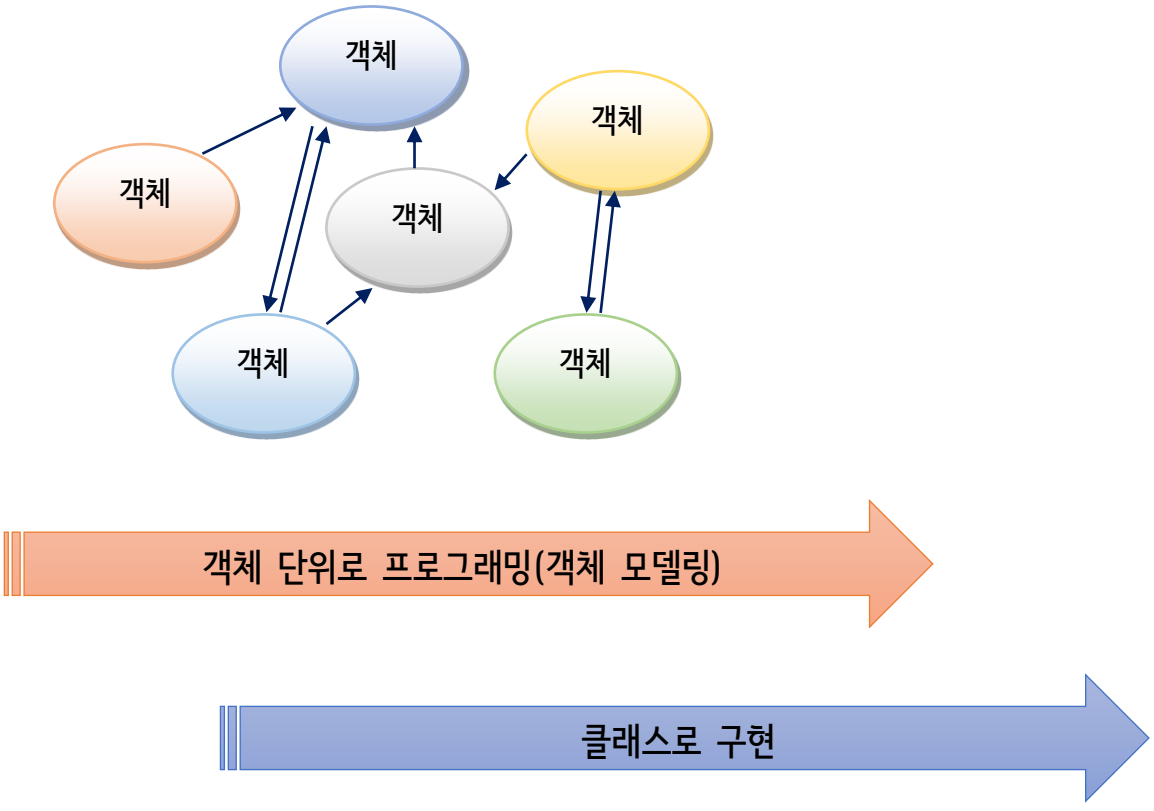
자바의 OOP 구문

객체지향 프로그래밍(Object-Oriented Programming, OOP)은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하려는 것이다. Java는 객체지향 프로그래밍언어로 분류되며 Java라는 언어로 프로그램을 개발한다는 것은 프로그램에서 요구되는 독립된 기능들을 객체들로 만드는 것을 의미하는데 각 기능의 객체들에 대하여 데이터와 행위로 나누어 설계한 것이 바로 클래스이다. 그리고 이 클래스라는 것은 Java 프로그램의 기본 구조이다.

■ 객체지향 프로그래밍

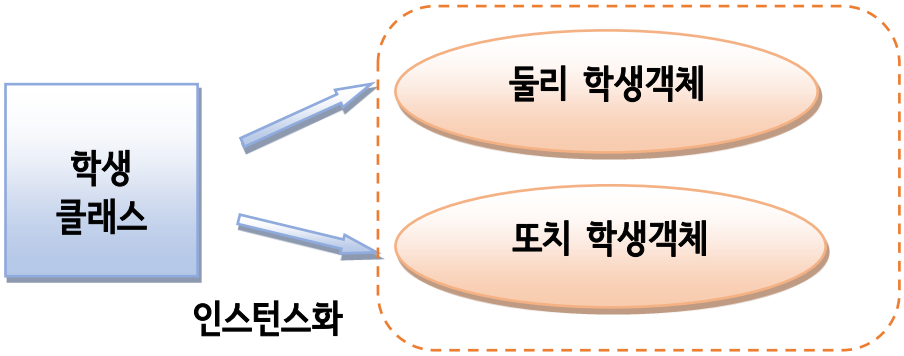
객체지향 프로그래밍(Object-Oriented Programming, OOP)은 컴퓨터 프로그래밍 패러다임의 하나로서 **실세계를 모델링하여 소프트웨어를 개발하는 방법**이다. 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다. 각각의 객체들은 메시지를 주고받으면서 데이터를 처리하게 된다.



■ 클래스

객체들은 객체들 고유의 속성(데이터)과 동작(behavior)을 가지고 있으며 서로 간에 상호 동작을 통해서 작업을 처리하게 된다. 클래스란 **'객체를 정의해 놓은 것'** 으로서 **'객체의 설계도 또는 템플릿'**이라고 정의할 수 있다. 클래스는 객체를 생성하는데 사용되며, 객체는 클래스에 정의된 대로 생성된다.

클래스로부터 객체를 만드는 과정을 클래스의 인스턴스화(instantiate)라고 하며, 어떠한 클래스로부터 만들어진 객체를 모두 그 클래스의 인스턴스(instance)라고 한다. 그러므로 객체와 인스턴스는 거의 비슷한 개념으로 사용된다.



