        int b = Integer.parseInt(scan.nextLine());
        if (b < 0) b *= -1;
        int result = a + b;
        System.out.println("결과는  " + result);
        a = Integer.parseInt(scan.nextLine());
        if (a < 0) a *= -1;
        b = Integer.parseInt(scan.nextLine());
        if (b < 0) b *= -1;
        result = a + b;
        System.out.println("결과는  " + result);
        System.out.println("결과는  " + sum(-2, 3));
    }
}
```

```
-2
-3
결과는 5
2
-3
결과는 5
결과는 5
```

메소드 구조

- 메소드는 머리(header)와 몸체(body)로 구성됨



  - 반환 자료형(return type)
    - 메소드에서 반환하는 결과 값의 자료형을 명시
  - 함수 시그니처(function signature)
    - 메소드를 구별하는데 사용됨
  - 구현
    - 메소드에서 처리해야 하는 작업을 코드로 작성한 부분

메소드 이름과 매개변수

- 메소드 이름은 식별자 이름 짓는 규칙을 따름
  - 하는 일이 무엇인지 알 수 있게 단어를 조합
  - 소문자 영문 알파벳으로 시작하고 영문자와 숫자의 조합 사용
- 메소드 매개변수 리스트
  - 메소드에 입력으로 전달될 값을 저장할 변수를 정의
  - 두 개 이상의 변수가 정의되면 ','로 분리
  - 왜 매개변수를 사용할까?
    - 함수에 매개변수를 사용하면 융통성과 재사용성을 높일 수 있음

```
float add(float a, float b) { return a + b; }
void printString(String str) {
    System.out.println(str);
}
printString("This is a reusable method"); // This is a reusable method
float value = add(10.5f, 20.2f);
printString("The value is " + value); // The value is 30.7
```

매소드 매개변수와 자동 형변환

- 메소드에 인자를 전달할 때 자동 형변환 발생

```
float add(float a, float b) {
    return a + b;
}
float value = add(3, 4); // 7.0 int->float implicit type conversion
value = add(3.2, 4.3); // Error: incompatible types: possible lossy conversion
from double to float
```

메소드 구현 순서

- 메소드가 클래스 내부에 있을 때와 JShell에서 독립적으로 존재할 때 매개변수의 사용법이 다를 수 있음
  - 클래스 내부에 만들어지는 멤버 메소드들은 같은 클래스에 있는 모든 멤버 변수에 직접 접근하고 사용 가능
  - 같은 클래스 내부에 구현된 메소드들은 서로 호출 가능. 즉, 메소드의 위치와 순서에 상관없음.
  - JShell에서는 메소드만 독립적으로 존재 가능함. 이때에는 함수에 대한 모든 입력을 매개변수를 통해 전달. 메소드 위치와 순서 중요함.

```
jshell> void sayHelloA() {
    String hello = "Hello";
    System.out.println(hello);
    printA();   // printA()는 현재 구현되어 있지 않음
}  created method sayHelloA(), however, it cannot be invoked until method
printA() is declared
```

JShell에서 메소드 구현 순서

```
// printA() 때문에 호출할 수 없음
jshell> sayHelloA();
|  attempted to call method sayHelloA() which cannot be invoked until method printA()
is declared

jshell> void printA() {
    System.out.println("A");
}
|  created method printA()

// printA()가 구현되었으므로 sayHello() 함수 호출 가능
jshell> sayHelloA();
Hello
A
```

메소드 리턴

- 리턴문은 기본형 값이나 참조값(레퍼런스)을 반환함
- 메소드 내부의 코드 실행을 중단시키고 실행 흐름을 메소드 호출한 곳으로 되돌림
- 두 가지 용도
  - 값의 반환

    return 값;
  - 반환값 없이 함수 실행 중단

    return;

```
// 정수값이 100보다 작은지 확인하는 메소드
boolean isLessThan100(int num) {
   return (num < 100) ? true : false;
}
```

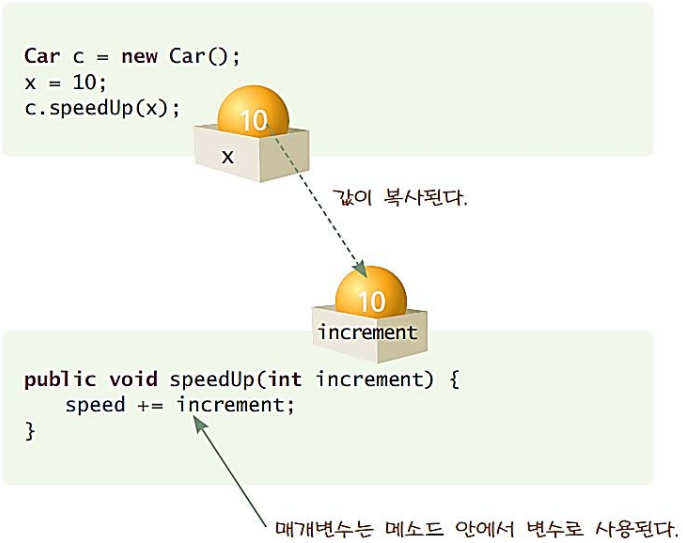
메소드의 참조형 리턴

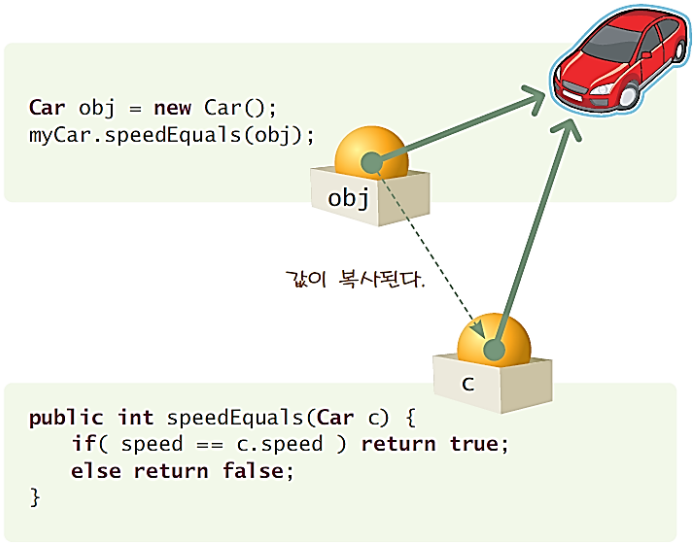
- 메소드의 참조형 리턴
  - 참조형의 레퍼런스 리턴
  - 메소드가 리턴하는 참조형의 타입은 리턴 받는 참조형 타입과 일치해야 함

