

JAVA

필드와 메소드





1 필드와 메소드

1. 필드

2. 메소드





필드

1) 필드와 변수의 종류

변수는 선언되는 위치에 따라서 다음과 같은 종류가 있다.

- 필드(field) 또는 인스턴스 변수 : 클래스 안에서 선언되는 멤버 변수
- 지역 변수(local variable) : 메소드나 블록 안에서 선언되는 변수
- 매개 변수(parameter): 메소드 선언 시 () 안에 선언되는 변수
메소드를 실행할 때 필요한 데이터를 외부에서 받기 위해 정의.



필드

1) 필드와 변수의 종류

```
class 클래스명{  
    public int num; ←----- 필드선언  
    ...  
    void start(int n)←{----- 매개변수  
        int t; ←----- 지역변수  
        ...  
    }  
}
```



필드의 선언 형식

Car 클래스는 speed라는 필드를 선언하고 있었다.

자료형

필드명: 소문자로 시작

```
public int speed ;
```

접근 수식자: private이나 public

필드의 접근 수식자는 어떤 클래스가 필드에 접근할 수 있는지를 표시한다.

- **public** : 이 필드는 모든 클래스로부터 접근가능하다.
- **private** : 클래스 내부에서만 접근이 가능하다.

필드의 사용 범위

```
class Account {  
    public void deposit(int amount){ //예금  
        balance += amount;  
    }  
    public int withdraw(int amount){ //인출  
        if(balance < amount)  
            return 0;  
        balance -= amount;  
        return amount;  
    }  
    public String accountNo; //계좌번호  
    public String ownerName; //예금주명  
    public int balance; //잔액  
}
```

선언 위치와
는 상관없이
어디서나 사
용이 가능하
다.



필드와 지역변수

모든 변수는 사용될 수 있는 범위(scope)가 결정되어 있다.

```
class Box{  
    private int width; ←----- 필드: 클래스안에서 사용가능  
    private int height;  
    ...  
    public int start(int n) {  
        int rectangleArea; ←-----  
        rectangleArea = width*height ;  
        return rectangleArea;  
    }  
}
```

지역변수: 메서드 안에 서만 사용가능



필드의 초기화

필드값은 선언과 동시에 초기화 될 수 있다.

```
public class ClassName {  
    public int capacity = 1;  
    public boolean use = false;  
}
```

필드의 초기값이 지정되지 않는다면 디폴트 값은 int형과 같은 수치인 경우에는 0이고 논리형은 false, 참조형은 null이 된다.



필드의 초기화

❖ 필드의 기본 초기값

- 초기값 지정되지 않은 필드
 - 객체 생성시 자동으로 기본값으로 초기화

데이터 타입	초기화	데이터 타입	초기화
byte	0	short	0
int	0	long	0l
float	0.0f	double	0.0
char	'\u0000'(null)	boolean	false
배열	null	클래스 (String 포함)	null



필드와 지역변수

```
public class BugProgram {  
    ...  
    public int getAverage(int n1, int n2) {  
        int sum;  
        sum += n1;  
        sum += n2;  
        return sum/2;  
    }  
}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The local variable sum may not have been initialized



예제

Car.java

```
public class Car {  
    // 필드 정의  
    public int speed;    // 속도  
  
    public String toString() {  
        return "속도: "+speed;  
    }  
}
```

CarTest.java

