JAVA CLASS & OBJECT

프로그램은 사람이 이해하는 코드를 작성.

느려도 꾸준하면 경기에서 이긴다.

작성자 : 홍효상
이메일 : hyomee@naver.com
소스 : https://github.com/hyomee/JAVA_EDU

Content

5. Class & Object

1. 객체지향프로그램(OOPL)
2. Class
3. Field
4. Method
5. Constructor
6. 자바 제어자

1. 객체지향 프로그램 (OOPL)

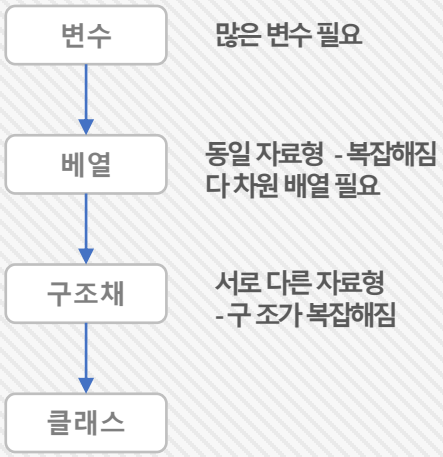
" 객체 지향 프로그래밍(OOP) 모델을 기반으로 하는 고급 프로그래밍 언어 "

객체지향 프로그램 언어 (Object-Oriented Programming Language)

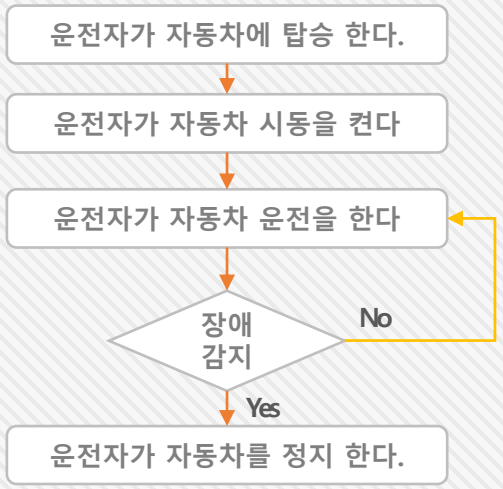
- 논리적 클래스, 객체, 메소드, 관계 및 기타 프로세스를 소프트웨어 및 애플리케이션 설계와 통합
- 객체 지향 프로그래밍 언어의 프로그래밍 구문은 하나 이상의 객체를 기반
  절차적 언어는 논리적 절차
- 데이터 추상화, 상속, 캡슐화, 클래스 생성 및 관련 개체를 포함하는 기본 개체 지향 기능을 나타내야 한다.

01. 왜 Class가 만들어 졌을까 ?

- Data 관점
  - 절차적 언어 문법으로 복잡한 Data를 표시에 한계 있음



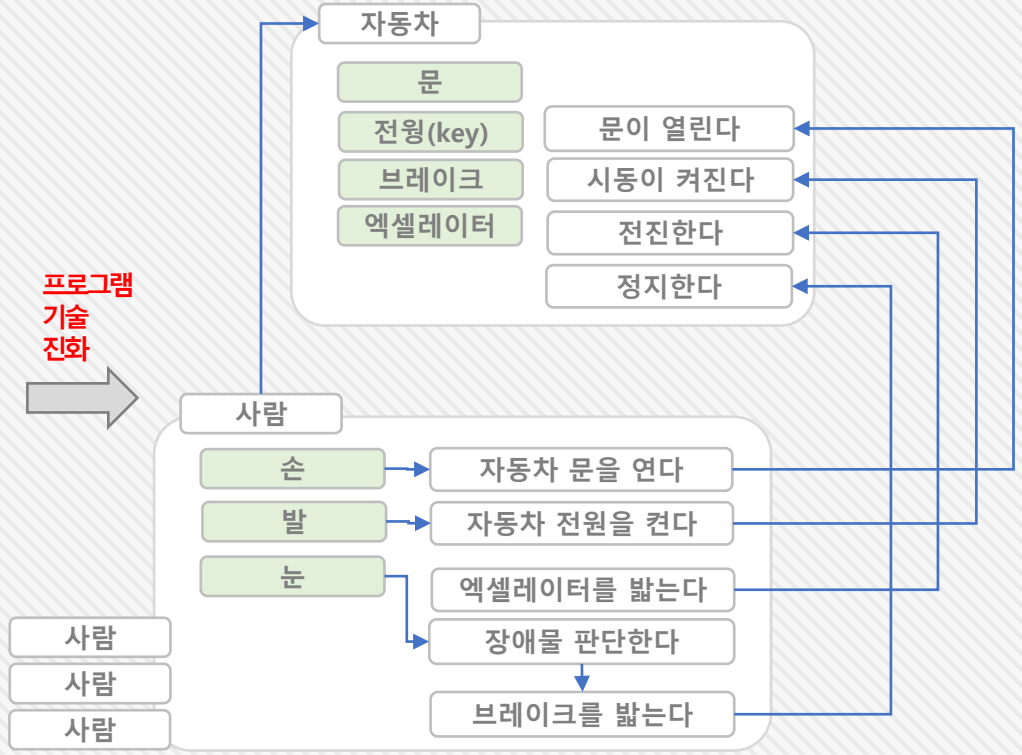
- 처리 관점
  - 순차적으로 처리하는 기능 중심의 프로그램은 프로그램이 복잡해짐



자동차의 기능과 사람의기능이 혼재 되어 있음

다른 사람이 운전을 한다면 ...