```
// 원소가 5개인 배열을 생성하는 메소드
int[] makeArray() {
   int[] temp = new int[5];
   return temp;
}
```

메소드 매개변수 전달 방식

- 자바의 메소드 매개변수로 전달방식(Parameter Passing)
  - 기본형이나 참조형 모두 **"값 전달"** 방법이 사용됨
- **값 전달(call by value 또는 pass by value)** 방법
  - 매개변수에 변수값을 전달할 때 복사본을 생성



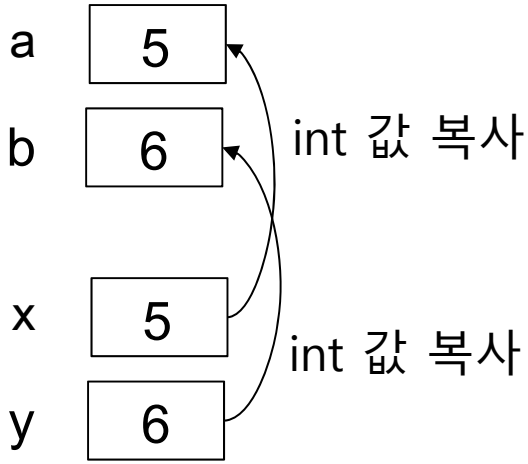


메소드 매개변수 전달 방식 – 기본형

- 기본형(int, double, char, boolean, etc)
  - 기본형 값이 복사되어 전달
  - 메소드의 매개 변수가 변경되어도 호출한 실제 인자 값은 변경되지 않음

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

```
int x = 5;
int y = 6;
int sum = sum(x, y);
```

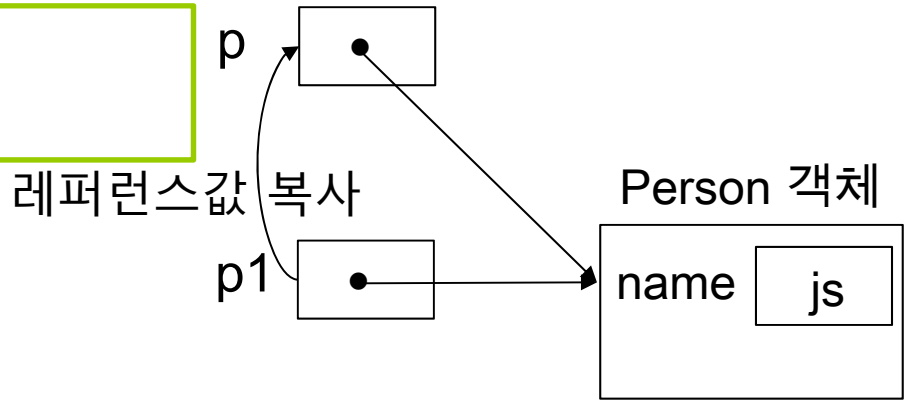


메소드 매개변수 전달 방법

- 참조형(class, interface, array, String, etc)
  - 참조값 (reference)을 전달
    - 객체 혹은 배열이 통째로 복사되어 전달되는 것이 아님
  - 메소드의 매개 변수와 호출한 실인자가 객체나 배열을 공유하게 됨

