

# JAVA 객체지향

**Chap01. 객체지향  
프로그래밍이란?**

**Chap02. 객체, 클래스**

**Chap03. 필드**

**Chap04. 메소드**

**Chap05. 생성자**

## 객체지향(Object-Oriented)

### 객체지향 언어

현실 세계는 사물이나 개념처럼 독립되고 구분되는 각각의 객체로 이루어져 있으며, 발생하는 모든 사건들은 객체간의 상호 작용이다. 이 개념을 컴퓨터로 옮겨 놓아 만들어낸 것이 객체지향 언어이다.

### 객체지향 프로그래밍

프로그래밍에서는 현실세계의 객체(사물, 개념)를 클래스(class)와 객체(Object)의 개념으로 컴퓨터에서 구현한다.

## 객체지향(Object-Oriented)

### 객체지향언어의 역사

- 과학, 군사적 모의실험(simulation)을 위해 컴퓨터를 이용한 가상세계를 구현하려고 객체지향이론이 시작됨
- 1960년대 최초의 객체지향언어 Simula 탄생
- 1980년대 절차방식의 프로그래밍의 한계를 객체지향방식으로 극복하려고 노력함

### 객체지향언어의 특징

- 코드의 재사용성이 높다.
  - 새로운 코드를 작성할 때 기존의 코드를 이용해서 쉽게 작성할 수 있다.
- 코드의 관리가 쉬워졌다.
  - 코드간의 관계를 맺어줌으로써 보다 적은 노력으로 코드변경이 가능하다.

# 클래스, 추상화

## 클래스

객체를 정의해 놓은 것. 객체의 설계도, 틀.

사물이나 개념의 공통 요소(속성, 기능)를 용도에 맞게 추상화(abstraction) 함.

예) 제품의 설계도.

## 추상화(abstraction)

중요한것은 남기고 불필요한 것은 제거하는 일

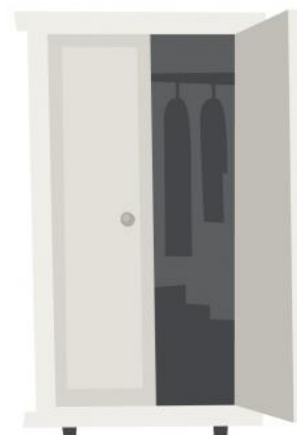
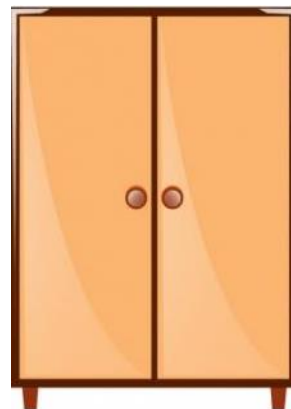
주어진 문제나 시스템을 중요하고 관계 있는 부분만 분리해 내어 간결하고 이해하기 쉽게 만드는 작업. 이러한 과정은 원래의 문제에서 구체적인 사항은 되도록 생략하고 핵심이 되는 원리만을 따지기 때문에 원래의 문제와는 전혀 관계가 없어 보이는 수학적 모델이 나오기도 한다. 이 기법은 복잡한 문제나 시스템을 이해하거나 설계하는 데 없어서는 안될 중요한 요소이다.

## 클래스 - 추상화

추상(abstraction) : 대상으로부터 모양을 뽑아내는 것  
즉, 어떤 대상에서 특징만을 뽑아낸 것



# 객체지향 프로그래밍



# 객체지향 프로그래밍



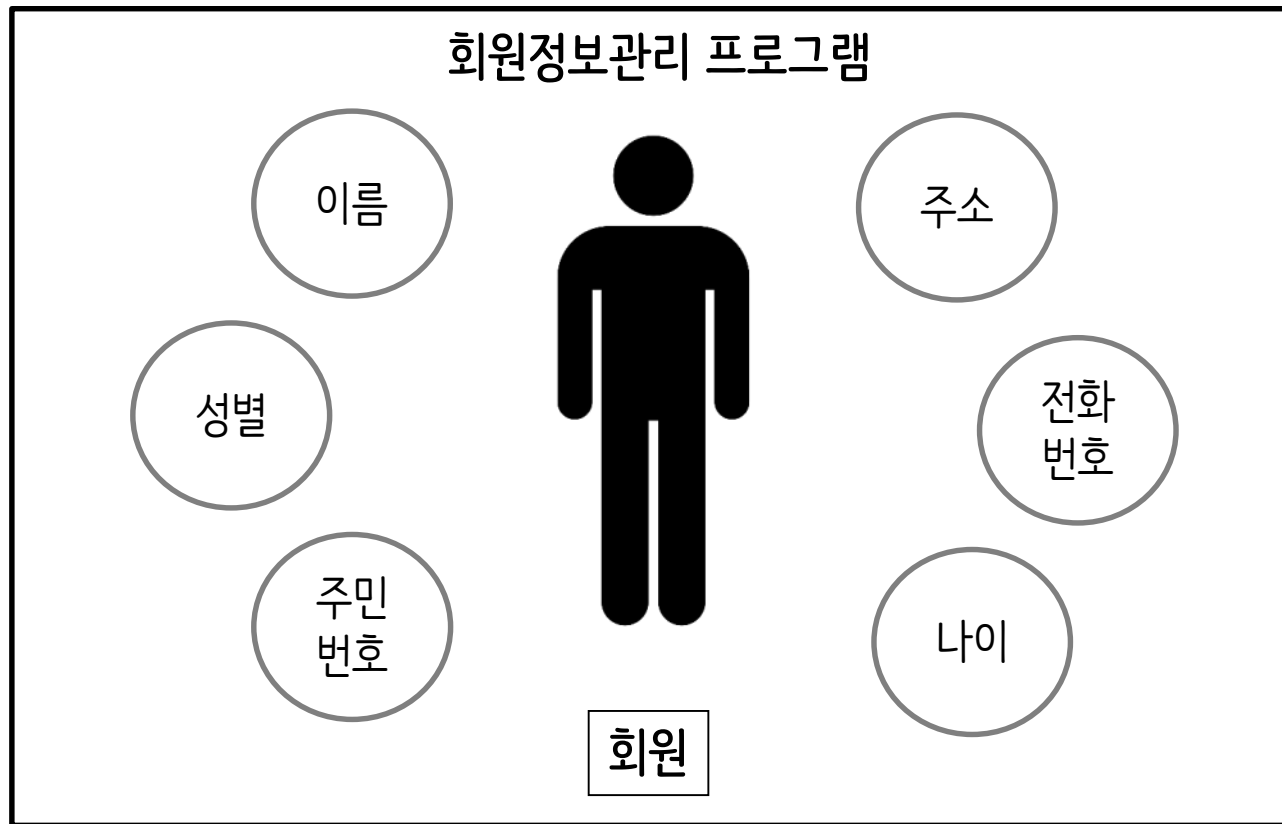
클래스

객체





기업에서 회원정보관리 프로그램을 만들려고 할 때,  
프로그램에서 요구되는 “회원 객체”를 만들기 위해 추상화해본다.



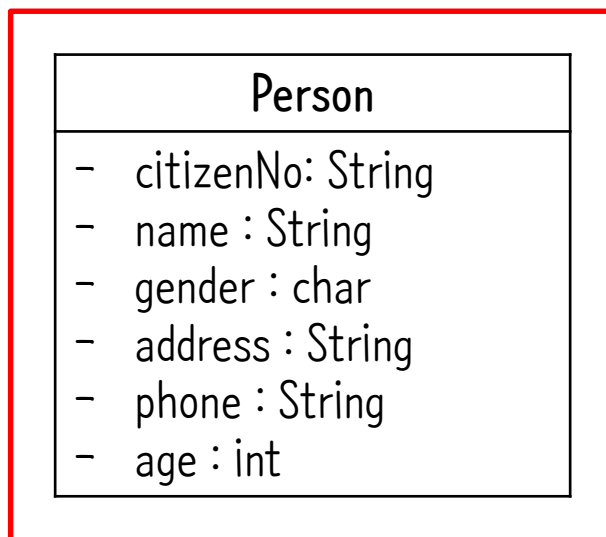
## 추상화(abstraction) → 클래스 작성

추상화한 결과물을 객체 지향 프로그래밍 언어를 사용해서 변수명(데이터이름)과 자료형(데이터타입)을 작성.