프로그램 기술 진화



1. 객체지향 프로그램 (OOPL)

" 객체 지향 프로그래밍(OOP) 모델을 기반으로 하는 고급 프로그래밍 언어 "

OOP 3대 요소

- **캡슐화(Encapsulation) = 정보 은닉**
  - 프로그램 내에서 같은 기능의 목적으로 작성된 코드를 모아서 다른 곳 (Class) 에서 안 보이게 숨기는 것
  - Class 속성을 숨기고 (private), 공개 (public)

- **상속 (Inheritance) = 재사용 + 확장**
  - Class와 Class관계 정의 (부모와 자식)
  - 자식 Class는 부모 Class 속성 및 기능을 사용 할 수 있음

- **다양성(polymorphism; 폴리모피즘) = 사용 편의**
  - 하나의 객체가 여러 가지 형태를 가질 수 있는 것.
  - 오버라이딩(재정의 : Overriding), 오버로딩(기능확장 : Overloading)

OOP 5원칙 (SOLID)

- **단일 책임 원칙(Single Responsibility Principle)**
  - 모든 클래스는 각각 하나의 책임만 가져야 한다.
  - 특수한 목적을 수행하도록 만든 클래스는 해당 목적 외에 다른 기능을 수행하면 안된다.

- **개방-폐쇄 원칙(Open Closed Principle)**
  - 클래스는 확장에는 열려 있고 수정에는 닫혀 있어야 한다.
  - 기존의 코드를 변경하지 않으면서 기능을 추가할 수 있도록 설계 되어야함

- **리스코프 치환 원칙(Liskov Substitution Principle)**
  - 자식 클래스는 언제나 자신의 부모 클래스를 대체할 수 있어야 한다
  - 자식 클래스는 부모 클래스의 책임을 무시하거나 재정의하지 않고 확장만 수행하도록 해야 한다.

- **인터페이스 분리 원칙(Interface Segregation Principle)**
  - 한 클래스는 자신이 사용하지 않는 인터페이스는 구현하지 말아야 함.
  - 하나의 일반적인 인터페이스보다 여러 개의 구체적인 인터페이스가 낫다

- **의존 관계 역전 원칙(Dependency Inversion Principle)**
  - 의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것보다는 변화하기 어려운 것, 거의 변화가 없는 것에 의존해야 한다.
  - 구체적인 클래스보다는 인터페이스나 추상 클래스와 관계를 맺어야 한다.

1. Class

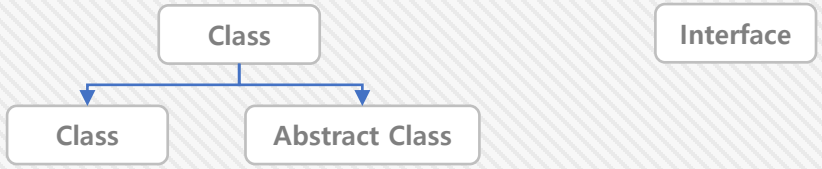
"객체 지향의 가장 기본적인 구조는 Class, 대등한 요소로는 Instance"

Class

- **Class :** 분류 또는 종류라고 하는 동종의 모임
- **Instance :** 구체적인 것 (Class의 생성자로 객체를 만드는 과정 : Instance 화)

문법 요소

- **Class :** 일반 Class, 추상 Class
- **Interface**



Class 구조

```
package ....
import ....
class 클래스명 { ... }

public class 클래스명 {

    int member = 5;                          // 필드

    public 클래스명 (매개변수) { ... }       // 생성자

    public 자료형 메서드명(매개변수) { ..... } // 메서드

    class 이너클래스명 { .... }              // innerClass

}
```

클래스 외부 구성 요소

클래스 내부 구성 요소

Class 구조 설명

- **외부 구성 요소**
  - **package** : 자바 Class 를 모아 놓은 디렉토리
  - **import** : 다른 package를 사용 할 때 포함 해야 함
  - **class** : external class로 public를 키워드를 붙일 수
- **내부 구성 요소**
  - **field** : Class의 속성, Class내부에서 사용 하는 변수
  - **constructor** : 객체를 생성하는 역할 담당. 생략 하면 기본 생성자 자동 생김
  - **method** : Class가 가지고 있는 기능(함수)
  - **innerClass** : Class내부에 있는 Class

1. Class

01. 객체 (Class)