```
public class CarTest {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.speed = 0;  
        System.out.println(myCar.toString());  
    }  
}
```



접근자와 설정자

클래스 선언할 때 필드는 일반적으로 **private** 접근 제한

- 외부에서 엉뚱한 값으로 변경할 수 없도록 (Setter의 필요성)
- 읽기 전용 필드가 있을 수 있음 (Getter의 필요성)

필드와 관련된 두가지 메소드가 있다.

필요할 경우 필드 값을 가공.

설정자(mutator) - Setter 필드의 값을 설정하는 메소드	접근자(accessor) - Getter 필드의 값을 반환(리턴)하는 메소드
형식	
set FieldName(매개변수) 메소드	get FieldName() 메소드



접근자와 설정자의 사용 이유

접근자와 설정자를 사용하여서 필드를 간접적으로 접근하는 것은 다음과 같은 이점을 가져다 준다.

- 설정자에게 매개변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있다. (Setter의 필요성)
- 필요할 때마다 필드값을 계산하여 반환할 수 있다.
- 접근자만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있다. (Getter의 필요성)



this

❖this

- 객체(인스턴스) 자신의 참조(번지)를 가지고 있는 키워드
- 객체 내부에서 인스턴스 멤버임을 명확히 하기 위해 **this.** 사용
- 매개변수와 필드명이 동일할 때 인스턴스 필드임을 명확히 하기 위해 사용

```
class Car{  
    private int speed;  
  
    public void setSpeed(int speed) {  
        this.speed = speed;  
    }  
}
```



설정자와 접근자

설정자와 접근자를 설정하면 다음과 같다.

```
class Box{  
    private int width;  
    private int height;  
  
    public int getWidth() {  
        return width;  
    }  
    public void setWidth(int width) {  
        this.width = width;  
    }  
  
    ...  
}
```



중간점검

1. 자바에는 몇 가지 종류의 변수가 존재하는가?
2. 필드의 값을 직접 접근하지 않고 접근자와 변경자 메소드를 사용하는 이유는 무엇인가?
3. 지역변수와 필드의 차이점은 무엇인가?



메소드

메서드는 반복적으로 실행하고자 하는 명령어들의 집합이다. 한번 선언으로 여러 번 호출해 사용할 수 있어서 코드의 재사용을 높여준다.

멤버 메서드는 클래스 내에서 메서드를 말하며 객체가 해야 하는 기능을 정의한 함수이다.

반환형: 메소드로부터 반환되는 데이터의 타입

```
public int sum(int x, int y){  
    return x + y;  
}
```

매개변수 : 외부로부터 전달받는 데이터

접근 제어 지정자: public, private와 같은 접근제어를 나타낸다

메소드가 값을 반환할 때 return 사용.

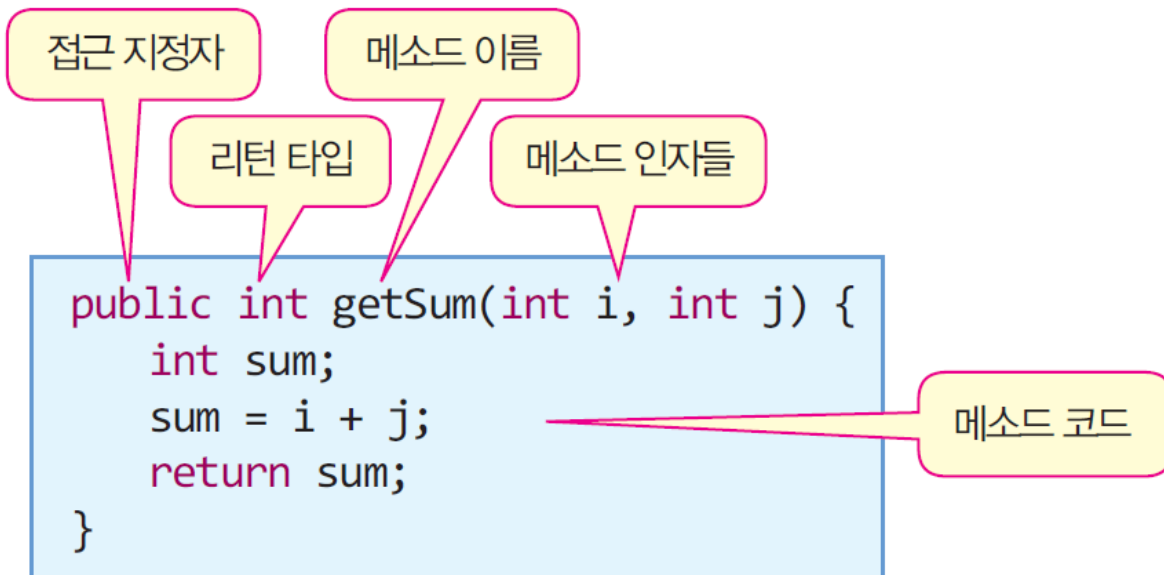
메소드 형식

메소드

- 클래스의 멤버 함수, C/C++의 함수와 동일
- 자바의 모든 메소드는 반드시 클래스 안에 있어야 함(캡슐화 원칙)

메소드 구성 형식

- 접근 지정자
 - public, private, protected, 디폴트(접근 지정자 생략된 경우)
- 리턴 타입
 - 메소드가 반환하는 값의 데이터 타입





메소드

메소드는 외부에서 값을 받을 수 있다. 바로 메소드에 정의된 매개변수를 통하여 값을 받을 수 있다.

- 메소드 호출시 전달하는 값을 인수(argument)
- 메소드에서 값을 받을 때 사용하는 변수를 매개 변수(parameter)



메소드(method)

❖ 메소드 매개변수 선언

- 매개변수는 메소드를 실행할 때 필요한 데이터를 외부에서 받기 위해 사용
- 매개변수도 필요 없을 수 있음

[메소드 선언]

