


클래스와 인스턴스

Object-oriented Programming

- 객체(Object)
 - 사물 / 물건, 대상, 개념
 - 객체지향 프로그래밍 : 객체 중심의 프로그래밍
- “나는 과일장수에게 두 개의 사과를 구매했다!”
- 객체지향 프로그래밍에서는 나, 과일장수, 사과라는 객체를 등장 시켜서 두 개의 사과 구매라는 행위를 실체화한다.
- 객체를 이루는 것은 데이터와 기능.

클래스(class) - 틀

```
1. class FruitSeller {  
2.     final int APPLE_PRICE = 1000;  
3.     int numOfApple = 20;  
4.     int myMoney = 0;  
5.  
6.     public int saleApple(int money) {  
7.         int num = money/1000;  
8.         numOfApple -= num;  
9.         myMoney += money;  
10.        return num;  
11.    }  
12.    public void showSaleResult() {  
13.        System.out.println("남은 사과 : " + numOfApple);  
14.        System.out.println("판매 수익 : " + myMoney);  
15.    }  
16. }
```



변수 선언



메소드 정의



메소드 정의

클래스(class) - 틀

```
1. class FruitBuyer {  
2.     int myMoney = 10000;  
3.     int numOfApple = 0;  
4.  
5.     public void buyApple(FruitSeller seller, int money) {  
6.         numOfApple += seller.saleApple(money);  
7.         myMoney -= money;  
8.     }  
9.     public void showBuyResult() {  
10.         System.out.println("현재 잔액 : " + myMoney);  
11.         System.out.println("사과 개수 : " + numOfApple);  
12.     }  
13. }
```

클래스 기반의 객체 생성

- 방법1

FruitSeller seller; // 참조변수의 선언

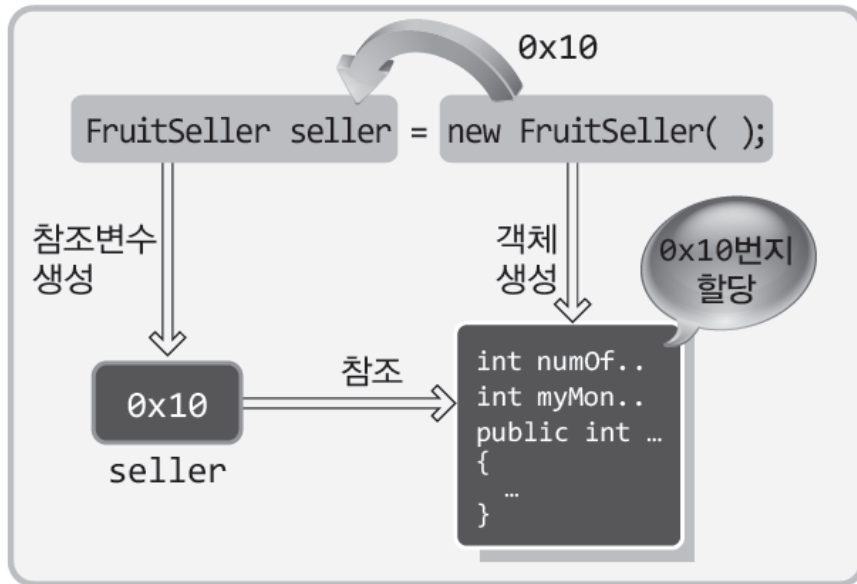
seller = new FruitSeller(); // 인스턴스의 생성

- 방법2

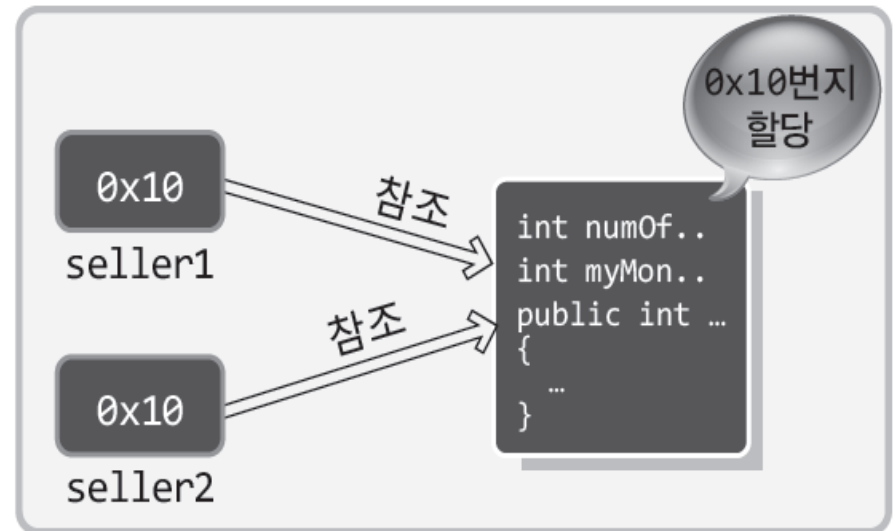
FruitSeller seller = new FruitSeller();

FruitBuyer buyer = new FruitBuyer();

객체 생성과 참조의 관계



```
FruitSeller seller1 = new FruitSeller();  
FruitSeller seller2 = seller1;
```