```
void sayHi(Person p) {
    System.out.println(" Hi " + p.getName());
}
```

```
Person p1 = new Person(" js ");
sayHi(p1);
```

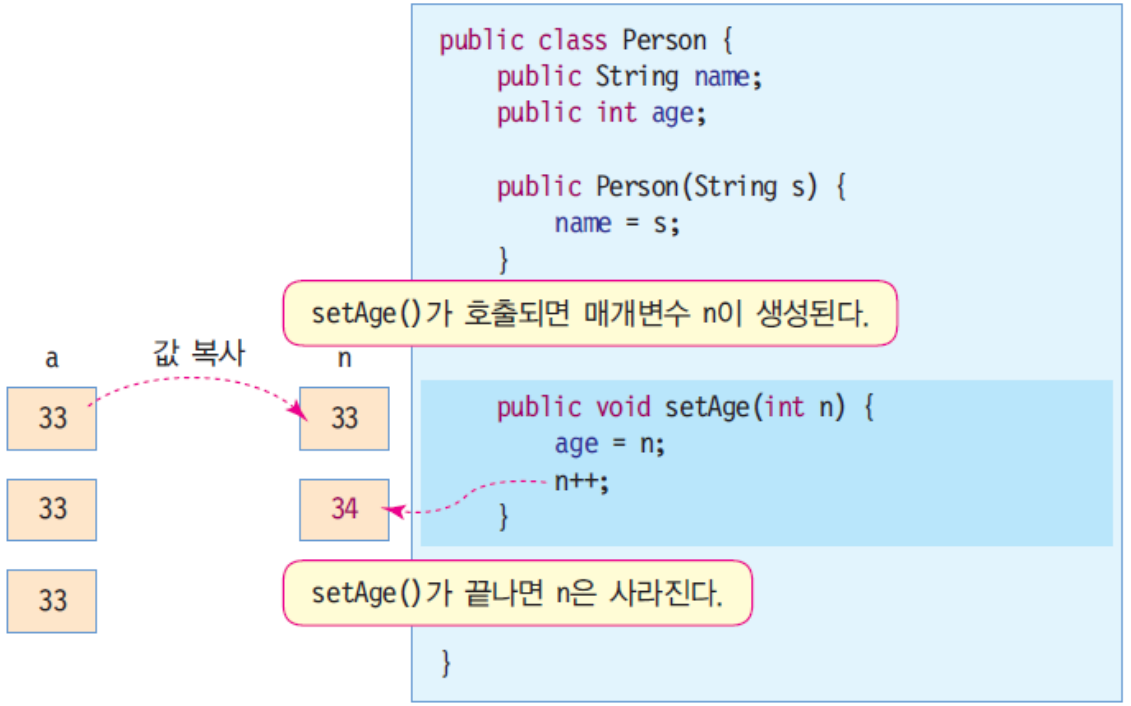


예시: 기본형의 전달

```
public class CallByValue {
    public static void main (String args[]) {
        Person aPerson = new Person("홍길동");
        int a = 33;

        aPerson.setAge(a);

        System.out.println(a);
    }
}
```



33

예시: 객체의 전달

```
class MyInt {
    int val;
    MyInt(int i) {
        val = i;
    }
}
public class CallByValueObject {
    public static void main(String args[]) {
        Person aPerson = new Person("홍길동");
        MyInt a = new MyInt(33);

        aPerson.setAge(a);

        System.out.println(a.val);
    }
}
```

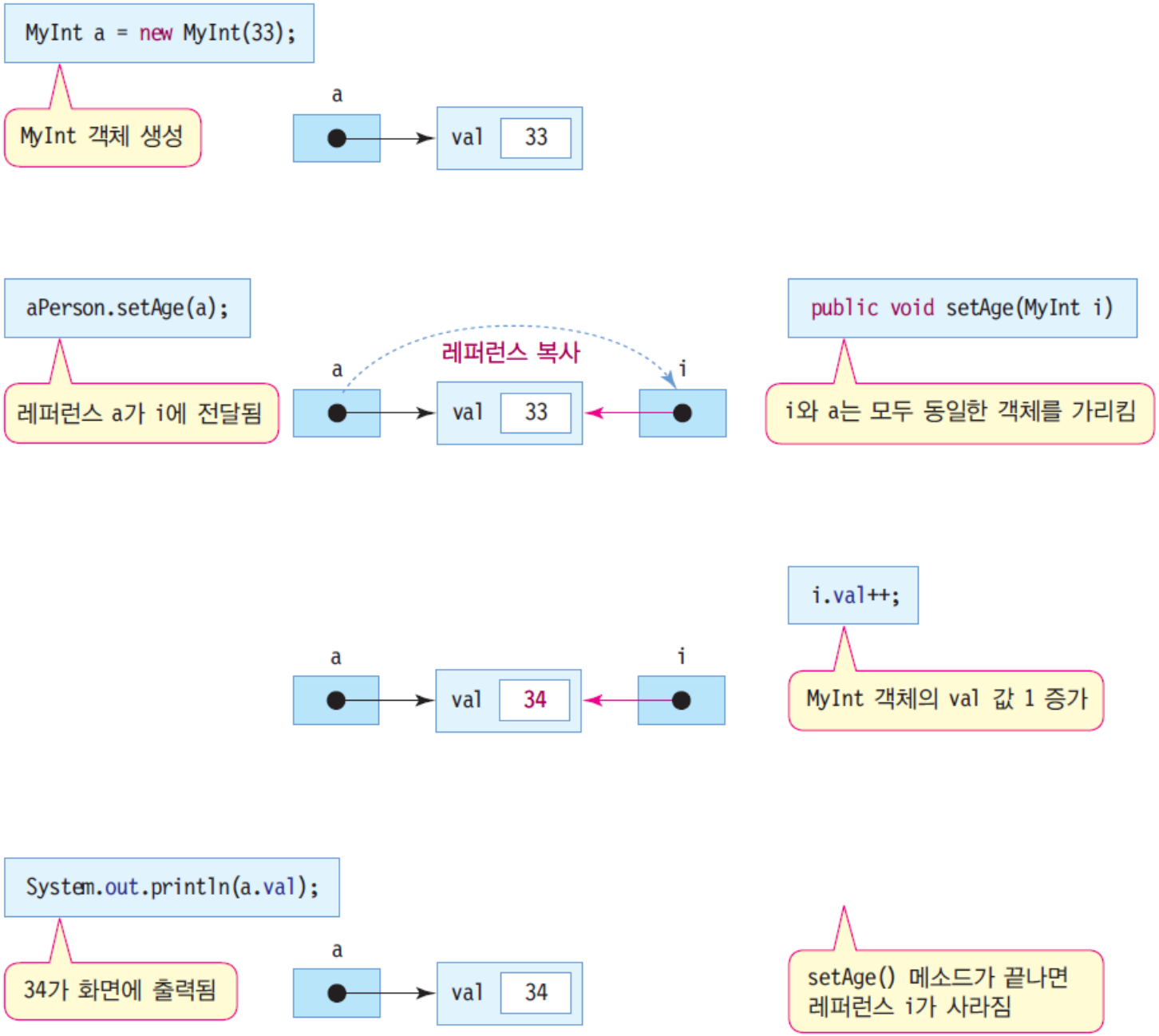
호출

