객체지향 프로그래밍
5장

메소드(함수), 유효범위

조용주
ycho@smu.ac.kr

함수란?

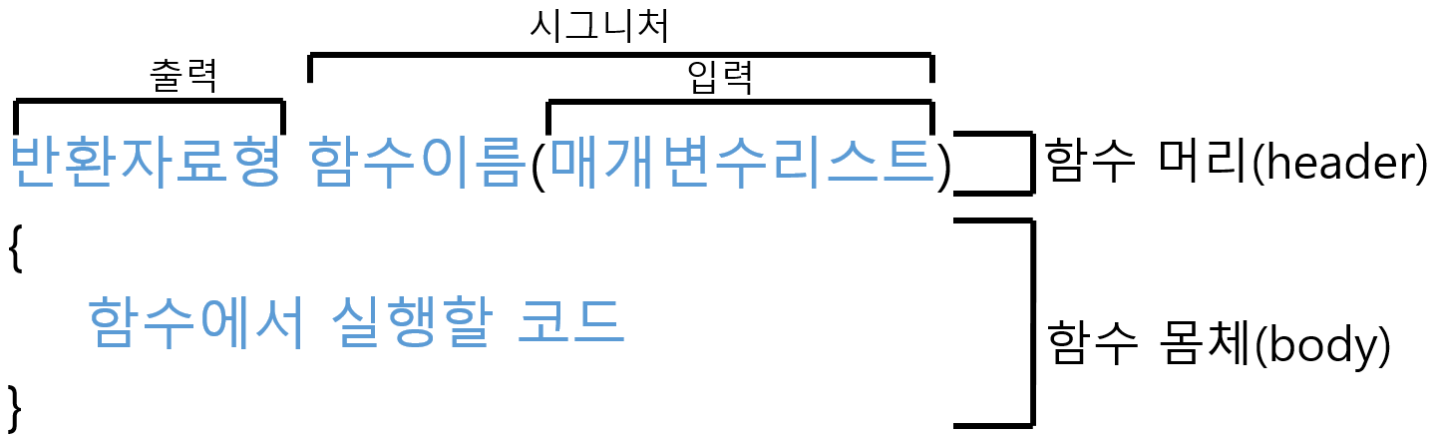
- 함수는 특정 작업을 처리할 수 있도록 만들어진 코드의 묶음
- 함수 구현
  - 여러 가지 명령문들을 조합해서 특정 작업을 처리할 수 있도록 코드를 작성하고 이름을 붙이는 것
  - 프로그램에서 함수를 사용한다는 것은 해당 함수가 할 수 있는 작업을 의뢰하는 것이며 "함수를 호출한다"라고 말함
  - 자바에서는 클래스 내부에서만 함수를 구현할 수 있고 호출할 수 있음
  - 예외적으로 JShell에서는 클래스 없이 함수만 구현하고 호출 가능

함수는 왜 필요한가?

- 문제를 작게 나누어서 해결
- 코드의 재사용이 수월함
- 코드 수정의 편의성이 높아짐
- 검증된 코드를 사용
- 코드 작성을 단순화시키고 가독성을 높일 수 있음

함수의 머리와 몸체

- 함수는 머리(header)와 몸체(body)로 구성됨



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

함수 이름과 매개변수 리스트

- 함수 이름은 식별자 이름 짓는 규칙을 따름
  - 하는 일이 무엇인지 알 수 있게 단어를 조합
  - 소문자 영문 알파벳으로 시작하고 영문자와 숫자의 조합 사용
  - 카멜 표기법을 이용
- 매개변수 리스트
  - 메소드에 입력으로 전달될 값을 저장할 변수를 정의

    ```
    자료형 변수_이름
    ```

  - 두 개 이상의 변수가 정의되면 ','로 분리

    ```
    자료형 변수_이름, 자료형 변수_이름
    ```

함수 정의

- 함수 정의 예

```
void printHelloString() {
    System.out.println("hello");
}

void printString(String str) {
    System.out.println(str);
}

int add(int num1, int num2) {
    int res;
    res = num1 + num2;
    return res;
}
```

왜 매개변수를 사용할까?

- 함수에 매개변수를 사용하면 융통성과 재사용성을 높일 수 있음
  - 매개변수를 이용하면 함수에서 작업하는 내용 중에 바뀔 수 있는 부분 대체 가능
  - 앞에서 본 printHelloString()과 printString() 함수를 비교

```
jshell> printString("hello");
hello

jshell> printString("This is a reusable function.");
This is a reusable function.

jshell> String str1 = "hello";

jshell> printString(str1 + " world");
hello world
```

매개변수와 자동 형 변환

- 함수에 인자를 전달할 때 자동 형 변환 발생 가능
  - 인자로 전달되는 값이 작은 범위의 자료형이고 매개변수가 더 큰 숫자 자료형이면 자동 형 변환 발생

```
jshell> float add(float a, float b) {
            return a + b;
        }

jshell> add(3, 4);
$2 ==> 7.0

jshell> add(3.7, 4.4);
|  Error:
|  incompatible types: possible lossy conversion
from double to float
|  add(3.7, 4.4);
|      ^-^
```