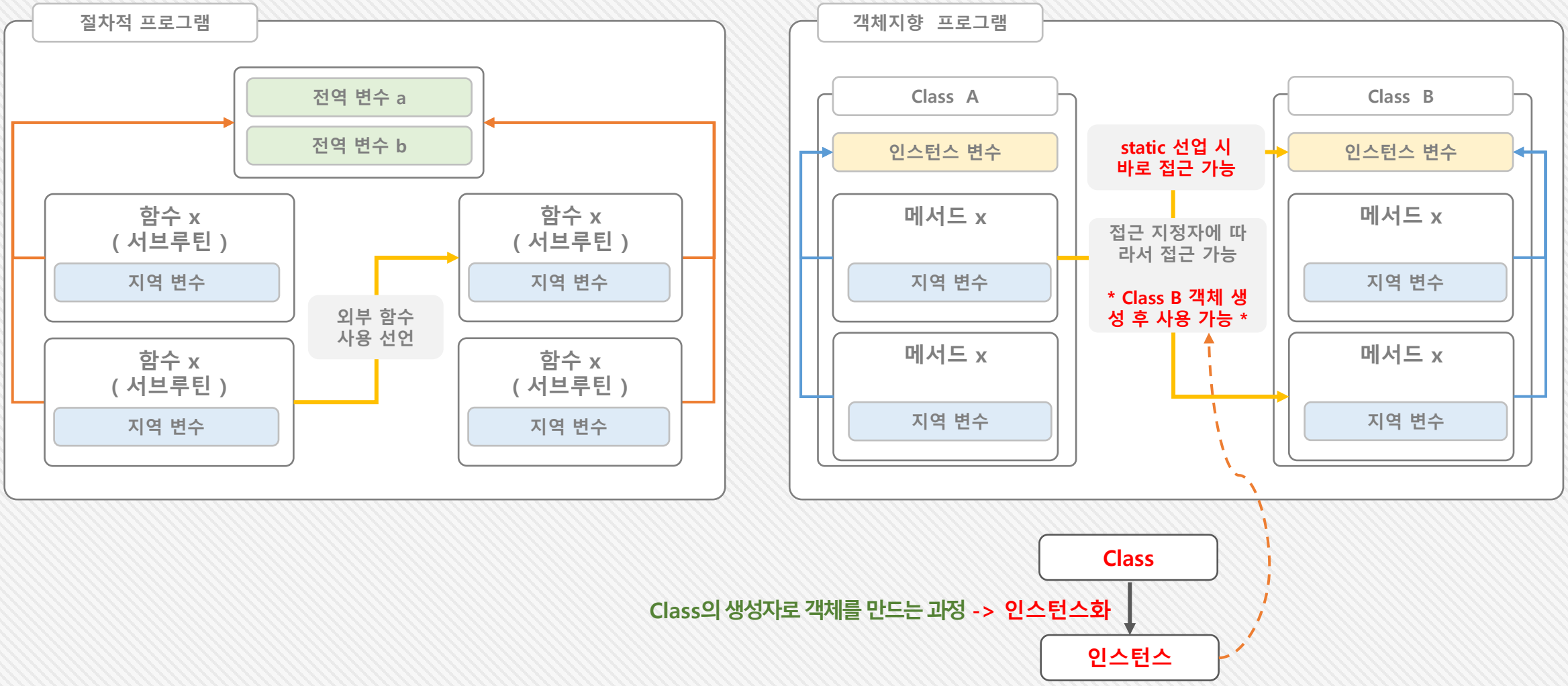
- 변수와 메소드를 정의 하는 프로토타입
- Field(맴버변수:객체의상태) + Operation(Method:객체의 행위)
- 클래스 이름은 대문자로 시작, 다음 단어의 시작은 대문자
- 사용자 정의 자료형, 객체의 자료형
  (Sample sample = new Sample())
- Class 키워드로 선언, 논리적인 개체, 한번만 선언
- 선언 시 키워드
  :public - 접근지정자가 맨 처음,
  :abstract - 추상클래스를 선언
  :final - 더 이상 자식으로 상속되지 않음을 명시,
  :ClassName - 클래스 이름
  :extends - 다름 클래스를 상속,
  :implements - 인터페이스 구현)
- 초기화 순서
  : 메모리에 적재된 후 한 번 초기화
  - 모든 클래스 변수 (static 변수) 가 디폴트 값으로 초기화

02. 객체 (Object)

- new 키워드에 의해서 만들어지며, 클래스의 인스턴스, 물리적인 개체 필요할 때 마다 생성
- type이 Class인 변수
- 객체 이름은 소문자로 시작, 다음 단어의 시작은 대문자

2. 절차적 프로그램 구조와 차이점

프로그램 구조적 차이



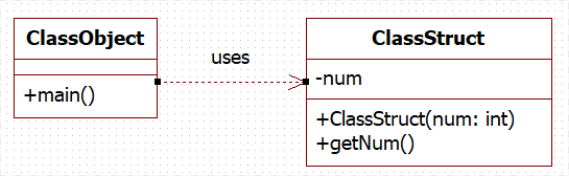
3. Instance

객체 생성

- 클래스(Class)의 생성자로 객체(Object)를 만드는 과정을 **인스턴스화 (Instantiation)**이라 하며 이 과정에서 만들어진 객체를 **인스턴스(Instance) 객체**라 함