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

## UML 다이어그램 표현

앞 페이지에서 정리된 변수명과 자료형을 클래스 다이어그램(UML)으로 표현한다면 아래와 같다.



클래스

다음 상황에서 객체지향적으로 프로그래밍을 하고자한다.  
각각 클래스를 설계해보자.

- 1) 야구 스포츠게임에서 플레이어객체를 만들고자 한다. 실제 야구게임에서 모티브를 얻어 추상화 해보자.
- 2) 결혼정보프로그램을 만들고자 한다. 회원객체를 설계해보자.
- 3) 가구점에서 판매할 상품들에 대해 판매정보를 객체화하려고 한다. 어떤 클래스를 어떻게 설계하겠는가.

클래스(Class)  
공통 특징, 서술

```
public class Dog {  
  
    String name;  
  
    public void move() {}  
    public void bark() {}  
    public void run() {}  
  
}
```

```
public static void main(String[] args) {  
  
    Dog d = new Dog();  
    d.bark();  
  
}
```

오브젝트(object)  
고유성, 구체, 실제 존재

[접근제한자] [예약어] class 클래스명{

[접근제한자] 자료형 변수명;  
[접근제한자] 자료형 변수명;

속성

[접근제한자] 생성자명(){}  
  
[접근제한자][예약어]리턴형 메소드명(){  
    //기능정의  
}

기능

}

예제

```
public class Person{
```

```
    private String name;  
    private int age;
```

속성

```
    public Person(){}  
  
    public String getName(){  
        return name;  
    }  
    public void introduce(){  
        System.out.println("반갑습니다. "  
            +age+"살, "+name+"입니다.");  
    }  
}
```

기능

## 접근제한자란?

- 객체 간의 허용 가능한 접근 범위 제어 및 상속관계에서 부모클래스가 자식클래스에게 허용하는 정보의 범위를 제어하는 지시자

## 접근제한자의 종류

- `public` : 모든 패키지 밖에서 `import` 하여 사용 가능
- `protected` : 비상속시 동일 패키지 내부, 상속 시 패키지 밖의 상속받은 후손클래스 내부에서 사용 가능
- `default` : 동일 패키지 내부에서 사용 가능
- `private` : 클래스 내부에서만 사용 가능



## 접근제한자

구분		같은 패키지 내	전체
+	public	0	0
~	(default)	0	

예) public class 클래스명 {

.....

}

class 클래스명{//생략하면 default클래스

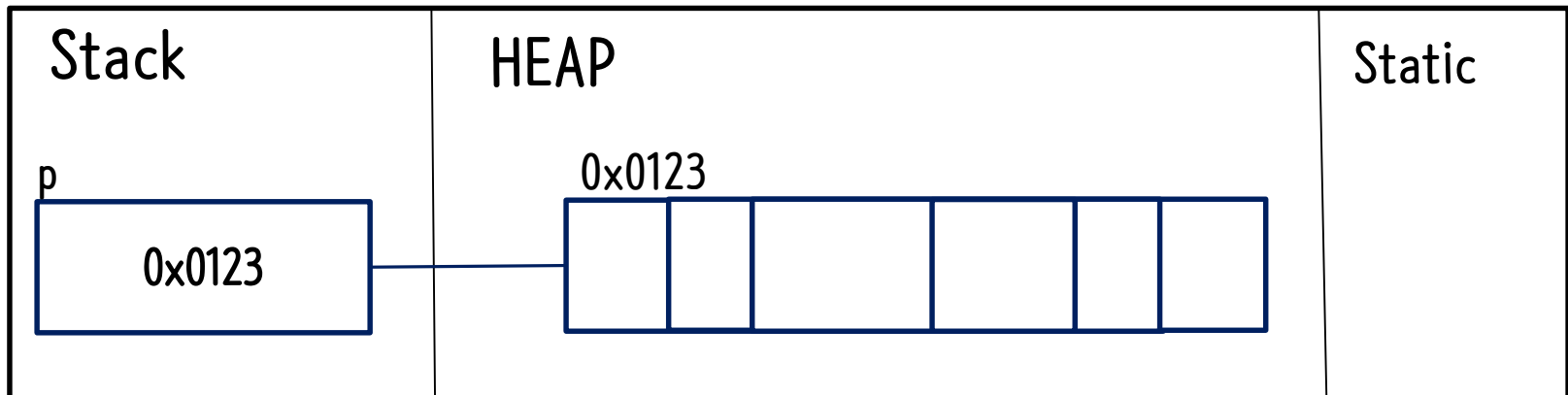
.....

}

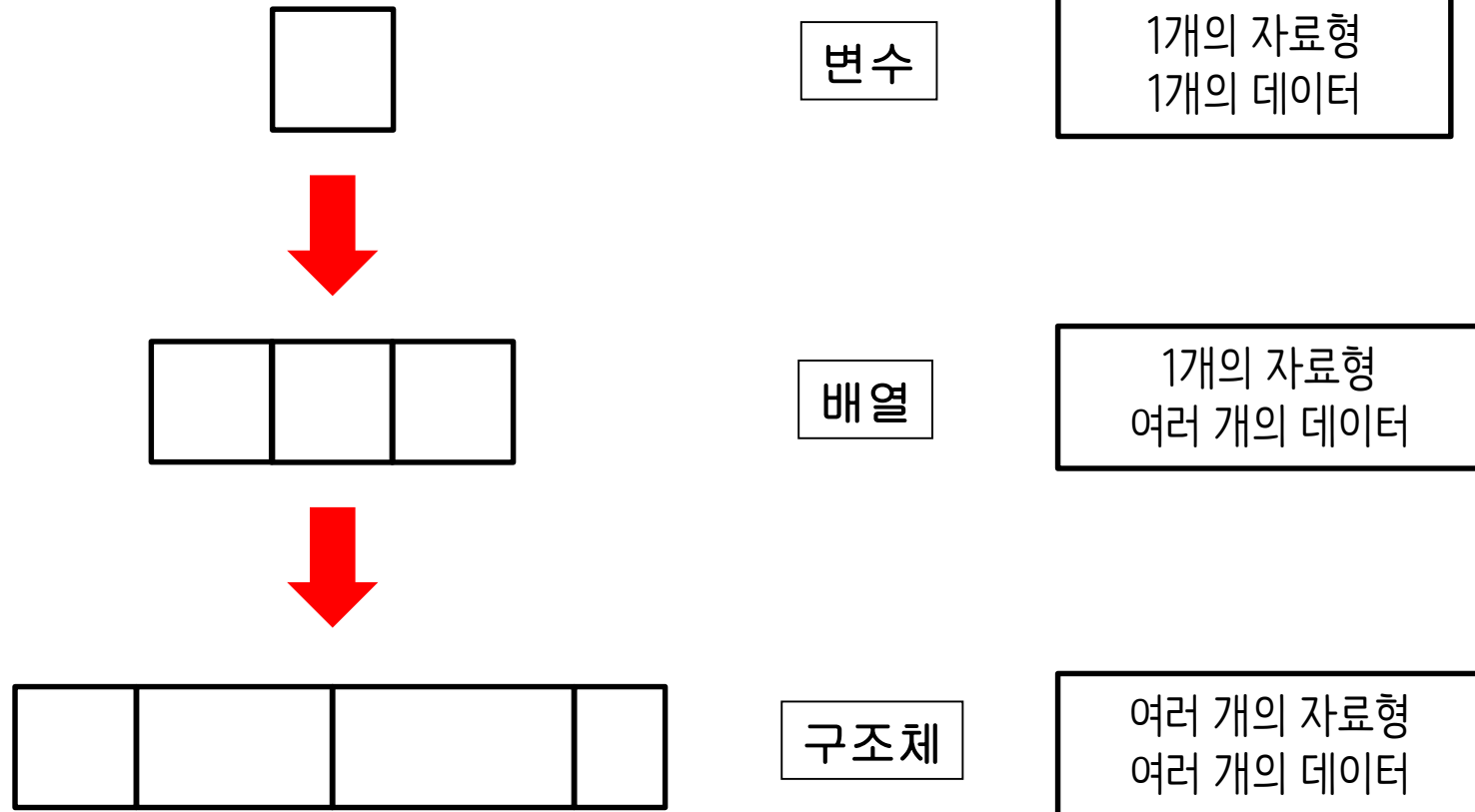
## 객체(Instance)의 할당

new 연산자(생성자)를 사용하여 객체를 생성하면 heap 메모리 공간에 서로 다른 자료형의 데이터가 연속으로 나열 할당된 객체 공간이 만들어진다. 이것을 인스턴스(instance)라고 한다.