클래스의 멤버 함수와 멤버 변수

- 함수가 클래스 내부에 있을 때와 독립적으로 존재할 때 매개변수의 사용법이 다를 수 있음
  - 컴파일러를 이용할 때 함수는 클래스 내부에만 구현 가능
  - JShell에서는 함수만 독립적으로 존재 가능
- 독립적으로 함수를 만들 때에는 함수에 대한 모든 입력을 매개변수를 통해 전달
- 클래스 내부에 만들어지는 멤버 함수들은 같은 클래스에 있는 모든 멤버 변수에 직접 접근하고 사용 가능
  - 매개변수가 줄어들면 함수 사용법이 단순해짐
  - 클래스 내부의 결합성을 높일 수 있음

클래스 내에서 함수 구현 순서 또는 위치

- 같은 클래스 내부에 구현된 함수들은 서로 호출 가능 → 함수의 위치와 순서에 상관없음

```
class PrintName {
    String name;

    void setName(String nm) { name = nm; }
    void printName() { printString(name); }
    void printString(String str) {
        System.out.println(str);
    }
    public static void main(String[] args) {
        PrintName pn = new PrintName();
        pn.setName("Yongjoo Cho");
        pn.printName();
    }
}
```

JShell에서 함수 구현 순서

- JShell에서는 클래스 내부에 함수를 구현할 수도 있고, 클래스 없이 함수를 구현하고 호출도 가능
- 클래스 내부에 만들어지는 함수들은 위치와 순서 상관없이 서로 호출 가능
- 독립적으로 만들어지는 함수들은 먼저 함수를 정의해야 사용할 수 있음

```
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 함수 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
```

실습 문제 1: 함수를 이용해서 정수 출력하기

- 1~100 이내의 정수값을 입력 받아 화면에 출력하는 프로그램을 작성
  - 범위 밖의 정수값이 입력되었다면 "1~100 범위 밖 정수가 입력되었습니다"라는 문자열 출력
  - 정수를 입력 받아 화면에 출력하는 함수와 주어진 정수가 일정 범위 안에 있는지 확인하는 함수를 구현하고 사용
- 요구사항
  - 인자로 정수값 한 개를 입력 받아 화면에 출력하는 함수 작성
  - 정수값 한 개와 범위를 나타내는 정수 두 개를 입력 받는 함수를 구현하고, 정수값이 범위 안에 포함되면 true 아니면 false 반환
  - Scanner 클래스를 생성하고, 정수를 입력 받는 과정에 문제 없다고 가정

리턴문(return statement)

- 리턴문은 값이나 참조를 반환할 수 있고, 함수 내부의 코드 실행을 중단시키고 실행 흐름을 함수를 호출한 곳을 되돌림
- 두 가지 용도
  - 값의 반환

    ```
    return 값;
    ```

    - 함수 내부에 있는 코드 실행을 중단하고 실행 흐름을 함수를 호출한 곳으로 되돌림
    - 값 또는 참조값을 함수의 결과로 반환
      - return 키워드는 값이나 참조값을 생성할 수 있는 표현식과 함께 사용 가능

리턴문(return statement)

- 반환하는 값 또는 참조값의 자료형과 함수 머리의 반환 자료형은 일치해야 함
- 반환 자료형이 명시되어 있다면 값을 반환하는 리턴문이 한 개 이상 있어야 함
- 반환 자료형이 명시되어 있으면 값이 없는 리턴문은 사용할 수 없음

- 반환값 없이 함수 실행 중단

```
return ;
```

  - 값을 반환하지 않는 함수(반환 자료형이 void로 선언된 함수)에서 코드 실행을 종료시키고 함수를 호출했던 코드로 실행 흐름을 되돌림

실습 문제 2: 정수값이 100보다 작은지 확인하는 함수 구현

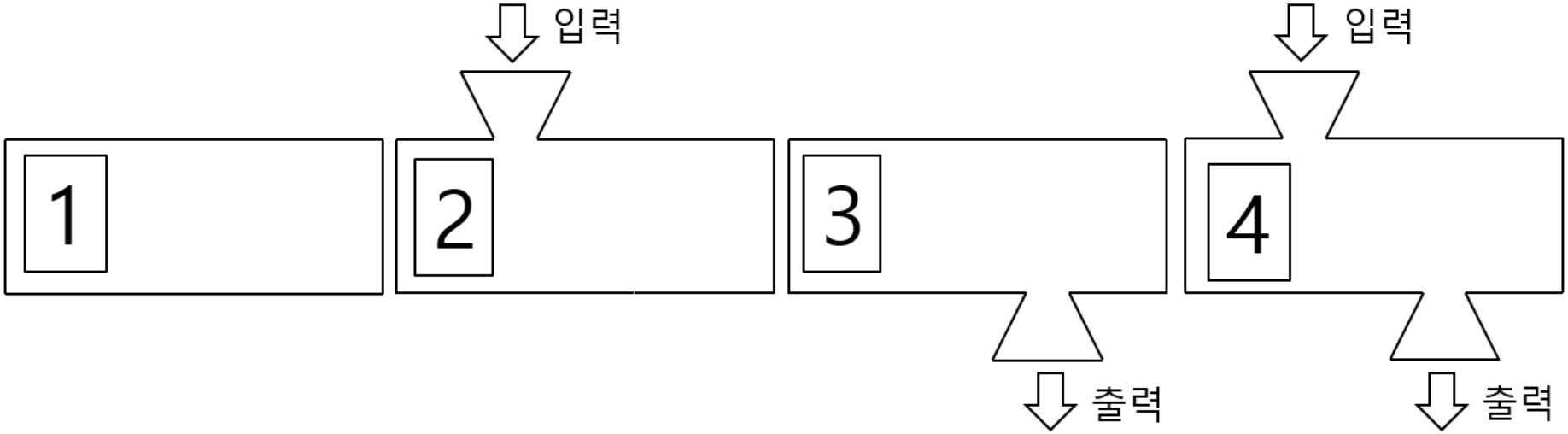
- 문제
  - 함수에 전달된 정수값이 100보다 작으면 true를 아니면 false를 반환하는 함수를 구현
- 요구사항
  - 함수를 구현하고 해당 함수를 사용하는 main() 함수를 같은 클래스에 함께 구현

