Chapter 19. 자바의 메모리 모델과 Object 클래스

나는 정말 JAVA를 공부한 적이 없다고

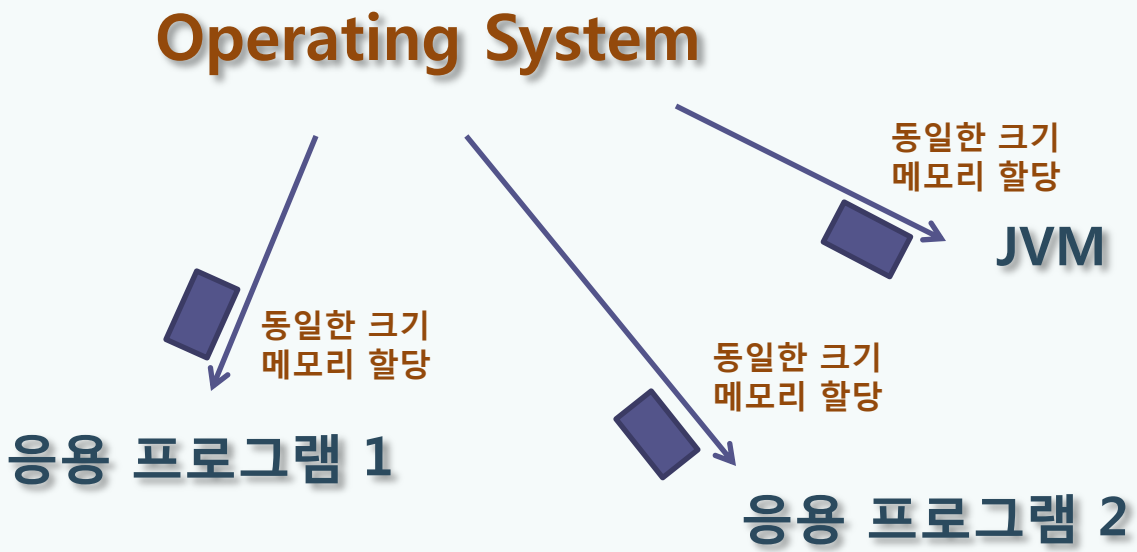
19-1. 자바 가상머신의 메모리 모델

■ JVM은 운영체제 위에서 동작한다.

· 운영체제가 JVM을 포함해서 모든 응용 프로그램에게 동일한 크기의 메모리 공간을 할당할 수 있는 이유는 **가상 메모리 기술**에서 찾을 수 있다.

·JVM은 운영체제로부터 할당받은 메모리 공간을 기반으로 자바 프로그램을 실행해야 한다.

· JVM은 운영체제로부터 할당받은 메모리 공간을 이용해서 자기 자신도 실행을 하고, 자바 프로그램도 실행을 한다.



■ JVM의 메모리 살림살이

나 정말 JAVA를 공부한 적 없는데

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

나 정말 JAVA를 공부한 적 없는데요

메소드 영역에 대한 설명

- 자바 바이트코드(bytecode) : 자바 가상머신의 의해서 실행되는 코드
- 메소드의 **자바 바이트코드**는 JVM이 구분하는 메모리 공간 중에서 **메소드 영역**에 저장된다.
- static으로 선언된 **클래스 변수**도 **메소드 영역**에 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

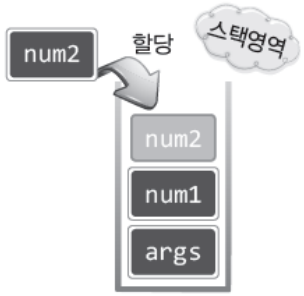
- 매개변수, 지역변수가 할당되는 메모리 공간
- 프로그램이 실행되는 도중에 임시로 할당되었다가 바로 이어서 소멸되는 특징이 있는 변수가 할당된다.
- 메소드의 실행을 위한 메모리 공간으로도 정의할 수 있다.