예) Person p = new Person();

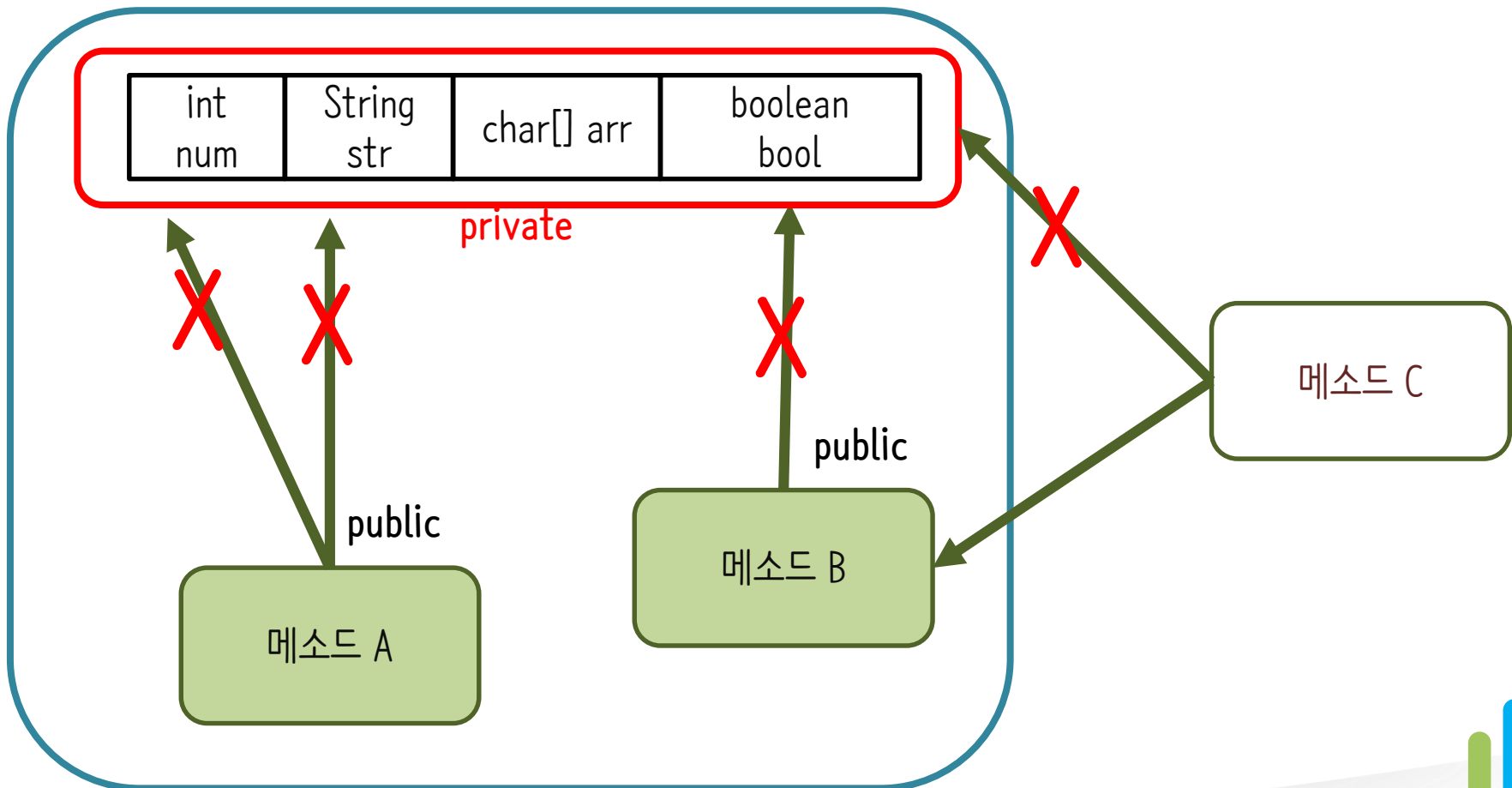


# 클래스의 등장 배경

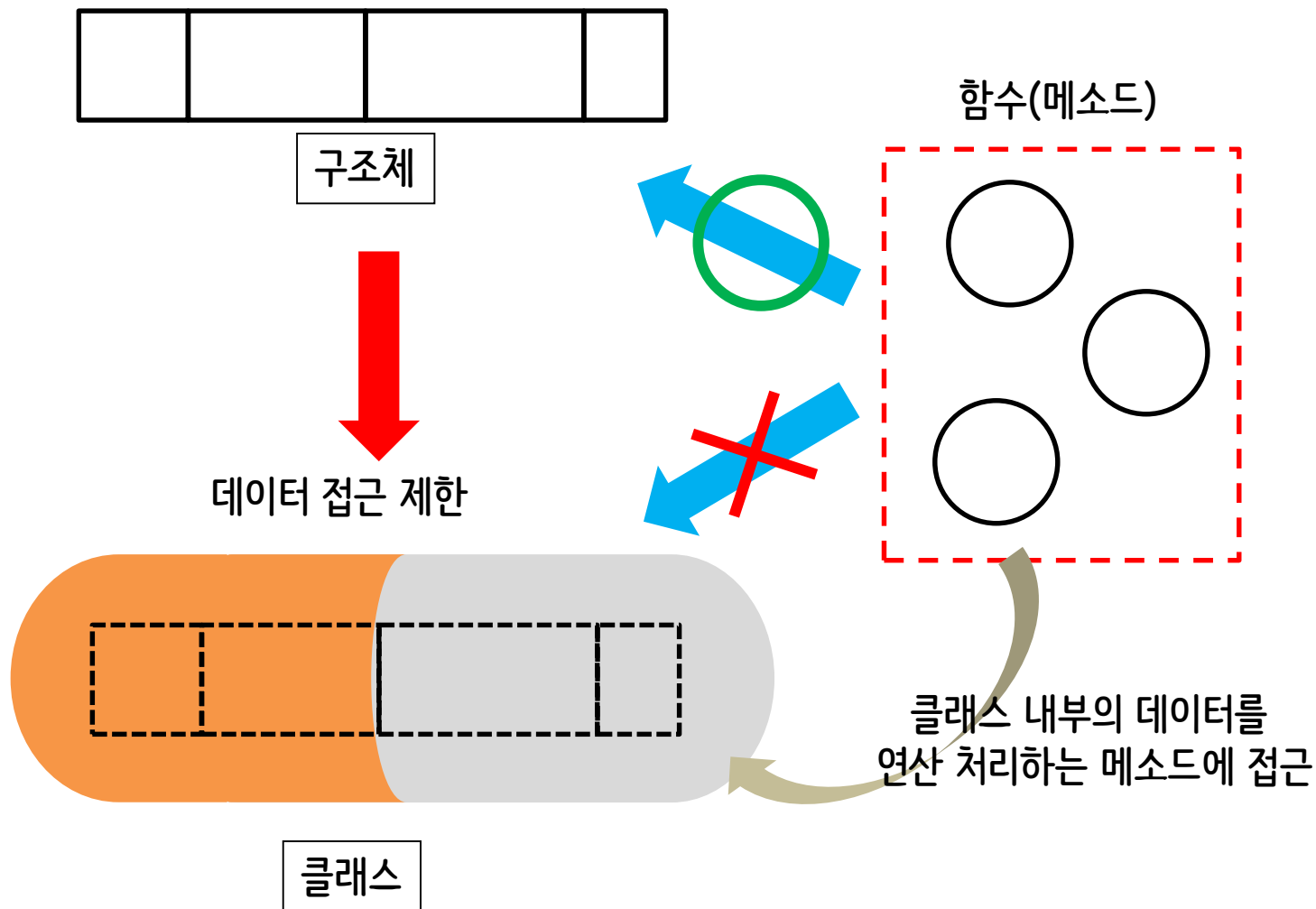


# 클래스의 등장 : 필드 / 메소드

여러 데이터타입을 가진 구조체의 보안문제로 인해 직접접근을 금지  
→ 이게 데이터냐?