실습 문제 3: 입력된 숫자의 범위에 따라 다른 일 하기 (반환 값 없이 리턴문 사용)

- 문제
  - 함수가 호출될 때마다 전달된 정수의 범위에 따라 다음에서 설명한 것처럼 동작하는 프로그램을 작성
    - 값이 100보다 크면, 멤버 변수에 있던 값에 더해서 합을 구하고 다시 멤버 변수에 저장한 후에 화면에 결과 출력
    - 값이 50보다 크고 100보다 작거나 같으면, 현재 멤버 변수에 저장되어 있는 값을 화면에 출력
    - 값이 50보다 작거나 같을 때에는 아무것도 하지 않고 함수를 종료
- 요구사항
  - 합계를 저장하는 멤버 변수 값은 0으로 초기화

함수의 종류와 사용법

- 자바 언어의 함수는 네 가지 종류가 있음
  - 입력과 출력이 모두 없는 함수
  - 입력은 있고 출력이 없는 함수
  - 입력은 없고 출력만 있는 함수
  - 입력과 출력이 모두 존재하는 함수



종류별 함수의 예

- 입력과 출력이 모두 없는 함수

```
void printHello() {
    System.out.println("hello");
}
```

- 입력은 있고 출력이 없는 함수

```
void printString(String str) {
    System.out.println(str);
}
```

종류별 함수의 예

- 입력은 없고 출력만 있는 함수

```
double getAverage() {
    double average = (2 + 3 + 5 + 4) / 4.0;
    return average;
}

boolean printHello2() {
    System.out.println("안녕하세요");
    return true;
}
```

종류별 함수의 예

- 입력과 출력이 모두 존재하는 함수

```
double getAverage(double n1, double n2,
double n3, double n4) {
    return (n1 + n2 + n3 + n4) / 4.0;
}

boolean printString2(String str) {
    if (str != null) {
      System.out.println(str);
        return true;
    }
    else {
        return false;
    }
}
```

사용 예

- 사용법

```
// 1 입출력이 모두 없음
함수_이름();

// 2 입력 있고 출력 없음
함수_이름(입력값);

// 3 입력 없고 출력 있음
변수 = 함수_이름();
함수_이름();

// 4 입출력 모두 있음
변수 = 함수_이름(입력값);
함수_이름(입력값);
```

사용 예

```
printHello(); // 1 입출력 모두 없음
printHello2(); // 3 입력 없고 출력 있음. 출력 무시
printString("hello"); // 2 입력 있고 출력 없음
printString2("hello"); // 4 입출력 있음. 출력 무시

// 3 입력 없고 출력 있음
double average = getAverage();

// 4 입출력이 모두 있음
double average2 = getAverage(1.3, 2.5, 3.7, 2.4);
// 3 입력 없고 출력 있음
boolean result = printString();
// 4 입력과 출력이 모두 있음
boolean result = printString2("hello");
```

함수를 이용해서 중복 코드를 줄이는 예

- 줄이기 전 원본 코드

```
 1    int[] arr1 = { 1, 5, 2, 9 };
 2    int sum = 0;
 3    for (int i = 0; i < arr1.length; i++) {
 4        sum += arr1[i];
 5    }
 6    System.out.println(sum);
 7
 8    int[] arr2 = { 1, 2, 3, 4, 5, 6 };
 9    sum = 0
10    for (int i = 0; i < arr2.length; i++) {
11        sum += arr2[i];
12    }
13    System.out.println(sum);
```

함수를 이용해서 중복 코드를 줄이는 예

- 비슷한 코드가 사용되는 부분
  - 입력 데이터를 정의하는 코드
  - 입력 데이터의 합계를 계산하는 코드
  - 결과를 화면에 출력하는 코드
- sumOfArray()를 만들어 중복을 제거

- sumOfArray함수 버전 1

```
void sumOfArray(int[] arr1) {
    int sum = 0;
    for (int i = 0; i < arr1.length; i++)
        sum += arr1[i];
    System.out.println(sum);
}
```

함수를 이용해서 중복 코드를 줄이는 예

```
jshell> int[] arr1 = { 1, 5, 2, 9 };

jshell> sumOfArray(arr1);
17

jshell> int[] arr2 = { 1, 2, 3, 4, 5, 6 };

jshell> sumOfArray(arr2);
21
```

함수를 이용해서 중복 코드를 줄이는 예

- sumOfArray 함수 버전 2
  - 버전 1은 두 가지 일을 하는 함수가 됨
    - 합계를 구하는 일
    - 합계를 화면에 출력하는 일
  - 합계만 구하는 부분만 따로 함수로 만들기
- sumOfArray 함수 버전 2

```
int sumOfArray(int[] arr1) {
    int sum = 0;
    for (int i = 0; i < arr1.length; i++)
        sum += arr1[i];
    return sum;
}
```

