

제8장

객체와 클래스 - 2



1. 변수의 범위(scope)

- 변수의 선언 위치에 따라 변수의 종류(멤버변수, 지역변수)와 범위(scope)이 결정된다.

```
public class Student {  
    //인스턴스 멤버변수(필드)  
    String name;  
    int age;  
  
    //정적(static)멤버 변수  
    static int hakbun;  
  
    public void method() {  
        //지역변수는 반드시 초기화해야 한다.  
        int temp=0;  
        System.out.println(temp);  
    }  
}
```

변수의 종류	선언 위치	생성시기
클래스, 정적 (static) 변수	클래스 영역	클래스가 메모리에 올라갈 때 (인스턴스 생성하지 않아도 사용 가능)
인스턴스 변수		무조건, new연산자로 인스턴스 생성시
지역 변수	메서드 영역	메서드가 호출될 때, 생겨나고 메서드 종료 시 소멸됨.(초기화가 반드시 필요함,static변수 사용 못함)

2. 변수의 종류

♠ 인스턴스 변수(instance variable)

- new연산자로 생성한 인스턴스는 무조건 독립적인 저장공간을 지님.
- 각각의 인스턴스에 다른 값 저장 가능함.
- **인스턴스 생성 후, '참조변수명.인스턴스변수명' 으로 접근함.**
- 인스턴스 생성 후, 참조변수가 참조를 없애버리면 자동 쓰레기 인스턴스가 됨 (Garbage Collector에 의해 자동 소멸됨.)

♠ 클래스(정적) 변수(class(static) variable)

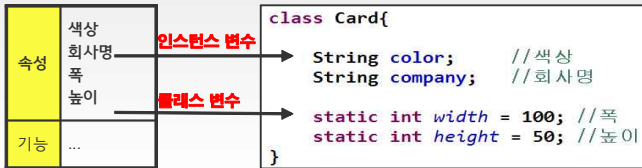
- 같은 클래스의 모든 인스턴스들이 공유하는 변수(공유변수)
- **인스턴스 생성없이 '클래스명.클래스변수명' 으로 접근함.**
- 클래스가 로딩될 때, 생성되고 프로그램이 종료될 때 소멸됨(public이 붙으면 전역변수임)

♠ 지역 변수(Local Variable)

- **메서드 내에 선언되며, 메서드의 종료와 함께 소멸됨.(초기화 반드시 필요함)**
- 메서드 { } 내에 선언된 지역변수는 블록을 벗어나면 소멸됨.

3. 클래스(정적) 변수와 인스턴스 변수

앞에서 말했듯이, **인스턴스 변수**는 생성될 때마다 독립적인 공간이 생겨 각기 다른 값을 유지할 수 있지만, **클래스 변수**는 모든 인스턴스가 하나의 저장공간을 공유하므로 **항상 똑같은 값을 갖는 것**에 주목하자.

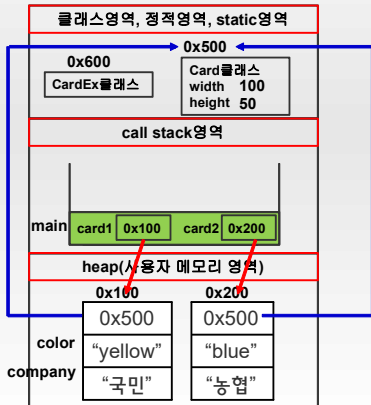


4. 클래스 변수와 인스턴스 변수의 메모리 형태

```
class Card {  
    String color;    //색상  
    String company;  //회사명  
  
    static int width = 100;    //폭  
    static int height = 50;    //높이  
}
```

```
public class CardEx {  
    public static void main(String[] args) {  
  
        Card card1 = new Card();  
        card1.color = "yellow";  
        card1.company = "국민";  
  
        System.out.println("card1은 " + card1.color + "색이며, 회사는 " +  
            card1.company + "이며, 크기는 (" + card1.width +  
            "," + card1.height + ")");  
        //static 변수 접근시 클래스명, 정적변수명으로 접근해야 올바른 방법이다.  
        card1.width = 50;  
        card1.height = 80;  
  
        Card card2 = new Card();  
        card2.color = "blue";  
        card2.company = "농협";  
  
        System.out.println("card2은 " + card2.color + "색이며, 회사는 " +  
            card2.company + "이며, 크기는 (" + Card.width +  
            "," + Card.height + ")");  
    }  
}
```