```
void powerOn(){ ... }  
double divide(int x, int y){ ... }
```

[메소드 호출]

```
powerOn();  
double result = divide(10, 20);
```

```
int b1 = 10;  
int b2 = 20;  
double result = divide(b1, b2);
```



메소드(method)

❖ 리턴(return) 문

- 메소드 실행을 중지하고 리턴값 지정하는 역할
- 리턴값이 있는 메소드
 - 반드시 리턴(return)문 사용해 리턴값 지정해야

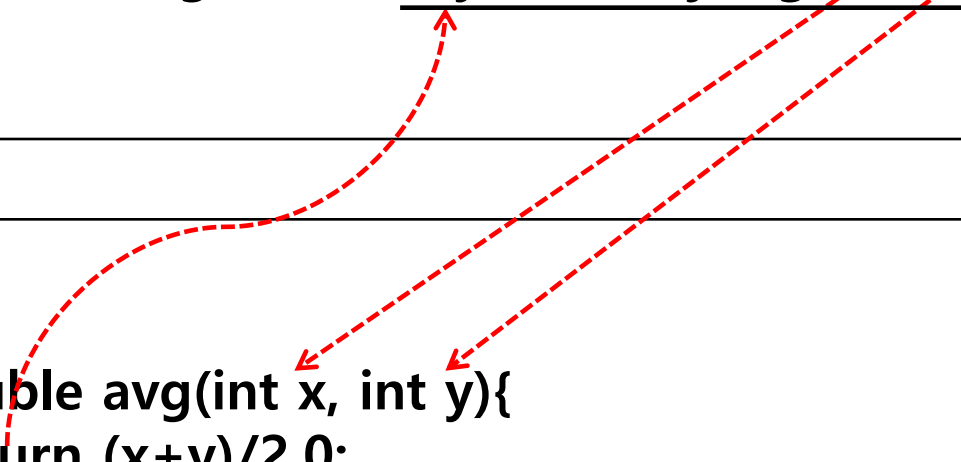
```
int plus(int x, int y){  
    int result = x + y;  
    return result;  
}
```

- return 문 뒤에 실행문 올 수 없음
- 리턴값이 없는 메소드
 - 메소드 실행을 강제 종료 시키는 역할

메소드

```
class MyMathTest {  
    public static void main(String[] arg) {  
        MyMath myMathObj = new MyMath();  
        double avgData = myMathObj.avg(10, 20);  
    }  
}
```