■ 스택의 흐름

Ⅰ

```
public static void main(String[ ] args)
{
  int num1=10;
  int num2=20;
  adder(num1, num2);
  . . . .
}

public static void adder(int n1, int n2)
{
  int result=n1+n2;
  return result;
}
```

현재 실행위치



·지역변수는 스택에 할당된다.

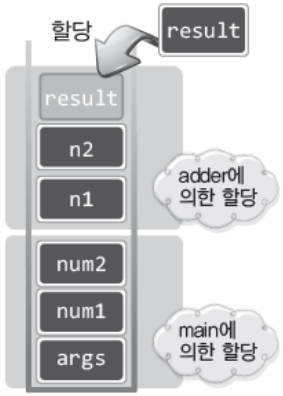
·스택에 할당된 지역변수는 해당 메소드를 빠져 나가면 소멸된다.

Ⅱ

```
public static void main(String[ ] args)
{
  int num1=10;
  int num2=20;
  adder(num1, num2);
  . . . .
}

public static void adder(int n1, int n2)
{
  int result=n1+n2;
  return result;
}
```

메소드가 호출됨

현재 실행위치

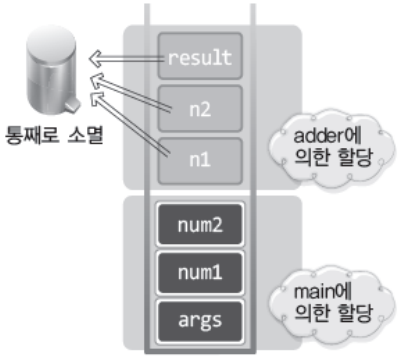


· 할당 및 소멸의 특성상 그 형태가 접시를 쌓는 것과 유사하다 따라서 스택이라 이름 지어졌다.

Ⅲ

```
public static void main(String[ ] args)
{
  int num1=10;
  int num2=20;
  adder(num1, num2);
  . . . .
}

public static void adder(int n1, int n2)
{
  int result=n1+n2;
  return result;
}
```

현재 실행위치



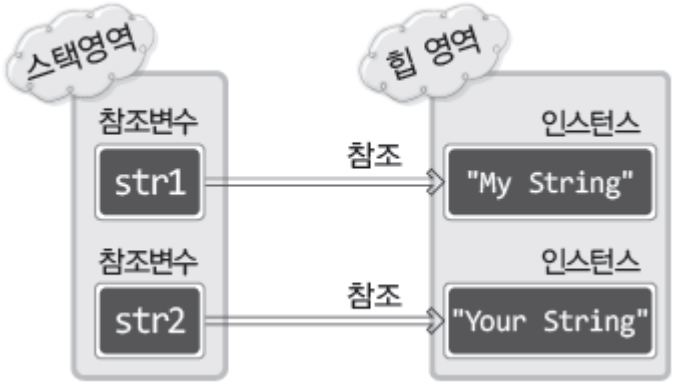
할당 및 소멸의 특성상

메소드 별 스택이 구분이 된다!

■ 힙 영역

힙 영역에 대한 설명

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



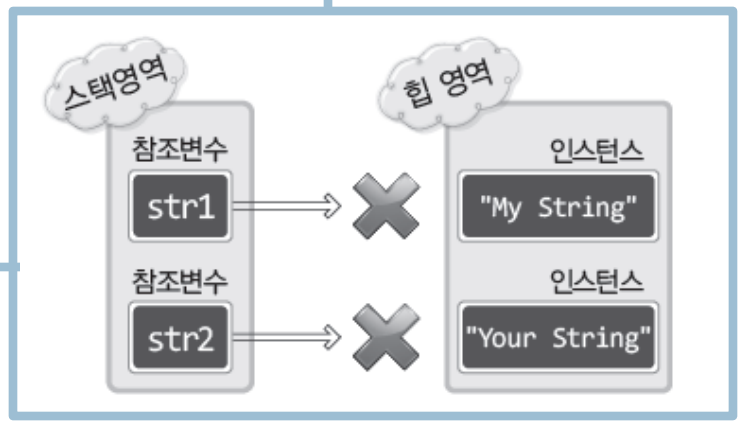
메소드 내에서 인스턴스를 생성한다면 위의 그림에서 설명하듯이 참조변수는 스택에 인스턴스는 힙에 저장된다.