01. 객체 생성

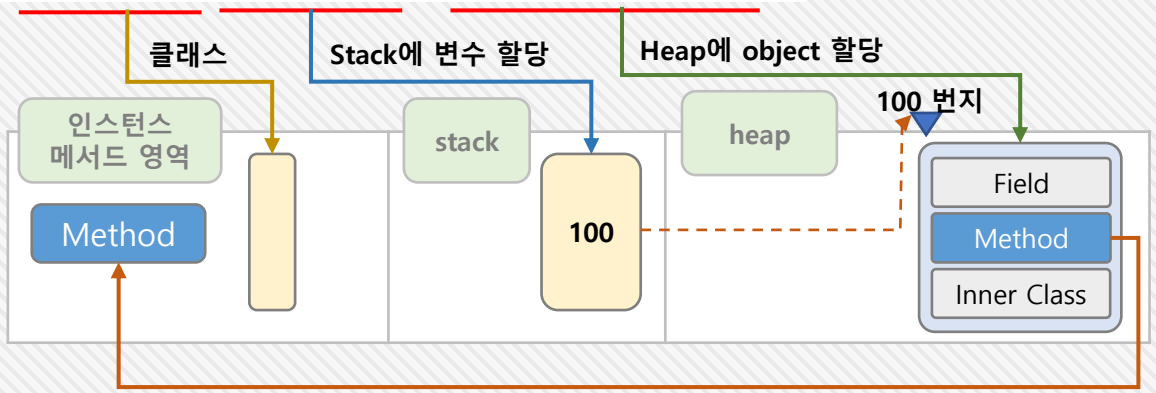
```
클래스명 참조변수명 = new 생성자();
```




```
public class ClassObject {
    public static void main(String[] args) {
        ClassStruct classStruct = new ClassStruct();
        classStruct.setNum(10);
        int num = classStruct.getNum();
        System.out.println("num = " + num);
    }
}
```

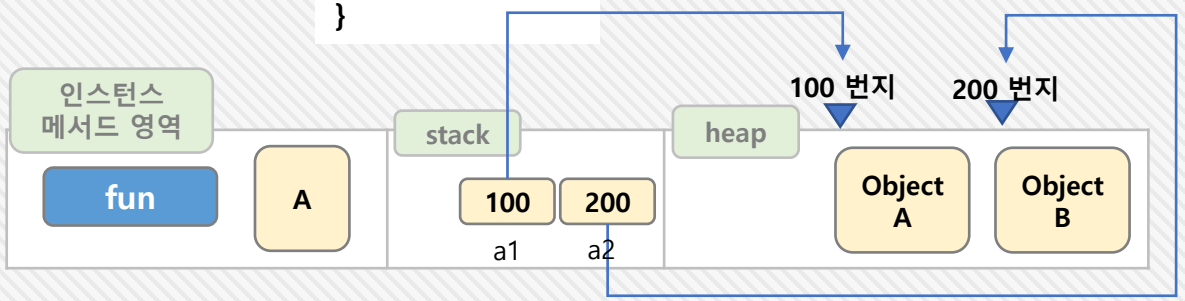
02. 메모리 구조

```
ClassStruct classStruct = new ClassStruct();
```




```
A a1 = new A();
A a2 = new A();
```

```
class A {
  void fun();
}
```



1. Field

Field

- 인스턴스 변수 (or 멤버 변수) : 클래스(Class)에 포함된 변수로 클래스 안에 있는 모든 곳에서 접근 가능 하다 -> 클래스의 전역 변수
- 지역 변수 : 메서드(Method)에 포함된 변수로 해당 메소드에서만 유효함

01. Field 선언

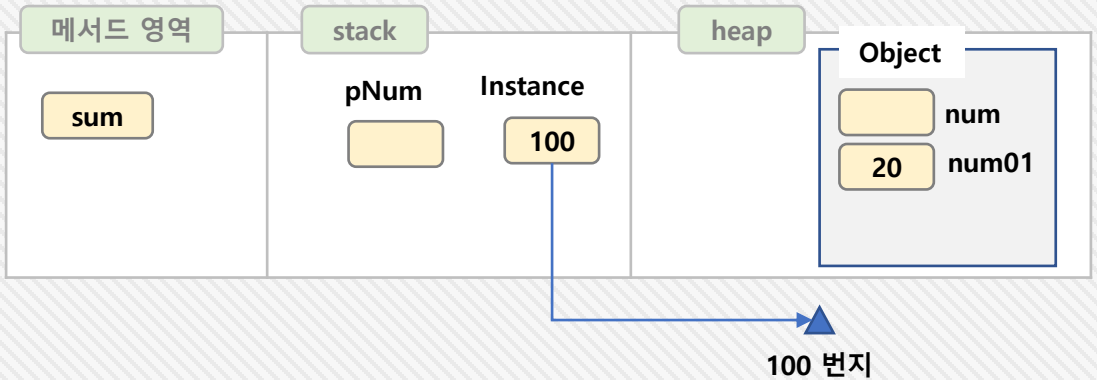
```
private 자료형 변수이름 [ = 초기값 ]
```

```
private int num;
private int num01 = 20;
```

02. 메모리 구조

- 인스턴스 변수는 객체 안에 저장 (heap)
- 지역 변수는 stack에 저장

```
public int sum(int pNum) {
    return this.num + (this.num01 + pNum);
}
```



03. 인스턴스 변수는 초기값이 강제 설정됨

```
int num = classStruct.getNum();
int num01 = classStruct.getNum01();
System.out.println("num = " + num);
System.out.println("num01 = " + num01);
```

```
num = 0
num01 = 20
```