객체 접근 방법

- 변수 접근

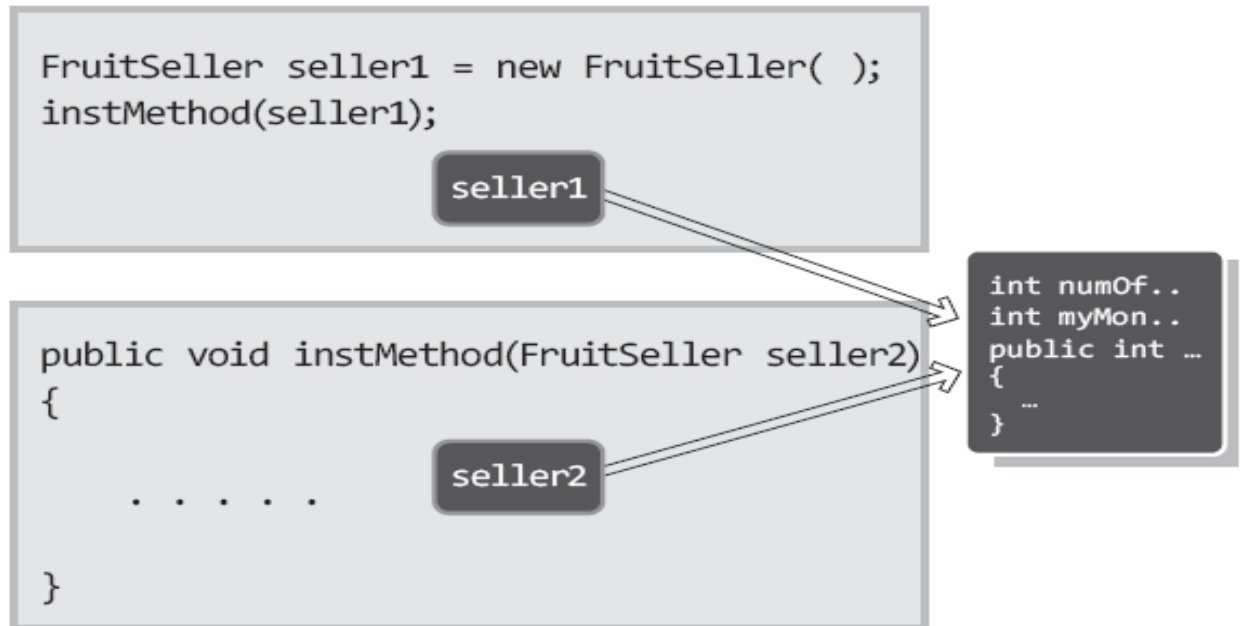
```
FruitSeller seller = new FruitSeller();  
seller.numOfApple = 20;
```

- 메소드 접근

```
FruitSeller seller = new FruitSeller();  
seller.saleApple(10);
```

참조변수와 메소드 관계

```
public void myMethod() {  
    FruitSeller seller1 = new FruitSeller();  
    instMethod(seller1);  
    ...  
}  
public void instMethod(FruitSeller seller2) {  
    ...  
}
```



Exam.

```
1. class Number {  
2.     int num = 0;  
3.     public void addNum(int n) {  
4.         num += n;  
5.     }  
6.     public int getNumber() {  
7.         return num;  
8.     }  
9. }
```

nInst = null; // 참조변수 nInst는 참조하는
//객체가 없다는 의미

```
10. class Exam6_1 {  
11.     public static void main(String[] args) {  
12.         Number nInst = new Number();  
13.         System.out.println("메소드 호출 전 : " + nInst.getNumber());  
14.         simpleMethod(nInst);  
15.         System.out.println("메소드 호출 후 : " + nInst.getNumber());  
16.     }  
17.     public static void simpleMethod(Number numb) {  
18.         numb.addNum(12);  
19.     }  
20. }
```

시뮬레이션

```
1. public class Exam6_2 {
2.     public static void main(String[] args) {
3.         FruitSeller seller = new FruitSeller();
4.         FruitBuyer buyer = new FruitBuyer();
5.
6.         buyer.buyApple(seller, 2000); // 메소드 호출 -> 메시지 전달
7.                                     // (Message Passing)
8.         System.out.println("과일 판매자의 현재 상황");
9.         seller.showSaleResult();
10.
11.        System.out.println("과일 구매자의 현재 상황");
12.        buyer.showBuyResult();
13.    }
14. }
```

생성자(Constructor)

- 클래스의 이름과 동일한 이름의 메소드.
- 인스턴스 생성시 딱 한 번 호출되는 메소드.
- 인스턴스 변수의 초기화를 목적으로 정의.
- 반환형이 선언되어 있지 않으면서, 반환하지 않는 메소드.

```
Number num = new Number( );
```

new Number

Number의 인스턴스 생성!

Number()

Number() 생성자 호출!

- 자바의 인스턴스 생성시에는 반드시 생성자가 호출.

```
1. class FruitSeller {
2.     int numOfApple;
3.     int myMoney;
4.     final int APPLE_PRICE;
5.     public FruitSeller(int money, int appleNum, int price) {
6.         myMoney = money;
7.         numOfApple = appleNum;
8.         APPLE_PRICE = price;
9.     }
10.    ...
11. }
12. class FruitBuyer {
13.     int myMoney;
14.     int numOfApple;
15.     public FruitBuyer(int money) {
16.         myMoney = money;
17.         numOfApple = 0;
18.     }
19.     ...
20. }
```

```
class Exam6_3 {
    public static void main(String[] args) {
        FruitSeller seller1 = new FruitSeller(0, 30, 1500);
        FruitSeller seller2 = new FruitSeller(0, 20, 1000);
        FruitBuyer buyer = new FruitBuyer(10000);
        .....
    }
}
```

디폴트 생성자(Default Constructor)

- 생성자를 정의하지 않았을 때에만 자동 삽입!

```
public FruitSeller()  
{  
    // 비어있다!!!  
}
```

정보은닉
/접근제어지시자
/캡슐화

정보은닉(Information Hiding)

```
1. class Exam8_3 {  
2.     public static void main(String[] args) {  
3.         FruitSeller seller = new FruitSeller(0, 30, 1500);  
4.         FruitBuyer buyer = new FruitBuyer(10000);  
5.  
6.         seller.myMoney += 500; // 돈 500원 내고!  
7.         buyer.myMoney -= 500;  
8.  
9.         seller.numOfApple -= 20;  
10.        buyer.numOfApple += 20; // 사과 스무 개 가져가는 꼴이네!  
11.  
12.        System.out.println("과일 판매자1의 현재 상황");  
13.        seller.showSaleResult();  
14.  
15.        System.out.println("과일 구매자의 현재 상황");  
16.        buyer.showBuyResult();  
17.    }  
18. }
```