학생 클래스로부터 생성된 인스턴스들-학생객체

프로그래밍에서의 객체는 클래스에 정의된 내용대로 메모리에 생성되어 사용 가능한 상태 및 수행 가능한 상태가 된 것을 뜻한다.

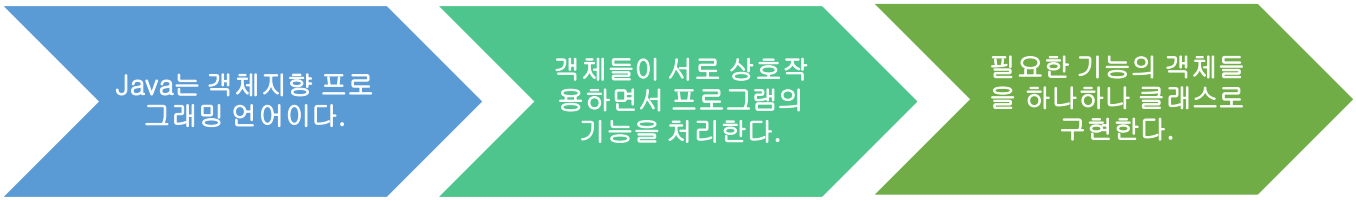
▶ 클래스의 구성 요소

클래스라는 객체를 만드는데 사용되는 설계도 또는 템플릿이다. 클래스를 만들 때는 어떠한 객체에 대한 설계도인가에 따라 객체에서 처리하게 될 데이터들은 변수로 객체에서 데이터를 처리하기 위해 필요한 행위는 메서드로 만든다. 바로 이 변수와 메서드가 클래스를 구성하는 요소들이며 클래스의 멤버라고 한다.

객체의 데이터 --→ 클래스의 변수
객체의 행위 --→ 클래스의 메서드

Java 프로그램의 구조

Java 프로그램의 기본 단위는 클래스이다. 이유는 Java는 완벽한 객체지향 프로그래밍 언어이기 때문이며 다음에 제시한 시나리오가 적용되기 때문이다.



Java에서의 클래스라는 단어의 의미는 2가지이다. 하나는 **객체를 설계한 프로그램의 구조**이며 다른 하나는 **Java만의 바이트 코드 형식으로 되어 있는 Java의 실행 파일**이다.

Java 프로그램의 소스 코드를 담은 파일을 소스파일이라고 하며 소스 파일의 명칭은 어떠한 명칭을 지정하든 확장자는 반드시 .java 여야만 한다. 그리고 그 안에는 하나의 클래스 또는 여러 개의 클래스를 정의할 수 있다. 그러나 가급적이면 클래스명과 그 클래스를 담게될 Java 프로그램의 소스 파일명은 대소문자까지 적용하여 동일하게 지정하는 것이 유지보수 측면에서 좋다.

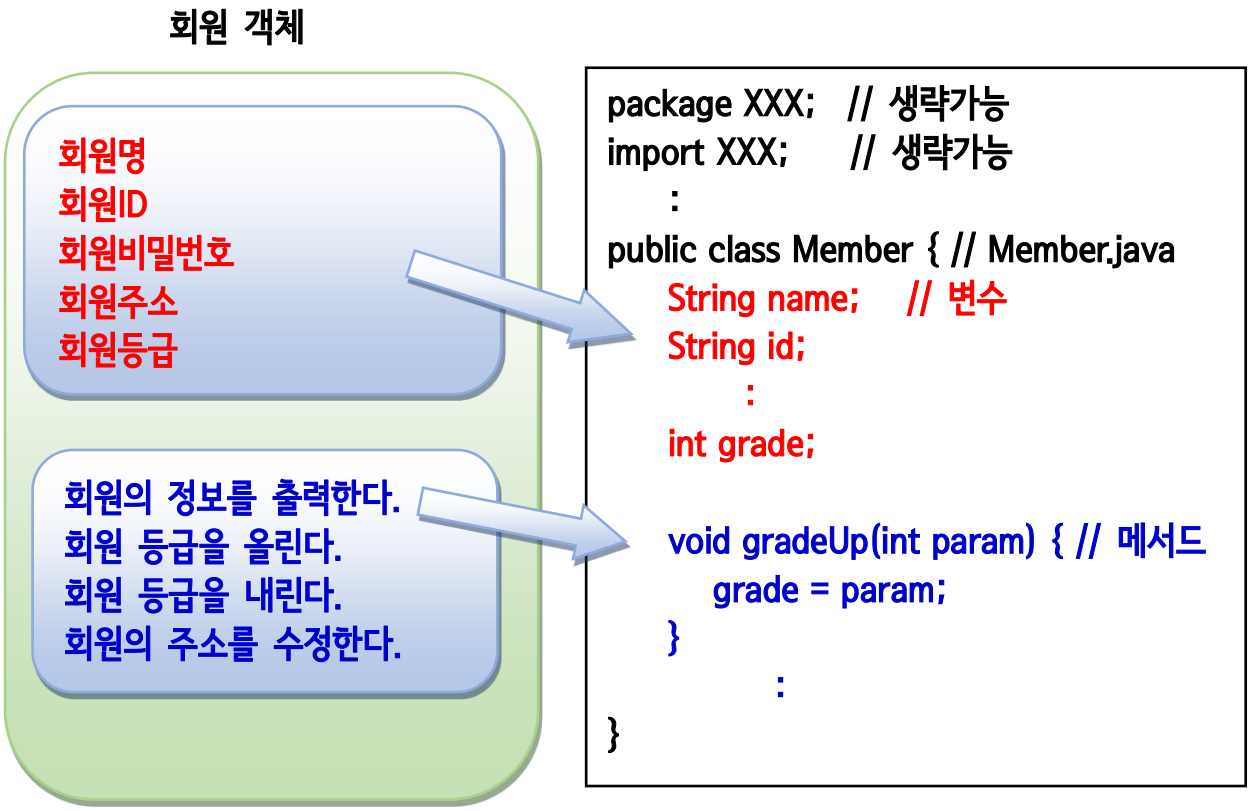
Java 프로그램의 소스 코드는 기본적으로 다음과 같은 순서로 구성된다.

```
[ 패키지 정의 ]
[ import 또는 import static 선언 ]
클래스 정의
[ 클래스 정의 ... ]
```

