객체

목차

Chap01.
객체지향언어

▶ 객체 지향 언어

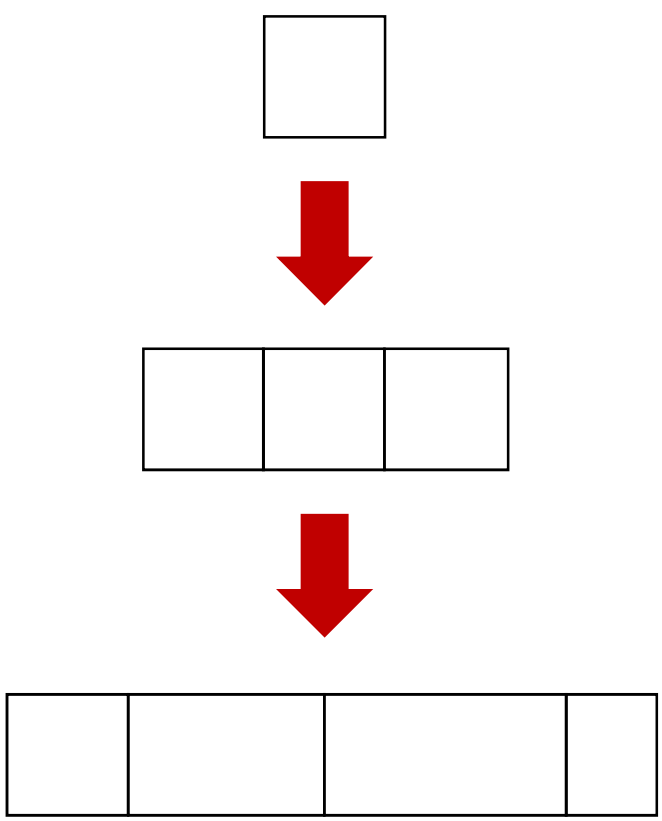
현실 세계는 사물이나 개념처럼 독립되고 구분되는 각각의 객체로
이루어져 있으며, 발생하는 모든 사건들은 객체간의 상호작용으로
이 개념을 컴퓨터로 옮겨 놓아 만들어낸 것이 객체지향 언어

✓ 자바에서의 객체(Object)