```
class MyMath {  
    ...  
    public double avg(int x, int y){  
        return (x+y)/2.0;  
    }  
}
```



The diagram illustrates the execution flow of the code. A red dashed arrow originates from the underlined method call `myMathObj.avg(10, 20);` in the `MyMathTest` class and points to the `avg` method definition in the `MyMath` class. Another red dashed arrow originates from the `avg` method definition and points back to the `avg` method call, indicating the return path.

메소드

```
class AClass{
    public int num;
    public static void main(String[] args) {
        AClass aObj = new AClass();
        aObj.num = 100;
        System.out.println(aObj.num);
        int b = 200;
        AClass cObj = aObj;
        cObj.num = 300;
        System.out.println(cObj.num);
    }
}
```

[메서드 영역]static main()

스택

힙

aObj=참조값

num = 100

[메서드 영역]static main()

스택

힙

b = 200

aObj=참조값

num = 100

[메서드 영역]static main()

스택

힙

cObj=참조값

b = 200

aObj=참조값

num = 300



인자 전달

자바의 인자 전달 방식

– 경우 1. 기본 타입의 값 전달

- 값이 복사되어 전달
- 메소드의 매개변수가 변경되어도 호출한 실인자 값은 변경되지 않음

– 경우 2. 객체 혹은 배열 전달

- 객체나 배열의 레퍼런스만 전달
 - 객체 혹은 배열이 통째로 복사되어 전달되는 것이 아님
- 메소드의 매개변수와 호출한 실인자 객체나 배열 공유

인자 전달 – 기본 타입의 값이 전달되는 경우

매개변수가 byte, int, double 등 기본 타입의 값일 때

- 호출자가 건네는 값이 매개변수에 복사되어 전달. 실인자 값은 변경되지 않음

```
public class ValuePassing {  
    public static void main(String args[]) {  
        int n = 10;  
  
        increase(n);  
  
        System.out.println(n);  
    }  
}
```

```
static void increase(int m) {  
    m = m + 1;  
}
```

호출

→ 실행 결과

10

main() 실행 시작

int n = 10;

n 10

increase(n);

n 10

n 10

값 복사

increase(int m) 실행 시작

10 m

11 m

m = m + 1;

increase(int m) 종료

System.out.println(n);

n 10

인자 전달 – 객체가 전달되는 경우

객체의 레퍼런스만 전달

- 매개 변수가 실인자 객체 공유

```
public class ReferencePassing {  
    public static void main (String args[]) {  
        Circle pizza = new Circle(10);  
  
        increase(pizza);  
  
        System.out.println(pizza.radius);  
    }  
}
```

호출

```
static void increase(Circle m) {  
    m.radius++;  
}
```

pizza.radius = 10;

→ 실행 결과

11

main() 실행 시작

pizza = new Circle(10); pizza

increase(pizza); pizza

increase(Circle m) 실행 시작

 pizza

m.radius++;

increase(Circle m) 종료

System.out.println(pizza.radius);

 pizza

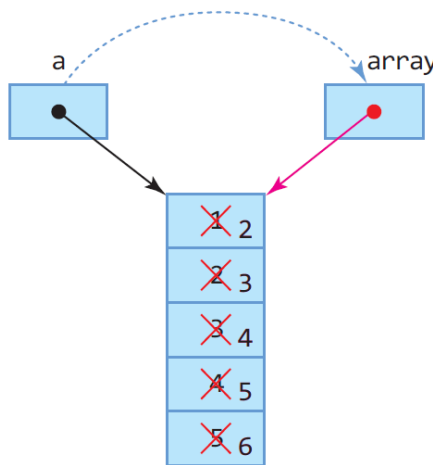
인자 전달 - 배열이 전달되는 경우

배열 레퍼런스만 매개 변수에 전달

- 배열 통째로 전달되지 않음
- 객체가 전달되는 경우와 동일
- 매개변수가 실인자의 배열을 공유