# 클래스의 등장 배경 - 캡슐화



**캡슐화(encapsulation)**

상속(inheritance)

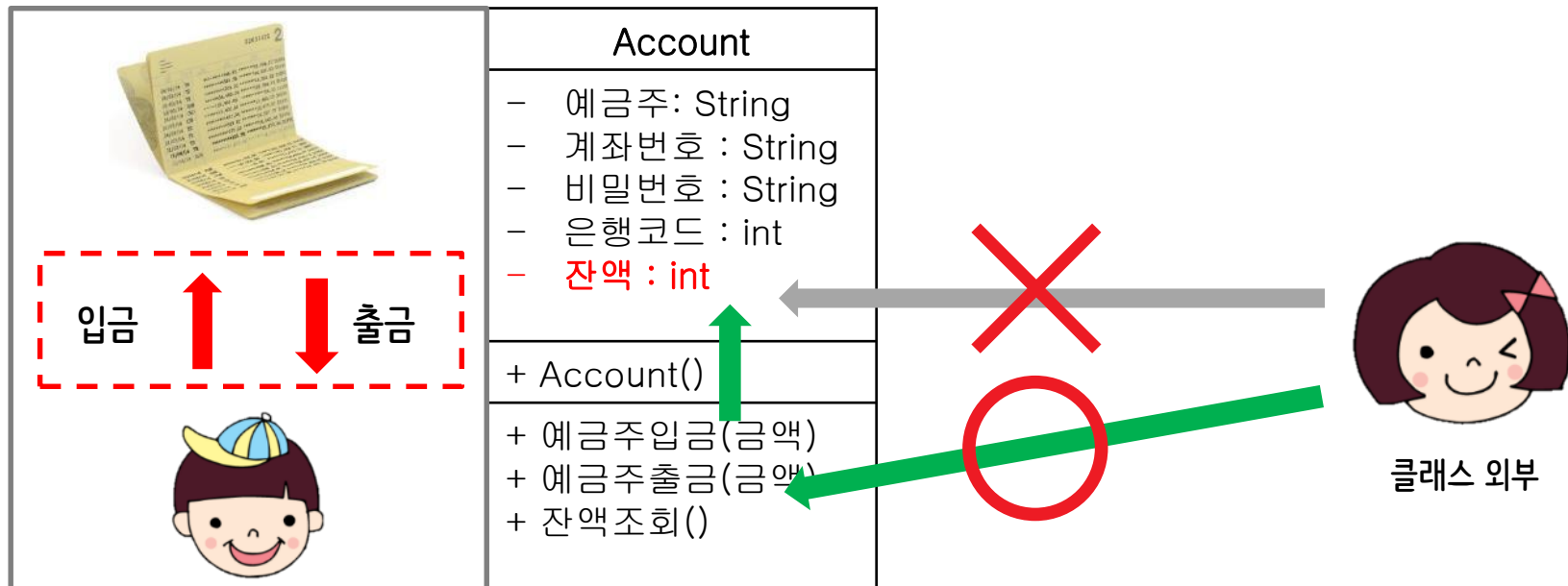
다형성(polymorphism)

추상화를 통해 정리된 데이터들과 기능을 하나로 묶어 관리하는 기법으로, 데이터의 접근 제한을 원칙으로 한다.

# 캡슐화 원칙

필드 : 클래스의 멤버 변수에 대한 접근권한은 **private**을 원칙으로 한다.

메소드 : 클래스의 멤버 변수에 대한 연산처리를 목적으로 하는 메소드를  
클래스 내부에 작성하고, 클래스 밖에서 접근할 수 있도록  
**public**으로 설정한다.

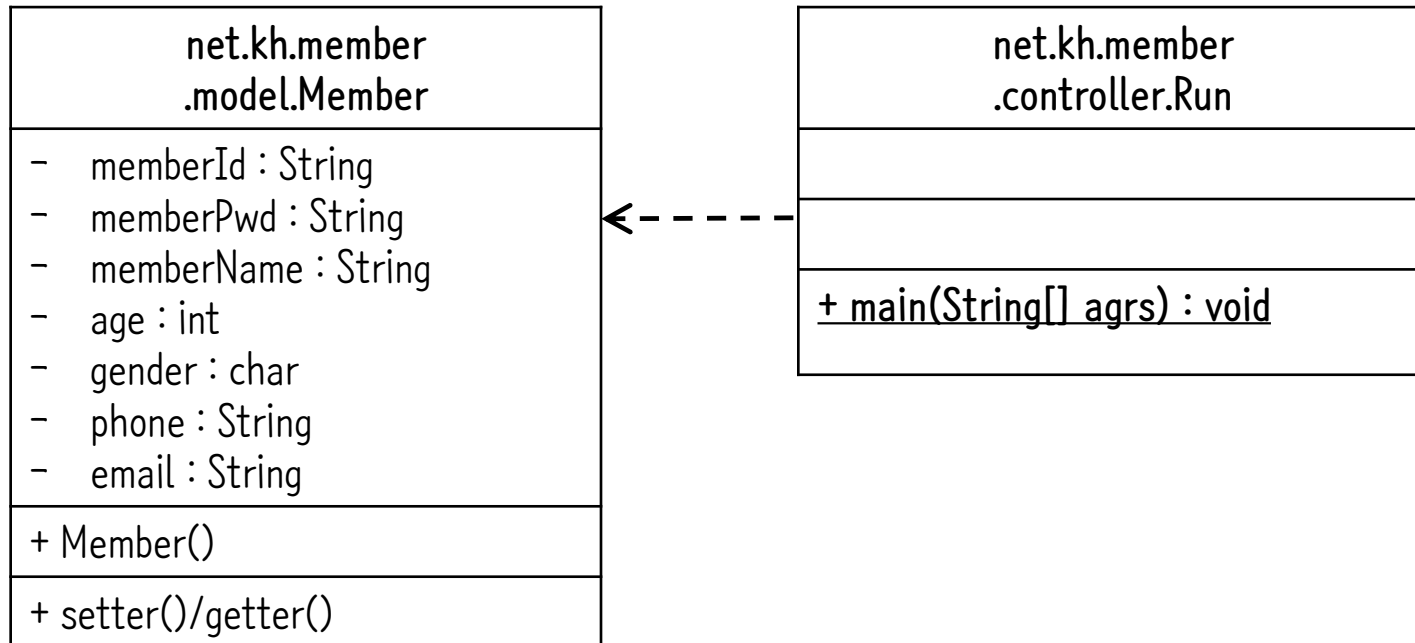


Account 클래스로 생성된 김철수학생 명의의 계좌 객체



# 실습문제1

아래 클래스 다이어그램을 보고 클래스를 작성하세요.