**클래스(정적)변수 변경 시는
반드시 클래스명.static변수명으로
접근하여 변경하여야 올바른 방법임**



5. 메서드(method)의 정의와 작성지침

- 메서드는 앞선 강의에서 살펴본 바 있다.
- 중요한 개념이므로 다시 한번 상기시키기 위해 포함하였다.
- 메서드는 작업(기능)을 수행하기 위한 명령문들의 집합이라고 보면 된다.
- 또한, 메서드는 선언부와 구현부로 나뉜다.(선언부가 구현부보다 훨씬 중요하다.)
(호출할 때, 필요하 건 선언부이기 때문이다.)
- 매개변수(인자값, Arguments, 파라미터)로 값을 받아서 그 결과를 리턴값(반환값)으로 돌려준다.(단, 매개변수가 없을 수 있으며, 리턴타입이 void라면 역시 결과를 돌려주지 않아도 문제없다.)
- 장점으로서는 반복적인 코드를 줄이고, 코드의 관리가 용이하다.
- 반복적으로 수행된다고 판단되는 여러 문장이 있다면, 메서드로 작성하는 것이 좋다.
- 아울러 하나의 메서드는 한가지 기능만 수행하도록 작성하는 편이 좋다.
 - * 코드 중복 제거, 누가 봐도 알아볼 수 있는 프로그램이 되도록 노력, 재 사용성 향상.
(프로그래머의 기본 자질임)

6. 메서드 구현 코드

- 메서드를 정의하는 방법(클래스 영역에만 정의할 수 있음)

```
public int add(int a, int b) { //선언부
                                // {} <-- 구현부(정의부)
    int result = a + b;
    //return의 값은 선언부의 데이터 타입과 반드시 일치
    return result;
}

//return값이 없는 경우, void타입을 사용한다.
public void power() {
    this.power = !power;
    //return값이 없을 경우, return만 써줘도 되고, 생략해도 무방하다.
    return;
}
```

7. return문(웬만하면 1개로 최소화 하자)

- 메서드의 종료가 정상적으로 이루어지는 경우
 - 메서드의 블록 { } 끝에 도달했을 때
 - 메서드의 블록 { } 수행 중 return문을 실행했을 때

- **return문**

- **현재 실행 중인 메서드를 즉시 종료하고, 호출한 곳으로 되돌아간다.**

리턴값이 없는 경우 – return문만 적어주면 된다.

```
//return값이 없을 경우, return만 써줘도 되고, 생략해도 무방하다.  
return;
```

리턴값이 있는 경우 – return문 뒤에 리턴값을 반드시 지정 해줘야 한다.

```
//return의 값은 선언부의 데이터 타입과 반드시 일치  
return result;
```


8. 메서드 호출 방법

● 메서드의 호출 방법

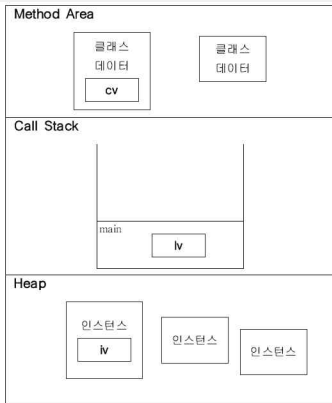
- 참조변수.메서드명(); //메서드에 매개변수가 없을 경우
- 참조변수.메서드명(값1, 값2, ...); //메서드에 매개변수가 있을 경우

```
Calculator cal = new Calculator();  
//divide메서드의 리턴값은 double타입이므로 double형으로  
//type이 일치되도록 반드시 받아주어야 한다.  
double result = cal.divide(50, 100);
```

```
class Calculator {  
    public double divide(int x, int y) {  
        double result = (double)x / y;  
        return result;  
        //아래와 같이 한줄로 줄일 수 있다.  
        //return (double)x / y;  
    }  
}
```

- 같은 클래스에서 static으로 선언된 메서드는 인스턴스 메서드를 호출할 수가 없다. (반대는 허용)
이유는 static 메서드는 클래스가 생성될 시 메모리의 클래스 영역에 곧바로 로딩되지만, 인스턴스 메서드는 인스턴스를 생성해야만 인스턴스 메서드를 사용 가능하기 때문이다.
인스턴스 메서드는 언제 생성될지 누가 알 수가 있는가?

9. JVM의 메모리 구조

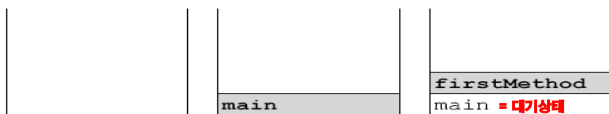


- ▶ 메서드영역(Method Area), 클래스 영역, static 영역
 - 클래스 정보와 클래스 변수가 저장되는 곳
 - **static이 붙은 변수나 메서드가 저장됨.**
- ▶ 호출스택(Call Stack) - **후입선출 개념**
 - 메서드의 작업공간. 메서드가 호출되면 메서드 수행에 필요한 메모리공간을 할당받고 메서드가 종료되면 사용하던 메모리를 반환한다.
- ▶ 힙(Heap)
 - 인스턴스가 생성되는 공간. **new연산자에 의해서 생성되는 배열과 객체는 모두 여기에 생성된다.**