정보 은닉

- 인스턴스 변수의 private
- class FruitSeller
 - {
 - `private` int numOfApple;
 - `private` int myMoney;
 - `private` final int APPLE_PRICE;
 -}
- class FruitBuyer
 - {
 - `private` int myMoney;
 - `private` int numOfApple;
 -}

접근제어 지시자 (Access Control Specifiers)

- public / protected / private와 같은 키워드.
- 접근의 허용 범위를 제한하는 용도로 사용.

```
class BBB
{
    public accessAAA(AAA inst)
    {
        inst.num=20;
        inst.setNum(20);
        System.out.println(inst.getNum());
    }
    . . . .
}
```

```
class AAA
{
    private int num;
    public void setNum(int n) { num=n; }
    public int getNum() { return num; }
    . . . .
}
```

inst.num=20; // num은 private 멤버이므로 컴파일 불가!

inst.setNum(20);

System.out.println(inst.getNum());

// setNum, getNum은 public이므로 호출 가능!

접근제어 지시자

- private – 클래스 내부(메소드)에서만 접근 가능.
- public – 어디서든 접근 가능(접근을 제한하지 않는다).
- default
 - 접근제어 지시자 선언을 하지 않은 경우.
 - 동일 패키지 내에서의 접근 허용.

BBB는 AAA와 동일패키지로 선언되었으므로 접근가능!

```
package orange;

class AAA    // package orange로 묶인다.
{
    int num; // default 선언!
    . . . .
}

class BBB    // package orange로 묶인다.
{
    public init(AAA a) { a.num=20; }
    . . . .
}
```

접근제어 지시자

- protected
 - 상속 관계에 놓여 있어도 접근을 허용.
 - default 선언으로 접근 가능한 영역 접근 가능.
 - 상속 관계에서도 접근 가능.

```
class AAA
{
    protected int num;
    . . . .
}

class BBB extends AAA // 상속
{
    protected int num;      // 상속된 인스턴스 변수
    public init(int n) { num=n; } // 상속된 변수 num의 접근!
    . . . .
}
```

접근제어 지시자의 관계

지시자	클래스 내부	동일 패키지	상속받은 클래스	이외의 영역
private	●	×	×	×
default	●	●	×	×
protected	●	●	●	×
public	●	●	●	●

- public > protected > default > private

default 클래스

- 동일한 패키지 내에 정의된 클래스에 의해서만 인스턴스 생성이 가능.

```
package apple;
class AAA    // default 클래스 선언
{
    . . . .
}

package peal;
class BBB    // default 클래스 선언
{
    public void make()
    {
        apple.AAA inst=new apple.AAA();
        . . . .    // 인스턴스 생성 불가! AAA와 BBB의 패키지가 다르므로!
    }
    . . . .
}
```

public 클래스

- 하나의 소스 파일에 하나의 클래스만 public으로 선언 가능.
- public 클래스 이름과 소스 파일 이름은 일치해야 한다.

[illegible]

생성자 / 클래스

```
public class AAA
{
    AAA(){...}
    . . . .
}
```

클래스는 public으로 선언되어서 파일을 대표하는 상황!
그럼에도 불구하고 생성자가 default로 선언되어서 동일
패키지 내에서만 인스턴스 생성을 허용하는 상황!

```
class BBB
{
    public BBB(){...}
    . . . .
}
```

생성자가 public임에도 클래스가 default로 선언
되어서 동일 패키지 내에서만 인스턴스 생성이
허용되는 상황!

default 생성자

- 디폴트 생성자의 접근제어 지시자는 클래스의 선언 형태에 따라서 결정.

```
public class AAA
{
    public AAA() {...} // public 클래스에 디폴트로 삽입되는 생성자
    . . . .
}
```

```
class BBB
{
    BBB() {...} // default 클래스에 디폴트로 삽입되는 생성자
    . . . .
}
```


public 선언 클래스

```
public class Calculator
{
    private Adder adder;
    private Subtractor subtractor;

    public Calculator()
    {
        adder = new Adder();
        subtractor = new Subtractor();
    }

    public int addTwoNumber(int num1, int num2)
    {
        return adder.addTwoNumber(num1, num2);
    }

    public int subTwoNumber(int num1, int num2)
    {
        return subtractor.subTwoNumber(num1, num2);
    }

    public void showOperatingTimes()
    {
        System.out.println("덧셈 횟수 : " + adder.getCntAdd());
        System.out.println("뺄셈 횟수 : " + subtractor.getCntSub());
    }
}
```

- 외부에서는 Calculaor 클래스의 존재만 알면 된다.
- Adder와 Subtractor 클래스의 존재는 알 필요 없다.
- 이렇게 외부에 노출시킬 클래스를 public으로 선언한다.

```
class Adder
{
    private int cntAdd;

    Adder() { cntAdd=0; }
    int getCntAdd() { return cntAdd; }
    int addTwoNumber(int num1, int num2)
    {
        cntAdd++;
        return num1 + num2;
    }
}
```

```
class Subtractor
{
    private int cntSub;

    Subtractor() { cntSub=0; }
    int getCntSub() { return cntSub; }
    int subTwoNumber(int num1, int num2)
    {
        cntSub++;
        return num1 - num2;
    }
}
```

클래스 구분의 필요성

계산기 기능의 완성을 위해서 Calculator 클래스 이외에 Adder, Subtractor 클래스를 별도로 구분 할 필요가 있는가?



- 변경이 있을 때, 변경되는 클래스의 범위를 줄일 수 있다.
- 작은 크기의 클래스를 다른 클래스의 정의에 활용할 수 있다.

⇒ 객체지향에서는 아주 큰 하나의 클래스보다, 아주 작은 열 개의 클래스가 더 큰 힘과 위력을 발휘한다!