클래스에 정의된 내용대로 new 연산자를 통해 메모리 영역에 생성된 것

객체 지향 언어 - 클래스

✓ 클래스의 등장 배경



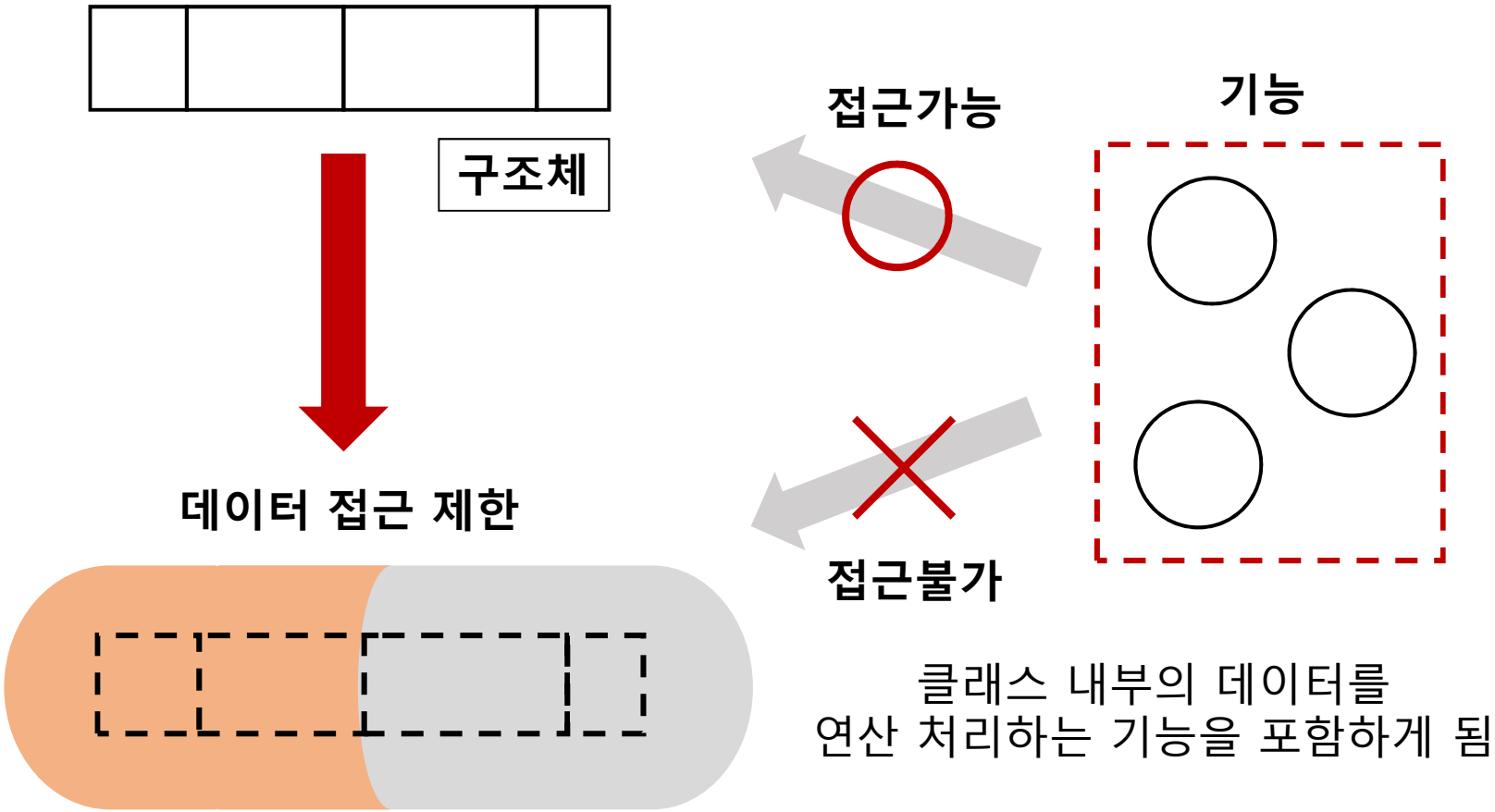
변수 — 1개의 자료형 1개의 데이터

배열 — 1개의 자료형 여러 개의 데이터

구조체 — 여러 개의 자료형 여러 개의 데이터

객체 지향 언어 - 클래스

✓ 클래스의 등장 배경



클래스 내부의 데이터를
연산 처리하는 기능을 포함하게 됨

▶ 객체 지향 언어 - 클래스

✓ 클래스의 등장 배경

객체의 특성에 대한 정의를 한 것으로 캡슐화를 통해 기능을 포함한 개념,
사물이나 개념의 공통 요소를 추상화(abstraction)하여 정의
ex) 제품의 설계도, 빵 틀

✓ 추상화(abstraction)

유연성을 확보하기 위해 구체적인 것은 제거한다는 의미
프로그램에서 필요한 공통점을 추출하고,
불필요한 공통점을 제거하는 과정

객체 지향 언어 - 추상화

✓ 추상화(abstraction) 예시

국가에서 국민 정보 관리용 프로그램을 만들려고 할 때,
프로그램에서 요구되는 “국민 한 사람”의 정보를 추상화 한다면?

▶ 객체 지향 언어 - 추상화

✓ 추상화(abstraction) 예시

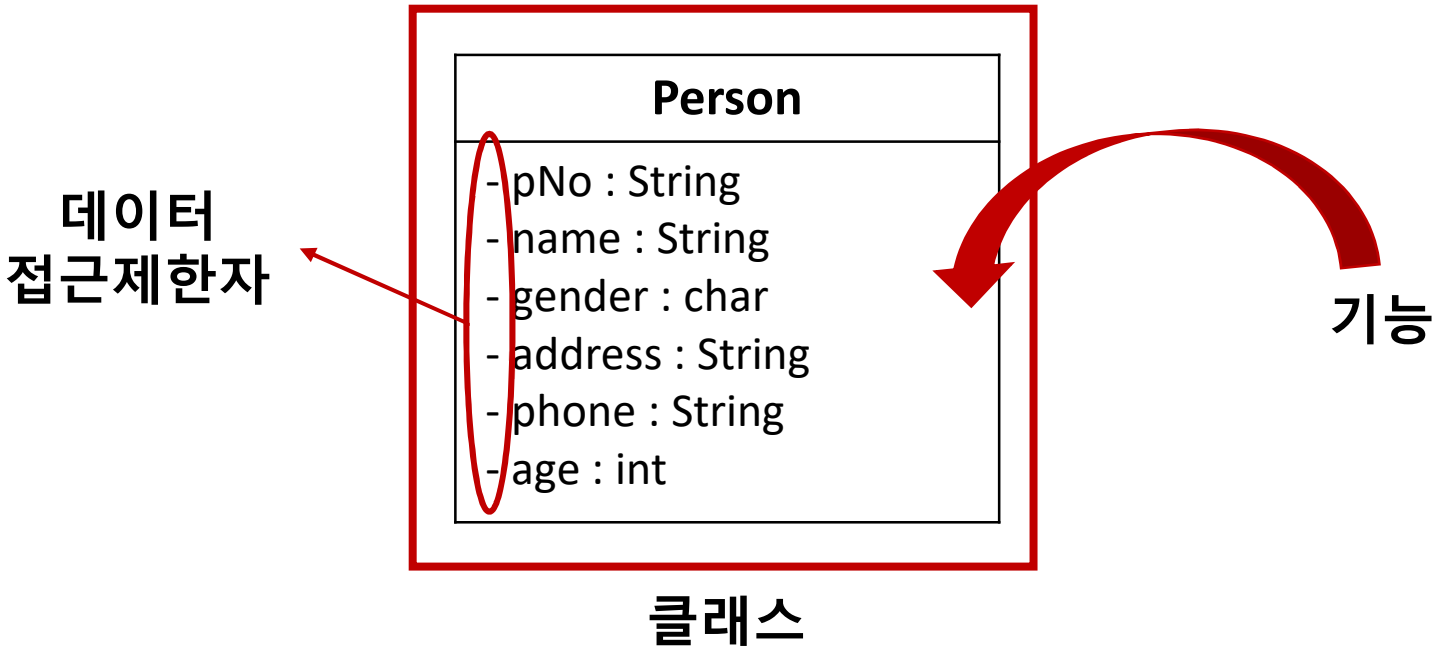
앞 페이지에서 추상화한 결과물을 객체 지향 프로그래밍 언어를
사용해서 변수명(데이터 이름)과 자료형(데이터 타입) 정리