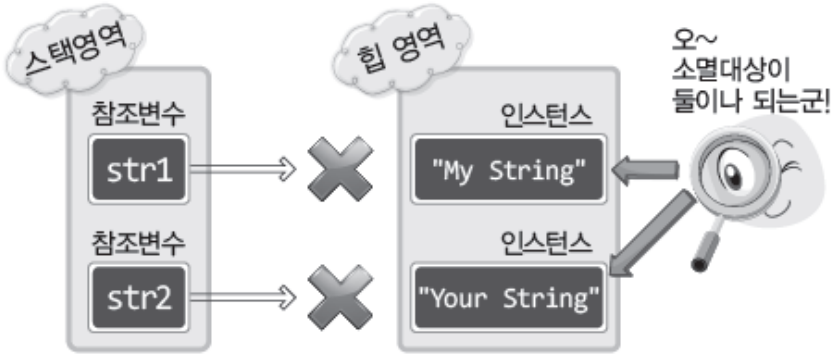
■ 인스턴스의 소멸시기


```
public staic 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 참조관계를

통해서 소멸 대상을 결정한다!

19-2. Object 클래스

■ 인스턴스 소멸 시 호출되는 finalize 메소드


```
protected void finalize( ) throws Throwable
```

인스턴스가 완전히 소멸되기 직전 호출되는 메소드,

Object 클래스의 멤버이므로 모든 인스턴스에는 이 메소드가 존재한다.

```
class MyName
{
    String objName;
    public MyName(String name)
    {
        objName=name;
    }
    protected void finalize() throws Thr
    {
        super.finalize();
        System.out.println(objName+"이 소멸되었습니다.");
    }
}
```

```
public static void main(String[] args)
{
    MyName obj1=new MyName("인스턴스1");
    MyName obj2=new MyName("인스턴스2");
    obj1=null;
    obj2=null;

    System.out.println("프로그램을 종료합니다.");
    // System.gc();
    // System.runFinalization();
}
```

위 예제의 실행과정에서 finalize 메소드는 호출되지 않을 수 있다.

Garbage Collection이 실행되는 시기와 인스턴스의 완전한 소멸의 시기는 차이가 날 수 있기 때문이다.

■ Garbage Collection에 대한 추가 설명

나 정말 JAVA를 공부한 적이 없다구요

Garbage Collection에 대한 추가적인 설명

- GC는 한 번도 발생하지 않을 수 있다.
- GC가 발생하면, 소멸의 대상이 되는 인스턴스는 결정되지만 이것이 실제 소멸로 바로 이어지지는 않는다.
- 인스턴스의 실제 소멸로 이어지지 않은 상태에서 프로그램이 종료될 수도 있다. 종료가 되면 어차피 인스턴스는 소멸 되니까
- 따라서 반드시 finalize 메소드가 반드시 호출되기 원한다면 아래에서 보이는 코드가 추가로 삽입되어야 한다.

```
System.gc();
System.runFinalization();
```

Garbage Collection을 명령함!

GC에 의해서 소멸이 결정된 인스턴스를 즉시 소멸하라!

finalize 메소드의 오버라이딩의 예


```
protected void finalize() throws Throwable
{
    super.finalize();
    System.out.println(objName+"이 소멸되었습니다.");
}
```

이것은 모범이 되는 메소드 오버라이딩의 예! 이다.

super.finalize( )

- Object 클래스에 정의되어 있는 finalize 메소드에 중요한 코드가 삽입되어 있는지 확인한 바 없다!
- 만약에 중요한 코드가 삽입되어 있다면? 단순한 오버라이딩으로 인해서 중요한 코드의 실행을 방해할 수 있다!
- 따라서! 대상 메소드에 대한 정보가 부족한 경우에는 오버라이딩 된 메소드도 호출이 되도록 오버라이딩을 하자! 이것이 오버라이딩의 기본 원칙이다.

인스턴스 비교


```
class IntNumber
{
    int num;
    public IntNumber(int num) { this.num=num; }
    public boolean isEquals(IntNumber numObj)
    {
        if(this.num==numObj.num)
            return true;
        else
            return false;
    }
}
```

이전에 언급했듯이 == 연산자는 참조 값 비교를 한다. 따라서 인스턴스간 내용비교를 위해서는 내용비교 기능의 메소드가 필요하다.