✓ 지역 변수는 stack에 저장 되므로 초기화 없으면 오류 발생

```
int localNum ;
System.out.println("localNum = " + localNum);
```

Variable 'localNum' might not have been initialized

1. Method

Method

- 클래스의 기능 – Operation

01. Method 선언

```
자바제어자 반환Type Method명( 매개변수 ) {

   실행 문 ;

   [ return 값 ]   // 반환 type 이 void 이면 생략
}
```

- 매개변수 : 입력 매개 변수로 복수인 경우 , 로 구분

```
자료형(Type) 매개변수이름 [ , 자료형(Type) 매개변수이름 ,... ]
```

02. 자바 제어자

- 접근 지정자 : 접근 가능 범위 선언
  - public, private, default, protected
- **static :**
  - 클래스의 멤버(필드, 메서드, 이너 클래스)에 사용
  - 정적 멤버 (Static Member)
  - 객체 생성 없이 "객체명.정적멤버명"으로 사용
- **final : 불변 값 표현**
  - 필드, 지역변수, 메서드, 클래스 앞에 위치 하며 각 의미가 틀림
- abstract :
  - 기능을 정의 하지 않는 추상한 것을 상속 받은 곳에서 구현
  - 추상 메서드(abstract method), 추상 클래스 (abstract class)

접근 지정자	접근(사용) 가능 범위
public	동일 패키지의 모든 메서드 + 다른 패키지의 모든 클래스
protected	동일 패키지의 모든 메서드 + 다른 패키지의 자식 클래스
default (or package)	동일 패키지의 모든 메서드 (접근 지정자 생략 가능)
private	동일 클래스

03. 메서드 오버로딩 (Method Overloading)

- 메서드 시그너처 (Method signature)
  - 메서드명과 입력 매개변수로 MM은 메서드명은 같은데 입력 매개변수의 개수가 틀리면 다르게 인식 하는 것을 이용 하는 것으로 이것을 메서드 오버로딩 (Method Overloading)이라 함

04. 가변 입력 매개 변수

- 배열입력 매개변수하고 하며, 여러 개의 입력 값을 하나로 전달"
- " ... " 을 사용함

```
반환Type Method명( 자료형 ...참조변수명 ) {

   실행 문 ;

   [ return 값 ]   // 반환 type 이 void 이면 생략
}
```

2. 메서드 오버로딩 (Method Overloading)

Method

- 클래스의 기능 – Operation

01. Method 선언

```
자바제어자 반환Type Method명( 매개변수 ) {

   실행 문 ;

   [ return 값 ]   // 반환 type 이 void 이면 생략
}
```

- 매개변수 : 입력 매개 변수로 복수인 경우 , 로 구분

```
자료형(Type) 매개변수이름 [ , 자료형(Type) 매개변수이름 ,... ]
```

02. 자바 제어자

- 접근 지정자 : 접근 가능 범위 선언
  - public, private, default, protected
- **static :**
  - 클래스의 멤버(필드, 메서드, 이너 클래스)에 사용
  - 정적 멤버 (Static Member)
  - 객체 생성 없이 "객체명.정적멤버명"으로 사용
- **final : 불변 값 표현**
  - 필드, 지역변수, 메서드, 클래스 앞에 위치 하며 각 의미가 틀림
- abstract :
  - 기능을 정의 하지 않는 추상한 것을 상속 받은 곳에서 구현
  - 추상 메서드(abstract method), 추상 클래스 (abstract class)

접근 지정자	접근(사용) 가능 범위
public	동일 패키지의 모든 메서드 + 다른 패키지의 모든 클래스
protected	동일 패키지의 모든 메서드 + 다른 패키지의 자식 클래스
default (or package)	동일 패키지의 모든 메서드 (접근 지정자 생략 가능)
private	동일 클래스

03. 메서드 오버로딩 (Method Overloading)

- 메서드 시그너처 (Method signature)
  - 메서드명과 입력 매개변수로 MM은 메서드명은 같은데 입력 매개변수의 개수가 틀리면 다르게 인식 하는 것을 이용 하는 것으로 이것을 메서드 오버로딩 (Method Overloading)이라 함

04. 가변 입력 매개 변수

- 배열입력 매개변수하고 하며, 여러 개의 입력 값을 하나로 전달"
- " ... " 을 사용함

```
반환Type Method명( 자료형 ...참조변수명 ) {

   실행 문 ;

   [ return 값 ]   // 반환 type 이 void 이면 생략
}
```

2. 메서드 오버로딩 (Method Overloading)

메서드 오버로딩 (Method Overloading)

- ...

3. 가변 입력 매개 변수

가변 입력 매개 변수

- …

1. Constructor

Constructor

- ...

1. 접근지정자

접근 지정자

- 멤버, 생성자, 메서드를 접근 하기 위한 지정자로 선언한 지정자에 따라서 접근 범위가 결정 된다.