캡슐화(Encapsulation)

```
class SinivelCap    // 콧물 처치용 캡슐
{
    public void take(){ . . . . }
}
class SneezeCap     // 재채기 처치용 캡슐
{
    public void take() { . . . . }
}
class SnuffleCap    // 코막힘 처치용 캡슐
{
    public void take() { . . . . }
}
```

세 개의 클래스가 하나의 목적인 '콧물감기의 치료'라는 일치된 목적을 갖고 있다. 그럼에도 불구하고 클래스가 나뉘어 있다.

SinivelCap, SneezeCap, SnuffleCap의 연관 관계가 깊다면 캡슐화가 이뤄지지 않은 상태.

```
public static void main(String[] args)
{
    ColdPatient sufferer = new ColdPatient();
    sufferer.takeSinivelCap(new SinivelCap());
    sufferer.takeSneezeCap(new SneezeCap());
    sufferer.takeSnuffleCap(new SnuffleCap());
}
```

약의 복용 순서가 정해져 있다면?

약을 복용하는 사람은 약의 복용과 관련해서 추가적인 지식이 필요하다. 캡슐화가 이뤄지지 않으면, 클래스의 사용을 위해서 알아야 할 것들이 많아진다.

캡슐화

```
class CONTAC600
{
    SinivelCap sin;
    SneezeCap sne;
    SnuffleCap snu;

    public CONTAC600()
    {
        sin=new SinivelCap();
        sne=new SneezeCap();
        snu=new SnuffleCap();
    }
    public void take()
    {
        sin.take();
        sne.take();
        snu.take();
    }
}
```

- 캡슐화는 관련 있는 모든 메소드와 변수를 하나의 클래스로 묶는 것!
- 둘 이상의 클래스를 묶어서 캡슐화를 완성할 수도 있다. 캡슐화는 메소드와 변수가 코드레벨에서 묶이는 것을 의미하지 않는다. 캡슐화는 개념적인 의미의 묶음을 의미한다.
- take 메소드 내에 약의 복용순서가 그대로 기록되어 있다. 따라서 약의 복용을 위해 알아야 할 것이 take 메소드 하나이다! 이것이 캡슐화의 목적 및 장점.

오버로딩
(Overloading)

메서드 오버로딩

- 동일한 이름의 메소드를 둘 이상 동시에 정의하는 것.
- 메소드의 매개변수 선언(개수 또는 자료형)이 다르면 메소드 오버로딩 성립.
- 오버로딩 된 메소드는 호출 시 전달하는 인자를 통해서 구분된다.

ex)

```
class MethodOverload
{
    void isYourFunc( int n ) { ... }
    void isYourFunc( int n1, int n2 ) {...}
    void isYourFunc( int n1, double n2 ) {...}
}
```

```
MethodOverload inst = new MethodOverload();
inst.isYourFunc(10);
inst.isYourFunc(10, 20);
inst.isYourfunc(12, 3.15);
```

=> 전달되는 인자의 유형을 통해서 호출되는 함수가 결정된다.

메서드 오버로딩시 주의 사항 I

- 형변환의 규칙까지 적용해야만 메소드가 구분되는 애매한 상황은 만들지 말자!

ex)

```
class MethodOverload {  
    void isYourFunc( int n ) { ... }  
    void isYourFunc( int n1, int n2 ) {...}  
    void isYourFunc( int n1, double n2 ) {...}  
}
```

```
MethodOverload inst = new MethodOverload();
```

```
inst.isYourFunc(10, 'a');
```

=> 무엇이 호출 되겠는가? 문자 'a'는 int형으로도, double형으로도 변환이 가능하다!

- ▶ 결론적으로, 형변환 규칙을 적용하되 가장 가까운 위치의 자료형으로 변환이 이루어진다. 따라서 isYourFunc(int n1, int n2)가 호출된다.

메서드 오버로딩시 주의 사항 II

- 반환형이 다른 것은 메소드 오버로딩이 성립 안된다.

ex)

```
class MethodOverload
{
    int  isYourFunc( int n ) { ... }
    boolean  isYourFunc( int n ) {...}
    .....
}
```


생성자도 오버로딩의 대상

- 생성자의 오버로딩은 하나의 클래스를 기반으로 다양한 형태의 인스턴스 생성을 가능하게 한다.

```
class Person {
    private int perID;
    private int milID;

    public Person(int pID, int mID) {
        perID=pID;
        milID=mID;
    }
    public Person(int pID) {
        perID=pID;
        milID=0;
    }

    public void showInfo() {
        System.out.println("민번: "+perID);
        if(milID!=0)
            System.out.println("군번: "+milID+'Wn');
        else
            System.out.println("군과 관계 없음 Wn");
    }
}
```

```
class Overloading
{
    public static void main(String[] args)
    {
        Person man =
            new Person(950123, 880102);
        Person woman =
            new Person(941125);

        man.showInfo();
        woman.showInfo();
    }
}
```

=> 군을 제대한 남성과 여성을 의미하는
인스턴스의 생성이 가능하다!

키워드 this를 이용한 다른 생성자의 호출

- 키워드 this를 이용하면 생성자 내에서 다른 생성자를 호출할 수 있다.
- 이는 생성자의 추가 정의에 대한 편의를 제공한다.
- 생성자마다 중복되는 초기화 과정의 중복을 피할 수 있다.

```
class Person {  
    private int perID;  
    private int milID;  
    private int birthYear;  
    private int birthMonth;  
    private int birthDay;  
  
    public Person(int perID, int milID, int bYear, int bMonth, int bDay) {  
        this.perID = perID;  
        this.milID = milID;  
        birthYear = bYear;  
        birthMonth = bMonth;  
        birthDay = bDay;  
    }  
  
    public Person(int perID, int bYear, int bMonth, int bDay) {  
        this(perID, 0, bYear, bMonth, bDay);  
    }  
    .....  
}
```