값과 객체를 함수에 전달하기

- 기본형과 참조형을 매개변수로 전달하는 것은 사용법이 같음
- 참조형인 객체를 전달해보기

```
void tellUsYourName(String name) {
    System.out.println("Hi, my name is " + name);
}
```

- 사용 예

```
jshell> String myName = "js";

jshell> tellUsYourName(myName);
Hi, my name is js

 jshell> tellUsYourName("Cho");
 Hi, my name is Cho
```

값과 객체를 함수에 전달하기

- Person 클래스를 만들고 객체를 함수에 전달해서 이름을 출력

```
class Person {
    String name;
    Person(String n) {
        name = n;
    }
    String getName() {
        return name;
    }
    void setName(String anotherName) {
        name = anotherName;
    }
}
```

값과 객체를 함수에 전달하기

- Person 객체를 인자로 전달 받는 함수

```
void tellUsPersonName(Person p) {
    System.out.println("Hi, my name is "
                       + p.getName());
}
```

- JShell에서 테스트

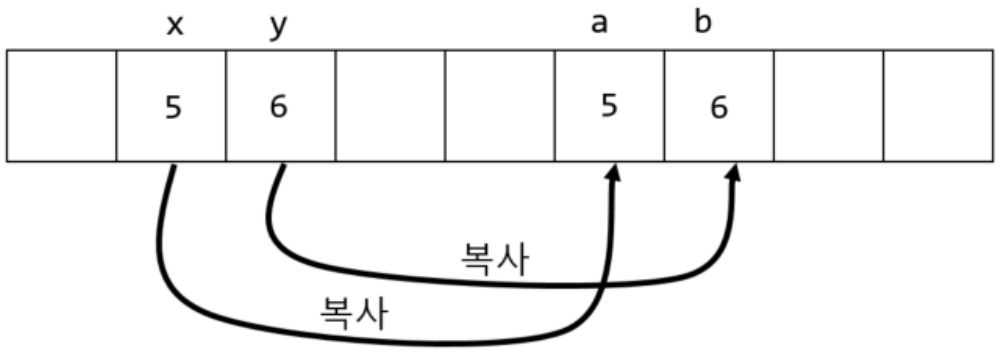
```
jshell> Person p1 = new Person("js");

jshell> tellUsPersonName(p1);
Hi, my name is js
```

매개변수 전달 방법

- 기본형이나 참조형 모두 매개변수로 전달될 때 값 전달 방법이 사용됨
- 값 전달(call by value 또는 pass by value) 방법
  - 매개변수에 변수값을 전달할 때 복사본을 생성
- 기본형 전달

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

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

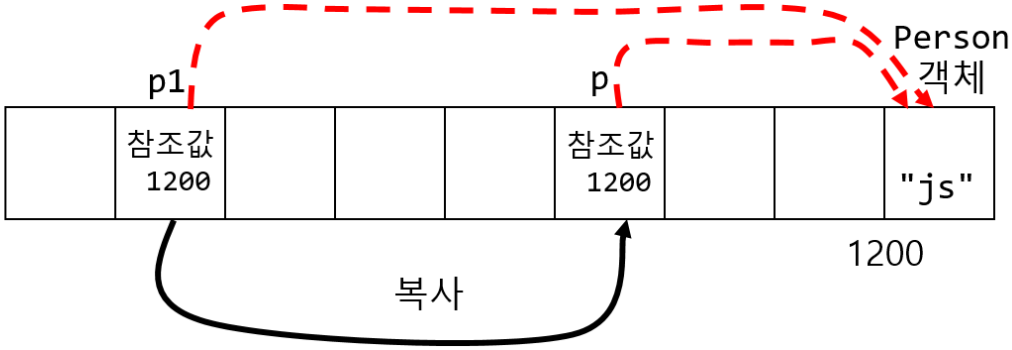


매개변수 전달 방법

참조형 전달

```
void tellUsYourName(Person p) {
    System.out.println("Hi my name is " + p.getName());
}
```

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



매개변수 변경

- 기본형이나 참조형 모두 매개변수 값을 바꾸는 것은 함수 내부에서만 영향을 미침
  - 함수 외부에는 영향을 주지 않음
- 참조형을 전달했을 때, 매개변수를 이용해서 객체의 멤버 변수를 변경하면 이는 함수 외부에 영향을 줌

매개변수 변경

- 기본형 전달

```
jshell> void changeVar(int a) {
    a = 123;
    System.out.printf("changeVar: a = %d\n", a);
}

jshell> int a = 1;

jshell> changeVar(a);
changeVar: a = 123;

jshell> System.out.printf("After calling
changeVar: a = %d\n", a);
After calling changeVar: a = 1
```

매개변수 변경

- 참조형 전달

```
jshell> void changeVar(Integer a) {
    System.out.println("*** HashCode before: "
                    + System.identityHashCode(a));
    a = 123;
    System.out.println("*** HashCode after: "
                    + System.identityHashCode(a));
}

jshell> Integer i = 1;

jshell> System.out.println("HashCode before calling
changeVar: " + System.identityHashCode(i));
HashCode before calling changeVar: 1555690610
```