항목	변수명	자료형(type)
주민등록번호	pNo	String
이름	name	String
성별	gender	char
주소	address	String
전화번호	phone	String
나이	age	int

객체 지향 언어 - 추상화

✓ 추상화(abstraction) 예시

앞 페이지에서 정리된 변수명과 자료형을 클래스 다이어그램으로 표현 시 아래와 같음



▶ 객체 지향 언어

✓ 객체

현실에 존재하는 독립적이면서 하나로 취급되는 사물이나 개념으로
객체 지향 언어에서 객체의 개념은 클래스에 정의된 내용대로 메모리에
할당된 결과물(Object)

클래스 (Class) **객체(Instance)**

학생이 가지는 공통적인
요소를 추상화 하여
클래스를 정의

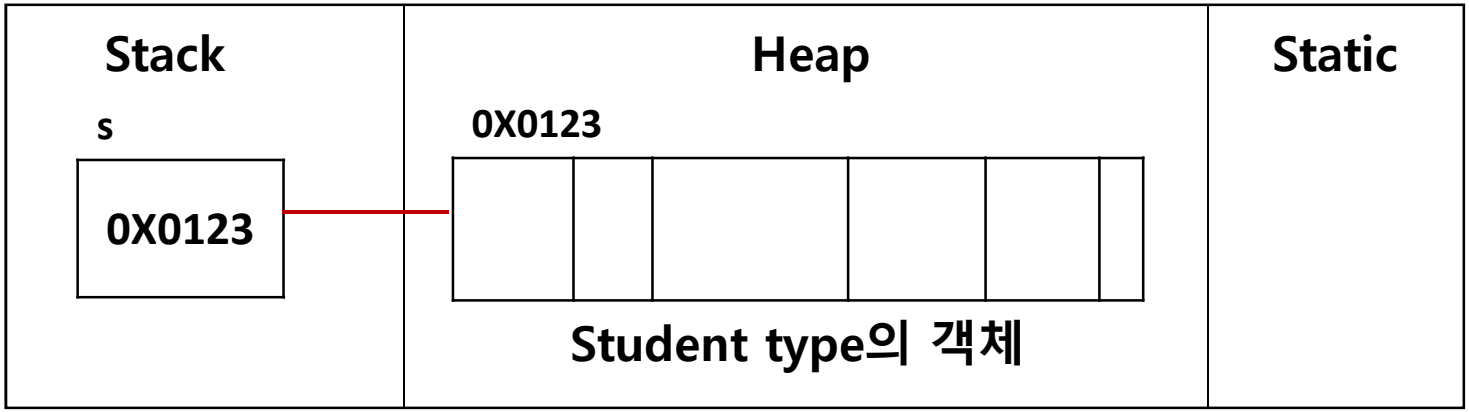
현실세계에 존재하는
고유 객체를
메모리에 할당

객체 지향 언어

✓ 객체(Instance)의 할당

new 연산자와 생성자를 사용하여 객체 생성 시 heap 메모리 공간에
서로 다른 자료형의 데이터가 연속으로 나열/할당된 객체 공간

예) Student s = new Student();



