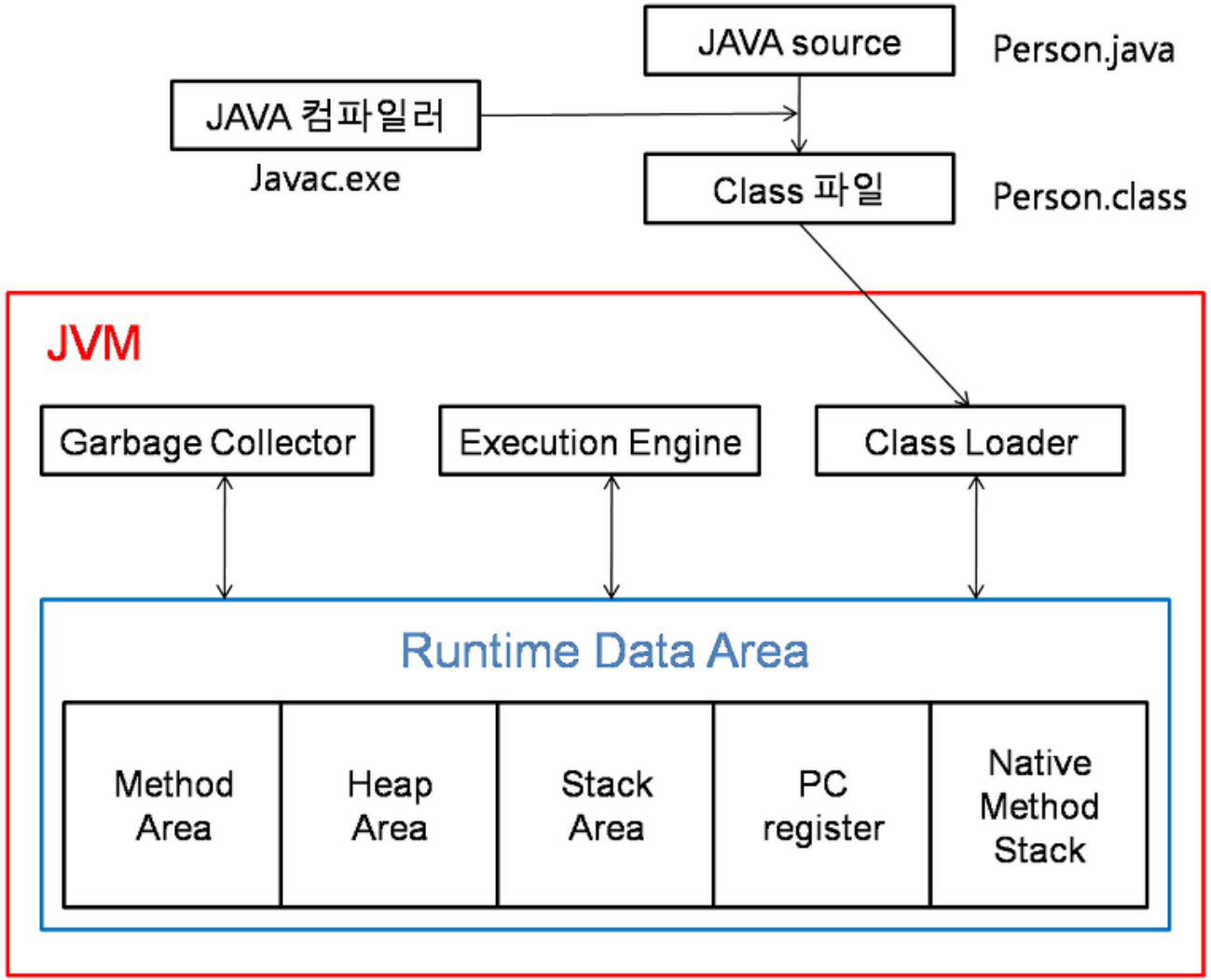
Java 기초

제어자

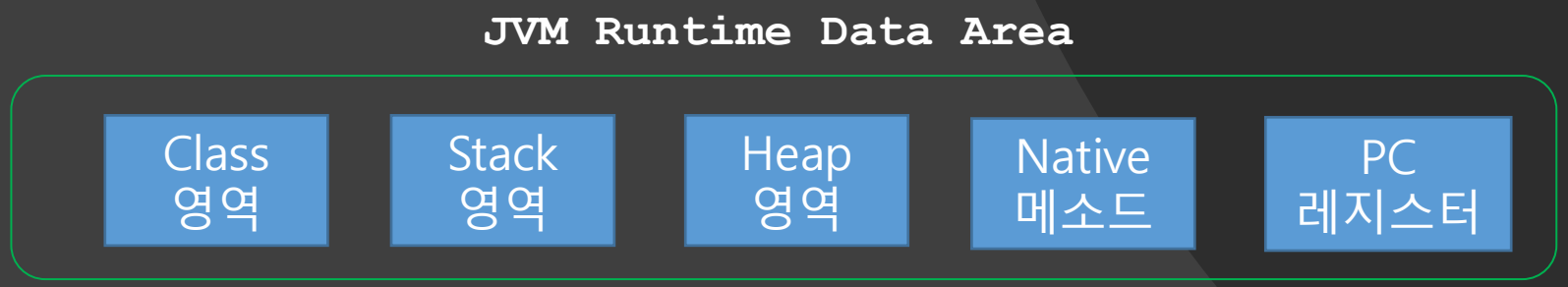
JVM 메모리 구조

JVM 메모리 구조



JVM 메모리 구조

- Runtime Data Area
  - JVM이 OS가 실행될 때 할당받는 데이터 영역
  - 크게 다섯가지 영역으로 나뉘며 각각의 영역은 다른 역할을 수행한다