■ 클래스의 정의

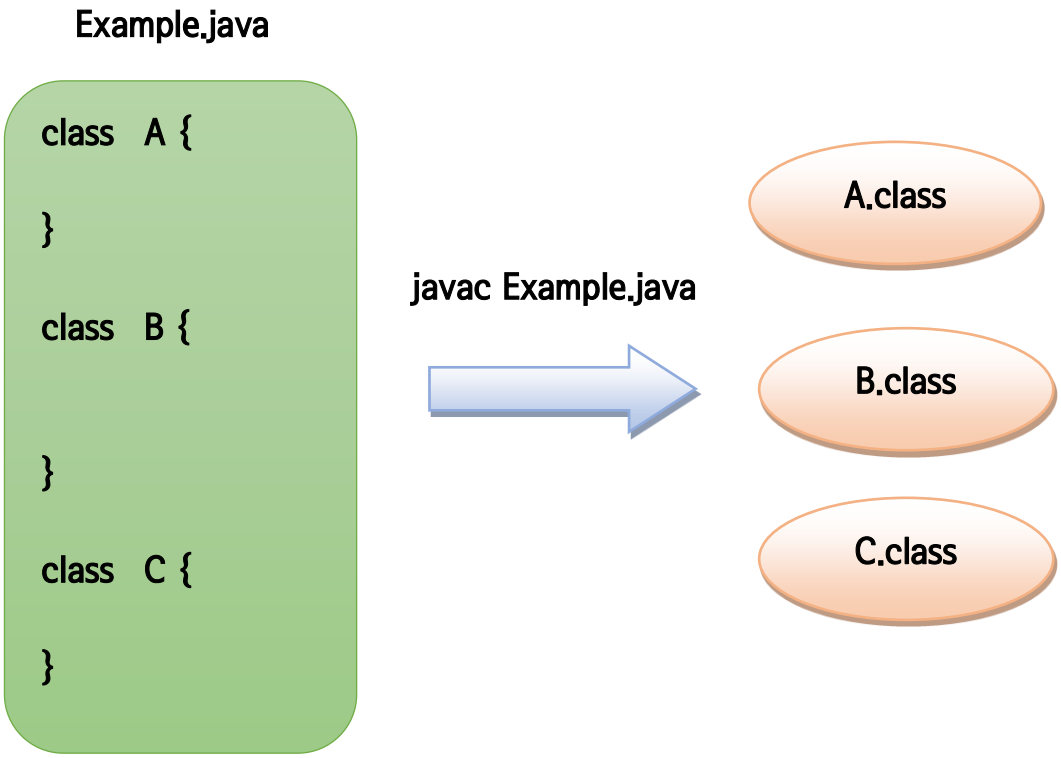
클래스를 만들 때는 이 클래스가 어떠한 객체에 대한 설계도(템플릿) 역할을 하게 될 것인지를 고려하여 객체의 데이터는 클래스의 변수로 객체의 행위는 메서드로 정의한다.

예를 들어 회원 객체에 대한 클래스를 만든다면 다음 그림과 같은 방식의 구현이 적용된다.



필요에 따라서는 하나의 Java 소스에 여러 개의 클래스를 정의할 수 있다. 다만 하나의 클래스만을 public으로 지정할 수 있다. Java 컴파일러가 주어진 Java 소스를 컴파일할 때는 소스 단위로 클래

스 파일을 만드는 것이 아니며 소스에 정의된 클래스 단위로 클래스 파일을 만들게 되므로 다음과 같이 하나의 Java 소스에 여러 개의 클래스가 정의되어 있는 경우에는 여러 개의 클래스 파일이 만들어진다.



■ 인스턴스의 활용

클래스로부터 객체를 만드는 과정을 인스턴스화(instantiate)라고 하며 어떤 클래스로부터 만들어진 객체를 인스턴스(instance)라고 한다. 생성되는 인스턴스는 멤버 연산자를 통해서 사용하게 되며 프로그램 수행에서 더 이상 필요하지 않은 인스턴스들은 적당한 시점에 JVM 에 의해 삭제된다.

포함(composite)관계에 기반한 클래스활용

다른 클래스의 기능을 포함하여 새로운 클래스를 만들기 위해서는 사용하고자 하는 클래스의 멤버가 클래스 멤버(static)인지 인스턴스 멤버(non-static)인지를 판단하여 클래스 멤버이면 인스턴스를 생성하지 않고 클래스명만으로 사용 가능하며 인스턴스 멤버인 경우에는 인스턴스를 생성해야 포함하고자 하는 기능을 사용할 수 있다.

■ 인스턴스 생성과 멤버(.) 연산자

인스턴스는 클래스로부터 만들어진 객체로서 클래스에 정의된 멤버들 중에서 인스턴스 멤버들을 메모리 할당하고 사용 가능한 상태로 만든 것이다. 인스턴스는 힙이라는 메모리 공간에 만들어지며 더 이상 사용되지 않는 인스턴스는 가비지 콜렉션(GC)이 발생되는 시점에서 JVM 에 의해 자동으로 메모리에서 삭제된다.

생성된 인스턴스를 한 번만 사용할 목적이라면 변수에 담아서 사용하는 것은 선택적이지만 여러 번 사용하기 위해서는 생성된 인스턴스를 변수에 담아서 사용한다. 이 때의 변수는 해당 인스턴스를 참조할 수 있는 타입의 변수여야 하며 일반적으로 다음과 같이 인스턴스의 클래스 타입과 동일한 타입으로 변수를 선언한다.