클래스 → **인스턴스(객체)**

인스턴스화

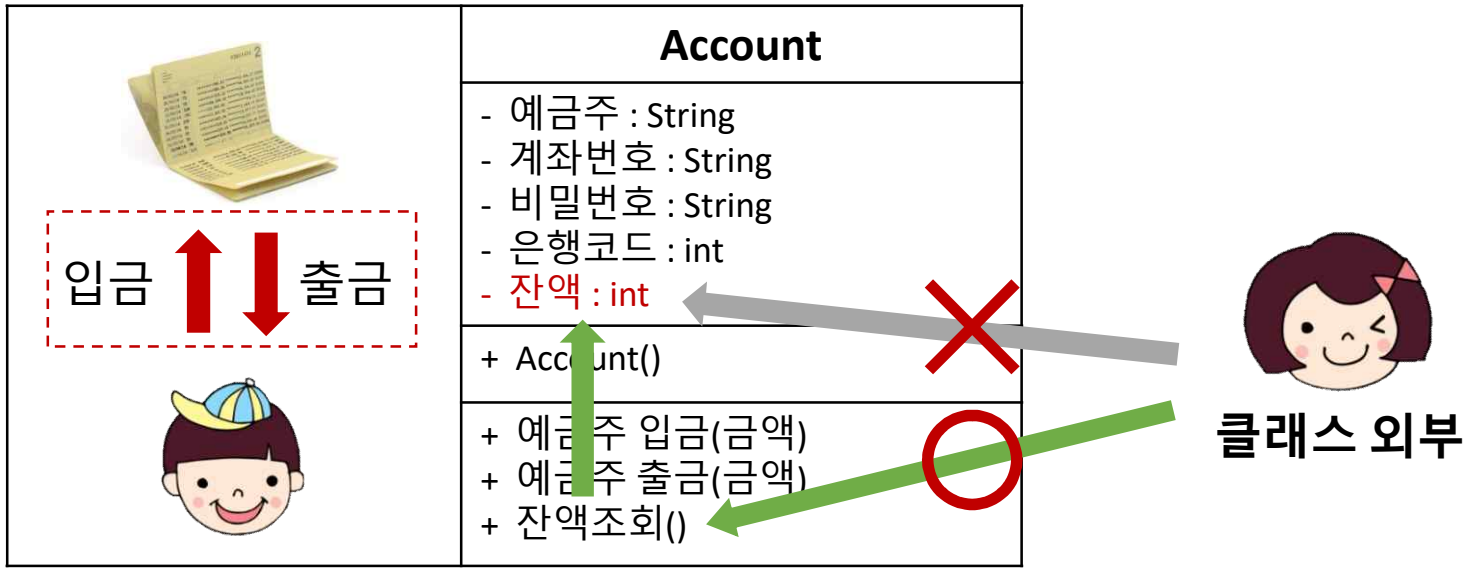
▶ 객체 지향 3대 특징

1. 캡슐화 (Encapsulation)
2. 상속 (Inheritance)
3. 다형성 (Polymorphism)

객체 지향 언어 - 캡슐화

✓ 캡슐화 원칙

1. 클래스의 **멤버 변수에 대한 접근 권한은 private을 원칙**으로 한다.
2. 클래스의 멤버 변수에 대한 연산처리를 목적으로 하는 함수들을 클래스 내부에 작성한다.
3. **멤버 함수는 클래스 밖에서 접근할 수 있도록 public으로 설정**한다.