10. Call Stack(호출 스택)

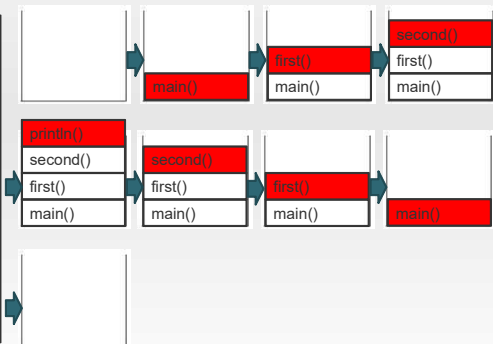
■ 호출 스택의 특징

- 프로그램 시작점인 `main()`가 시작되면서, 프로그램은 수많은 인스턴스도 생성하고 메서드도 수 없이 호출된다. 이 때, **생성자 호출이나 메서드 호출 시에 필요한 메모리 공간을 스택 공간에 할당한다.**
- 물론, 메서드의 수행이 끝이 나면 할당된 메모리 공간이 반환이 된다.
- **호출 스택에서 맨 위에 있는 메서드가 현재 실행 중이며, 아래에 있는 메서드가 바로 위에 메서드를 호출한 메서드이며 대기 상태로 있다.**



11. Call Stack(호출 스택)예제와 메모리 상태

```
public static void main(String[] args) {  
    //static은 static만 부를수 있다.  
    CallStackExample.first();  
}  
  
public static void first() {  
    second();  
}  
  
public static void second() {  
    System.out.println("second()");  
}
```



12. 기본형 매개변수와 참조형 매개변수

■ 기본형 매개변수 – read only

- 흔히, C언어에서 **call by value**라고 불리며, 메서드 호출 시에 매개변수로 넘겨주는 값은 메서드의 지역변수로 복사가 이루어지는 형태
(수정을 해도 호출한 메서드의 값에는 전혀 영향을 미치지 않는다.)

■ 참조형 매개변수 – read & write

- C언어에서 **call by reference**라고 불리며, 메서드 호출 시에 매개변수로 넘겨주는 값은 주소값을 넘겨주는 형태
(호출된 메서드에서 수정을 하면, 호출한 메서드의 값에도 직접적으로 영향을 미친다.)

13. 기본형 매개변수를 넘길 때

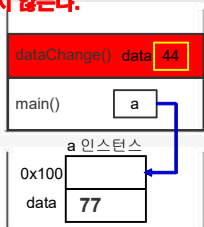
```
class A {
    int data;
}

public class PrimitiveParam {

    public static void main(String[] args) {

        A a = new A();
        a.data = 77;
        System.out.println("main()메서드 내 data 값 : " + a.data);
        //아래 메서드 호출은 실제 값을 넘기고 있다. (Call by value)
        //즉, dataChange(int data)의 매개변수로 복사가 되어 지고 있다.
        PrimitiveParam.dataChange(a.data);
        System.out.println("dataChange()호출 후");
        System.out.println("main()메서드 내 data 값 : " + a.data);
    }
    //static은 static만 호출할 수 있다.
    public static void dataChange(int data) {
        data = 44;
        System.out.println("dataChange()메서드 내 data값 : " + data);
    }
}
```

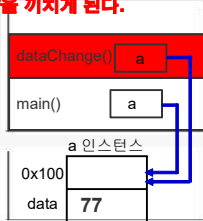
call by value는 값에 의한 복사의 개념으로, dataChange(a.data)를 호출 시, 매개변수가 호출한 메서드의 매개변수로 복사가 되어진다. 하여, 메인메서드가 스택에 먼저 쌓이고, dataChange(int data)가 위에 쌓이는 형태가 되는 것이다. 결론은 값 복사는 호출한 곳에는 전혀 영향을 끼치지 않는다.



14. 참조형 매개변수를 넘길 때

```
class A {  
    int data;  
}  
public class PrimitiveParam {  
    public static void main(String[] args) {  
        A a = new A();  
        a.data = 77;  
        System.out.println("main()메서드 내 data 값 : " + a.data);  
        //아래 메서드 호출은 인스턴스의 주소를 넘기고 있다. (Call by reference)  
        //즉, dataChange(a)의 매개변수로 주소값이 넘어가고 있다.  
        PrimitiveParam.dataChange(a);  
        System.out.println("dataChange()호출 후");  
        System.out.println("main()메서드 내 data 값 : " + a.data);  
    }  
    //static은 static만 호출할 수 있다.  
    public static void dataChange(A a) {  
        a.data = 44;  
        System.out.println("dataChange()메서드 내 data값 : " + a.data);  
    }  
}
```

call by reference는 주소에 의한 호출의 개념으로, dataChange(a)를 호출 시, 매개변수가 호출한 메서드의 매개변수로 주소가 넘어가게 되어 주소공유가 일어난다. 하여, 메인메서드가 스택에 먼저 쌓이고, dataChange(A a)가 위에 쌓이는 형태가 되는 것이다. 결론은 주소에 의한 호출은 호출한 곳에 직접적으로 영향을 끼치게 된다.



감사합니다.