```
public static void main(String[] args)
{
    IntNumber num1=new IntNumber(10);
    IntNumber num2=new IntNumber(12);
    IntNumber num3=new IntNumber(10);

    if(num1.isEquals(num2))
        System.out.println("num1과 num2는 동일한 정수");
    else
        System.out.println("num1과 num2는 다른 정수");

    if(num1.isEquals(num3))
        System.out.println("num1과 num3는 동일한 정수");
    else
        System.out.println("num1과 num3는 다른 정수");
}
```

실행 결과

```
num1과 num2는 다른 정수
num1과 num3는 동일한 정수
```

equals 메소드

```
class IntNumber
{
    int num;
    public IntNumber(int num)
    {
        this.num=num;
    }
    public boolean equals(Object obj)
    {
        if(this.num==((IntNumber)obj).num)
            return true;
        else
            return false;
    }
}
```

JAVA에서는 인스턴스간의 내용 비교를 목적으로 Object 클래스에 equals 메소드를 정의해 놓았다.
따라서 새로 정의되는 클래스의 내용 비교가 가능하도록 이 메소드를 오버라이딩 하는 것이 좋다!

Object 클래스의 equals 메소드를 인스턴스의 내용비교 메소드로 지정해 놓았기 때문에, 처음 접하는 클래스의 인스턴스라 하더라도 equals 메소드의 호출을 통해서 인스턴스간 내용 비교를 할 수 있다.

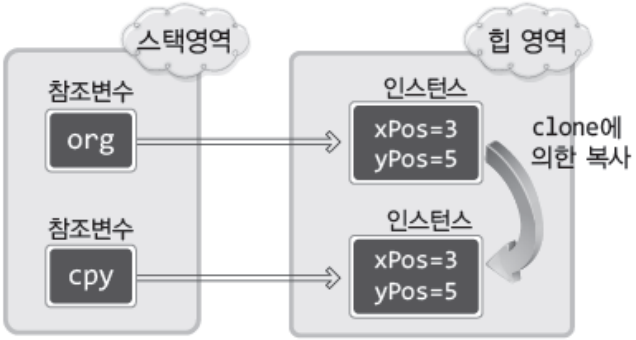
인스턴스의 복사(복제): clone 메소드

- Object 클래스에는 인스턴스의 복사를 목적으로 clone이라는 이름의 메소드가 정의되어 있다.
- 단, 이 메소드는 Cloneable 인터페이스를 구현하는 클래스의 인스턴스에서만 호출될 수 있다.
- Cloneable 인터페이스의 구현은 다음의 의미를 지닌다.
  "이 클래스의 인스턴스는 복사를 해도 됩니다."
- 사실 인스턴스의 복사는 매우 민감한 작업이다. 따라서 클래스를 정의할 때 복사의 허용여부를 결정하도록 Cloneable 인터페이스를 통해서 요구하고 있다.

```
class Point implements Cloneable
{
    private int xPos;
    private int yPos;
    . . . . .
    public Object clone() throws CloneNotSupportedExcepti
    {
        return super.clone();
    }
}
```

clone 메소드는 protected로 선언되어 있다.
따라서 외부 호출이 가능하도록 public으로 오버라이딩!

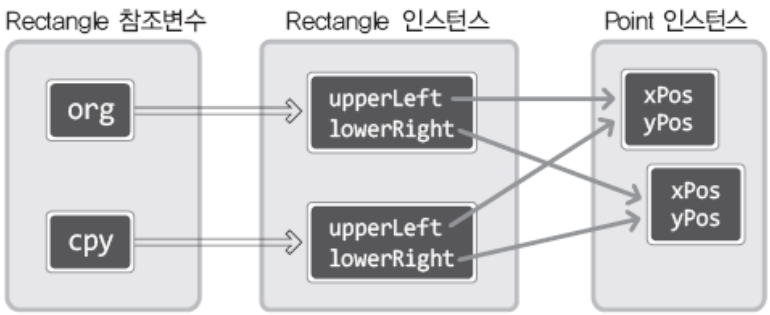
인스턴스를 통째로 복사한다!



■ 얕은(Shallow) 복사의 예