접근 지정자	접근(사용) 가능 범위
public	동일 패키지의 모든 메서드 + 다른 패키지의 모든 클래스
protected	동일 패키지의 모든 메서드 + 다른 패키지의 자식 클래스
default (or package)	동일 패키지의 모든 메서드 (접근 지정자 생략)
private	동일 클래스

```java
package com.hyomee.classMethod;
import com.hyomee.classobject.ClassMethod;
public class ClassMethodTest {
  public void DefaultMethodTest() {
    ClassMethod classMethod = new ClassMethod();
    classMethod.publicMethod();
    classMethod.protectedMethod();
    classMethod.defaultMethod();
    classMethod.privateMethod();
  }
}
```

```java
package com.hyomee.classobject;
public class ClassPackageMethodTest {
  public void ClassPackageMethodTest() {
    ClassMethod classMethod = new ClassMethod();
    classMethod.publicMethod();
    classMethod.protectedMethod();
    classMethod.defaultMethod();
    classMethod.privateMethod();
  }
}
```

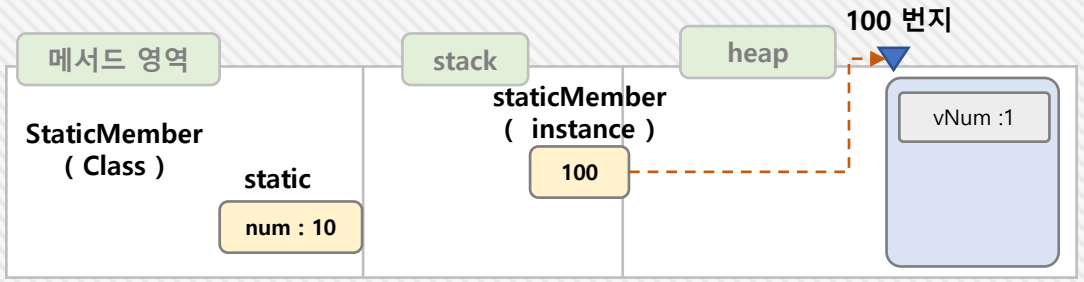
```java
package com.hyomee.classobject;
public class ClassMethod {
  public ClassMethod() {
    System.out.println("==== ClassMethod 생성 ");
  }
  public void publicMethod() {
    System.out.println("==== publicMethod    ");
  }
  protected void protectedMethod() {
    System.out.println("==== protectedMethod ");
  }
  void defaultMethod() {
    System.out.println("==== defaultMethod   ");
  }
  private void privateMethod() {
    System.out.println("==== privateMethod   ");
  }
}
```

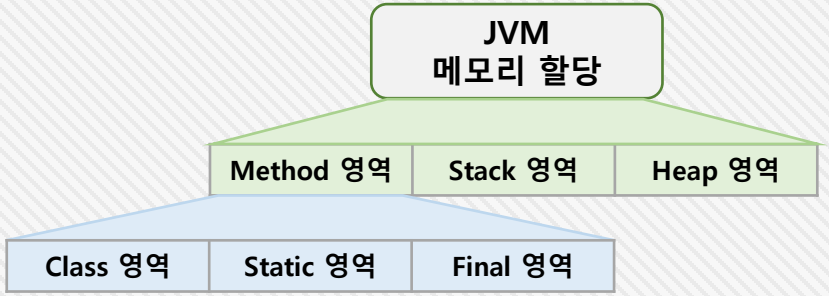
2. static

static Field

- 클래스의 멤버(필드, 메서드, 이너 클래스)에 사용
- 정적 멤버 (Static Member)
- 객체 생성 없이 "객체명.정적멤버명"으로 사용

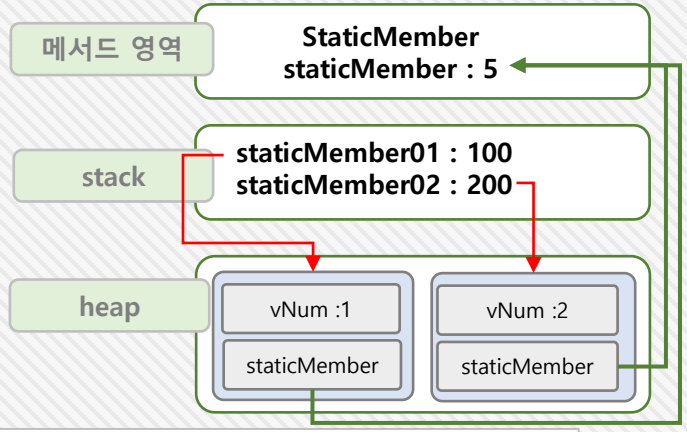
```
public class StaticMember {
   int vNum = 1;
   static int num = 10;
}
```





✓ **Static Member (정적 변수)는 인스턴스 객체로 각각 생성이 되어도 값은 공유 한다.**

```
public class StaticMember {
  int vNum = 1;
  static int num = 10;
  static int staticNum;

  void setStaticNum(int num) {
    staticNum = num;
  }

  int getStaticNum() {
    return staticNum;
  }
}
```




```
StaticMember staticMember01 = new StaticMember();
StaticMember staticMember02 = new StaticMember();

staticMember01.vNum = 1;
staticMember02.vNum = 2;
staticMember01.setStaticNum(5);

System.out.println("instance member num " + staticMember01.vNum);
System.out.println("instance member num " + staticMember02.vNum);
System.out.println("static member 01 staticNum "
        + staticMember01.getStaticNum());
System.out.println("static member 01 staticNum "
        + staticMember02.getStaticNum());
```

```
instance member num 1
instance member num 2
static member 01 staticNum 5
static member 01 staticNum 5
```