```
public class ArrayPassing {  
    public static void main(String args[]) {  
        int a[] = {1, 2, 3, 4, 5};  
  
        increase(a);  
  
        for(int i=0; i<a.length; i++)  
            System.out.print(a[i]+" ");  
    }  
}
```

레퍼런스 복사



```
static void increase(int[] array) {  
    for(int i=0; i<array.length; i++) {  
        array[i]++;  
    }  
}
```

⇒ 실행 결과

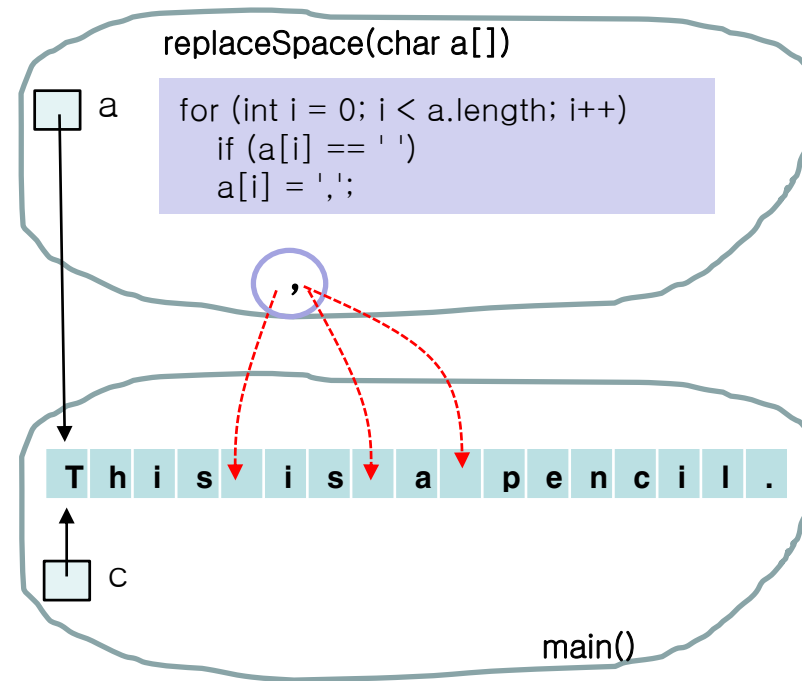
2 3 4 5 6

예제 4-8 : 인자로 배열이 전달되는 예

char[] 배열을 전달받아 출력하는 printCharArray() 메소드와 배열 속의 공백(' ') 문자를 ','로 대체하는 replaceSpace() 메소드를 작성하라.

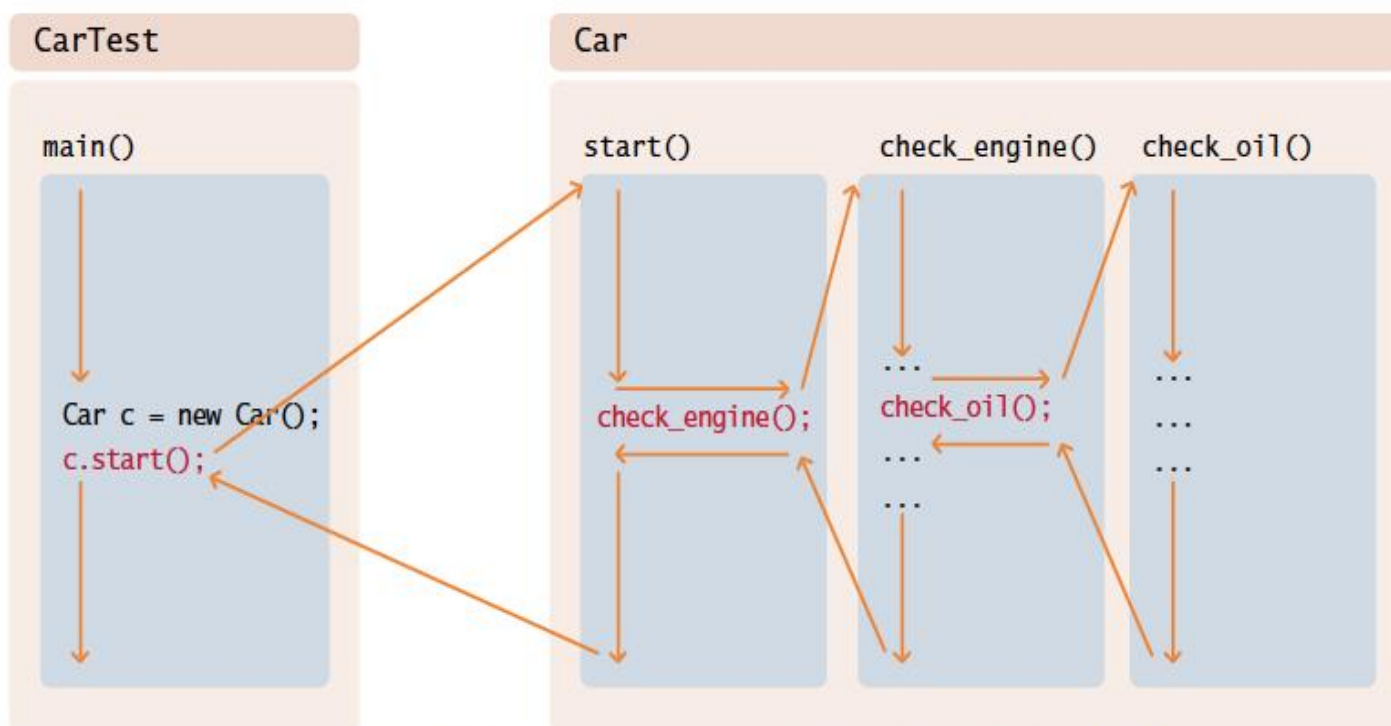
```
public class ArrayParameterEx {
    static void replaceSpace(char a[]) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == ' ')
                a[i] = ',';
    }