```
public class Person {
    public String name;
    public int age;
    public Person(String s) {
        name = s;
    }

    public void setAge(MyInt i) {
        age = i.val;
        i.val++;
    }

}
```

* 객체가 복사되어 전달되는 것이 아님
객체에 대한 레퍼런스만 복사되어 전달


MyInt a = new MyInt(33);
MyInt 객체 생성
a
val
33
aPerson.setAge(a);
레퍼런스 a가 i에 전달됨
레퍼런스 복사
public void setAge(MyInt i)
i와 a는 모두 동일한 객체를 가리킴
34
i.val++;
MyInt 객체의 val 값 1 증가
System.out.println(a.val);
34가 화면에 출력됨
setAge() 메소드가 끝나면
레퍼런스 i가 사라짐

예시: 배열의 전달

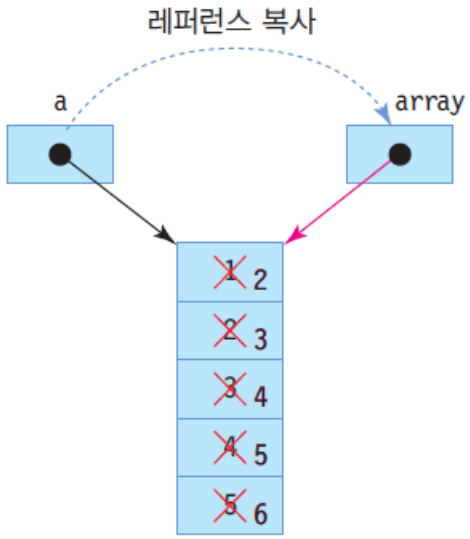
- 매개 변수에 배열의 레퍼런스만 전달

```
public class ArrayParameter {

    public static void main(String args[]) {
        int a[] = {1, 2, 3, 4, 5};


        increase(a);


        for(int i=0; i<a.length; i++)
            System.out.print(a[i]+" ");
    }
```



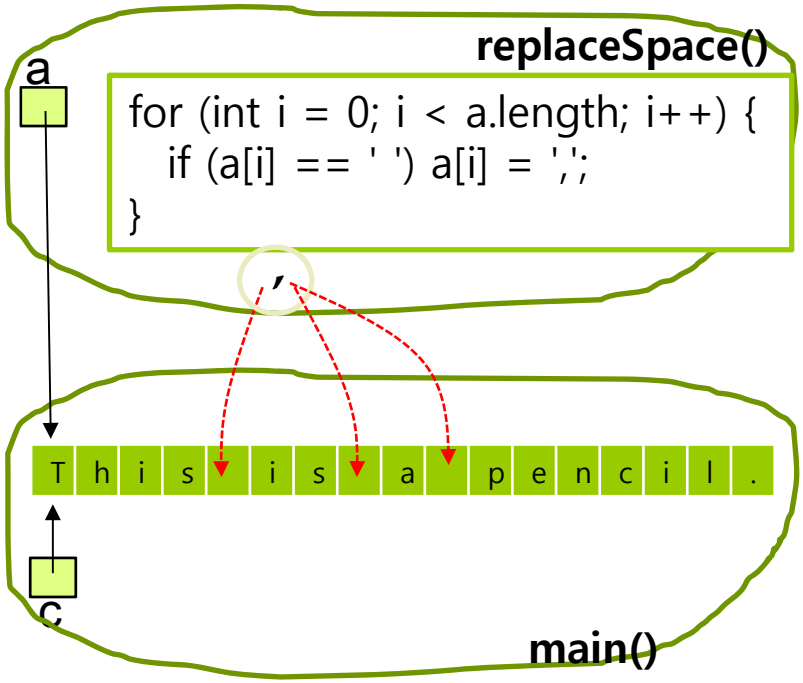

```
    static void increase(int[] array) {
        for(int i=0; i<array.length; i++) {
            array[i]++;
        }
    }
}
```

2 3 4 5 6

예제: 배열의 전달

```
public class ArrayParameter {
   static void replaceSpace(char[] a) {
      for (int i = 0; i < a.length; i++)
         if (a[i] == ' ')
            a[i] = ',';
   }
   static void printCharArray(char[] a) {
      for (int i = 0; i < a.length; i++)
         System.out.print(a[i]);
      System.out.println();
   }
   public static void main (String[] args) {
      char c[] = {'T','h','i','s',' ','i','s',' ','a',' '
','p','e','n','c','i','l','.'};
      printCharArray(c);
      replaceSpace(c);
      printCharArray(c);
   }
}
```

char 배열을 메소드의 인자로 전달하여 배열 안의 공백(' ') 문자를 ','로 대치하는 프로그램을 작성하라.