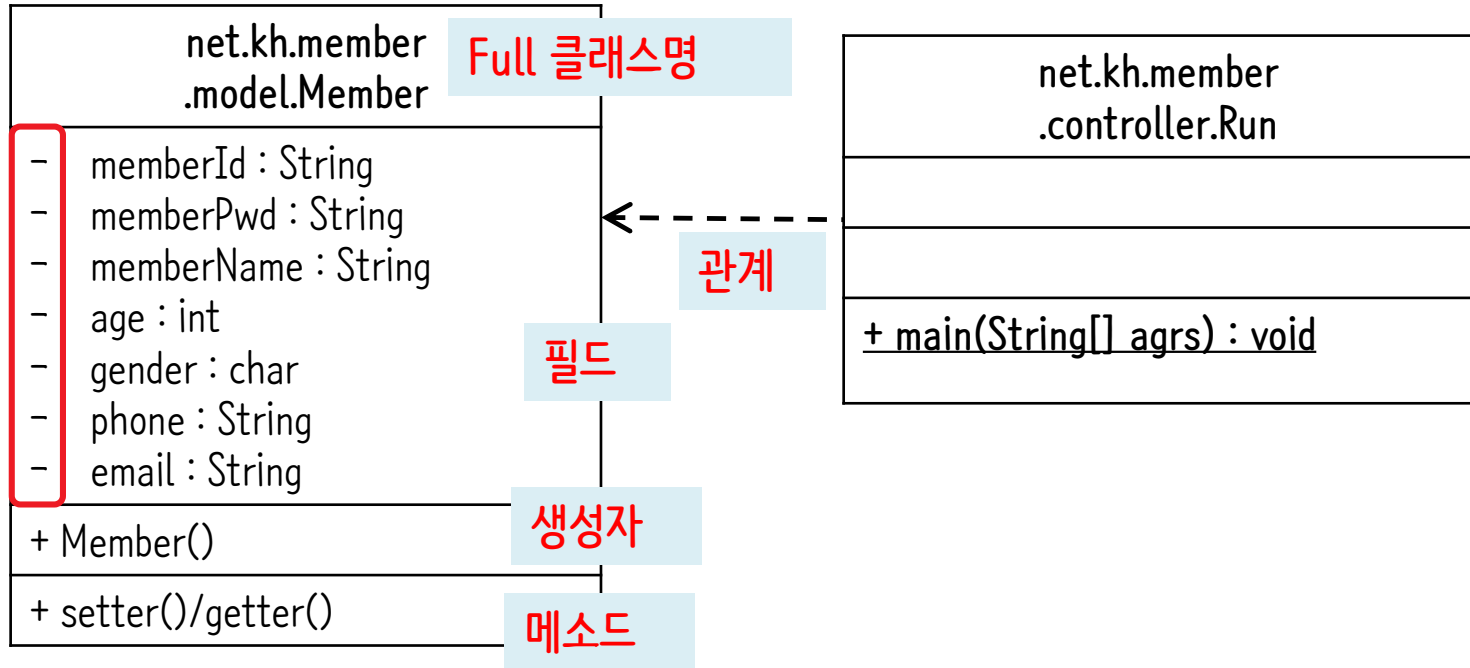


package	class	method	설명
net.kh.member.controller	Run	<u>Main(args:String[]) :</u> <u>void</u>	실행용 메소드 기본 생성자로 객체를 만들고 Setter를 이용해 값 변경 후 Getter를 이용해 출력



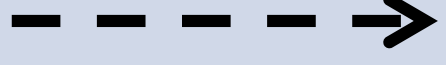



# UML 다이어그램 이해하기

## 접근제한자

- private  
+ public  
~ default  
# protected



# UML 다이어그램 이해하기

관계	UML 표기	설명
Genelization 일반화		상속관계 자식클래스 → 부모클래스
Realization 실체화		구현관계 구현클래스 → 인터페이스
Dependency 의존		객체생성, 객체사용 사용클래스 → 피사용클래스
Association 연관		필드로 다른 객체를 참조. 사용클래스 → 피사용클래스
Aggregation 약집합		필드로 다른 객체를 참조. 1:다 사용클래스 → 피사용클래스
Composition 강집합 (완전포함)		필드로 다른 객체를 참조. 1:1 사용클래스 → 피사용클래스

# 필드(field)

변수의 선언위치에 따라 구분한다.

1. 클래스변수 : 클래스영역에 static키워드를 가짐.
2. 멤버변수(인스턴스변수) : 클래스영역에 선언.
3. 지역변수 : 클래스영역이 아닌, 메서드, 생성자, 초기화블록내부에서 선언

```
class Var {  
    public static int size;  
    public int num;  
  
    public void test(){  
        int num = 100;  
    }  
}
```

클래스영역

메소드영역

# 변수에 따른 소멸시기

변수구분	소멸시기
클래스변수	프로그램 종료시
멤버변수(인스턴스변수)	객체소멸시(GC 소관)
지역변수	메소드 종료시

# 클래스변수 예약어 - static

static : 같은 타입의 여러 객체가 공유할 필드에  
사용하며, 프로그램 start시에 정적 메모리영역에  
자동 할당되는 멤버에 적용한다.

표현식

```
접근제한자 class 클래스명 {  
    접근제한자 static 자료형 변수명;  
}
```

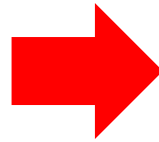
```
예) public class Student{  
    private static String academy = "kh정보교육원";  
    private String name;  
}
```

예) class Student{  
    public int var1;  
    protected int var2;  
    int var3; //접근제한자 생략하면 default임  
    private int var4; //캡슐화의 원칙임  
}



구분		해당클래스 내부	같은 패키지 내	후손 클래스 내	전체
+	public	0	0	0	0
#	protected	0	0	0	
~	(default)	0	0		
-	private	0			

kh.java.edu.Hello
- a : String = "a" + b : int = 10 ~ c : String = "c" - <u>d : String = "d"</u>
+Hello()
+getter(), setter()



```
kh.java.World  
public static void main(String args[]){  
  
    Hello h = new Hello();  
  
    System.out.println(h.a);  
    System.out.println(h.b);  
    System.out.println(h.c);  
    System.out.println(h.d);  
  
}
```

final : 하나의 값만 계속 저장해야 하는 변수에 사용  
표현식

```
class 클래스명 {
```

```
    접근제한자 final 자료형 변수명 = 초기값;
```

```
}
```

예) class Product {

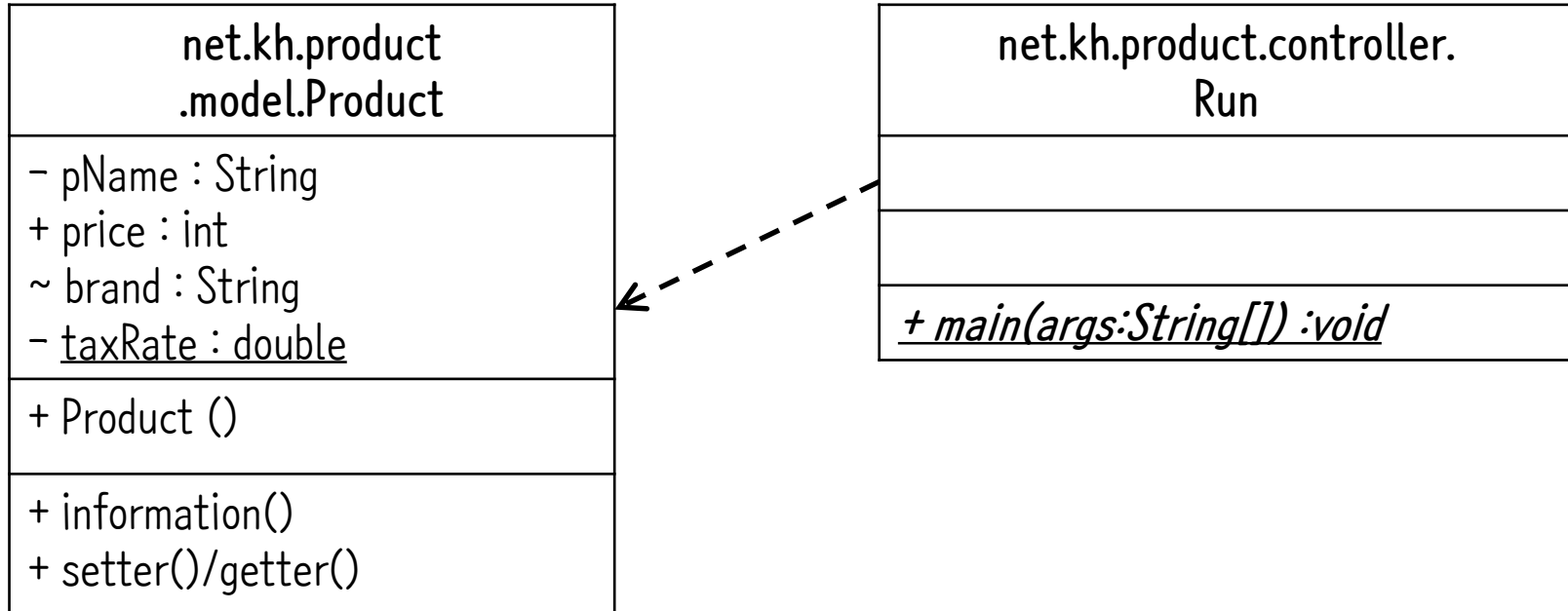
```
    public static final double TAX = 3.3;
```

```
    private int price;
```

```
}
```