매개변수 변경

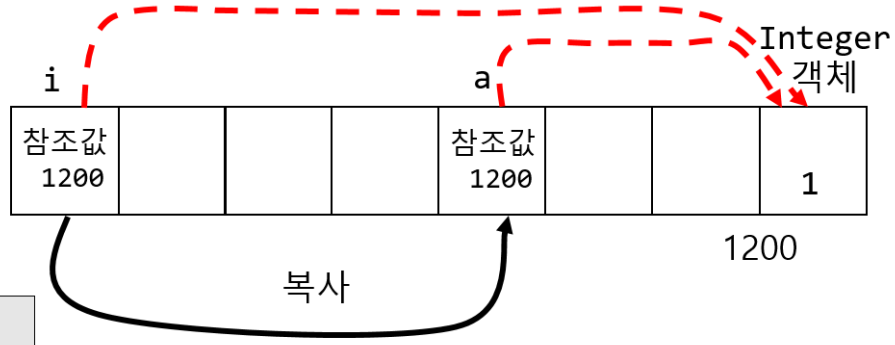
- 참조형 전달

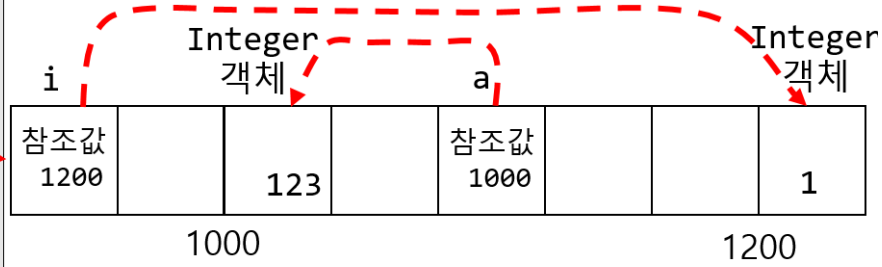
```
jshell> changeVar(i);
*** HashCode before: 1555690610
*** HashCode after: 1164371389

jshell> System.out.println("HashCode after calling
changeVar: " + System.identityHashCode(i));
HashCode after calling changeVar: 1555690610

jshell> System.out.printf("After calling changeVar:
i = %d\n", i);
After calling changeVar: i = 1
```

매개변수 변경

```
Integer i = 1;
changeVar(i);
```

```
void changeVar(Integer a) {
    System.out.println(
        "*** HashCode before: " +
        System.identityHashCode(a));
    a = 123;
    System.out.println(
        "*** HashCode after: " +
        System.identityHashCode(a));
}
```





매개변수 변경

- 참조값을 전달했을 때 매개변수가 멤버변수 값을 변경하면 외부에 영향을 미침

```
jshell> void changeName(Person p) {
    p.setName("local js"); // 객체의 속성 값을 변경
}

jshell> Person p1 = new Person("js");

jshell> System.out.printf("Before calling
changeName: %s\n", p1.getName());
Before calling changeName: js

jshell> changeName(p1);

jshell> System.out.printf("After calling
changeName: %s\n", p1.getName());
After calling changeName: local js
```

배열 전달

- 배열의 요소를 변경하는 함수

```
void changeArr(Integer[] n, int index, int n) {
    n[index] = num; // index 요소를 입력 값으로 변경
}
```

- JShell에서 테스트

```
jshell> Integer[] x = { 1, 2, 3, 4, 5 };

jshell> changeArr(x, 0, 1111);

jshell> System.out.printf("x[0] = %d\n", x[0]);
x[0] = 1111

jshell> System.out.printf("x[1] = %d\n", x[1]);
x[1] = 2
```

배열 전달

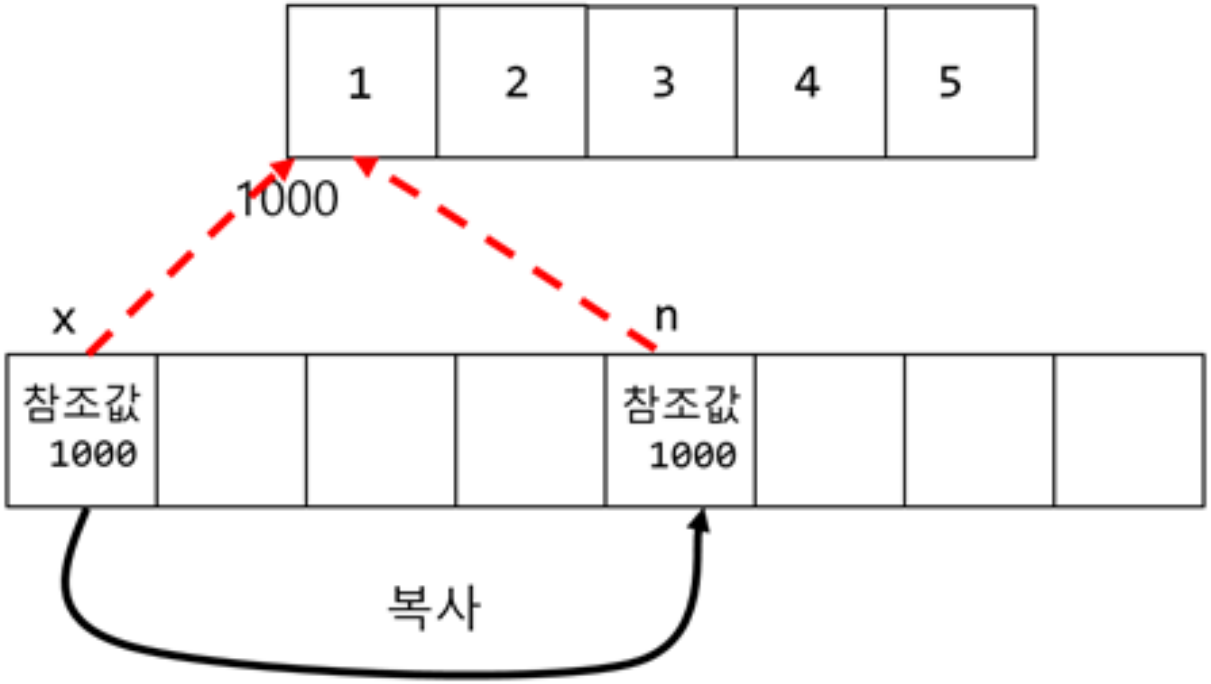
- 배열을 기본형으로 바꿔도 마찬가지
- 함수

```
void changeArr(int[] n, int index, int num) {
    n[index] = num;
}
```