    static void printCharArray(char a[]) {
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i]);
        System.out.println();
    }
    public static void main (String args[]) {
        char c[] = {'T','h','i','s',' ','i','s',' ',' ','a',' ','p','e','n','c','i','l','.'};
        printCharArray(c);
        replaceSpace(c);
        printCharArray(c);
    }
}
```

This is a pencil.
This,is,a,pencil.



메소드 호출

메소드 안에 들어 있는 문장을 실행하기 위해서는 메소드를 호출하여야 한다.



메소드가 다른
클래스에 있으면
객체를 통하여 호출

메소드가 같은 클래스에 있으면
메소드 이름을 통하여 호출



중복 메소드

자바에서는 같은 이름의 메소드가 여러 개 존재할 수 있다. 이것을 중복 메소드(overloading, 메소드 오버로딩)라고 한다. 중복 메소드는 여러 가지 데이터 타입에 같은 처리를 수행하는 경우에 많이 사용된다.

메소드(method)

❖ 메소드 오버로딩(Overloading)

- 클래스 내에 같은 이름의 메소드를 여러 개 선언하는 것
- 하나의 메소드 이름으로 다양한 매개값 받기 위해 메소드 오버로딩
- 오버로딩의 조건: 매개변수의 타입, 개수, 순서가 달라야