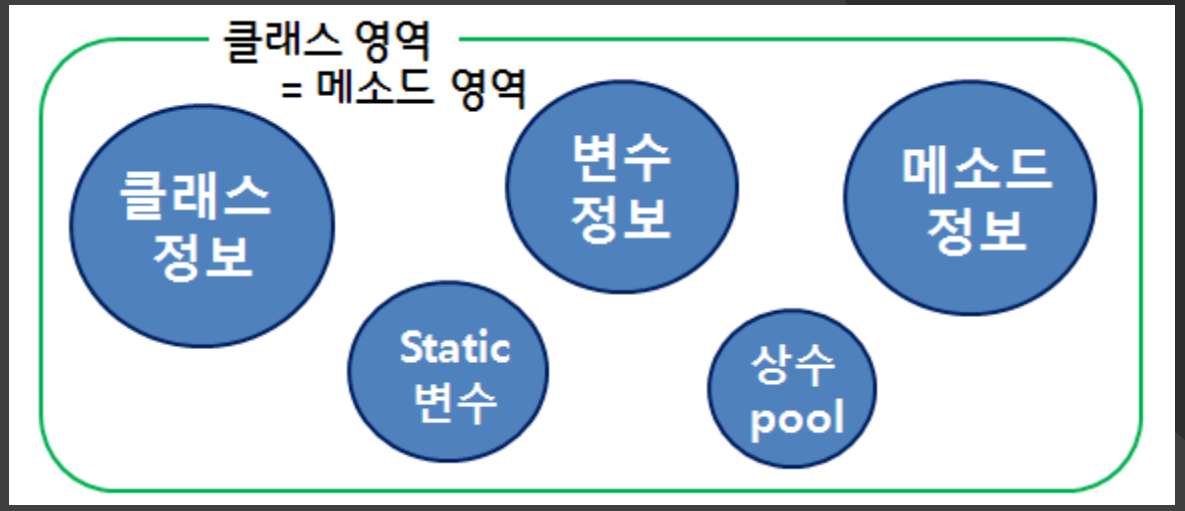


JVM 메모리 구조

- 주 JVM 기능
  - Class Loader
    - JVM 내로 클래스 파일을 로드하고, 링크를 통해 배치하는 작업을 수행하는 모듈. 런타임 시에 동적으로 클래스를 로드한다.
  - Execution Engine
    - 클래스 로더를 통해 JVM 내의 Runtime Data Area에 배치된 바이트 코드들을 명령어 단위로 읽어서 실행한다. 최초 JVM이 나왔을 당시에는 인터프리터 방식이었기 때문에 속도가 느리다는 단점이 있었지만 JIT 컴파일러 방식을 통해 이 점을 보완하였다. JIT는 바이트 코드를 어셈블러 같은 네이티브 코드로 바꿈으로써 실행이 빠르지만 역시 변환하는데 비용이 발생하였다. 이 같은 이유로 JVM은 모든 코드를 JIT 컴파일러 방식으로 실행하지 않고, 인터프리터 방식을 사용하다가 일정한 기준이 넘어가면 JIT 컴파일러 방식으로 실행한다.
  - Garbage Collector
    - Garbage Collector(GC)는 힙 메모리 영역에 생성된 객체들 중에서 참조되지 않은 객체들을 탐색 후 제거하는 역할을 한다.

JVM 메모리 구조

- Class 메모리 영역(Method 메모리 영역)
  - 클래스 멤버 변수의 이름, 데이터 타입, 접근 제어자 정보같은 필드 정보와 메소드의 이름, 리턴 타입, 파라미터, 접근 제어자 정보같은 메소드 정보, Type정보(Interface인지 class인지), Constant Pool(상수 풀 : 문자 상수, 타입, 필드, 객체 참조가 저장됨), static 변수, final class 변수등이 생성되는 영역이다.