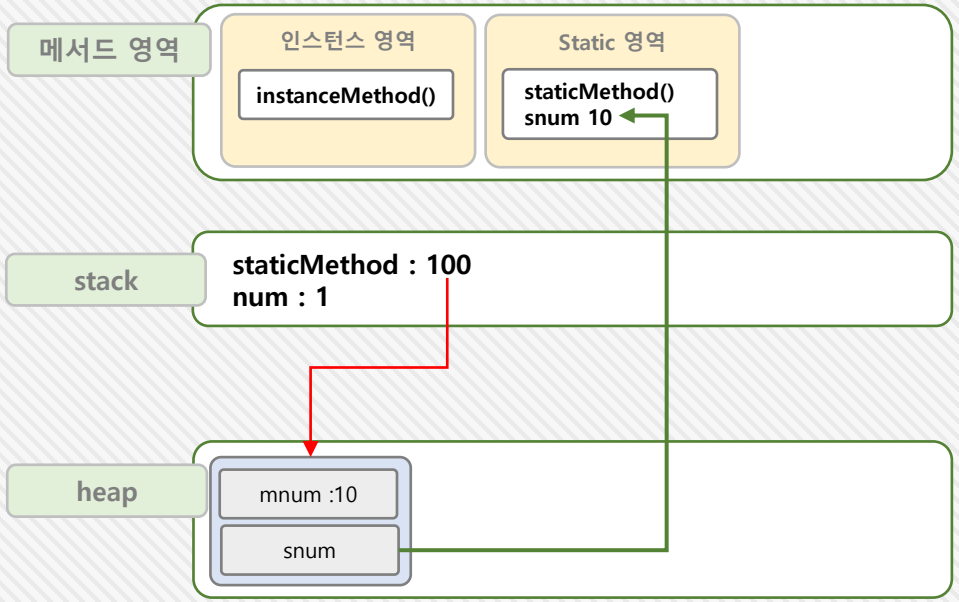
2. static

static Method

- 정적 메서드 (Static Member), 객체 생성 없이 "객체명.정적Method()"으로 사용
- 인스턴스 변수는 사용 불가, 메서드 내부에서 객체를 생성 하여 사용 하는 경우는 객체의 인스턴스 변수 사용 가능

```java
public class StaticMethod {
  int mnum = 10;
  static int snum = 20;
  void instanceMethod() {
    mnum = mnum + 1;
    snum = snum + 10;

    System.out.println("instanceMethod instance 변수 mnum : " + mnum);
    System.out.println("instanceMethod static 변수 snum : " + snum);
  }

  static void staticMethod() {
    int num = 1;
    num = num + 1;
    snum = snum + 10;

    // 오류 Cannot resolve symbol 'nnum'
    // static 에서는 static 변수만 접근 가능
    // mnum = nnum + 1;

    // 인스턴스 변수는 접근 가능
    StaticMember staticMember = new StaticMember();
    staticMember.vNum = staticMember.vNum + 1;

    System.out.println("staticMethod local num : " + num);
    System.out.println("staticMethod static snum : " + snum);

  }
}
```

```java
StaticMethod.staticMethod();
StaticMethod staticMethod = new StaticMethod();
staticMethod.instanceMethod();
```




```
staticMethod local num : 2
staticMethod static snum : 30
instanceMethod instance 변수 mnum : 11
instanceMethod static 변수 snum : 40
```

2. static

static 초기화 블럭

- 객체 생성 이전에 static field는 초기화 되지 않는다. (몰론 선언과 동시에 초기화 하는 경우 제외)
- static 블록을 사용 하여 초기화 한다. -> 메모리에 Class가 로딩 되는 시점에 가장 먼저 실행됨
- Reference variable는 재 할당 할 수 없지만 요소는 변경 가능 하다.

```
// 블럭 초기화
StaticBlockInit staticBlockInit = new StaticBlockInit();
System.out.println("static block 초기화 "  + StaticBlockInit.snum);;
```

```
public class StaticBlockInit {

  int num;
  static int snum;

  static {
    snum = 10;
    System.out.println("Static Block Init......");
  }

  public StaticBlockInit() {
    this.num = 3;
    System.out.println("생성 this.num Init......");
  }
}
```

```
Static Block Init......
생성 this.num Init......
static block 초기화 10
```

3. final

final

- Field, Local Variable, Method, Class 앞에 위치하며, 각각의 의미가 틀리다.
- Field, Local Variable 사용시 초기화를 하여야 한다.