클래스에 따라서는 한 개의 생성자 메서드를 제공하는 경우도 있지만 여러 개로 오버로딩하여 제공하는 경우가 대부분이며 이러한 경우에는 정의되어 있는 생성자 메서드들 중에서 인스턴스를 생성하는 목적에 적합한 생성자를 선택하여 사용한다.

클래스의 인스턴스가 생성되면 다음과 같이 멤버 연산자(.)를 사용하여 인스턴스의 멤버를 사용할 수 있다.


```
String daytime = new Date().toString();

Button obj1 = new Button("종료");

String label = obj1.getLabel();

GregorianCalendar obj2 = new GregorianCalendar(1996, 0, 1);

int day = obj2 .get(Calendar.DAY_OF_YEAR);

Integer obj3 = new Integer(10);

int code = obj3.hashCode();

File obj4 = new File("c:₩Windows₩Notepad.exe");

int size = obj4.length();
```

■ static 멤버(클래스 멤버)의 활용

포함관계로 클래스를 활용할 때 사용하고자 하는 멤버가 static 형(클래스 형)인 경우에는 인스턴스를 생성하지 않고 클래스명과 멤버 연산자로 변수에 접근하거나 메서드를 호출할 수 있다.

```
int num = Integer.parseInt("123");
double  randomSu = Math.random();
System.exit(0);
System.out.println("자바");
System.out.println(Integer.MAX_VALUE);
Thread.sleep(1000);
System.gc();
Collections.sort();
```

생성자 메서드의 구현과 메서드 오버로딩

구현하고자 하는 클래스의 기능에 따라 생성자 메서드의 기능과 호출시 전달되는 파라미터의 사양을 다양하게 정의할 수 있다. Java 에서 정의되는 메서드들은 오버로딩 구문이 적용될 수 있으며 생성자 메서드도 메서드이기 때문에 여러 개로 오버로딩하여 구현 가능하며 this 변수와 this() 메서드를 사용하여 효율적인 구현을 할 수 있다.

■ 생성자 메서드의 구현

클래스에는 생성자 메서드가 반드시 한 개 이상 정의되어 있어야 한다. 그러므로 클래스의 소스상에 생성자가 구현되어 있지 않은 경우에는 컴파일러에 의해서 파라미터를 받지 않은 생성자가 자동으로 생성된다. 컴파일러에 의해서 생성되는 생성자를 디폴트 생성자라고 한다.

```
public class Student{
    int studentNumber;       // 인스턴스 생성시 0 으로 자동 초기화 된다.
    String studentName;  // 인스턴스 생성시 null 로 자동 초기화 된다.
    String className;        // 인스턴스 생성시 null 로 자동 초기화 된다.
}

// 컴파일러에 의해 생성된 디폴트 생성자로 Student 의 인스턴스를 생성한다.
Student obj = new Student();
System.out.println("학생번호 : "+obj.studentNumber);
System.out.println("학생이름 : "+obj.studentName);
System.out.println("학생반명 : "+obj.className);
```

생성자 메서드는 클래스를 인스턴스화할 때 호출되는 인스턴스 초기화 메서드로서 메서드명은 클래스 명과 동일하며 리턴 값이 없는 메서드로 정해져 있기 때문에 리턴 값의 유형을 생략해야 한다.

```
[접근제어자] 클래스명([매개변수 선언]) {
    :
}
```

```
public class Example {
    public Example() {
        :
    }
}
```

다음은 생성자 메서드를 정의하고 있는 클래스의 예제이다.

```
// Student 클래스에 생성자 메서드가 정의되어 있지 않으므로 Student(){  } 와
// 같은 디폴트 생성자가 자동으로 생성되지 않는다.
public class Student{
    int studentNumber;
    String studentName;
    String className;
    public Student(int num, String name) {
        studentNumber = num;
        studentName = name;
    }
}

Student obj = new Student();    // 오류 발생
// 클래스에 정의되어 있는 생성자로 Student 의 인스턴스를 생성한다.
Student obj = new Student(1,"자바" );
System.out.println("학생번호 : "+obj.studentNumber);
System.out.println("학생이름 : "+obj.studentName);
System.out.println("학생반명 : "+obj.className);
```

▶ 생성자 메서드의 오버로딩

생성자가 구현되어 있는 클래스의 경우에는 디폴트 생성자가 자동 생성되지 않는다. 파라미터를 선택적으로 전달하여 인스턴스 생성이 가능하도록 하려면 생성자 메서드를 여러 개 오버로딩하여 구현하는 것도 가능하다.

생성자 메서드를 오버로딩한다는 것은 매개변수 선언을 달리하여 생성자 메서드를 여러 개 정의하는 것을 말한다. 여러 개의 생성자 메서드를 제공하는 클래스는 정의되어 있는 생성자의 사양과 인스턴스 생성 목적에 따라 다양한 파라미터를 전달할 수 있다. 다음은 Student 클래스의 생성자 메서드를 세 개로 오버로딩한 예제이다.

```
// Student 클래스에 생성자 메서드가 여러 개로 오버로딩되어 있다.
public class Student{
    int studentNumber;
    String studentName;
    String className;
    public Student() {
        studentNumber = 0;
        studentName = "";
        className = "미정";
    }
    public Student(int num, String name) {
        studentNumber = num;
        studentName = name;
        className = "미정";
    }
    public Student(int num, String name, String cname) {
        studentNumber = num;
        studentName = name;
        className = cname;
    }
}

Student obj1= new Student();
System.out.println("학생번호 : "+obj1.studentNumber);
System.out.println("학생이름 : "+obj1.studentName);
System.out.println("학생반명 : "+obj1.className);

Student obj2 = new Student(1,"자바", "에디슨");
System.out.println("학생번호 : "+obj2.studentNumber);
System.out.println("학생이름 : "+obj2.studentName);
System.out.println("학생반명 : "+obj2.className);
```

▶ this

생성자 메서드와 인스턴스 메서드에서 사용 가능한 변수로서 클래스의 인스턴스 생성시에 자기 자신에 대한 인스턴스의 참조 값을 갖도록 자동으로 메모리 할당되고 초기화되는 변수이다.