- JShell에서 테스트

```
jshell> int[] x = { 1, 2, 3, 4, 5 };

jshell> changeArr(x, 0, 1111);

jshell> System.out.printf("x[0] = %d\n", x[0]);
x[0] = 1111

jshell> System.out.printf("x[1] = %d\n", x[1]);
x[1] = 2
```

배열 전달



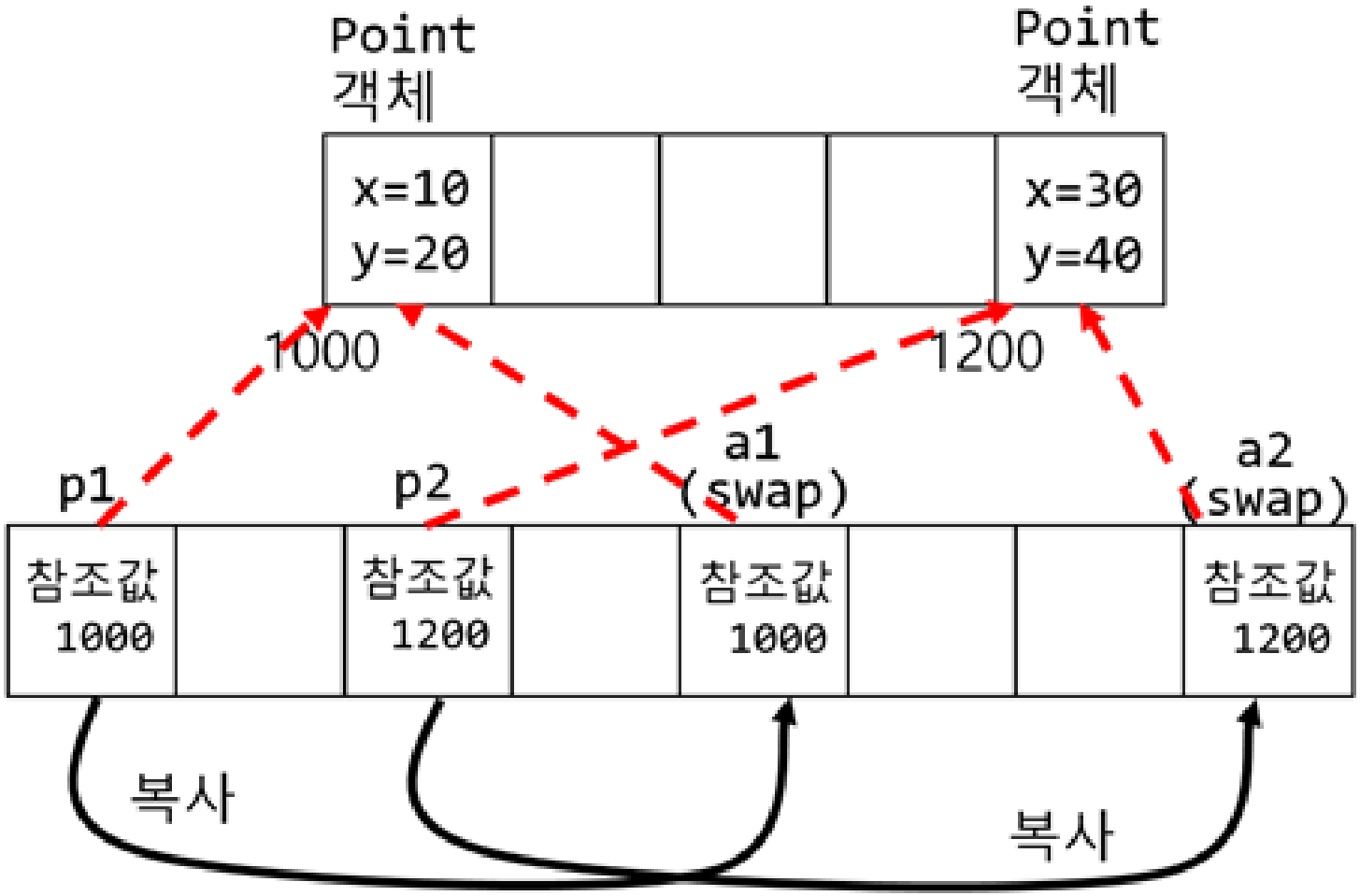
swap() 함수 구현이 안됨

- 자바에서는 매개변수에 전달되는 두 값을 서로 바꾸는 swap() 함수 구현이 불가능
- java.awt모듈의 Point 클래스 객체를 두 개 바꿔보기로 함
- 두 개 매개변수의 값을 교환하는 swap() 함수 구현

```
void swap(Point a1, Point a2) {
    Point temp = a1;
    a1 = a2;
    a2 = temp;
    System.out.printf("IN SWAP: a1 %s, a2 %s",
                      a1.toString(), a2.toString());
}
```

```
jshell> import java.awt.Point;

jshell> Point p1 = new Point(10, 20);

jshell> Point p2 = new Point(30, 40);

jshell> System.out.printf("BEFORE swapping: p1:
%s p2: %s\n", p1.toString(), p2.toString());
BEFORE swapping: p1: java.awt.Point[x=10,y=20]
p2: java.awt.Point[x=30,y=40]

jshell> swap(p1, p2);
IN SWAP: a1 java.awt.Point[x=30,y=40], a2
java.awt.Point[x=30,y=40]

jshell> System.out.printf("AFTER swapping:
p1:%s p2:%s\n", p1.toString(), p2.toString());
AFTER swapping: p1: java.awt.Point[x=10,y=20]
p2: java.awt.Point[x=30,y=40]
```

swap() 함수 구현이 안됨



Point 객체 swap 해보기

- Point 객체의 내부 값들을 바꿔 swap() 함수 구현

```
jshell> import java.awt.Point;

jshell> void swap(Point a1, Point a2) {
    Point temp1 = new Point(a1.x, a1.y);
    a1.x = a2.x;
    a1.y = a2.y;
    a2.x = temp1.x;
    a2.y = temp1.y;
}

jshell> Point p1 = new Point(10, 20);

jshell> Point p2 = new Point(30, 40);
```

Point 객체 swap 해보기

```
jshell> System.out.printf("BEFORE swapping: p1:
%s p2: %s\n", p1.toString(), p2.toString());
BEFORE swapping: p1: java.awt.Point[x=10,y=20]
p2: java.awt.Point[x=30,y=40]

jshell> swap(p1, p2);

jshell> System.out.printf("AFTER swapping: p1:%s
p2:%s\n", p1.toString(), p2.toString());
AFTER swapping: p1: java.awt.Point[x=30,y=40]
p2: java.awt.Point[x=10,y=20]
```

변수의 유효 범위

- 프로그래밍 언어에서는 단어(변수, 함수 이름, 클래스 이름)들은 코드에서 사용할 수 있는 영역이 정해져 있음 → 유효범위(scope)
- 변수나 함수 등을 유효 범위를 벗어나서 사용하려고 하면 컴파일 오류 발생
- 유효범위를 네 가지 영역으로 분류
  - 클래스
  - 메소드
  - for반복문
  - 코드 블록

변수의 유효 범위

- 클래스 유효범위
  - 멤버 변수 또는 멤버 함수를 사용할 수 있는 영역