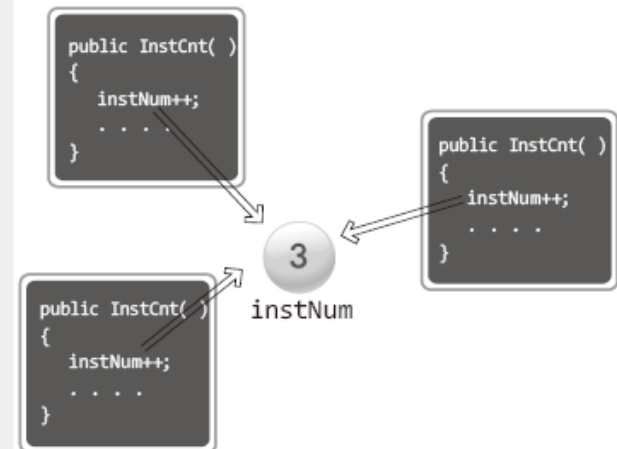
클래스 변수 &
클래스 메소드

static 변수(클래스 변수)

- 인스턴스의 생성과 상관없이 초기화되는 변수.
- 하나만 선언되는 변수.
- public으로 선언되면 누구나 어디서든 접근 가능.

```
class InstCnt
{
    static int instNum=0;
    public InstCnt()
    {
        instNum++;
        System.out.println("인스턴스 생성 : "+instNum);
    }
}

class ClassVar
{
    public static void main(String[] args)
    {
        InstCnt cnt1=new InstCnt();
        InstCnt cnt2=new InstCnt();
        InstCnt cnt3=new InstCnt();
    }
}
```



인스턴스 생성 : 1
인스턴스 생성 : 2
인스턴스 생성 : 3

static 변수의 접근 방법

```
1. class AccessWay {
2.     static int num=0;
3.     AccessWay() {
4.         incrCnt();
5.     }
6.     public void incrCnt(){ num++; } // 클래스 내부 접근 방법
7. }
8.
9. class ClassVarAccess {
10.    public static void main(String[] args) {
11.        AccessWay way = new AccessWay();
12.        way.num++; // 인스턴스의 이름을 이용한 접근 방법
13.        AccessWay.num++; // 클래스의 이름을 이용한 접근 방법
14.        System.out.println("num=" + AccessWay.num ); }
15. }
```

static 변수의 초기화 시점

- JVM은 실행과정에서 필요한 클래스의 정보를 메모리에 로딩한다.
- 로딩 시점에서 static 변수가 초기화 된다.

```
1.  class InstCnt {
2.      static int instNum=100;
3.      public InstCnt() {
4.          instNum++;
5.          System.out.println("인스턴스 생성: " + instNum);
6.      }
7.  }
8.
9.  class StaticValNoInst {
10.     public static void main(String[] args)      {
11.         InstCnt.instNum -= 15;    // 인스턴스 생성 없이 사용.
12.         System.out.println(InstCnt.instNum);
13.     }
14. }
```

static 변수의 활용

- 동일한 클래스의 인스턴스 사이에서의 데이터 공유가 필요할 때, static 변수는 유용하게 활용된다.
- 클래스 내부, 또는 외부에서 참조의 목적으로 선언된 변수는 static final로 선언한다.

```
1. class Circle {  
2.     static final double PI=3.1415;  
3.     private double radius;  
4.  
5.     public Circle(double rad) { radius=rad; }  
6.     public void showPerimeter() { // 둘레 출력  
7.         double peri=(radius*2)*PI;  
8.         System.out.println("둘레: "+peri);  
9.     }  
10.    public void showArea() { // 넓이 출력  
11.        double area=(radius*radius)*PI;  
12.        System.out.println("넓이: "+area);  
13.    }  
14. }
```

```
class ClassVarUse {  
    public static void main(String[] args)  
    {  
        Circle cl = new Circle(1.2);  
        cl.showPerimeter();  
        cl.showArea();  
    }  
}
```

static 메소드(클래스 메소드)

- 기본적인 특성과 접근 방법은 static 변수와 동일.

```
1. class NumberPrinter {
2.     public static void showInt(int n) { System.out.println(n); }
3.     public static void showDouble(double n) { System.out.println(n); }
4. }
5.
6. class ClassMethod {
7.     public static void main(String[] args) {
8.         NumberPrinter.showInt(20); // 클래스의 이름을 통한 호출
9.
10.        NumberPrinter np = new NumberPrinter();
11.        np.showDouble(3.15); // 인스턴스의 이름을 통한 호출
12.    }
13. }
```

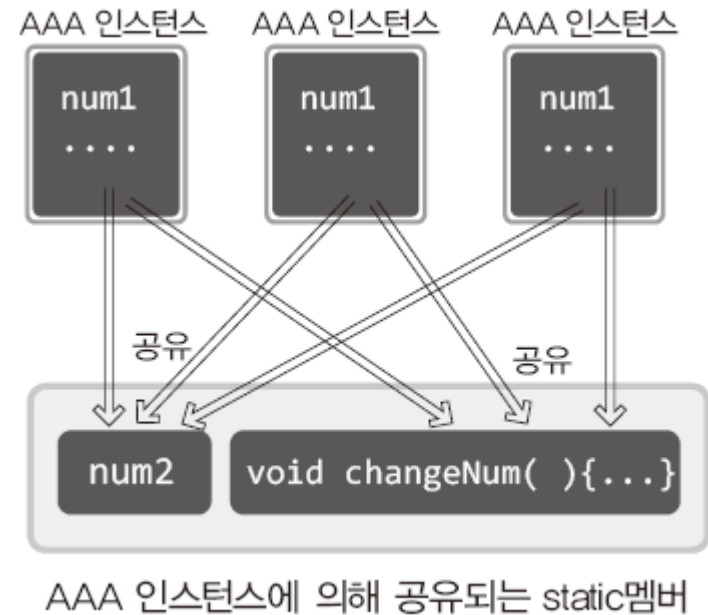

static 메소드의 활용

```
1. class SimpleMath { // 단순 계산 클래스
2.     public static final double PI=3.1415;
3.     public static double add(double n1, double n2){ return n1+ n2; }
4.     public static double min(double n1, double n2){ return n1 - n2; }
5.     public static double mul(double n1, double n2){ return n1* n2; }
6. }
7. class AreaMath { // 넓이 계산 클래스
8.     public static double calCircleArea(double rad) {
9.         double result=SimpleMath.mul(rad, rad);
10.        result=SimpleMath.mul(result, SimpleMath.PI);
11.        return result;
12.    }
13.    public static double calRectangleArea(double width, double height) {
14.        return SimpleMath.mul(width, height);
15.    }
16. }
17. class ChangeToStaticMethod {
18.     public static void main(String[] args) {
19.         System.out.println("원의 넓이: "+AreaMath.calCircleArea(2.4));
20.     }
21. }
```