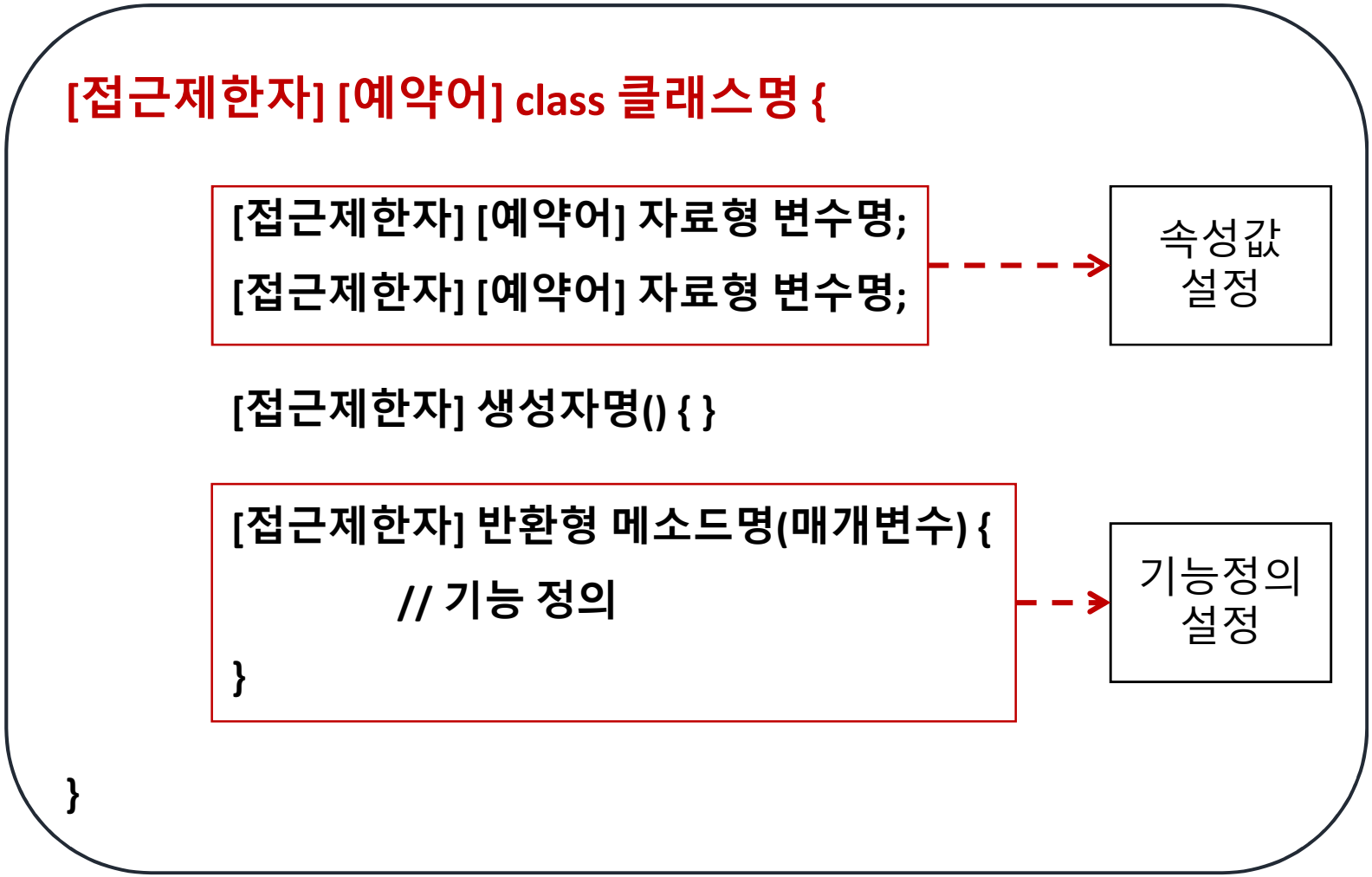


Account 클래스로 생성된 김철수 학생 명의의 계좌 객체

Chap02.
클래스 (Class)

▶ 클래스 (Class)

✓ 클래스 선언



▶ 클래스 (Class)

✓ 클래스 예시

```
public class Member {
	private String name;
	private int age;

	public Member() {}

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```

▶ 클래스 (Class)

✓ 클래스 접근제한자

구분		같은 패키지 내	전체
+	public	O	O
~	(default)	O	

예)

```
public class 클래스명 {
        // ..............
}

class 클래스명 {
        // ..............
}
```

Chap03.
package와 import

▶ 소스파일

✓ 소스파일 구성 순서

1. package문
2. import문
3. 클래스 선언

✓ 소스파일 구성 예시

```
package kh.academy; //package문

import java.util.Date; //import문

public class ImportTest { //클래스 선언
        public static void main(String[] args) {

        }
}
```

▶ package

서로 관련된 클래스 혹은 인터페이스의 묶음으로 폴더와 비슷
패키지는 서브 패키지를 가질 수 있으며 '.'으로 구분

예) Scanner 클래스의 full name은 패키지명이 포함된 java.util.Scanner 이다.

✓ 패키지의 선언

소스파일 첫 번째 문장에 단 한번 선언하며 하나의 소스파일에
둘 이상의 클래스가 포함된 경우, 모두 같은 패키지에 속함
모든 클래스는 하나의 패키지에 속하며, 패키지가 선언되지 않은
클래스는 자동적으로 이름없는 패키지(default)에 속하게 됨

예) package java.util;

▶ import

사용할 클래스가 속한 패키지를 지정하는 데 사용

import문을 사용하면 클래스를 사용할 때 패키지 명 생략 가능

java.lang패키지의 클래스는 import를 하지 않고도 사용 가능

* java.lang 패키지 내의 클래스 → String, Object, System....

✓ import문의 선언

import문은 패키지문과 클래스 선언의 사이에 선언하며 컴파일 시에 처리되므로 프로그램 성능에 영향을 주지 않음

지정된 패키지에 포함된 클래스는 import 가능하지만 서브 패키지에 속한 모든 클래스까지는 불가능

```
예) import java.util.Date;
    import java.util.*;          // java.util 패키지 내의 모든 클래스 (단, 서브클래스는 X)
    import java.*;               // 불가능
```

▶ import

✓ import문 주의사항

이름이 같은 클래스가 속한 두 패키지를 import 할 때는 클래스 앞에 패키지 명을 붙여 구분해 주어야 함

예)
```
package kh.academy;

   import java.sql.Date;

   public class ImportTest {

      public static void main(String[] args) {
         java.util.Date today = new java.util.Date();
      }

   }
```

Chap04.
필드 (Field)

▶ 필드 (Field)

✓ 필드 표현식

[접근제한자] [예약어] class 클래스명 {

[접근제한자] [예약어] 자료형 변수명 [= 초기값];

}

✓ 필드 예시

```
public class Academy {
        public int temp1;
        protected int temp2;
        int temp3;          // 접근제한자 생략 시 default
        private int temp4; // 캡슐화 원칙으로 private 사용
}
```

▶ 필드 (Field)

✓ 필드 접근제한자

구분		해당 클래스 내부	같은 패키지 내	후손 클래스 내	전체
+	public	O	O	O	O
#	protected	O	O	O	
~	(default)	O	O		
-	**private**	O			

▶ 필드 (Field)

✓ 필드 예약어 - static

같은 타입의 여러 객체가 공유할 목적의 필드에 사용하며,

프로그램 start시에 정적 메모리 영역에 자동 할당되는 멤버에 적용

✓ static 표현식