  - 멤버 변수는 같은 클래스내의 모든 멤버 함수에서 사용 가능
  - 여러 메소드에서 공통적으로 변수를 사용해야 할 때 멤버 변수로 정의
- 메소드 범위
  - 메소드 내에서만 사용할 수 있는 영역
  - 메소드 내부에서 임시로 사용하기 위해 정의하는 변수
  - 매개변수 또는 메소드의 코드 영역에서 생성되는 변수들이 메소드 범위에 포함됨

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(); // 사용 못함
    }
}
```

변수의 유효 범위

- for 반복문 범위
  - for문에서 생성되는 변수들은 해당 반복문 내부에서만 사용 가능

```
for (명령문1; 명령문2; 명령문3) {
    반복_코드;
}
```

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

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

변수의 유효 범위

- for문에서 변수를 여러 개 정의

```
// ForScope.java
class ForScope {
    public static void main(String[] args) {
        for (int i = 0, j = 0; i < 5; i++, j++) {
            System.out.printf("i = %d, j = %d\n",
                                i, j);
        }
        // System.out.printf("i = %d, j = %d\n",
                              i, j); // 오류 발생
    }
}
```

변수의 유효 범위

- 코드 블록 범위
  - 조건문, 반복문, 또는 임의의 코드 블록 내에서 변수를 정의하면, 그 변수의 유효 범위는 코드 블록 내부가 됨

```
// CodeBlockScope.java
class CodeBlockScope {
    public static void main(String[] args) {
        if (true) {
            int i = 3;
            int j = 4; // 이 영역을 벗어나면 사용 못함
            System.out.printf("조건문 i = %d\n", i);
        }
        for (int n = 0; n < 3; n++) {
            int i = 4;
            int k = 5; // 이 영역을 벗어나면 사용 못함
            System.out.printf("반복문 i = %d\n", i);
        }
```

변수의 유효 범위

```
        {  // 새로운 코드 블록 시작
              int i = 5;
              System.out.printf("새로운 코드 블록 i =
%d\n", i);
         }  // 새로운 코드 블록 끝
         //System.out.printf("i = %d, j = %d\n", i,
j); // 오류 발생
    }
}
```

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

변수의 유효 범위

```
jshell> void codeBlockScope() {
    int i = 3;
    if (true) {
        int i = 4;
        int j = 4; // 이 영역을 벗어나면 사용 못함
        System.out.printf("조건문 i = %d\n", i);
    }
}
|  Error:
|  variable i is already defined in method
codeBlockScope()
|          int i = 4;
|          ^--------^
```

유효 범위의 우선 순위

- 유효범위가 겹치는 영역에서 같은 이름의 변수가 두 개 이상 정의되었을 때 어떤 변수가 사용될 것인지 결정하는 것은 유효 범위의 우선 순위를 따름
- 예를 들어 클래스의 멤버 변수와 똑같은 이름의 변수가 메소드 내부에서 다시 정의된다면?