static 메소드의 인스턴스 접근 불가

- static 메소드는 인스턴스에 속하지 않기 때문에 인스턴스 멤버에 접근이 불가능하다.

```
class AAA
{
    int num1;
    static int num2;
    static void changeNum()
    {
        num1++;    // 문제 됨!
        num2++;    // 문제 안됨!
    }
    . . . .
}
```



System.out.println()

- System : java.lang 패키지에 묶여있는 클래스의 이름.
 - import java.lang.* ; 자동 삽입되므로 System이란 이름을 직접 쓸 수 있다.
- out : static 변수이되 인스턴스를 참조하는 참조 변수
 - PrintStream 이라는 클래스의 참조 변수

```
public class System
{
    public static final PrintStream out;
    . . . .
} // static final로 선언되었으니, 인스턴스의 생성없이
// System.out 이라는 이름으로 접근 가능하다.
```

➤ System.out.println()은 System 클래스의 멤버 out이 참조하는 인스턴스의 println 메소드를 호출하는 문장이다.

메소드 특징

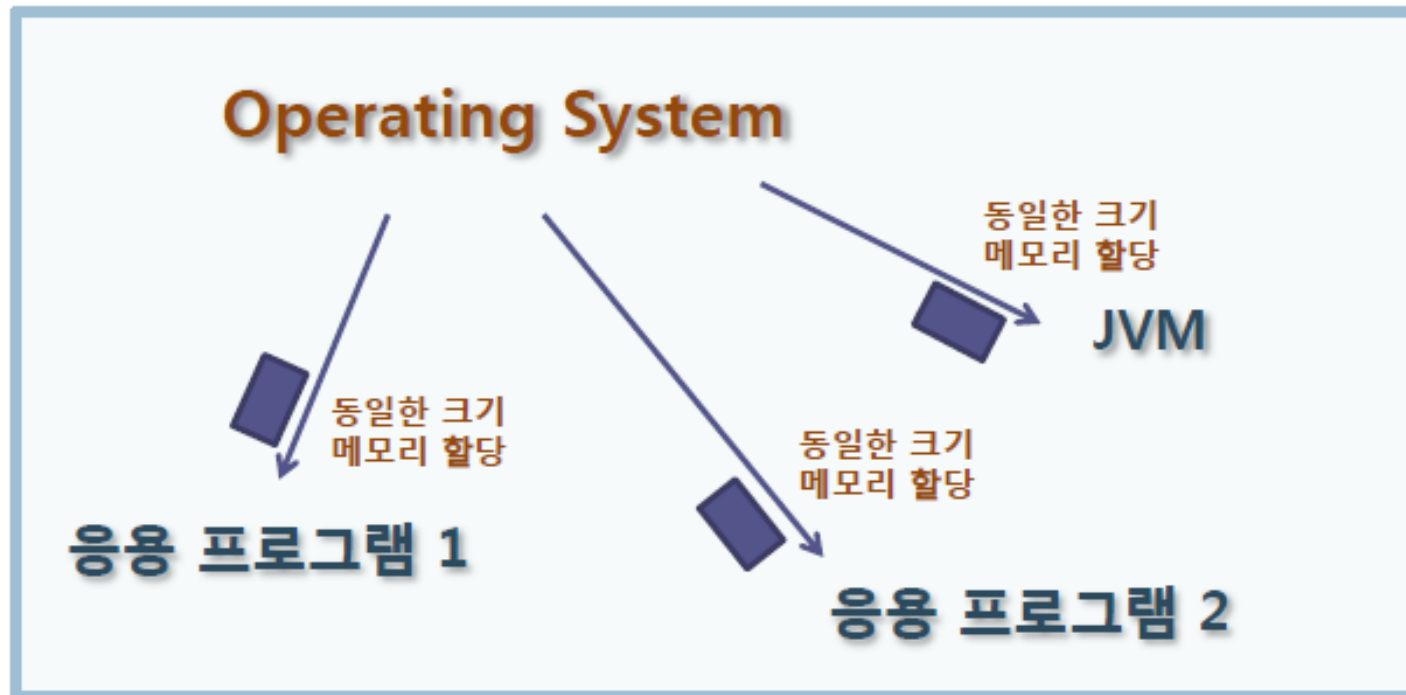
- 모든 메소드는 자신이 속한 클래스의 인스턴스 생성이 가능!
- 이는 main 메소드도 마찬가지!
- 따라서 main 메소드는 어디든 존재할 수 있다.

```
class AAA
{
    public static void makeAAA()
    {
        AAA a1 = new AAA();
        . . . .
    }
    . . . .
}
```

자바의 메모리 모델

☞ JVM은 운영체제 위에서 동작한다.