```
class 클래스{  
    리턴타입 메서드명(타입 변수, ...){ ... }
```

무관

동일

매개변수의 타입, 개수, 순서가 달라야함

```
    리턴타입 메서드명(타입 변수, ...){ ... }
```

```
}
```

```
void println(){ ... }  
void println(boolean x){ ... }  
void println(char x){...}  
void println(double x){...}  
void println(float x){ ... }  
void println(int x){ ... }  
void println(long x){ ... }  
void println(String x){ ... }  
void println(Object x){ ... }
```

오버로딩된 메소드 호출

// 메소드 오버로딩이 성공한 사례

```
class MethodOverloading {  
    public int getSum(int i, int j) {  
        return i + j;  
    }  
    public int getSum(int i, int j, int k) {  
        return i + j + k;  
    }  
}
```

// 메소드 오버로딩이 실패한 사례

```
class MethodOverloadingFail {  
    public int getSum(int i, int j) {  
        return i + j;  
    }  
    public double getSum(int i, int j) {  
        return (double)(i + j);  
    }  
}
```

```
public static void main(String args[]) {  
    MethodSample a = new MethodSample();  
  
    int i = a.getSum(1, 2);  
    int j = a.getSum(1, 2, 3);  
    double k = a.getSum(1.1, 2.2);  
}
```

매개 변수의 개수와 타입이
서로 다른 3 함수 호출

```
public class MethodSample {  
    public int getSum(int i, int j) {  
        return i + j;  
    }  
    public int getSum(int i, int j, int k) {  
        return i + j + k;  
    }  
    public double getSum(double i, double j) {  
        return i + j;  
    }  
}
```

메소드(method)

```
int plus(int x, int y) {  
    int result = x + y;  
    return result;  
}
```

← plus(3, 7);

```
double plus(double x, double y) {  
    double result = x + y;  
    return result;  
}
```

← plus(10.3, 34.9);

```
int divide(int x, int y) { ... }  
int divide(int a, int b) { ... }
```

X



중복 메소드

제곱값을 구하는 메소드 square()를 포함한 클래스가 존재.