위치	의미
Local Variable	값 변경 불가
Field	값 변경 불가
Method	Override 불가
Class	상속 불가

```
public class FinalMain {

  public static void main(String[] args) {
    FinalMethodChild finalMethodChild = new FinalMethodChild();
    finalMethodChild.instanceMethod();
    finalMethodChild.finalInstanceMethod();

    FinalChild finalChild = new FinalChild();
    finalChild.runFinalParent();
  }
}
```

✓ 다음 페이지

✓ Field, Local Variable 예제

```
public class FinalClass {

  final int instanceNum01 ;
  final int instanceNum02 = 10;
  final int[] instanceArray = new int[]{10,20,30};

  public FinalClass(int num) {
    this.instanceNum01 = num;
  }

  public void finalVarible() {
    int localNum = 10;
    final int finalLocalNum ;

    finalLocalNum = 30;
    localNum = localNum + 1;

    // 오류 : Variable 'finalLocalNum' might already have been assigned to
    // finalLocalNum = finalLocalNum + 2; // final 지역변수
    // instanceNum02 = instanceNum02 + 2; // final instance 변수

    instanceArray[2] = 30;
    // 오류 :  Cannot assign a value to final variable 'instanceArray'
    // instanceArray = new int[]{10,20,30};

  }
}
```

3. final

final

✓ final Method

```
public class FinalMethodParent {
  protected void instanceMethod() {
    System.out.println("Parent Instance Method ... ");
  }

  final protected void finalInstanceMethod() {
    System.out.println("Parent final Instance Method ... ");
  }
}
```

```
public class FinalMethodChild extends FinalMethodParent {
  public void instanceMethod() {
    super.instanceMethod();
    super.finalInstanceMethod();
    System.out.println("Child Instance Method ... ");
  }

/*
  오류 : 'finalInstanceMethod()' cannot override 'finalInstanceMethod()'
  in 'com.hyomee.finaltest.FinalMethodParent'; overridden method is final
  final public void finalInstanceMethod() {
    System.out.println("Parent final Instance Method ... ");
  }
*/
}
```

X

```
Parent Instance Method ...
Parent final Instance Method ...
Child Instance Method ...
Parent final Instance Method ...
```

✓ final Class

```
public final class FinalParent {

  private void instancePrivateMethod() {
    System.out.println("Parent instancePrivateMethod Method ... ");
  }

  // Method declared 'final' in 'final' class
  // final public void finalInstancePublicMethod() {
  public void finalInstancePublicMethod() {
    System.out.println("Parent final finalInstancePublicMethod Method ... ");
    instancePrivateMethod();
  }

  // Class member declared 'protected' in 'final' class
  // protected void finalInstanceMethod() {
  void finalInstanceMethod() {
    System.out.println("Parent final Instance Method ... ");
  }
}
```

```
/*
  오류 Cannot inherit from final 'com.hyomee.finaltest.FinalParent'
  public class FinalChild extends FinalParent{
  }
*/
public class FinalChild {
  public void runFinalParent() {
    FinalParent finalParent = new FinalParent();
    finalParent.finalInstanceMethod();
    finalParent.finalInstancePublicMethod();
  }
}
```

```
Parent final Instance Method ...
Parent final finalInstancePublicMethod Method ...
Parent instancePrivateMethod Method ...
```

4. abstract

abstract

- 사전적의미 "추상적인"으로 추상 메소드(Abstract Method)와 추상 클래스 (Abstract Class)가 있다.
- **추상 메서드(Abstract Method)**: Method선언 만 하고 기능이 없는 미완성 Method로 상속 받은 객체에서 기능 구현하는 것으로 세미콜론(;)을 끝나야 함
- **추상 클래스 (Abstract Class)** : 하나이상의 추상 메소드(Abstract Method)가 포함되어 있는 Class로 new 로 직접 생성 할 수 없고 상속을 통해서 만 가능

선언 방법 : abstract 자료형 메서드명();

```
abstract int abstractmethod();      미완성 코드

protected void method() {           완성 코드
}
```

```
public class AbstractMain {
  public static void main(String[] args) {
    // 오류 : 'AbastractClass' is abstract; cannot be instantiated
    // AbastractClass abastractClass = new AbastractClass() ;
    AbastractClass abastractClass = new AbastractClass() {
      @Override
      int abstractmethod() {
        // 구현 .........
        System.out.println("main에서 abstractmethod ..... 구현 ..... ");
        return 0;
      }
    };
    System.out.println("main.abastractClass 객체 생성 추상 메서드 구현 : "
            + abastractClass.abstractmethod());
    System.out.println("main.abastractClass 객체 생성 Method 실행 : "
            + abastractClass.method());

    AbstractChild abstractChild = new AbstractChild();
    System.out.println("abstractChild 객체 생성 : "
            + abstractChild.abstractmethod()); ;
  }
}
```

```
main에서 abstractmethod ..... 구현 .....
main.abastractClass 객체 생성 추상 메서드 구현 : 0
AbastractClass.method 완성된 코드 .....
main.abastractClass 객체 생성 Method 실행 : 0
AbstractChild.abstractmethod : Override 구현 .....
AbastractClass.method 완성된 코드 .....
abstractChild 객체 생성 : 0
```

```
abstract class AbastractClass {

  abstract int abstractmethod();

  protected int method() {
    System.out.println("AbastractClass.method 완성된 코드 ..... ");
    return 0;
  }

}
```

```
public class AbstractChild extends AbastractClass{

  @Override
  public int abstractmethod() {
    // 오류 : Abstract method 'abstractmethod()'
    //        cannot be accessed directly
    // super.abstractmethod();
    System.out.println("AbstractChild.abstractmethod : Override 구현 ..... ");
    super.method();
    return 0;
  }
}
```