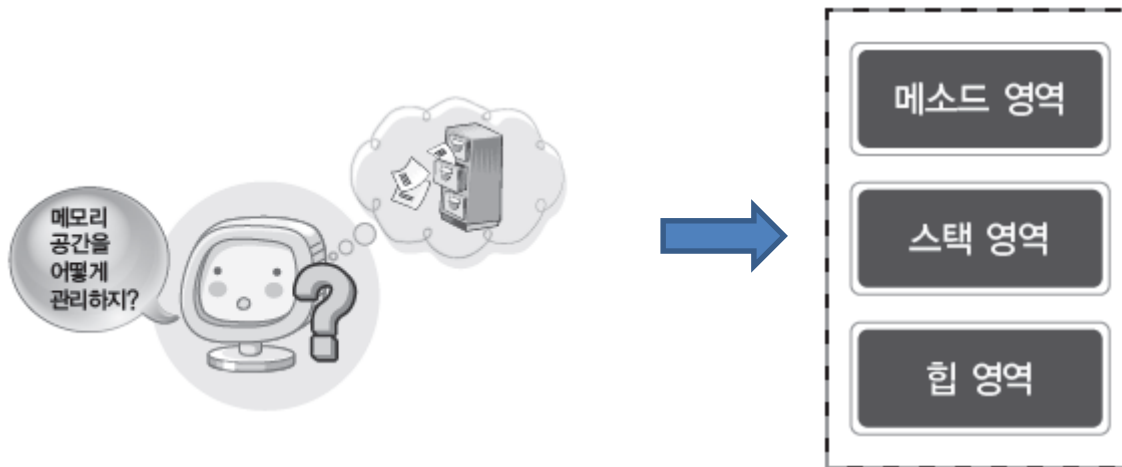
- 운영체제가 JVM을 포함해서 모든 응용 프로그램에게 동일한 크기의 메모리 공간을 할당할 수 있는 이유는 가상메모리 기술에서 찾을 수 있다.
- JVM은 운영체제로 부터 할당 받은 메모리공간을 기반으로 자바프로그램을 실행해야 한다.
- JVM은 운영체제로 부터 할당 받은 메모리 공간을 이용해서 자기 자신도 실행을 하고, 자바 프로그램도 실행을 한다.



👉 JVM의 메모리 살림살이

JVM의 메모리 구분 및 관리 기준

- | | |
|------------------------|-----------------------|
| • 메소드 영역 (method area) | 메소드의 바이트코드, static 변수 |
| • 스택 영역 (stack area) | 지역변수, 매개변수 |
| • 힙 영역 (heap area) | 인스턴스 |



메모리 공간을 용도에 따라서 별도로 나누는 이유는, 서랍장의 칸을 구분하고, 칸 별로 용도를 지정하는 이유와 차이가 없다!

메소드 영역과 스택의 특성

메소드 영역에 대한 설명

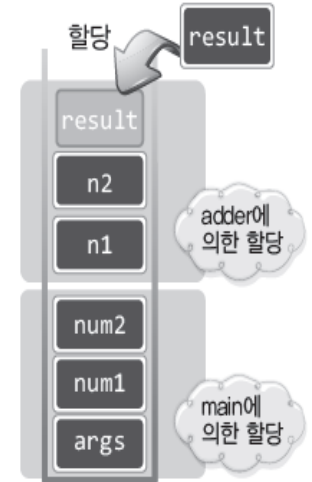
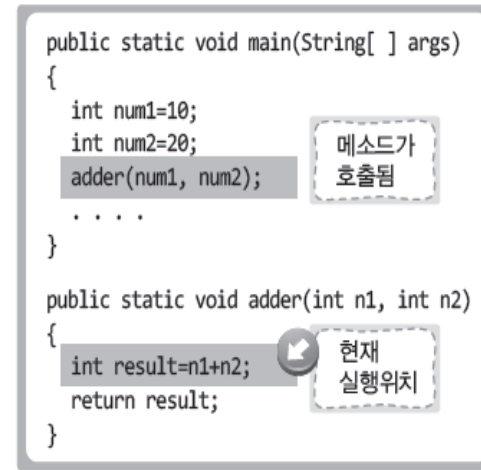
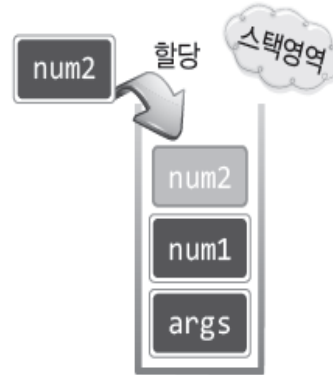
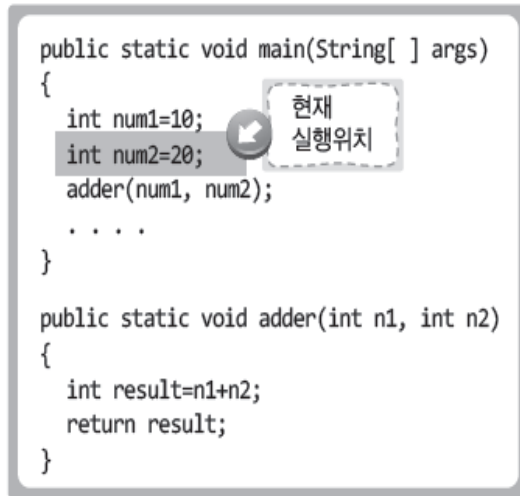
- ✓ 자바 바이트코드(bytecode) : 자바 가상머신에 의해서 실행되는 코드.
- ✓ 메소드의 자바 바이트코드는 JVM이 구분하는 메모리 공간 중에서 메소드 영역에 저장된다.
- ✓ static으로 선언된 클래스 변수도 메소드 영역에 저장된다.
- ✓ 정리하면, 클래스의 정보가 JVM의 메모리 공간에 LOAD 될 때 할당 및 초기화되는 대상은 메소드 영역에 할당된다.

참고로, 메소드의 바이트코드는 실행에 필요한 바이트코드 전부를 의미한다. 자바 프로그램의 실행은 메소드 내에 정의된 문장들의 실행으로 완성되기 때문이다.