```
public class Academy {
        private static int temp1;
}
```

▶ 필드 (Field)

✓ 필드 예약어 - final

하나의 값만 계속 저장해야 하는 변수에 사용하는 예약어

✓ final 표현식

```
public class Academy {
        private final int TEMP1 = 100;
        private int temp4;
}
```

▶ 필드 (Field) - 초기화 블럭

✓ 초기화 블럭

- **인스턴스 블럭 ({ })**

  인스턴스 변수를 초기화 시키는 블럭으로 객체 생성시 마다 초기화

- **static(클래스) 블럭 (static{ })**

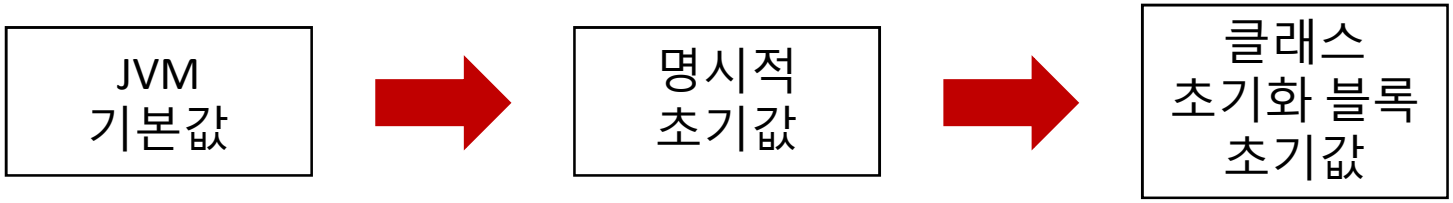
  static 필드를 초기화 시키는 블럭으로 프로그램 시작 시 한 번만 초기화

✓ 초기화 블럭 표현식

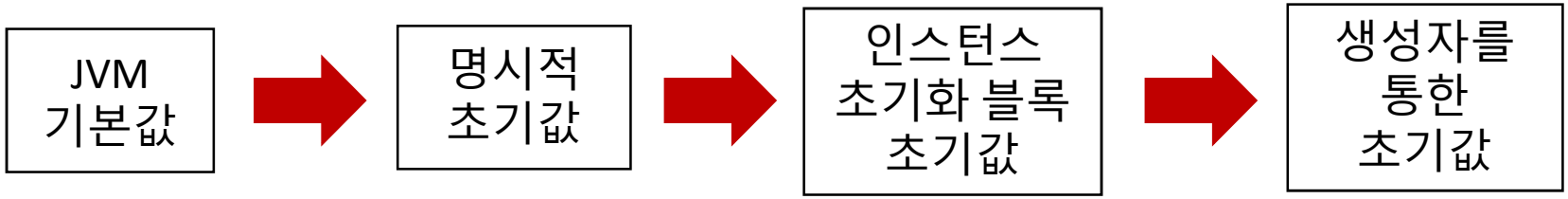
```
[접근제한자] [예약어] class 클래스명 {
        [접근제한자] static 자료형 필드1;
        [접근제한자] 자료형 필드2;

        static{ 필드1 = 초기값; }
        { 필드2 = 초기값; }
}
```

필드 (Field) – 초기화 순서

✓ 클래스 변수



✓ 인스턴스 변수



Chap04.
생성자(Constructor)

▶ 생성자 (Constructor)

객체가 new연산자를 통해 Heap메모리 영역에 할당될 때

객체 안에서 만들어지는 필드 초기화

생성자는 일종의 메소드로, 전달된 초기 값을 받아서 객체의 필드에 기록

✓ 생성자 규칙

생성자의 선언은 메소드 선언과 유사하나 반환 값이 없으며

생성자명을 클래스명과 똑같이 지정해주어야 함

▶ 생성자 (Constructor)

✓ 생성자 표현식

[접근제한자] [예약어] class 클래스명 {

[접근제한자] 클래스명() { }

[접근제한자] 클래스명(매개변수) { (this.)필드명 = 매개변수; }

}

```
public class Academy {
        private int studentNo;
        private String name;

        // 기본 생성자
        public Academy() {}

        //매개변수 있는 생성자
        public Academy(int studentNo, String name) {
                this.studentNo = studentNo;
                this.name = name;
        }
 }
```

▶ 생성자 (Constructor)

✓ 기본 생성자

작성하지 않은 경우, 클래스 사용 시 **JVM이 자동으로 기본 생성자 생성**

✓ 매개변수 생성자

- 객체 생성 시 전달받은 값으로 객체를 초기화 하기 위해 사용
- **매개변수 생성자 작성 시 JVM이 기본 생성자를 자동으로 생성해주지 않음**
- 상속에서 사용 시 반드시 기본 생성자를 작성
- 오버로딩을 이용하여 작성

▶ 오버로딩

한 클래스 내에 동일한 이름의 메소드를 여러 개 작성하는 기법

✓ 오버로딩 조건

- 같은 메소드 이름
- 다른 매개변수의 개수 또는 다른 매개변수 타입

▶ this

모든 인스턴스 메소드에 숨겨진 채 존재하는 레퍼런스로 할당된 객체를 가리킴

함수 실행 시 전달되는 객체의 주소를 자동으로 받음

✓ this 사용 예시