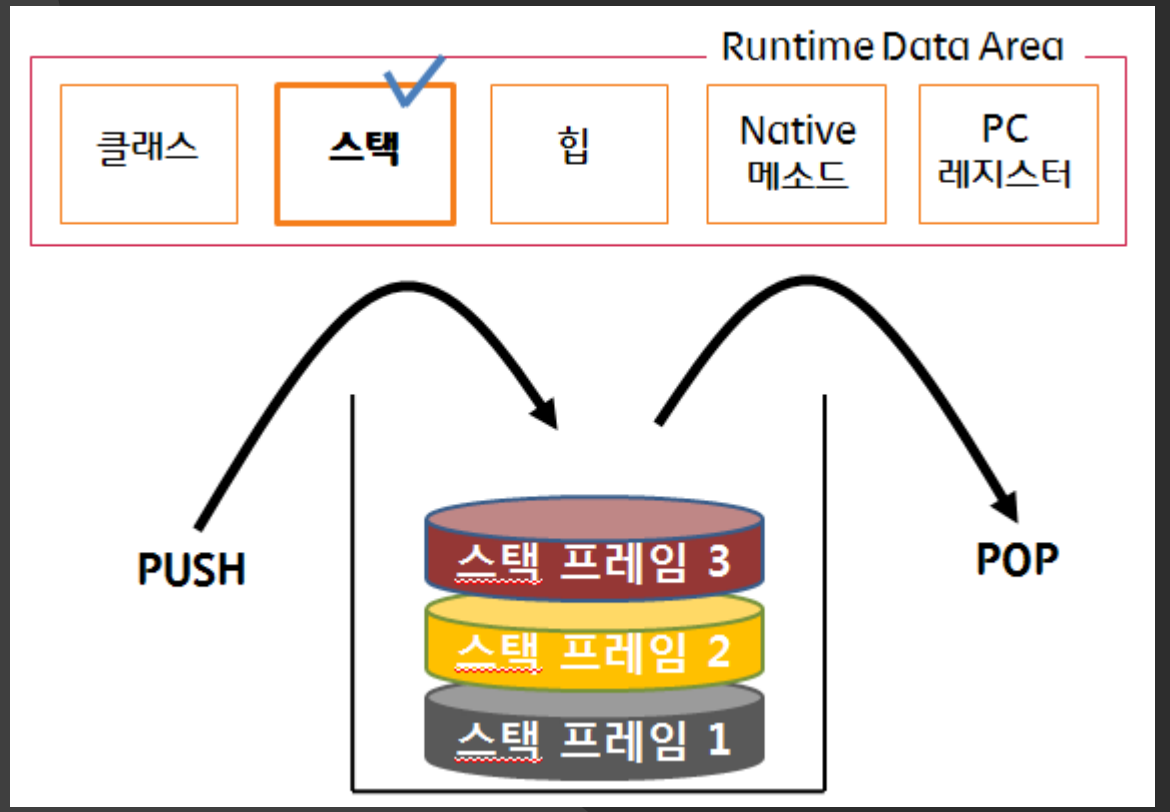


JVM 메모리 구조

- 힙 메모리 영역(heap)
  - new 키워드로 생성된 객체와 배열이 생성되는 영역.
  - 메소드 영역에 로드된 클래스만 생성이 가능하고 Garbage Collector가 참조되지 않는 메모리를 확인하고 제거하는 영역.

JVM 메모리 구조

- call stack 메모리 영역(heap)
  - 지역 변수, 파라미터, 리턴 값, 연산에 사용되는 임시 값 등이 생성되는 영역.
  - int a = 10; 이라는 소스를 작성했다면 정수 값이 할당될 수 있는 메모리공간을 a라고 잡아두고 그 메모리 영역에 값이 10이 들어간다. 즉, 스택에 메모리에 이름이 a라고 붙여주고 값이 10인 메모리 공간을 만든다.
  - 클래스 Person p = new Person(); 이라는 소스를 작성했다면 Person p는 스택 영역에 생성되고 new로 생성된 Person 클래스의 인스턴스는 힙 영역에 생성된다.



JVM 메모리 구조

- PC Register (PC 레지스터)
  - Thread(쓰레드)가 생성될 때마다 생성되는 영역으로 Program Counter 즉, 현재 쓰레드가 실행되는 부분의 주소와 명령을 저장하고 있는 영역이다. (*CPU의 레지스터와 다름)
  - 이것을 이용해서 쓰레드를 돌아가면서 수행할 수 있게 한다.

- Native method stack
  - 자바 외 언어로 작성된 네이티브 코드를 위한 메모리 영역이다.
  - 보통 C/C++등의 코드를 수행하기 위한 스택이다. (JNI)

JVM 메모리 구조