# 실습문제2

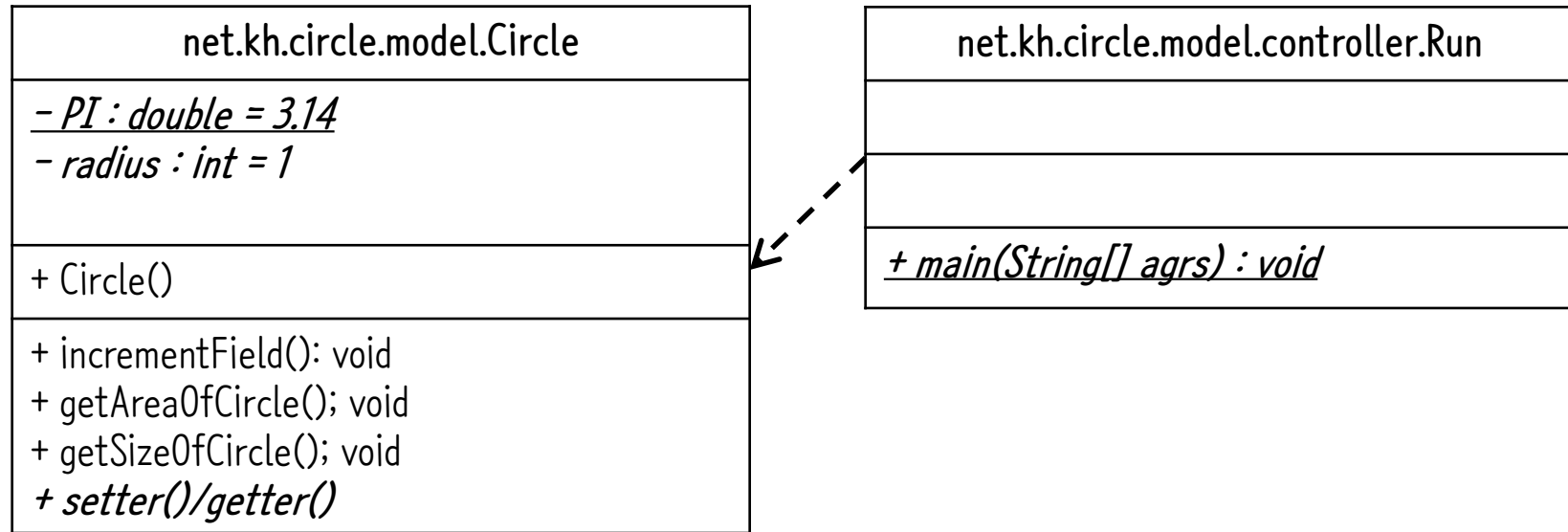
아래 클래스 다이어그램을 보고 클래스를 작성하세요.



package	class	method	설명
net.kh.product.controller	Run	<u>Main(args:String[])</u> : void	실행용 메소드 Product객체생성후 setter를 통해 값을 지정 information()으로 출력  각필드에 직접접근해서 값을 출력. 접근이 불가능한 필드가 있다면, 그 이유에 대해 주석달기.

# 실습문제3

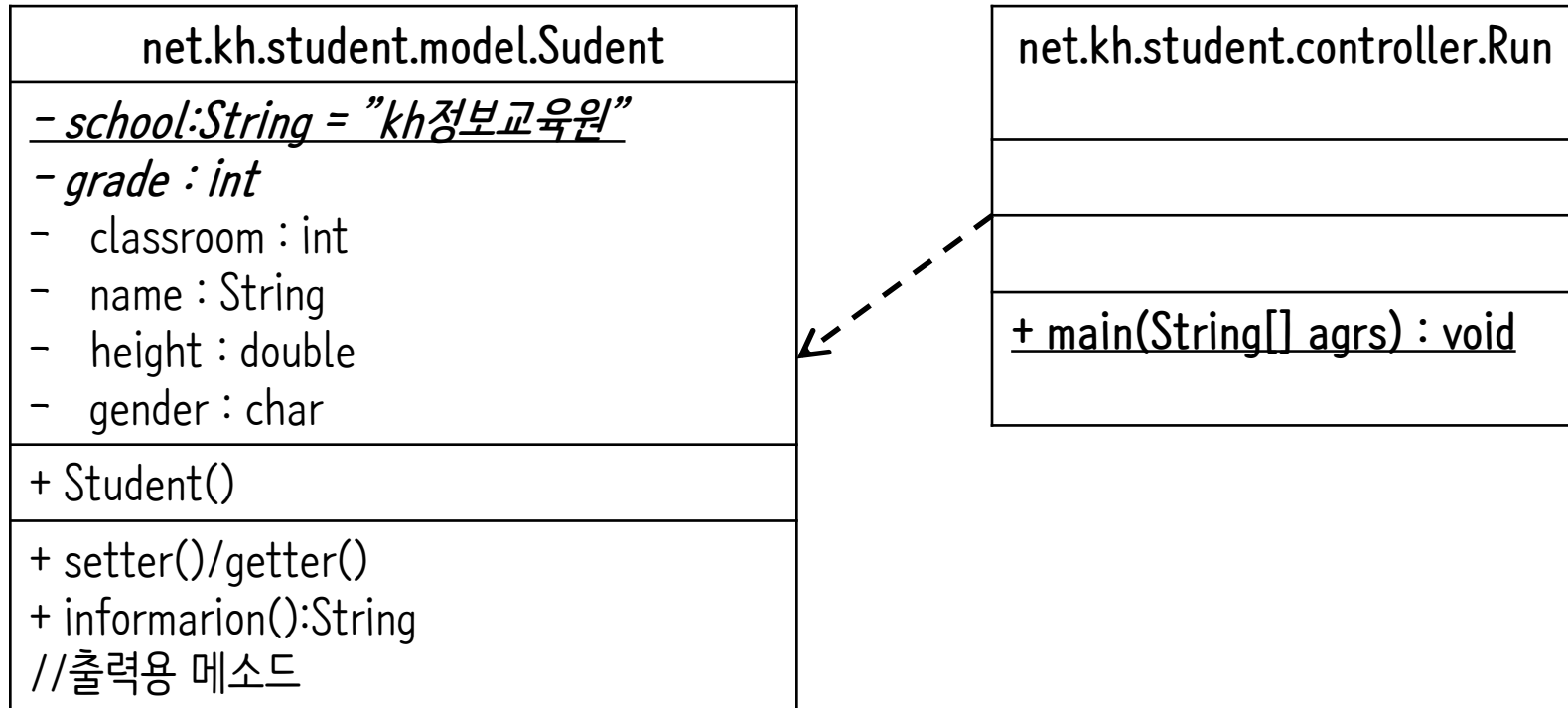
아래 클래스 다이어그램을 보고 클래스를 작성하세요.



package	class	method	설명
net.kh.circle.controller	Run	<u>Main(args:String[] ) : void</u>	실행용 메소드 필드에 값 셋팅후 원주, 원둘레 구하기. incrementField를 통해 반지름 1증가후, 다시 원주/원둘레 구하기