```
public class Academy{
        private String name;
        public Academy() { }
        public Academy(String name) { this.name = name;  }
}
```

* 위와 같이 매개변수를 가지는 생성자에서 **매개변수 명이 필드명과 같은 경우**
매개변수의 변수명이 우선이므로 this 객체를 이용하여 대입되는
변수가 필드라는 것을 구분해줌

▶ this()

생성자, 같은 클래스의 다른 생성자를 호출할 때 사용하며 반드시 첫 줄에 선언

✓ this() 사용 예시

```
public class Academy{
        private int age;
        private String name;
        public Academy() { this(20, "김철수"); }
        public Academy(int age, String name) {
                this.age = age;     this.name = name;
        }
 }
```

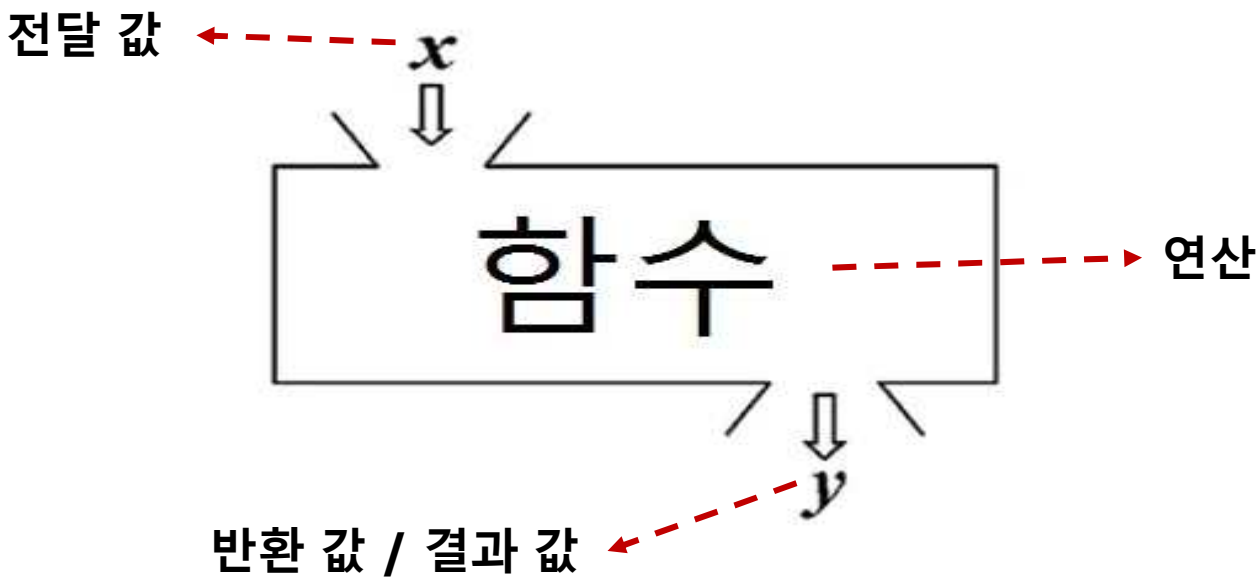
Chap05.
메소드(Method)

▶ 메소드 (Method)

수학의 함수와 비슷하며 호출을 통해 사용

전달 값이 없는 상태로 호출하거나 어떤 값을 전달하여 호출

함수 내에 작성된 연산 수행 후 반환 값/결과 값은 있거나 없을 수 있음



▶ 메소드 (Method)

✓ 메소드 표현식

```
[접근제한자] [예약어] 반환형 메소드명( [매개변수] ) {
        // 기능 정의
}
```

```
public void information() {
        System.out.println(studentNo);
}
```

▶ 메소드 (Method)

✓ 메소드 접근제한자

구분		클래스	패키지	자손 클래스	전체
+	public	O	O	O	O
#	protected	O	O	O	
~	(default)	O	O		
-	private	O			

▶ 메소드 (Method)

✓ 메소드 예약어

구분	전체
static	static 영역에 할당하여 객체 생성 없이 사용
final	종단의 의미, 상속 시 오버라이딩 불가능
abstract	미완성된, 상속하여 오버라이딩으로 완성시켜 사용해야 함
synchronized	동기화 처리, 공유 자원에 한 개의 스레드만 접근 가능함
static final (final static)	static과 final의 의미를 둘 다 가짐

▶ 메소드 (Method)

✓ 메소드 반환형

구분	전체
void	반환형이 없음을 의미, 반환 값이 없을 경우 반드시 작성
기본 자료형	연산 수행 후 반환 값이 기본 자료형일 경우 사용
배열	연산 수행후 반환 값이 배열인 경우 배열의 주소값이 반환
클래스	연산 수행후 반환 값이 해당 클래스 타입의 객체일 경우 해당 객체의 주소값이 반환 (클래스 == 타입)

▶ 메소드 (Method)

✓ 메소드 매개변수

구분	전체
( )	매개변수가 없는 것을 의미
기본 자료형	기본형 매개변수 사용 시 값을 복사하여 전달 매개변수 값을 변경하여도 본래 값은 변경되지 않음
배열	배열, 클래스 등 참조형을 매개변수로 전달 시에는 데이터의 주소 값을 전달하기 때문에 매개변수를 수정하면 본래의 데이터가 수정됨(얕은 복사)
클래스	
가변인자	매개변수의 개수를 유동적으로 설정하는 방법으로 가변 매개변수 외 다른 매개변수가 있으면 가변 매개변수를 마지막에 설정 * 방법 : (자료형 ... 변수명)

* 매개변수의 수에 제한이 없다.

▶ 메소드 (Method)

✓ 메소드 표현식 – 매개변수가 없고 리턴 값이 있을 때