```
This is a pencil.
This,is,a,pencil.
```

Method 매개변수 값변경

- 기본형이나 참조형 모두 매개변수 값을 바꾸는 것은 함수 내부에서만 영향을 미침
  - 함수 외부에는 영향을 주지 않음

```
public static void square1(int x) {
    x *= x;
    System.out.println("Inside x=" + x);
    System.out.println(System.identityHashCode(x)); // 798154996(주소값)
}
public static void main (String[] args) {
    int i = 5;
    System.out.println("Before i=" + i); // Before i=5
    System.out.println(System.identityHashCode(i)); // 925858445(주소값)
    square1(i); // Inside x=25
    System.out.println("After i=" + i); // After i=5
    System.out.println(System.identityHashCode(i)); // 925858445(주소값)
}
```

Method 매개변수 값변경

- 참조형을 전달했을 때, 매개변수를 이용해서 객체의 멤버 변수를 변경하면 이는 함수 외부에 영향을 줌

```
public static void square2(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] *= arr[i];
    }
    System.out.println("Inside arr=" + Arrays.toString(arr));
    System.out.println(System.identityHashCode(arr)); // 681842940(주소값)
}
public static void main (String[] args) {
    int[] arr = {2, 3, 4};
    System.out.println("Before arr=" + Arrays.toString(arr)); // Before arr=[2, 3, 4]
    System.out.println(System.identityHashCode(arr)); // 681842940(주소값)
    square2(arr); // Inside arr=[4, 9, 16]
    System.out.println("After arr=" + Arrays.toString(arr)); // After arr=[4, 9, 16]
    System.out.println(System.identityHashCode(arr)); // 681842940(주소값)
}
```

Method 매개변수 값변경

```
class IntValue {
    public int value;
    public IntValue(int value) {
        this.value = value;
    }
}
public static void square3(IntValue x) {
    x.value *= x.value;
    System.out.println("Inside x.value=" + x.value);
    System.out.println(System.identityHashCode(x)); // 523429237(주소값)
}
public static void main (String[] args) {
    IntValue v = new IntValue(5);
    System.out.println("Before v.value=" + v.value); // Before v.value=5
    System.out.println(System.identityHashCode(v)); // 523429237(주소값)
    square3(v); // Inside v.value=25
    System.out.println("After v.value=" + v.value); // After v.value=25
    System.out.println(System.identityHashCode(v)); // 523429237(주소값)
}
```

Method 매개변수 값변경

```
static void changeArray1(int[] arr) {
    arr[0]=888; // arr -> myArray이므로 원본 배열의 첫번째 값은 888로 변경
    System.out.println("changeArray1 arr=" + Arrays.toString(arr));
    System.out.println(System.identityHashCode(arr)); // 664740647(주소값)
    arr = new int[] {-3, -1, -2, -3, -4}; // local 변수로 새롭게 할당하여 지정 그러나
원본 배열 변경 안됨
    System.out.println("changeArray1 arr=" + Arrays.toString(arr));
    System.out.println(System.identityHashCode(arr)); // 804564176(주소값)
}
public static void main(String[] args) {
    int[] myArray = {1, 4, 5};
    System.out.println("Before myArray=" + Arrays.toString(myArray));
    System.out.println(System.identityHashCode(myArray)); // 664740647(주소값)
    changeArray1(myArray);
    System.out.println("After changeArray1 myArray=" + Arrays.toString(myArray));
    System.out.println(System.identityHashCode(myArray)); // 664740647(주소값)
}
```

```
Before myArray=[1, 4, 5]
changeArray1 arr=[888, 4, 5]
changeArray1 arr=[-3, -1, -2, -3, -4]
After changeArray1 myArray=[888, 4, 5]
```

Method 매개변수 값변경

```
static void changeArray2(IntValue[] arr) {
    arr[0].value=888; // arr -> myArray이므로 원본 배열의 첫번째 값은 888로 변경
    System.out.println("changeArray2 arr=" + Arrays.toString(arr));
    System.out.println(System.identityHashCode(arr)); // 1421795058(주소값)
    arr = new IntValue[] {new IntValue(-3), new IntValue(-1), new IntValue(-2), new
IntValue(-3), new IntValue(-4)}; // local 변수로 새롭게 할당하여 지정 그러나 원본
배열 변경 안됨
    System.out.println("changeArray2 arr=" + Arrays.toString(arr));
    System.out.println(System.identityHashCode(arr)); // 1555009629(주소값)
}
public static void main(String[] args) {
    IntValue[] myArray2 = {new IntValue(1), new IntValue(4), new IntValue(5)};
    System.out.println("Before myArray2=" + Arrays.toString(myArray2));
    System.out.println(System.identityHashCode(myArray2)); // 1421795058(주소값)
    changeArray2(myArray2);
    System.out.println("After changeArray2 myArray2=" + Arrays.toString(myArray2));
    System.out.println(System.identityHashCode(myArray2)); // 1421795058(주소값)
}
```