```
class Rectangle implements Cloneable
{
    Point upperLeft, lowerRight;

    public Rectangle(int x1, int y1, int x2, int y2)
    {
        upperLeft=new Point(x1, y1);
        lowerRight=new Point(x2, y2);
    }

    . . . . . .

    public Object clone() throws CloneNotSupportedExcepti
    {
        return super.clone();
    }
}
```

```
public static void main(String[] args)
{
    Rectangle org=new Rectangle(1, 1, 9, 9);
    Rectangle cpy;

    try
    {
        cpy=(Rectangle)org.clone();
        org.changePos(2, 2, 7, 7);
        org.showPosition();
        cpy.showPosition();
    }
    catch(CloneNotSupportedException e)
    {
        e.printStackTrace();
    }
}
```

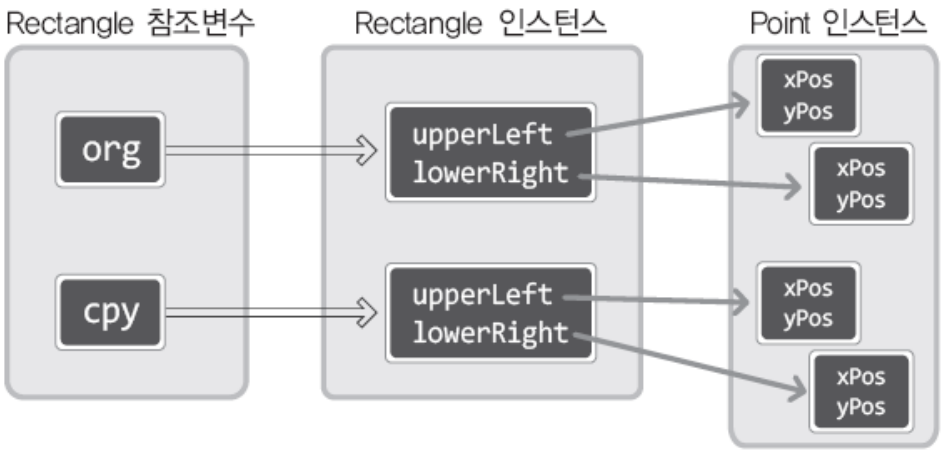


복사결과!
이러한 유형의 복사를 가리켜 얕은 복사라 한다!

Object 클래스의 clone 메소드는 인스턴스를 통째로 복사는 하지만, 인스턴스가 참조하고 있는 또 다른 인스턴스까지 복사하지는 않는다. 단순히 참조 값만을 복사할 뿐이다!

■ 깊은(Deep) 복사의 예


```
class Rectangle implements Cloneable
{
    // clone 메소드를 제외한 나머지는 ShallowCopy.java와 동일하므로 생략

    public Object clone() throws CloneNotSupportedException
    {
        Rectangle copy=(Rectangle)super.clone();
        copy.upperLeft=(Point)upperLeft.clone();
        copy.lowerRight=(Point)lowerRight.clone();
        return copy;
    }
}
```



복사결과!
이러한 유형의 복사를 가리켜 깊은 복사라 한다!

■ String 인스턴스와 배열 인스턴스 대상의 복사

• String 인스턴스에 저장되어 있는 문자열 정보는 변경되지 않는다. 따라서 굳이 String 인스턴스를 깊은 복사의 목록에 포함시킬 필요는 없다.

• 배열 대상의 clone 메소드의 호출 결과는 배열의 복사이다! 즉, 배열과 배열에 저장된 인스턴스의 참조 값은 복사가 되지만, 배열의 참조 값이 참조하는 인스턴스까지 복사가 진행되지는 않는다!