주로 지역 변수와 멤버 변수를 구분하는 용도로 사용한다. 메서드내에서 사용되는 변수는 가까이 선언되어 있는 변수가 우선적으로 사용되므로 지역 변수와 멤버 변수명이 동일할 때는 무조건 지역 변수를 사용하게 되는 결과가 되며 이 때 멤버 변수를 사용하려면 변수명 앞에 this 라는 참조형 변수를 사용하여 접근한다.

다음은 Strudent 클래스의 생성자 메서드에서 this 를 사용한 예이다.

```
public class Student{
    int studentNumber;
    String studentName;
    String className;
    public Student(int studentNumber, String studentName) {
        // 매개 변수와 멤버 변수를 구분하기 위해 this 를 사용하고 있다.
        this.studentNumber = studentNumber;
        this.studentName = studentName;
        className = "미정";
    }
    public Student(int num, String name, String cname) {
        studentNumber = num;
        studentName = name;
        className = cname;
    }
}
```

▶ this()

오버로딩을 적용하여 생성자 메서드들을 구현할 때 여러 생성자에 동일한 수행 문장이 반복하여 작성되는 경우가 많다 이런 경우에는 this() 라는 생성자 호출 기능의 메서드를 사용하여 생성자 메서드 작성을 효율적으로 하고 코드의 재사용성을 높일 수 있다.

this() 메서드는 같은 클래스의 다른 생성자를 호출할 때 사용 가능하며 반드시 첫 번째 수행 문장으로만 호출될 수 있고 생성자 메서드에서만 사용 가능하다.

다음은 Strudent 클래스의 생성자 메서드에서 this() 를 사용한 예이다.

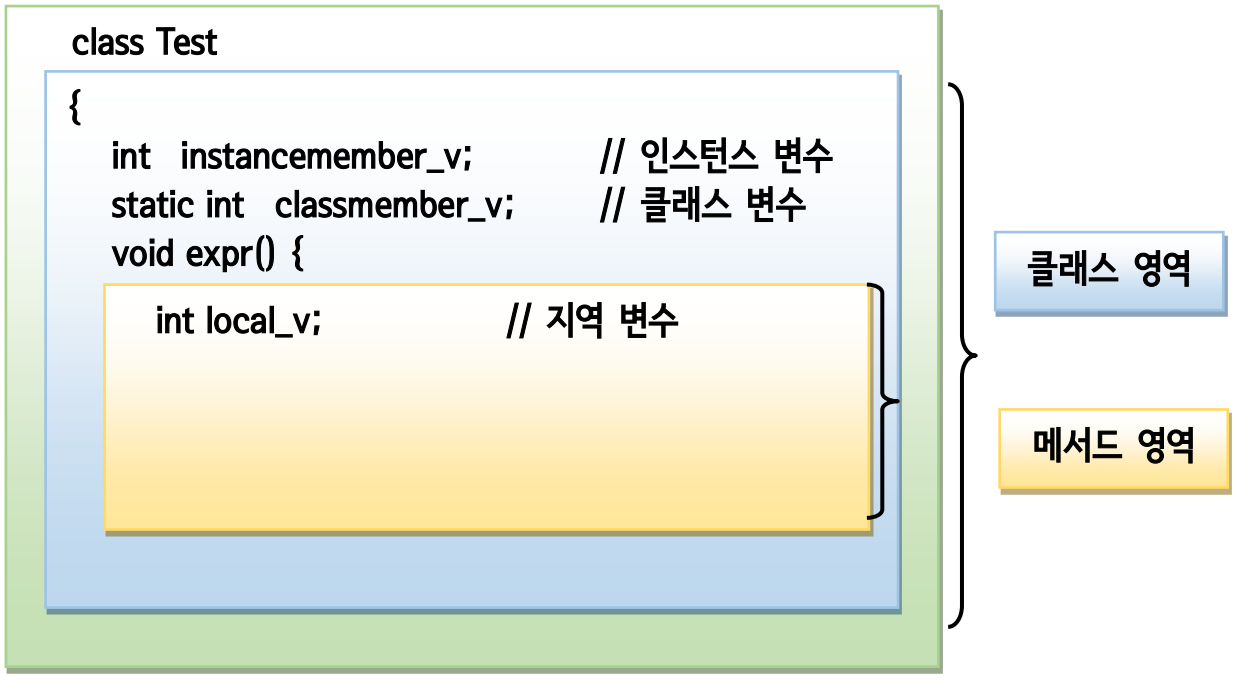
```
public class Student{
    int studentNumber;
    String studentName;
    String className;
    public Student() {
        this(0, "", "미정");   // 첫 번째 행으로만 호출 가능하다.
    }
    public Student(int studentNumber, String studentName) {
        this(studentNumber, studentName, "미정");
    }
    public Student(int num, String name, String cname) {
        studentNumber = num;
        studentName = name;
        className = cname;
    }
}
```

호출

호출

■ 선언 위치에 따른 변수의 종류

Java 에서의 변수는 클래스 변수, 인스턴스 변수 그리고 지역변수(매개변수)가 있다. 변수가 선언된 위치에 따라서 이렇게 세 가지로 나뉘며 어떠한 변수인가에 따라서 해당 변수의 메모리 할당 시기와 사용 범위(scope)가 결정된다.