```
Before myArray2=[value=1, value=4, value=5]
changeArray2 arr=[value=888, value=4, value=5]
changeArray2 arr=[value=-3, value=-1, value=-2, value=-3, value=-4]
After changeArray1 myArray2=[value=888, value=4, value=5]
```

swap() 구현 안됨

- 자바에서는 pass-by-value라서, 매개변수에 전달되는 두 개의 정수 값을 서로 바꾸는 swap() 구현이 안됨

```
static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    System.out.println("Inside swap: a=" + a + " b=" + b);
}
public static void main(String[] args) {
    int x = 10;
    int y = 20;
    swap(x, y);
    System.out.println("After swap: x=" + x + " y=" + y);
}
```

```
Inside swap a=20 b=10
After swap x=10 y=20
```

swap() 구현 안됨

- java.awt 모듈의 Point 클래스 객체를 두 개 바꿔보기로 함
- Point 객체 주소값을 pass-by-value해서, swap() 안됨

```
static void swap2(Point a, Point b) {
    Point temp = a;
    a = b;
    b = temp;
    System.out.println("Inside swap2: a=" + a + " b=" + b);
}
public static void main(String[] args) {
    Point x2 = new Point(10, 10);
    Point y2 = new Point(20, 20);
    swap2(x2, y2);
    System.out.println("After swap2: x2=" + x2 + " y2=" + y2);
}
```

```
Inside swap2 a=Point[x=20, y=20] b=Point[x=10, y=10]
After swap2 x2=Point[x=10, y=10] y2=Point[x=20, y=20]
```

swap() 구현

- Point 객체 두 값을 서로 바꾸는 swap() 구현을 위해, 객체의 멤버 변수를 변경하면 가능

```
static void swap3(Point a, Point b) {
    Point temp = new Point(a.x, a.y);
    a.setLocation(b);
    b.setLocation(temp);
    System.out.println("Inside swap3: a=" + a + " b=" + b);
}
public static void main(String[] args) {
    Point x3 = new Point(10, 10);
    Point y3 = new Point(20, 20);
    swap3(x3, y3);
    System.out.println("After swap3: x3=" + x3 + " y3=" + y3);
}
```

```
Inside swap3 a=Point[x=20, y=20] b=Point[x=10, y=10]
After swap3 x3=Point[x=20, y=20] y3=Point[x=10, y=10]
```

swap() 구현

- 정수 두 값을 서로 바꾸는 swap() 구현을 위해, int[]를 사용하고 배열의 멤버 변수를 변경하면 가능

```
static void swap4(int[] a, int[] b) {
    int temp = a[0];
    a[0] = b[0];
    b[0] = temp;
    System.out.println("Inside swap4: a=" + a[0] + " b=" + b[0]);
}
public static void main(String[] args) {
    int[] x4 = {10};
    int[] y4 = {20};
    swap4(x4, y4);
    System.out.println("After swap4: x4=" + x4[0] + " y4=" + y4[0]);
}
```

```
Inside swap4 a=20 b=10
After swap4 x4=20 y4=10
```

swap() 구현 안됨

- 배열 객체 주소값을 pass-by-value해서, swap() 안됨

```
static void swap5(int[] a, int[] b) {
    int[] temp = a;
    a = b;
    b = temp;
    System.out.println("Inside swap5: a=" + Arrays.toString(a) + " b=" +
Arrays.toString(b));
}
public static void main(String[] args) {
    int[] arr1 = {1, 2, 3};
    int[] arr2 = {4, 5, 6, 7, 8};
    swap5(arr1, arr2);
    System.out.println("After swap5: arr1=" + Arrays.toString(arr1) + "
arr2=" + Arrays.toString(arr2));
}
```

```
Inside swap5 a=[4,5,6,7,8] b=[1,2,3]
After swap5 arr1=[1,2,3] arr2=[4,5,6,7,8]
```

swap() 구현

- 배열을 swap() 구현 하기 위해서, 2차원배열을 사용함

```
static void swap6(int[][] a, int[][] b) {
    int[] temp = a[0];
    a[0] = b[0];
    b[0] = temp;
    System.out.println("Inside swap6: a=" + Arrays.toString(a[0]) + " b=" +
Arrays.toString(b[0]));
}
public static void main(String[] args) {
    int[][] arr3 = { new int[] {1, 2, 3} };
    int[][] arr4 = { new int[] { 4, 5, 6, 7, 8} };
    swap6(arr3, arr4);
    System.out.println("After swap6: arr3=" + Arrays.toString(arr3[0]) + "
arr4=" + Arrays.toString(arr4[0]));
}
```

```
Inside swap5 a=[4,5,6,7,8] b=[1,2,3]
After swap5 arr3=[4,5,6,7,8] arr4=[1,2,3]
```

변수의 유효 범위

- 유효범위(Scope)
  - 프로그래밍 언어에서 식별자(변수, 메소드, 클래스 등)는 코드에서 사용할 수 있는 영역이 정해져 있음
  - 변수나 메소드 등을 유효범위를 벗어나서 사용하려고 하면 컴파일 오류 발생
- 유효범위를 네 가지 영역으로 분류
  - 클래스
  - 메소드
  - for반복문
  - 코드 블록

변수의 유효 범위

- 클래스 영역