- GC(Garbage Collection)
  - 참조되지 않은 객체들을 탐색 후 삭제
  - 삭제된 객체의 메모리를 반환
  - Heap 메모리의 재사용
  - GC는 Minor GC와 Major GC로 나뉜다.

JVM 메모리 구조

- Minor GC : New 영역에서 일어나는 GC
  - 최초에 객체가 생성되면 Eden영역에 생성된다.
  - Eden영역에 객체가 가득 차게 되면 첫 번째 CG가 일어난다.
  - survivor1 영역에 Eden영역의 메모리를 그대로 복사된다. 그리고 survivor1 영역을 제외한 다른 영역의 객체를 제거한다.
  - Eden영역도 가득차고 survivor1영역도 가득 차게 된다면, Eden영역에 생성된 객체와 survivor1영역에 생성된 객체 중에 참조되고 있는 객체가 있는지 검사한다.
  - 참조 되고 있지 않은 객체는 내버려두고 참조되고 있는 객체만 survivor2영역에 복사한다.
  - survivor2영역을 제외한 다른 영역의 객체들을 제거한다.
  - 위의 과정중에 일정 횟수이상 참조되고 있는 객체들을 survivor2에서 Old영역으로 이동시킨다.
  - 위 과정을 계속 반복, survivor2 영역까지 꽉차기 전에 계속해서 Old로 비움

JVM 메모리 구조

- Major GC(Full GC) : Old 영역에서 일어나는 GC
  - Old 영역에 있는 모든 객체들을 검사하며 참조되고 있는지 확인한다.
  - 참조되지 않은 객체들을 모아 한 번에 제거한다.
  - Minor GC보다 시간이 훨씬 많이 걸리고 실행중에 GC를 제외한 모든 쓰레드가 중지한다.