▶ 인스턴스 변수

클래스 영역에 선언되는 클래스의 멤버 변수로서 인스턴스 생성시 메모리 영역이 할당되어 사용 가능한 상태가 되는 변수이다. 클래스의 인스턴스가 생성될 때마다 메모리 영역을 각각 할당되어 사용되므로 "인스턴스에 속하는 변수다" 또는 "인스턴스 생성을 통해서 사용 가능한 상태가 되는 변수다 " 하여 인스턴스 변수라고 부른다. 인스턴스가 메모리에서 삭제되는 시점에 메모리에서 인스턴스 변수의 영역이 해제된다. 클래스 내의 모든 인스턴스 메서드와 생성자 메서드에서 바로 접근할 수 있으며 다른 클래스에서 사용할 때는 **'참조형변수.멤버변수'** 와 같은 형식으로 사용되는 변수이다.

▶ 클래스 변수

클래스 영역에 선언되는 클래스의 멤버 변수로서 변수를 선언할 때 static 제어자가 설정되어 있는 경우 클래스 변수라고 한다. 인스턴스 생성과는 관계없이 JVM 이 클래스를 로딩할 때 메모리 영역이 할당되어 사용 가능한 상태가 되는 변수로서 할당된 메모리 영역은 프로그램이 종료될 때까지 유지된다. 클래스의 인스턴스가 여러 번 생성되더라도 클래스 변수와는 관계없으며 클래스 로딩시 할당된 클래스 변수를 여러 인스턴스가 공유하는 결과가 된다. 클래스 변수를 사용할 때는 어떠한 클래스에 속하는 변수인지를 나타내기 위해 **'클래스명.멤버변수'**와 같은 형식으로 사용하여 접근한다.

▶ 지역 변수

메서드, 생성자 메서드 그리고 초기화 블록 등에서 선언되는 변수로서 해당 변수 선언문이 수행될 때 메모리 영역이 생성되었다가 메서드가 수행이 종료되면 해제되는 변수이다. 선언된 블록내에서만 사용 가능하며 블록을 벗어나면 자동적으로 메모리 영역이 해제된다.

[클래스 변수와 인스턴스 변수의 메모리 할당 구조 그림]

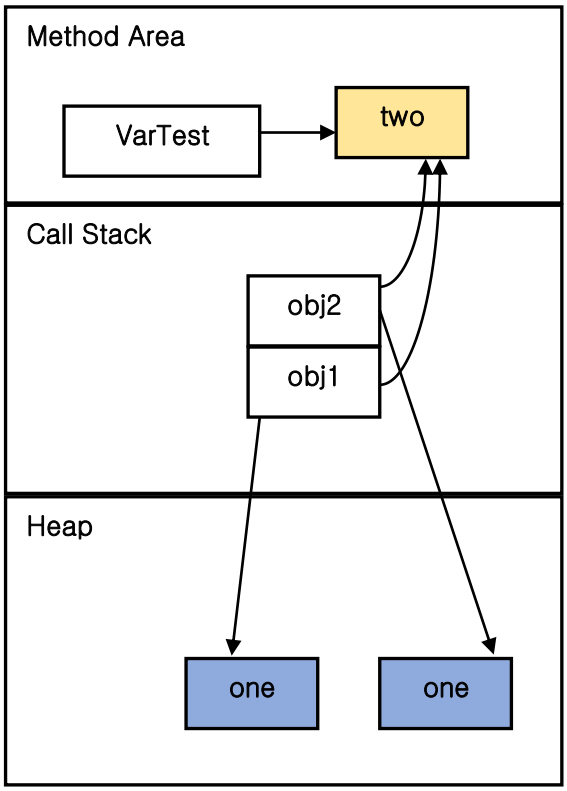
클래스의 내용이 JVM 메모리상에 올려지는 것을 클래스로딩이라고 한다. 클래스가 로딩되는 과정에서 클래스에 정의되어 있는 static 형(클래스형) 변수들의 공간이 Method Area 영역에 할당되며 이 영역은 프로그램의 수행이 끝날 때까지 고정된다.

클래스의 인스턴스 생성시에 클래스에 정의되어 있는 non-static 형(인스턴스형) 변수들의 공간이 Heap 영역에 할당되는데 인스턴스를 생성할 때마다 할당된다.

클래스 변수는 해당 클래스의 인스턴스를 참조하는 변수로도 접근할 수 있으며 클래스 명으로도 전근할 수 있지만 인스턴스 변수는 반드시 해당 인스턴스를 참조하는 변수를 통해서만 접근할 수 있다.

```
class   VarTest {
    int   one = 10;
    static int two = 20;
}

VarTest obj1 = new VarTest();
VarTest obj2 = new VarTest();
obj1.one++;
obj1.two++;
System.out.println(obj1.one);
System.out.println(obj2.one);
System.out.println(obj1.two);
System.out.println(obj2.two);
//오류
System.out.println(VarTest.one);
System.out.println(VarTest.two);
```



■ 클래스 메서드에서의 동일 클래스의 멤버 활용

클래스 로딩이라는 과정은 Java 프로그램이 수행하는 동안 수행하는데 있어서 추가로 필요한 클래스가 인식될 때마다 발생한다. 어떠한 클래스든 클래스 로딩이라는 과정은 한 번 발생하게 되며 이 때 클래스에 정의되어 있는 static 관련 요소들, 클래스 변수, 클래스 메서드 그리고 static 초기화 블록에 대하여 다음과 같은 기능이 수행된다.

1. 클래스 변수의 공간이 Method Area 영역에 할당된다.
2. 클래스 메서드들이 호출 가능한 상태가 된다.
3. static 초기화 블록의 구현 내용이 수행된다.