```
// TestScope.java
class Scope {
    int num1 = 3;
    int num2 = 4;

    void printNumbers() {
        System.out.printf("num1 = %d, num2 = %d\n",
                          num1, num2);
    }
```

```
    void printNumbers2() {
        int num1 = 5;
        System.out.printf("num1 = %d, num2 = %d\n",
                          num1, num2);
    }

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

유효 범위의 우선 순위

클래스 유효범위
num1, num2

메소드
유효범위
num1

메소드
유효범위
num1,
num2

```
class Scope {
    int num1 = 3;
    int num2 = 4;

    void printNumbers() {
        System.out.println("num1 = %d, num2 = %d\n", num1, num2);
    }

    void printNumbers2() {
        int num1 = 5;
        System.out.println("num1 = %d, num2 = %d\n", num1, num2);
    }

    void printNumbers3(int num1) {
        int num2 = 5;
        System.out.println("num1 = %d, num2 = %d\n", num1, num2);
    }
}
```

유효 범위의 우선 순위

- 클래스 멤버 변수와 메소드 내에 있는 코드 블록 간에 중복되는 변수가 있다면?
  - 안쪽 영역에 있는 변수들이 우선해서 사용됨

```
// ClassBlockScopePriority.java
class ClassBlockScopePriority {
    int j = 5;
    int i = 4;

    void printNumbers(int i) {
        if (true) {
            int k = 5;
            int j = 4;
            System.out.printf("조건문 i = %d, j =
%d, k = %d\n", i, j, k);
        }
```

유효 범위의 우선 순위

```
        System.out.printf("메소드 i = %d, j = %d\n", i, j);
    }

    public static void main(String[] args) {
        ClassBlockScopePriority classBlock = new ClassBlockScopePriority();
        classBlock.printNumbers(3);
    }
}
```

- 유효범위는 안쪽에서 바깥쪽으로 확장

메소드 오버로딩(overloading)

- 오버로딩은 동일한 클래스 내에서 **메소드의 이름은 같게, 매개변수는 다르게** 구현
  - 함수의 이름과 매개변수가 같고, 반환 자료형만 다른 경우 오버로딩이 될 수 없음
- 오버로딩이 지원되면 다른 입력에 대해 비슷한 일을 하는 함수들을 같은 이름으로 구현할 수 있어 프로그래머의 편의성과 코드의 가독성을 높일 수 있음

```
// Number.java
class Number {
    int add(int a, int b) { return a + b;}

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

메소드 오버로딩(overloading)

```
// TestNumber.java
class TestNumber {
    public static void main(String[] args) {
        Number number = new Number();
        int sumInt = number.add(2, 3);
        System.out.printf("2 + 3 = %d\n", sumInt);
        float sumFloat = number.add(2.0f, 3.0f);
        System.out.printf("2.0 + 3.0 = %f\n",
                          sumFloat);
    }
}
```

메소드 오버로딩(overloading)

- 함수가 오버로딩 되었을 때 **전달되는 인자를 취할 수 있는 시그니처를 가지는 함수가 없다면 컴파일 오류 발생**

```
// TestNumber2.java
class TestNumber2 {
  public static void main(String[] args) {
    Number number = new Number();
    int sumInt = number.add(2, 3);
    System.out.printf("2 + 3 = %d\n", sumInt);
    float sumFloat = number.add(2.0f, 3.0f);
    System.out.printf("2 + 3 = %f\n", sumFloat);
    double sumDouble = number.add(2.0, 3.0);
    System.out.printf("2 + 3 = %f\n", sumFloat);
  }
}
```

메소드 오버로딩(overloading)

컴파일 결과

```
C:\Code\java\05>javac TestNumber2.java
TestNumber2.java:9: error: no suitable method found for  add(double,double)
     double sumDouble = number.add(2.0, 3.0);
                                ^
 method Number.add(int,int) is not applicable
   (argument mismatch; possible lossy conversion from double to int)
 method Number.add(float,float) is not applicable
   (argument mismatch; possible lossy conversion from double to float)
1 error
```

오버로딩과 자동 형 변환

- 오버로딩 된 함수들을 사용할 때에도 인자와 매개변수 사이에 자동 형 변환 발생 가능
- 오버로딩 된 함수를 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

오버로딩과 자동 형 변환

```
// TestNumber3.java
class TestNumber3 {
    public static void main(String[] args) {
        Number number = new Number();
        byte b1 = 2;
        byte b2 = 3;
        int sumInt = number.add(b1, b2);
        System.out.printf("2 + 3 = %d\n", sumInt);

        short s1 = 2000;
        int n2 = 3000;
        sumInt = number.add(s1, n2);
        System.out.printf("2 + 3 = %d\n", sumInt);
    }
}
```