JVM 메모리 구조

- JVM 메모리 프로세스

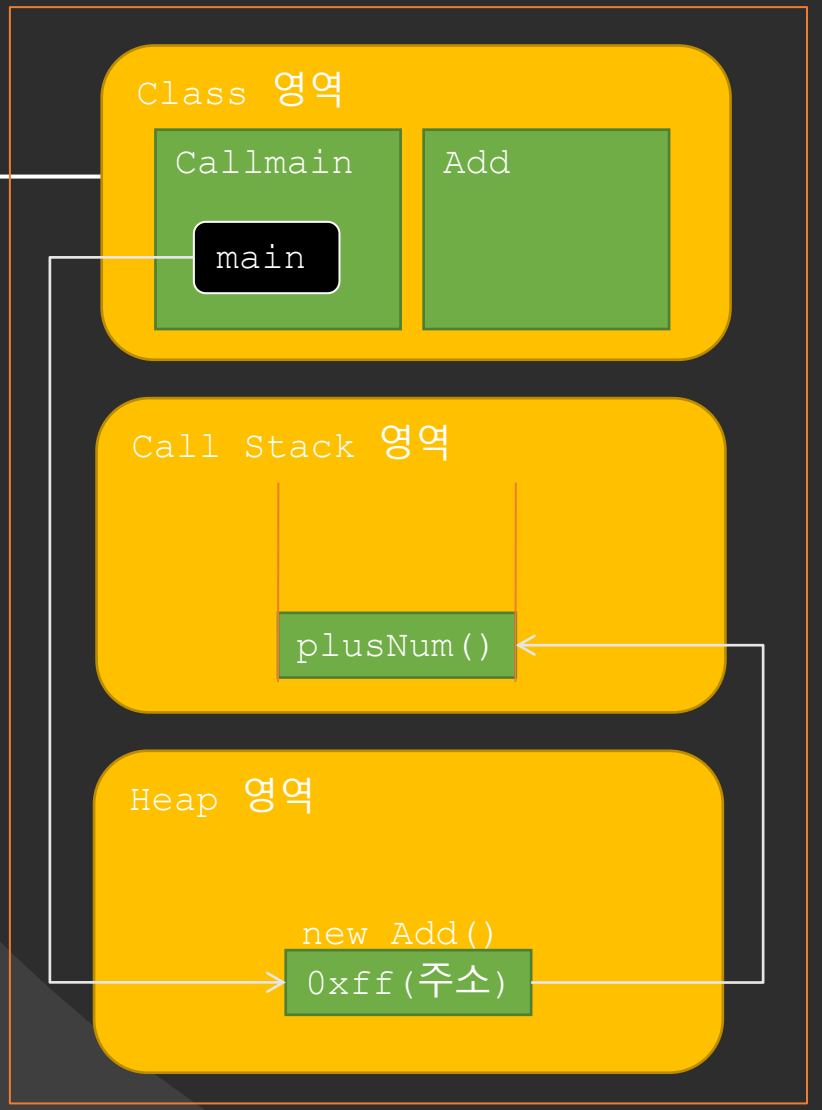
Ex)

```java
public class CallMain {

    public static void main(String[] args) {
        Add a = new Add();
        int x = a.plusNum(1, 2);
        System.out.println(x);
    }

}
public class Add {

    public int plusNum(int i, int j){
        return i+j;
    }
}
```



Static

- Static
  - 클래스 영역에 적재되는 메서드, 맴버변수들을 선언할 때 쓰임
  - static을 메서드나 맴버변수에 사용하게 되면 해당 맴버변수나 메서드는 모든 인스턴스가 공유한다.
  - 맴버변수는 static을 사용하게 되면 클래스 변수가 된다
  - 클래스 변수는 인스턴스를 생성하지 않고 사용 가능하다.
  - 클래스가 메모리에 로드될 때 클래스 변수가 생성된다.
  - 메서드에 static을 사용하게 되면 인스턴스를 생성하지 않고도 호출이 가능한 클래스 메서드가 된다.