▶ 클래스 메서드

클래스 메서드는 클래스 로딩시에 자동으로 호출 가능한 상태가 되는 메서드로서 대표적인 클래스 메서드는 main() 메서드가 있다. 클래스 메서드의 경우에는 클래스의 인스턴스를 생성하지 않고도 클래스명으로 바로 호출할 수 있기 때문에 메서드의 사용이라는 측면에서 간단하게 메서드 사용이 가능하다.

클래스 메서드에서는 동일 클래스에 존재하는 다른 클래스 멤버들은 사용 가능하지만 인스턴스 멤버들을 사용할 수 없다.

클래스 메서드의 정의 기준은 메서드가 수행하는데 있어서 동일 클래스에 존재하는 인스턴스 멤버를 사용하는지 여부에 의해 정할 수 있다. 동일 클래스에 존재하는 인스턴스 멤버를 사용하여 수행하는 메서드인 경우에는 인스턴스 메서드로 해야 하며 그렇지 않고 단독 수행이 가능한 경우에만 클래스 메서드로 설계해야 한다.

다음은 클래스 메서드와 인스턴스 메서드를 정의하고 있는 클래스를 활용하는 예제이다.

```
class  SumExam {
    long a, b;
    long add() {        // 인스턴스메서드
        log();
        return a + b;
    }
    static long add(long a, long b) { // 클래스메서드(static메서드)
        log();   // 오류. 클래스 메서드에서 인스턴스 메서드를 직접 호출 불가능
        return a + b;
    }
    private void log() { // 인스턴스메서드
        System.out.println("log : add() 메서드 수행!!");
    }
}
class  SumTest {
    public static void main(String args[]) {
        System.out.println(SumExam.add(200L,100L); // 클래스메서드 호출
        SumExam  obj = new  SumExam(); // 인스턴스 생성
        obj.a = 200L;
        obj.b = 100L;
        System.out.println(obj.add()); // 인스턴스메서드 호출
    }
}
```

■ 변수 초기화

변수의 초기화란 변수를 선언하고 처음으로 값을 저장하는 것을 의미하며 변수의 종류에 따라서 초기화하는 다양한 방법을 적용할 수 있다. 자동 초기화란 변수를 선언하고 값을 저장하지 않아도 변수의 메모리 영역이 할당될 때 자동적으로 값이 저장되는 것을 의미하며 명시적 초기화란 변수를 선언하면서 대입연산자로 값을 저장하는 것을 의미한다.

변수의 종류	초기화 방법
클래스 변수	(1) 따로 초기화를 구현하지 않으면 기본값으로 자동 초기화한다. (2) 변수 선언시에 대입 연산자로 명시적으로 초기화한다. (3) 복잡한 초기화 연산을 필요로 하는 경우에는 static 블록을 사용한다.
인스턴스 변수	(1) 따로 초기화를 구현하지 않으면 기본값으로 자동 초기화한다. (2) 변수 선언시에 대입 연산자로 명시적으로 초기화한다. (3) 생성자 메서드를 통해 전달된 파라미터를 사용하여 초기화한다. (3) 인스턴스 블록의 수행을 통해서 초기화한다.
지역 변수	(1) 지역변수는 자동초기화를 지원하지 않는다. (2) 변수 선언시에 대입 연산자로 명시적으로 초기화한다. (3) 선언만 해놓고 변수 사용전에 대입 연산자로 초기화한다.

▶ static 블록과 인스턴스 블록

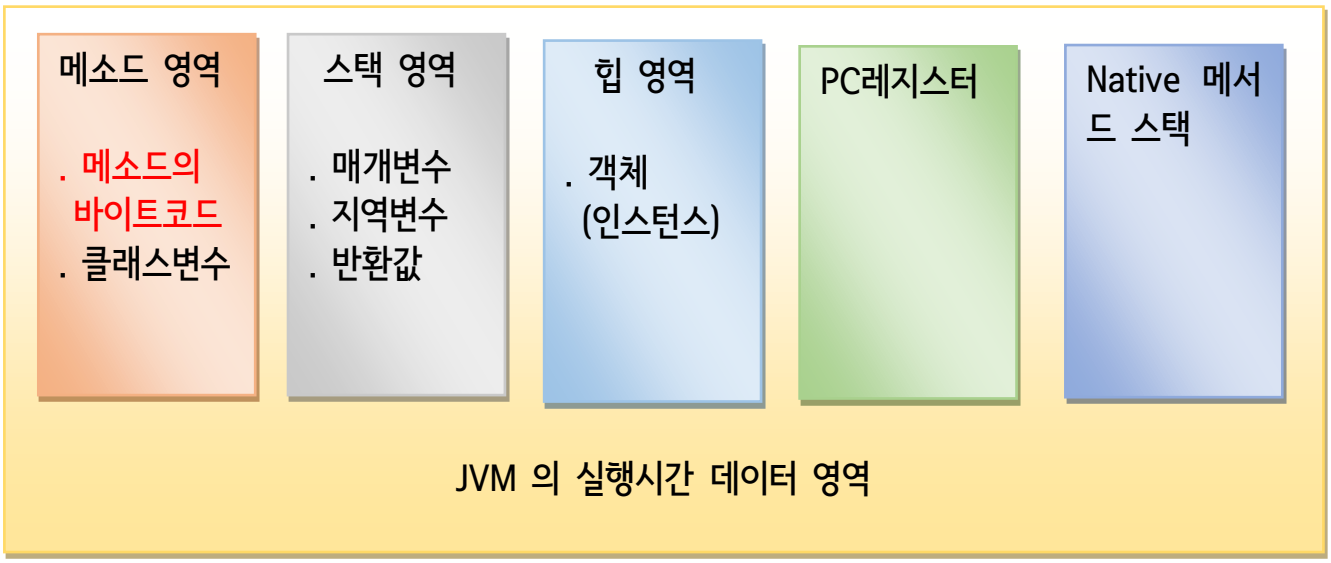
static 블록은 클래스가 로딩되는 시점에 수행되는 코드를 작성하는 블록이며 인스턴스 블록은 인스턴스 생성시 수행되는 코드를 작성하는 블록이다. 인스턴스 생성시 호출되어 수행되는 코드를 작성하는 메서드로 생성자 메서드라는 것도 있지만 생성자 메서드가 여러 개로 오버로딩되어 있고 여러 생성자에 공통으로 들어갈 수행코드의 일부가 있는 경우라면 인스턴스 블록을 사용하는 것도 방법이 될 수 있다.