```
[접근제한자] [예약어] 반환형 메소드명() {
        // 기능 정의
}
```

```
public int information() {
        return studentNo;
}
```

▶ 메소드 (Method)

✓ 메소드 표현식 – 매개변수가 없고 리턴 값이 없을 때

[접근제한자] [예약어] void 메소드명() {

```
        // 기능 정의
}
```

```
public void information() {
        System.out.println(studentNo);
}
```

▶ 메소드 (Method)

✓ 메소드 표현식 – 매개변수가 있고 리턴 값이 있을 때

```
[접근제한자] [예약어] 반환형 메소드명(자료형 변수명) {
        // 기능 정의
}
```

```
public String information(String studentName) {
        return studentNo + " " + studentName;
}
```

▶ 메소드 (Method)

✓ 메소드 표현식 – 매개변수가 있고 리턴 값이 없을 때

```
[접근제한자] [예약어] void 메소드명(자료형 변수명) {
        // 기능 정의
}

public void information(String studentName) {
      System.out.println(studentNo + " " + studentName);
}
```

▶ getter와 setter 메소드

✓ setter 메소드

필드에 변경할 값을 전달 받아서 필드 값을 변경하는 메소드

✓ 표현식

[접근제한자] [예약어] void set필드명(자료형 변수명) {

(this.)필드명 = 변수명;

}

```
public void setStudentNo(int studentNo) {
        this.studentNo = studentNo;
}
```

▶ getter와 setter 메소드

✓ getter 메소드

필드에 기록된 값을 읽어서 요청한 쪽으로 읽은 값을 넘기는 메소드

✓ 표현식

[접근제한자] [예약어] 반환형 get필드명() {

return 필드명;

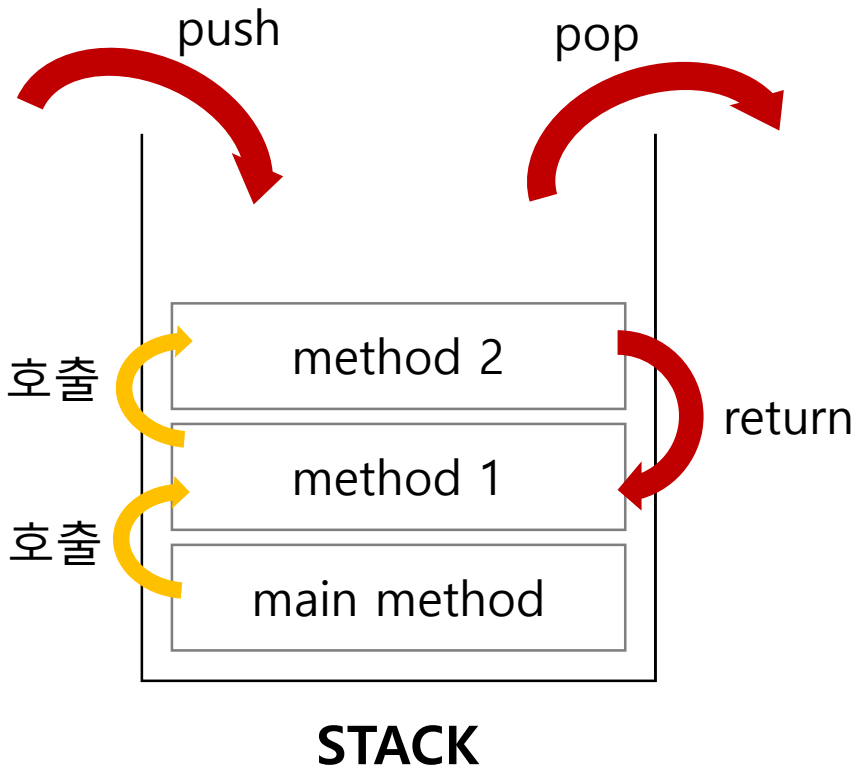
}

```
public int getStudentNo() {
        return studentNo;
}
```

return

해당 메소드를 종료하고 자신을 호출한 메소드로 돌아가는 예약어

반환 값이 있다면 반환 값을 가지고 자신을 호출한 메소드로 돌아감



* STACK의 자료구조 : LIFO(Last-Input-First-Out)