스택 영역에 대한 설명

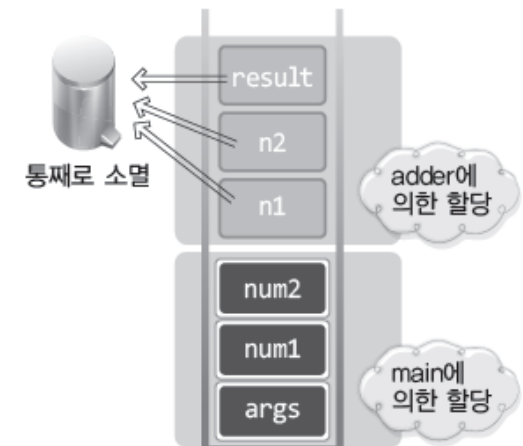
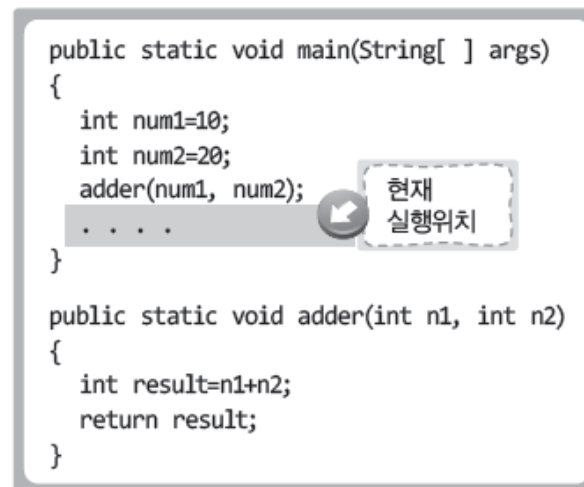
- ✓ 매개변수, 지역변수가 할당되는 메모리 공간.
- ✓ 프로그램이 실행되는 도중에 임시로 할당되었다가 바로 이어서 소멸되는 특징이 있는 변수가 할당된다.
- ✓ 메소드의 실행을 위한 메모리 공간으로도 정의할 수 있다.

👉 스택의 흐름



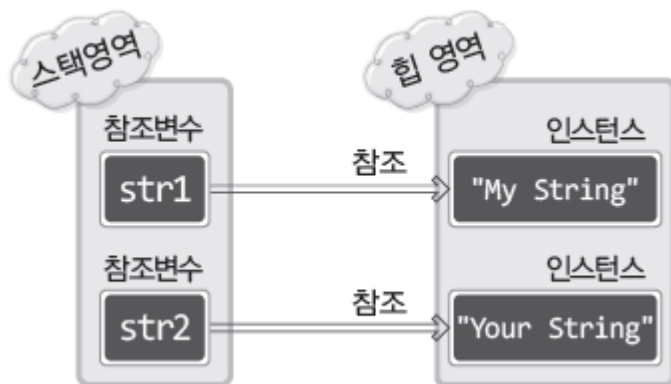
- 지역변수는 스택에 할당된다.
- 스택에 할당된 지역변수는 해당 메소드를 빠져 나가면 소멸된다. 할당 및 소멸의 특성상 그 형태가 접시를 쌓는 것과 유사하다. 따라서 스택이라 이름 지어졌다.

할당 및 소멸의 특성상
메소드 별 스택이 구분이
된다!



👉 힙 영역

- 인스턴스가 생성되는 메모리 공간
- JVM에 의한 메모리 공간의 정리(Garbage Collection)가 이뤄지는 공간
- 할당은 프로그래머가 소멸은 JVM이.
- 참조 변수에 의한 참조가 전혀 이뤄지지 않는 인스턴스가 소멸의 대상이 된다. 따라서 JVM은 인스턴스의 참조관계를 확인하고 소멸 할 대상을 선정한다.

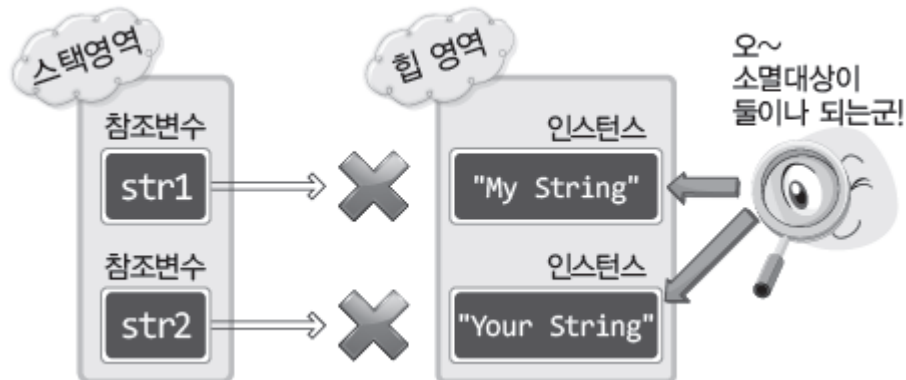
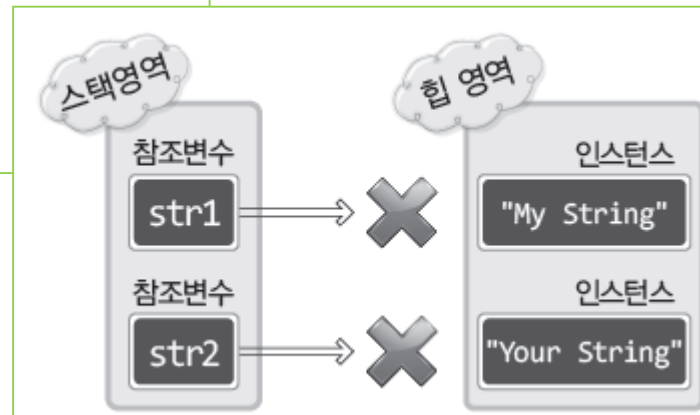


메소드 내에서 인스턴스를 생성한다면 위의 그림에서 설명하듯이 참조변수는 스택에 인스턴스는 힙에 저장된다.

👉 인스턴스의 소멸시기

```
public static void simpleMethod()  
{  
    String str1=new String("My String");  
    String str2=new String("Your String");  
    . . . .  
    str1=null;  
    str2=null;  
    . . . .  
}
```

참조가 이뤄지지 않으면
소멸의 대상이 된다!



JVM은 인스턴스의 참조관계
를 통해서 소멸 대상을 결정
한다!