다음은 static 블록과 인스턴스 블록이 작성된 클래스의 예이다.

```
class   Block {
    static {                                        // 클래스 로딩 시 수행되는 블록
        System.out.println("static 블록 수행");
    }
    {                                               // 인스턴스 생성 시 수행되는 블록
        System.out.println("인스턴스 블록 수행");
    }
    Block() {                                       // 인스턴스 생성 시 호출되는 생성자
        System.out.println("생성자 수행");
    }
}
Block obj1 = new Block();
Block obj2 = new Block();
```

가비지 컬렉션(GC)

■ JVM 의 메모리 구조



[메서드 영역(Method Area)]

로딩된 클래스의 메서드 코드가 저장되고 클래스 변수의 공간이 만들어지는 영역으로서 여러 객체에서 공유되는 영역이고 프로그램 수행이 끝날 때까지 고정되는 영역이다.

이 영역은 JVM에서 실행되고 있는 모든 스레드에 의해 공유되며 JVM은 여러 개의 스레드가 메서드를 정상적으로 사용하기 위해 동기화(synchronization)기법을 제공한다.

[힙 영역(Heap Area)]

프로그램 상에서 데이터를 저장하기 위해 동적으로(실행시간에) 할당하여 쓸 수 있는 메모리 영역 클래스의 인스턴스(객체)가 만들어지는 영역이다. 여러 스레드들이 공유하는 영역이다.

[스택 영역(Stack Area)]

메서드가 호출될 때마다 스택 프레임이라는 데이터 영역이 생성되며, 이것이 쌓여 스택을 구성한다. 수행되는 메서드 정보, 로컬변수, 매개변수, 연산 중 발생하는 임시데이터 등이 저장되는 공간이며 메소드가 수행되는 동안 필요로 되며 메서드의 수행이 끝나면 자동으로 해제된다.
즉 메서드가 호출될 때 필요로 되는 변수들을 스택에 저장하고, 메서드 실행이 끝나면 스택을 반환한다. 스레드별로 각각의 스택 영역을 가지게 된다.

스택에 있는 제일 위의 프레임은 현재 수행중인 메서드의 정보를 가지게 되며 바로 아래에 존재하는 스택 프레임의 메서드에 의해서 호출된 상태가 된다.

[Native 메서드 스택]

native 메서드는 Java 가 아닌 기존의 다른 언어에서 제공되는 메서드를 의미한다. 그러한 메서드의 매개변수, 지역변수 등을 저장하는 영역이다.

[PC 레지스터(register)]
스레드가 시작할 때 생성되어 JVM이 현재 수행할 명령어의 주소를 저장한다.

■ JVM 의 GC(가비지 컬렉션, Garbage Collection)

GC는 Garbage Collection의 약자로 Java 언어의 중요한 특징중의 하나이다. GC는 Java 프로그램에서 사용하지 않는 메모리를 자동으로 수거하는 기능을 말한다. 더 이상 사용되지 않는 메모리 공간을 JVM 이 관리해주므로 개발자는 메모리 관리로부터 좀더 자유롭게 되었다.

GC 가 수행되는 영역은 힙 영역으로서 더 이상 참조되지 않는 인스턴스의 공간을 모아서 사용 영역을 해제하고 사용 가능한 상태로 만드는 것이다. 가비지 컬렉터는 주 프로그램에 대하여 분리된 스레드처럼 독립적으로 실행되며, 사용하지 않는 온갖 메모리를 자동으로 해제한다. **더 이상 참조되지 않는 인스턴스를 찾는 방법은 참조 계수(reference-counting)를 점검한다. 참조 계수가 0 이라는 것은 이 인스턴스를 참조하는 변수가 존재하지 않는다는 것을 뜻한다.**

일반적으로 Application에서 사용되는 인스턴스는 오래 유지되는 인스턴스보다, 생성된 후에 얼마 지나지 않아 사용되지 않는 경우가 많다. 하여 JVM 은 힙이라는 영역에서도 NEW/Young 이라는 영역과 Old 라는 영역으로 인스턴스(객체)들을 나누어 보관하는데 새로이 생성되거나 생성된지 얼마 되지 않은 인스턴스들을 Young 영역에 보관하고 오래된 인스턴스들은 Old 영역에 보관한다.

GC 는 주로 NEW/Young 영역에서 일어나며 이 영역에서 일어나는 GC 를 Minor GC라고 하며 Old 영역에서 일어나는 GC 를 Full GC 라고 한다. Minor GC 는 짧은 시간 수행하지만 Full GC 는 속도가 느리며, Full GC가 일어나는 도중에는 순간적으로 Java Application이 멈춰 버리는 현상이 발생할 수도 있다.

▶ System.gc()

JVM 에게 GC수행을 요청하는 기능의 메서드로서 Runtime.getRuntime().gc() 와 동일한 기능이다. Full GC 를 수행하도록 하므로 꼭 필요한 경우가 아니라면 가급적 사용하지 않는 것이 좋다.

▶ finalize() 메서드

메모리에서 객체가 소멸될 때 가비지 컬렉터에 의해서 호출되는 기능의 메서드이다. **인스턴스(객체)가 소멸될 때 수행되어야 하는 코드가 있으면 이 메서드를 오버라이딩하여 구현한다.** finalize() 메서드는 최상위 클래스인 Object 클래스가 정의하고 있는 메서드이다.