# 실습문제4

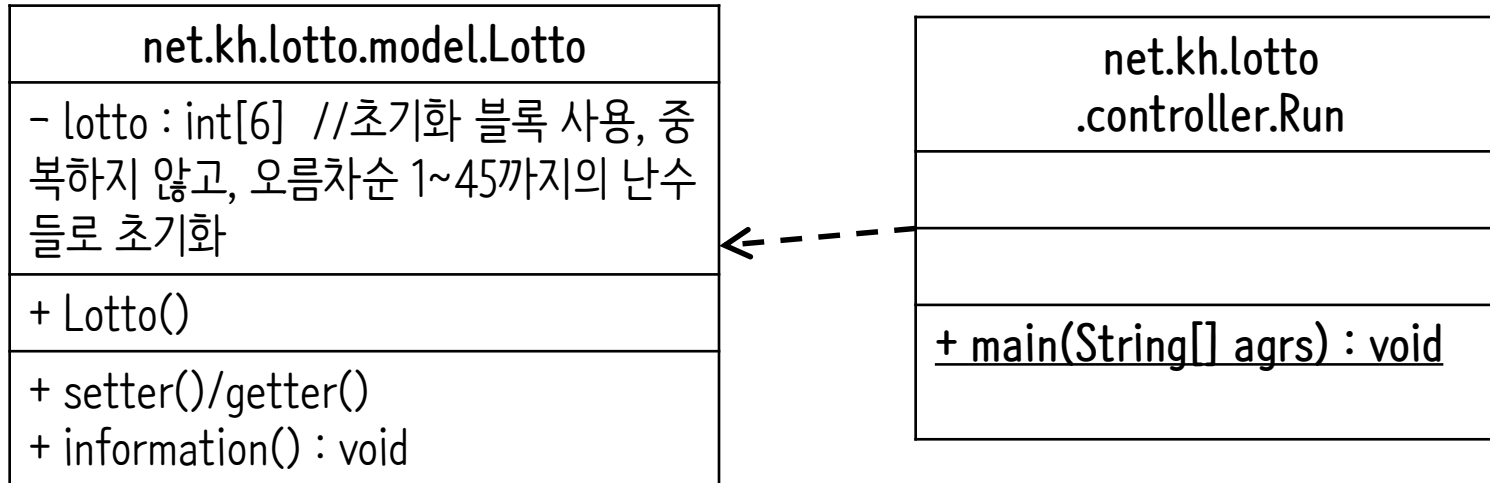
아래 클래스 다이어그램을 보고 클래스를 작성하세요.



package	class	method	설명
net.kh.student.controller	Run	<u>Main(args:String[] ) : void</u>	실행용 메소드 Student 객체 생성 후 setter이용 값 대입. Information()으로 출력

# 실습문제5

아래 클래스 다이어그램을 보고 클래스를 작성하세요.



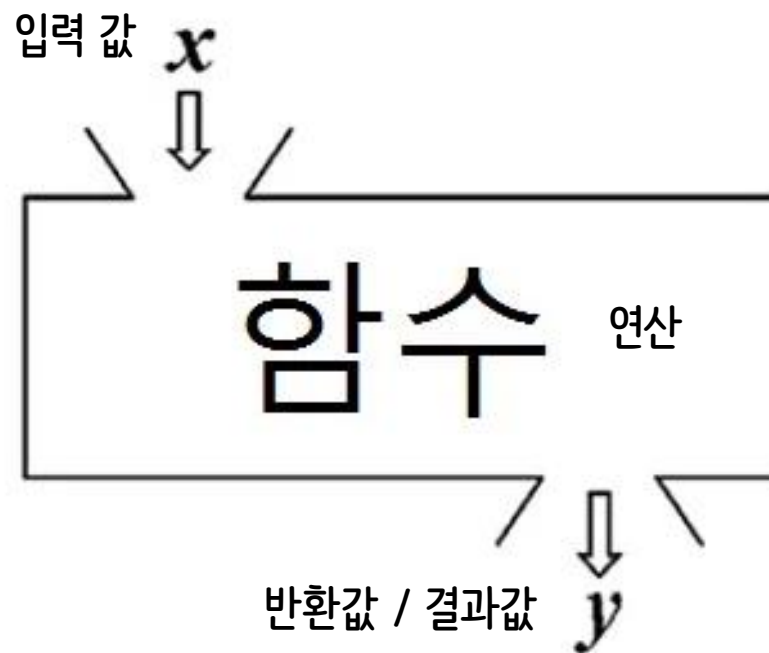
package	class	method	설명
net.kh.lotto.model.vo	Lotto	infor Marion():String	반복문을 이용하여 배열을 출력하는 메소드
net.kh.lotto.controller	Run	<u>Main(args:String[]) : void</u>	실행용 메소드 Lotto객체 생성 후 Information()으로 출력

# 메소드(method)



## 메소드란

수학의 함수와 비슷하며 호출을 통해 사용한다. 호출시 전달값은 있거나 없을수 있다. 함수가 호출되면, 내부에 작성된 연산을 수행하게되며, 연산 후 결과값/반환값은 있거나 없을 수 있다.



## 표현식

[접근제한자][예약어] 반환형 메소드명(매개변수){

//실행내용 작성

}

예) public void showInformation(){  
    System.out.println(userName);  
}

# 메소드의 접근제한자

구분		해당클래스 내부	같은 패키지 내	후손 클래스 내	전체
+	public	0	0	0	0
#	protected	0	0	0	
~	(default)	0	0		
-	private	0			

구 분	설 명
static	static 영역에 할당하여 객체 생성 없이 사용함
final	종단의 의미, 상속시 오버라이딩 불가능
abstract	미완성된, 상속하여 오버라이딩으로 완성시켜 사용해야 함
synchronized	동기화처리, 공유 자원에 한 개의 스레드만 접근 가능함
static final (final static)	static과 final의 의미를 둘 다 가짐

구 분	설 명
void	반환값이 없을 경우
기본자료형	반환값이 8가지 기본 자료형일 경우
배열	기본형, 클래스의 배열 모두 가능.
클래스	반환값이 해당 클래스 타입의 객체일 경우 사용함 (클래스 == 타입)

\* 반환형은 단 한 개만 가질 수 있다.

구 분	설 명
()	매개변수가 없는것을 의미함
기본자료형	기본형 매개변수 사용시 값을 복사하여 전달함. 매개변수 값을 변경하여도 원래 값은 변경되지 않는다.
배열	배열, 클래스등 참조형을 매개변수로 전달시에는 데이터의 주소값을 전달하므로 매개변수를 수정하면 본래 데이터가 수정된다.
클래스	
가변인자	매개변수의 개수를 유동적으로 설정하는 방법, 가변매개변수와 다른 매개변수가 있으면 가변매개변수를 마지막에 설정해야 함. 방법 : (자료형 ... 변수명)

\* 매개변수의 수에 제한이 없다.

# 오버로딩 Overloading

한 클래스 내에서 **파라미터선언부가 다르고, 이름이 같은 메서드**를 여러 개 정의하는 것.



오버로딩 성립조건 2가지

1. 메소드 이름이 같아야한다.
2. 매개변수 선언부가 달라야한다.
  - 매개변수 타입, 개수, 순서

오버로딩 주의점

1. 매개변수명은 상관치 않는다.
2. 리턴타입은 오버로딩시 상관치 않는다.

java.io.PrintStream의 println() 메소드의 오버로딩을 살펴보자.

# 오버로딩 Overloading - 메소드

```
public int test(){}  
public int test(int a){}  
public int test(int a, int b){}  
public int test(int a, String s){}
```

```
public int test(int b, int a){}  
public int test(String s, int a){}
```

```
public String test(int a, int b, String str){}  
private int test(String str, int a, int b){}  
public int test(int a, long b, String str){}
```



# 메소드의 종류

매개변수가 없고 리턴값이 있거나 없거나

표현식

- 리턴값 있는것

[접근제한자] 리턴형 메소드명() {

.....

return 반환값;

}

- 리턴값 없는것

[접근제한자] void 메소드명() {

.....

}

# 메소드의 종류

매개변수가 있고 리턴값이 있거나 없거나

표현식

- 리턴값 있는것

접근제한자 리턴형 메소드명(자료형 변수명){

.....

return 반환값;

}

- 리턴값 없는것

접근제한자 void 메소드명(자료형 변수명){

.....

}

호출할 클래스명, 메소드명을 적고 설정된 매개변수와 동일한 자료형의 값을 전달.

## 표현식

매개변수 있는 경우 : 참조형 변수명.메소드명 (매개변수);

매개변수 없는 경우 : 참조형 변수명.메소드명();

```
public void showInformation(){  
    Person ps=new Person();  
    ps.setAge(19);           //매개변수 있는 경우  
    int age = ps.getAge();   //매개변수 없는 경우  
}
```

## Setter 메소드

-필드에 변경할 값을 전달 받아서 필드값을 변경하는 메소드

-매개변수가 필드와 동일한 이름일 경우 this연산자를 붙여서 구분

```
접근제한자 void set필드명(자료형 변수) {  
    (this.)필드명 = 자료형 변수;  
}
```

## Getter 메소드

- 필드에 기록된 값을 읽어서 요구하는 쪽으로 읽은 값을 넘기는 메소드

```
접근제한자 반환형 get필드명() {  
    return 필드명;  
}
```

## Setter 메소드

```
public void setBalance(int balance){  
    this.balance = balance;  
}
```

## Getter 메소드

```
public int getBalance(){  
    return balance;  
}
```

모든 인스턴스의 메소드에 숨겨진 채 존재하는 지역변수로,  
해당 객체의 주소값을 담고 있다.