Static

- Static Example

```java
public class Ex01 {
    // static
    public static void main(String[] args) {
        StaticClass sc = new StaticClass();
        sc.method1();
        sc.method2();
        System.out.println(sc.a1);
        System.out.println(sc.b1);
        System.out.println(sc.c1);
        sc.a1 = 130;

        StaticClass.method1();
        //StaticClass.method2(); //error
        System.out.println(StaticClass.a1);
        System.out.println(StaticClass.b1);
        //System.out.println(StaticClass.c1); //error
        System.out.println(sc.a1);
    }

}
```

```java
public class StaticClass {
    public static int a1 = 3;
    public static int b1;
    static {
        b1 = 5;
    }
    public int c1;

    public static void method1() {
        System.out.println("static method1");
    }

    public void method2() {
        System.out.println("normal method2");
    }
}
```

```
static method1
normal method2
3
5
0
static method1
130
5
130
```

Static

- Static 사용 시 주의할 점
  - static을 쓰게 되면 class영역에 자동으로 생성이 된다
  - static영역의 매개변수와 메서드는 해당 힙과 스택의 동작에는 상관없이 계속 JVM의 메모리에 남아있다
  - 만약 이러한 static변수나 메서드가 계속 메모리 상에 머물게 되면 class 메모리의 비중이 커져 시스템의 부하를 많이 잡아먹게 된다.
  - 이후 시스템의 과부하의 주 원인이 된다.

  ※ static은 공용상수나 공용메서드에서만 사용이 가능하도록 한다.

Final

- Final 이란
  - 클래스, 메서드, 맴버변수와 지역변수를 바꿀 수 없도록 할 때 쓰인다.
  - 변수에 사용하면 처음 선언을 제외한 값을 변경할 수 없는 상수가 된다.
  - 메서드에 사용되면 오버라이딩이 불가능하다.
  - 클래스에 사용되면 확장이 될 수 없다. 즉 조상클래스가 될 수 없다.
  - 해당 final을 사용할 시에는 접근성에 고려해서 써야 한다
  - 대부분 공동 프로젝트에서 상대방이 수정해서 사용할 요소를 배제하기 위해 사용된다.
  - 반드시 필요하다 싶은 요소 중 변경되지 말아야 할 메서드에 사용이 된다

Final

- Final 예제

```java
public class Ex02 extends FinalClass{
    // final
    final int x; // 변수의 상수화
    // 보통 상수 앞에는 public static final로 선언한다.
    public static final int MAX_COUNT = 12;

    public Ex02(int x) {
        this.x = x;
    }

    public void finalMethod() {

    }

    public static void main(String[] args) {
        Ex02 ex02 = new Ex02(2);
    }
}
```

```java
//public final class FinalClass { // 상속 불가
public class FinalClass {
    //public final void finalMethod() { // 오버라이딩 불가
    public void finalMethod() {

    }
}
```

접근 제한자

- 접근 제한자
  - 외부에서 접근 할 시에 해당 클래스와 맴버변수 메서드에 접근을 제한할 때 사용한다.
  - 대부분 공동 프로젝트에서 상대방이 수정해서 사용할 요소를 배제하기 위해 사용된다.

접근 제어자가 사용될 수 있는 곳 - 클래스, 멤버변수, 메서드, 생성자

private - **같은 클래스** 내에서만 접근이 가능하다.
default - **같은 패키지** 내에서만 접근이 가능하다.
protected - 같은 패키지 내에서, 그리고 다른 패키지의 자손클래스에서 접근이 가능하다.
public - 접근 제한이 전혀 없다.

접근 제한자

- 접근 제한자
  - 외부에서 접근 할 시에 해당 클래스와 맴버변수 메서드에 접근을 제한할 때 사용한다.
  - 대부분 공동 프로젝트에서 상대방이 수정해서 사용할 요소를 배제하기 위해 사용된다.

접근 제어자가 사용될 수 있는 곳 - 클래스, 멤버변수, 메서드, 생성자

private - **같은 클래스** 내에서만 접근이 가능하다.
default - **같은 패키지** 내에서만 접근이 가능하다.
protected - 같은 패키지 내에서, 그리고 다른 패키지의 자손클래스에서 접근이 가능하다.
public - 접근 제한이 전혀 없다.