```
// Hello.java
class Hello {
    String toWhom = "world";
    Hello() { }
    Hello(String whom) {
        setWhom(whom);
    }
    void setWhom(String whom) {
        toWhom = whom;
    }
    void sayHello() {
        System.out.println("hello " + toWhom);
    }
}
```

변수의 유효 범위

- 메소드 영역

```
// TestHello.java
class TestHello {
    void callSayHello() {
        Hello hello = new Hello();
        hello.sayHello();
    }

    void anotherCallSayHello() {
        hello.sayHello(); // hello 객체가 없어서 사용 못함
    }
}
```

변수의 유효 범위

- for 반복문 범위
  - 명령문1에서 새로운 변수를 정의하고 초기값을 지정하는 경우, 이 변수는 for문에서만 사용할 수 있음

```
for (int i = 0; i < 5; i++) {
    System.out.println("number i = " + i);
}
System.out.println("i = " + i); // 오류 발생
```

- for문에서 변수를 여러 개 정의

```
for (int i = 0, j = 0; i < 5; i++, i++) {
    System.out.printf("number i = %d, j = %d\n " , i, j);
}
System.out.printf("number i = %d, j = %d\n " , i, j); // 오류 발생
```

변수의 유효 범위

- 코드 블록 범위
  - 메소드에서 정의된 매개 변수나 지역 변수와 동일한 이름의 변수는 코드 블록 내에 다시 만들 수 없음

```
class CodeBlockScope {
    public static void main(String[] args) {
        if (true) {
            int i = 3; // 이 영역을 벗어나면 사용 못함
            System.out.printf("조건문 i = %d\n", i);
        }
        for (int n = 0; n < 3; n++) {
            int i = 4; // 이 영역을 벗어나면 사용 못함
            System.out.printf("반복문 i = %d, n = %d\n", i, n);
        }
    }
}
```

유효 범위의 우선 순위

- 유효범위가 겹치는 동일 이름의 변수가 두 개 이상 정의되었을 때 어떤 변수가 사용되는가?
- 예를 들어 클래스의 멤버 변수와 똑같은 이름의 변수가 메소드 내부에서 다시 정의된다면?

클래스 유효범위
num1, num2

메소드
유효범위
num1

메소드
유효범위
num1,
num2

```
public class TestScope {
    static int num1 = 3;
    static int num2 = 4;
    static void printNumbers() {
        System.out.printf("num1 = %d, num2 = %d\n", num1, num2);
    }
    static void printNumbers2() {
        int num1 = 5;
        System.out.printf("num1 = %d, num2 = %d\n",  num1, num2);
    }
    static void printNumbers3(int num1) {
        int num2 = 5;
        System.out.printf("num1 = %d, num2 = %d\n", num1, num2);
    }
    public static void main(String[] args) {
        printNumbers();
        printNumbers2();
        printNumbers3(2);
    }
}
```

메소드 오버로딩

- 메소드 오버로딩(Overloading)
  - 한 클래스 내에서 두 개 이상의 이름이 같은 메소드 작성
    - 메소드 이름이 동일하여야 함
    - 매개 변수의 개수가 서로 다르거나, 타입이 서로 달라야 함
    - 리턴 타입은 오버로딩과 관련 없음

```
// 메소드 오버로딩이 성공한 사례
class MethodOverloading {
   public int sum(int i, int j) {
      return i + j;
   }
   public int sum(int i, int j, int k) {
      return i + j + k;
   }
   public double sum(double i, double j) {
      return i + j;
   }
}
```

```
// 메소드 오버로딩이 실패한 사례
class MethodOverloadingFail {
   public int sum(int i, int j) {
      return i + j;
   }
   public double sum(int i, int j) {
      return (double)(i + j);
   }
}
```

오버로딩된 메소드 호출

- 오버로딩이 지원되면 다른 입력에 대해 비슷한 일을 하는 함수들을 같은 이름으로 구현할 수 있어 프로그래머의 편의성과 코드의 가독성을 높일 수 있음

```
public class MethodOverloading {
    public static void main(String[] args) {
        int i = sum(2, 3);
        int j = sum(1, 2, 3);
        double k = sum(1.1, 2.2);
    }
}
```

```
public class MethodOverloading {
    static int sum(int i, int j) {
        return i + j;
    }
    static int sum(int i, int j, int k) {
        return i + j + k;
    }
    static double sum(double i, double j) {
        return i + j;
    }
}
```

오버로딩된 메소드 호출

- 메소드 오버로딩 사용시 **전달되는 인자를 취할 수 있는 함수 시그니처가 없다면 컴파일 오류 발생**

```
// TestNumber2.java
class TestNumber2 {
    static int add(int a, int b) {        return a + b;     }
    static float add(float a, float b) {        return a + b;     }
    public static void main(String[] args) {
        int sumInt = add(2, 3);
        System.out.printf("2 + 3 = %d\n", sumInt);
        float sumFloat = add(2.0f, 3.0f);
        System.out.printf("2.0 + 3.0 = %f\n",  sumFloat);
        double sumDouble = add(2.0, 3.0); // compile error – no suitable method
found for add(double, double)
        System.out.println("2 + 3 = " + sumDouble);
    }
}
```

메소드 오버로딩과 자동 형 변환

- 오버로딩 된 메소드을 사용할 때에도 인자와 매개변수 사이에 자동 형 변환 발생 가능
- 오버로딩 된 메소드 호출 하는 방법의 우선 순위
  - 인자의 자료형과 매개변수의 자료형이 정확하게 일치
  - 자동 형 변환을 통해 전달 가능
- 작은 범위에서 큰 범위의 자료형으로 값이 자동으로 변환되어 오버로딩된 함수가 호출되는 것을 프로모션(promotion)되었다고 함

메소드 오버로딩과 자동 형 변환