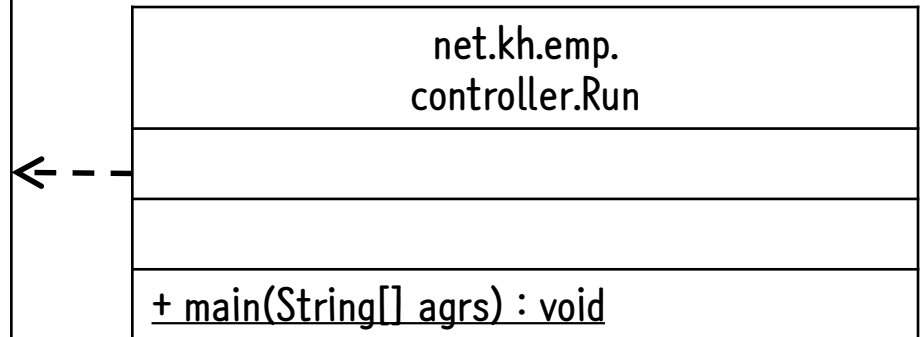
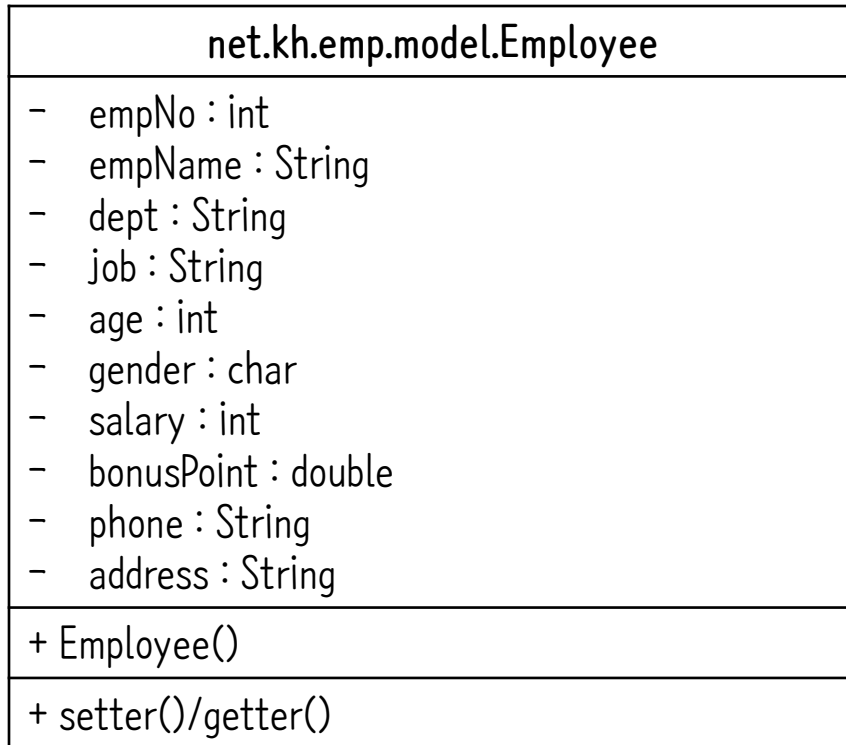
```
예) class Academy{  
    String name;  
  
    public Academy(){  
    public Academy(String name){  
        this.name = name;  
    }  
}
```

매개변수를 가지는 생성자 또는 메소드에서 매개변수명이 필드명과 같을 경우 매개변수의 변수명이 우선하기 때문에 this를 활용해서 매개변수와 필드를 구분한다.

# 실습문제7

아래 클래스 다이어그램을 보고 클래스를 작성하세요.

empNo	empName	dept	job	age	gender	salary	bonusPoint	phone	address
100	홍길동	영업부	과장	25	남	2500000	0.05	010-1234-5678	서울시 강남구



package	class	method	설명
net.kh.emp. controller	Run	<u>Main(args:String[]) :</u> <u>void</u>	실행용 메소드 기본 생성자로 객체를 만들고 Setter를 이용해 값 변경 후 Getter를 이용해 출력

# 생성자



# 클래스 생성자

## 생성자란

- 객체를 생성할 때 항상 실행되는 것으로, 메소드 중에서 맨 처음 실행되는 메소드
- 객체의 초기화란 클래스가 객체를 생성/호출하였을 때, 객체의 필드를 초기화하거나 메소드를 호출해서 객체를 사용할 준비를 하는 것으로 일련의 준비단계

# 클래스 생성자

## 생성자의 규칙

- 클래스에는 반드시 생성자가 존재해야 한다.
- 인스턴스 생성시 딱 한번 호출된다.
- 클래스 이름과 동일한 이름을 가진 메소드이다.
- 반환형이 존재하지 않는다. 즉 `return`이 없다.

# 클래스 생성자

## 생성자 종류

객체를 호출할 때 클래스는 생성자를 통해 객체를 초기화하게 되는데 구현 방법에 따라 기본 생성자, 매개변수가 있는 생성자로 나뉘게된다.

## 생성자 종류

### 기본 생성자 (매개변수가 없는 생성자)

- 작성하지 않은 경우, 클래스 사용시 JVM이 기본 생성자 자동 생성

### 매개변수 있는 생성자

- 객체 생성시 전달 받은 값으로 객체를 초기화 하기 위해 사용함.
- 매개변수 있는 생성자 작성시 JVM이 기본 생성자 자동 생성 하지 않음
- 상속 사용시 반드시 기본 생성자를 작성해야 함
- 오버로딩을 이용하여 작성함

# 클래스 매개변수 있는 생성자

표현식

```
class 클래스명 {
```

```
    [접근제한자] 클래스명(){} //기본 생성자
```

```
    [접근제한자] 클래스명(매개변수){
```

```
        (this.)필드명 = 매개변수
```

```
    };
```

```
}
```

# 클래스 매개변수 있는 생성자

클래스에 생성자를 만들면 디폴트 생성자가 자동으로 생성되지 않아 디폴트 생성자를 활용 하려면 코드를 추가 해줘야 함.

예) class Person{  
    private int age;  
    private String name;

```
public Person(){} //직접추가해야함.  
public Person(int age, String name){  
    this.age = age;  
    this.name = name;  
}
```

```
}
```

# 오버로딩 Overloading - 생성자

한 클래스 내에서 **파라미터선언부가 다른 생성자**를 여러 개 정의하는 것.



오버로딩 성립조건 2가지

1. 메소드 이름이 같아야한다.
2. 매개변수 선언부가 달라야한다.
  - 매개변수 타입, 개수, 순서

java.util.Scanner의 생성자 오버로딩을 살펴보자.

# 오버로딩 Overloading - 생성자

```
class Truck {  
    public Truck(){}  
  
    public Truck(String str){}  
  
    public Truck(String k){}  
  
    public Truck(String str, int x, int j){}  
  
    public Truck(int x, int j, String str){}  
}
```



# this()

생성자, 같은 클래스의 다른 생성자를 호출할 때 사용

**\*\* 반드시 첫번째 줄에 선언해야 한다.**

예)

```
class Student{
    int age;
    String name;

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

# this()를 통한 실행순서

Suit s = new Suit(100); 를 실행한다면??

```
public class Suit {  
    private int size;  
    private String brand;  
    private int price;  
  
    public Suit(int size) {  
        this(size, "Hazzys");  
        System.out.println("Suit(int) 호출!!");    //(1)  
    }  
    public Suit(int size, String brand){  
        this(size, brand, 100000);  
        System.out.println("Suit(int, String)호출");    //(2)  
    }  
    public Suit(int size, String brand, int price){  
        this.size = size;  
        this.brand = brand;  
        this.price = price;  
        System.out.println("Suit(int, String,int)호출");    //(3)  
    }  
}
```