접근 제한자

- 접근 제한자 사용 가능 유무

제어자	같은 클래스	같은 패키지	자손클래스	전 체
public				
protected				
default				
private				

접근 제한자

- 접근 제한자 예제

```
public class AccessClass {
    private int x;//private형
    protected void setX(int x){this.x = x;}//protected형
    int getX(){return x;}//void형
}
```

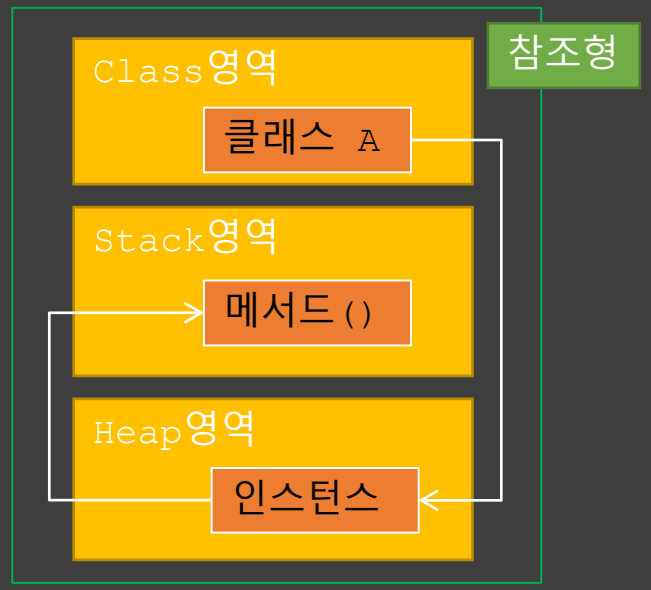
접근 제한자

대 상	사용가능한 제어자
클래스	public, (default), final, abstract
메서드	모든 접근 제어자, final, abstract, static
멤버변수	모든 접근 제어자, final, static
지역변수	final

- 제어자와의 조합
  1. 메서드에 static과 abstract를 함께 사용할 수 없다 : static메서드는 몸통(구현부)이 있는 메서드에만 사용할 수 있기 때문이다.
  2. 클래스에 abstract와 final을 동시에 사용할 수 없다 : 클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미이고, abstract는 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.
  3. abstract메서드의 접근제어자가 private일 수 없다 : abstract메서드는 자손클래스에서 구현해주어야 하는데 접근 제어자가 private이면, 자손클래스에서 접근할 수 없기 때문이다.
  4. 메서드에 private과 final을 같이 사용할 필요는 없다 : 접근 제어자가 private인 메서드는 오버라이딩될 수 없기 때문이다. 이 둘 중 하나만 사용해도 의미가 충분하다.

Call by value, Call by reference

- 주소를 담은 변수와 값을 담은 변수의 정의
  - 기본형 변수는 일반적으로 stack에 값이 있지만 참조형 변수는 heap영역 주소값을 가지고 있다.
  - 기본형 변수와 참조형 변수의 차이는 변수의 주소를 참조여부다..



Call by value, Call by reference

- 기본형 변수의 인자값 변화 예제

Ex)

```
public class CallByValue {

    public CallByValue(int a, int b) {
        a++;b++;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        new CallByValue(x,y);
        System.out.println("x : "+x+"   y : "+y);
    }

}
```

-기본형 변수를 다른 메서드에서 바꿔 입력해도 입력한 변수의 값은 바뀌지 않는다.

Result)

```
x : 1   y : 2
```

Call by value, Call by reference

- 참조형 변수의 인자값 변화 예제

Ex)

```
public class CallByReference {

    public CallByReference(StringBuffer sbm) {
        sbm.append(" World");
    }
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");
        new CallByReference(sb);
        System.out.println(sb.toString());
    }

}
```

-참조형 변수를 다른 메서드에서 바꿔 입력하면 참조형 변수의 값이 바뀌어서 출력되는 것을 볼 수 있다.
-참조형 변수는 해당 참조형 변수의 주소를 참조하기 때문이다.

Result)

```
Hello World
```