```
// TestNumber3.java
class TestNumber3 {
    static int add(int a, int b) {        return a + b;    }
    static float add(float a, float b) {        return a + b;    }
    public static void main(String[] args) {
        byte b1 = 2;
        byte b2 = 3;
        int sumInt = add(b1, b2); // b2, b2 byte->int implicit type conversion
        System.out.printf("2 + 3 = %d\n", sumInt);

        short s1 = 2000;
        int n2 = 3000;
        sumInt = add(s1, n2); // s1 short->int implicit type conversion
        System.out.printf("2 + 3 = %d\n", sumInt);
    }
}
```

예제: 인자 두 개를 전달 받아 합 또는 연결된 문자열을 반환 (메소드 오버로딩 사용)

```
class Add {
    static String add(String s1, String s2) {       return s1 + ' ' + s2;     }
    static int add(int n1, int n2) {         return n1 + n2;   }
    static double add(double d1, double d2) {   return d1+d2;   }
    public static void main(String[] args) {
        String newStr = add("hello", "world");
        int sumInt = add(5, 3);
        double sumDouble = add(5.1, 3.5);
        System.out.printf("newStr = %s\n", newStr);
        System.out.printf("sumInt = %d\n", sumInt);
        System.out.printf("sumInt = %f\n", sumDouble);
    }
}
```

재귀 호출

- 재귀호출/재귀함수(Recursive call 또는 Recursion)
  - 함수 코드 내부에서 자기 자신을 다시 호출하는 함수
- 함수 내부에서 사용되는 지역 변수의 값들은 자기 자신을 호출하기 전의 값들을 호출 후에도 그대로 보존함
  - 자기 자신을 다시 부르더라도 새로운 지역 변수들이 생성되는 것을 생각하면 됨
- 재귀호출을 빠져나갈 수 있도록 검사하고 종료하는 부분이 반드시 존재해야 함 (재귀호출 탈출 조건)
- 함수나 알고리즘에 따라서 재귀 호출이 오히려 쉽게 이해되는 경우가 있음
  - 팩토리얼 (factorial), 최대공약수, 피보나치 수열, 등

재귀호출

- 팩토리얼을 구현하는 재귀호출 작성
- 알고리즘
  - 만약 n이 1보다 작거나 같으면 값 1을 반환
  - 만약 n이 1보다 크다면 n * (n – 1)!을 반환

```
// n! = n * (n-1) * (n-2) …. * 1 (if n > 1)
// n! = 1 (if n == 1)
static int factorial(int n){
  if (n <= 1)
    return 1;
  else
    return(n * factorial(n-1));
 }
factorial(1);  // 1
factorial(5);  // 5! = 5 * 4 * 3 * 2 * 1 = 120
```

재귀호출

- 피보나치 수열 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233
- 알고리즘
  - 1항과 2항은 1
  - 3항 이후부터의 n항은 (n – 1)항 + (n – 2)항

```
// f(n) = f(n-1) + f(n-2) for n>=2, with f(0) = 0 and f(1) = 1
static long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;                    // 재귀호출 탈출 조건
    return fibonacci(n – 1) + fibonacci(n – 2);
}
fibonacci(1)                  // 1
fibonacci(2)                  // 1
fibonacci(3)                  // 2
fibonacci(6)                  // 8
```

재귀호출

- power 함수 작성
- 알고리즘
  - 만약 n이 0이면, 값 1을 반환
  - 만약 n이 0이 아니라면, b * power(b, n – 1)을 반환

```
// b^n = b * b^(n-1) (if n > 0)
// b^n = 1 (if n=0)
static int power(int b, int n){
  if (n == 0)
    return 1;
  else
    return(b * power(b, n-1));
 }
power(3, 2);  // 3^2 = 3 * 3 = 9
power(5, 5);  // 5^5 = 5 * 5 * 5 * 5 * 5 = 3125
```

재귀 호출

- 최대 공약수를 구하는 공식

```
gcd(a, b) = a;
            if ((a == b) && (a > 0) && (b > 0))
gcd(a, b) = gcd(a - b, b);
            if ((a > b) && (a > 0) && (b > 0))
gcd(a, b) = gcd(a, b - a);
            if ((b > a) && (a > 0) && (b > 0))
```

- 재귀호출로 구현

```
int gcd(int a, int b) {
    if (a == b) { return a; }
    else if (a > b) { return gcd(a - b, b); }
    else { return gcd(a, b - a); }
}
```

재귀 호출 예제

1~100까지의 합을 구하는 코드를 재귀 호출을 이용해서 구현

```
class TestRecursiveSum {
    static long sum(int n) {
        if (n == 1) { return 1; }
        else { return n + sum(n - 1); }
    }
    public static void main(String[] args) {
        System.out.println(sum(100));
    }
}
```