```
// 정수값을 제공하는 메소드
```

```
public int square(int i) {  
    return i*i;  
}
```

```
//실수값을 제공하는 메소드
```

```
public double square(double i) {  
    return i*i;  
}
```




중복 메소드

- 메소드 호출시 매개 변수를 보고 일치하는 메소드가 호출된다. 만약 `square(3.14)`와 같이 호출되면 컴파일러는 매개 변수의 개수, 타입, 순서 등을 봐서 두 번째 메소드를 호출한다.
- 메소드의 이름, 매개변수의 개수, 타입, 순서를 시그니처(signature)라고 한다. 중복 메소드는 이름은 같지만 시그니처는 반드시 달라야 한다.



중간점검

1. 같은 이름의 메소드를 중복하여 정의하는 것을 _____라고 한다.
2. void가 메소드 앞에 붙으면 어떤 의미인가?
3. 여러분의 이름을 출력하는 메소드 printMyName()를 작성하여 보라.



클래스 다이어그램

객체 지향 프로그래밍은 객체 단위로 작업하므로 먼저 해야 할 일은 객체를 구조화하는 객체 모델링이다. 객체 모델링이 완료되면 이것을 프로그래밍 언어로 표현할 수 있도록 '클래스 다이어그램'을 작성한다. 클래스 다이어그램은 객체 모델링에서 표현된 구조를 프로그래밍 언어로 표현하기 쉽게 작성한 것이다.

객체 모델링	클래스 다이어그램	설명
객체	클래스	객체를 정의하기 위한 틀로써 클래스명 정의
속성	필드	객체가 가진 고유한 특성의 정의
동작	메서드	객체가 할 수 있는 동작을 정의



클래스 다이어그램

객체 지향 프로그래밍에서도 프로그래머들은 애플리케이션을 구성하는 클래스들 간의 관계를 그리기 위하여 클래스 다이어그램(class diagram)을 사용한다. 가장 대표적인 클래스 다이어그램 표기법은 UML(Unified Modeling Language)이다.

※ 용어 정리

- **Class Diagram**는 시스템에서 사용되는 객체 타입(클래스)을 정의하고 그들 간에 존재하는 정적인 관계를 표현한 다이어그램입니다.
- **Use Case Diagram**는 사용자 관점에서 SW 시스템의 범위와 기능 정의하고, 시스템이 해야 할 무엇을 작성하는 모델입니다.

UML

UML: Unified Modeling Language의 약자이며 요구분석, 시스템설계, 시스템 구현 등의 시스템 개발 과정에서, 개발자간의 의사소통을 원활하게 이루어지게 하기 위하여 표준화한 모델링 언어입니다.

클래스의 이름을 적어준다.

Car

- speed: int
- gear: int
- color: String

+ speedUP(): void
+ speedDown(): void

클래스의 속성을 나타낸다.
즉 필드를 적어준다.

클래스의 동작을 나타낸다.
즉 메소드를 적어준다.



캡슐화 설계

(1) UML에서 캡슐화 표기

데이터 접근자(private)는 '-'로 정의하며 외부에서 접근하지 못하도록 설계한다. 메서드 접근자(public)는 '+'로 정의하여 외부에서 접근할 수 있도록 설계한다. 데이터뿐만 아니라 함수도 외부에 숨길 필요가 있는 경우 '-'로 정의하여 설계 한다.

UML	자바	C++
+	public	public
#	protected	protected
-	private	private
~	표기법 없음(No Keyword)	friend



캡슐화 설계

(2) 캡슐화 구현(자바에서 캡슐화 코드)

자바의 캡슐화 구현은 클래스 구현을 기반으로 추상화, 재사용, 그리고 정보은닉을 구현한다.

정보은닉
구현

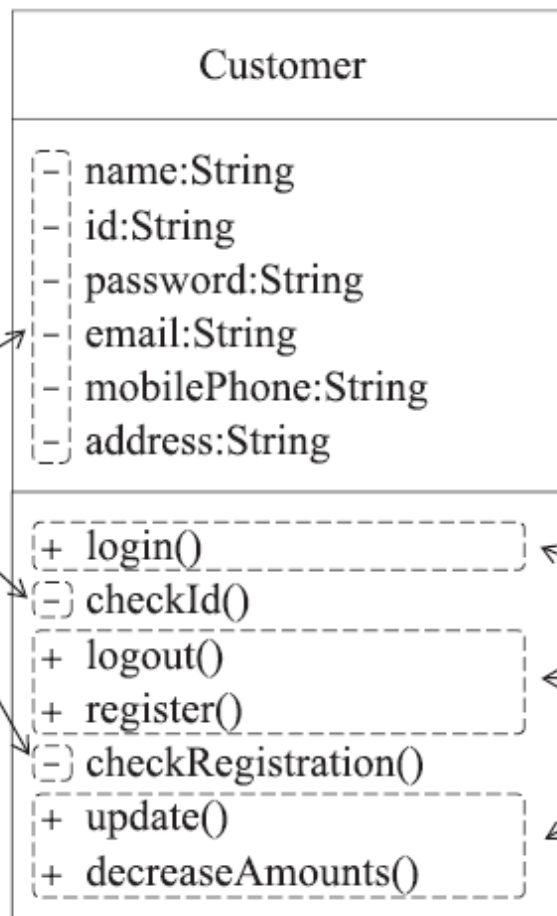
```
public class Product {  
    private String productId;  
    private int price;  
  
    public void addProduct(String productId) { ... }  
    public Boolean deleteProduct(String productId) { ... }  
    private Boolean checkProduct(String productId) { ... }  
}
```

Product 클래스와
결합도가 높은
데이터와 함수를
묶어 캡슐화 구현

(나) 자바 클래스의 캡슐화 코드

캡슐화 설계

데이터와 함수를
Customer 클래스 내부에
은닉하여 블랙박스화 설계



외부 통신을 위한
공개인터페이스 설계



캡슐화 설계

캡슐화를 고려하여 클래스 다이어그램을 설계한다.

① 도서대여시스템의 요구사항

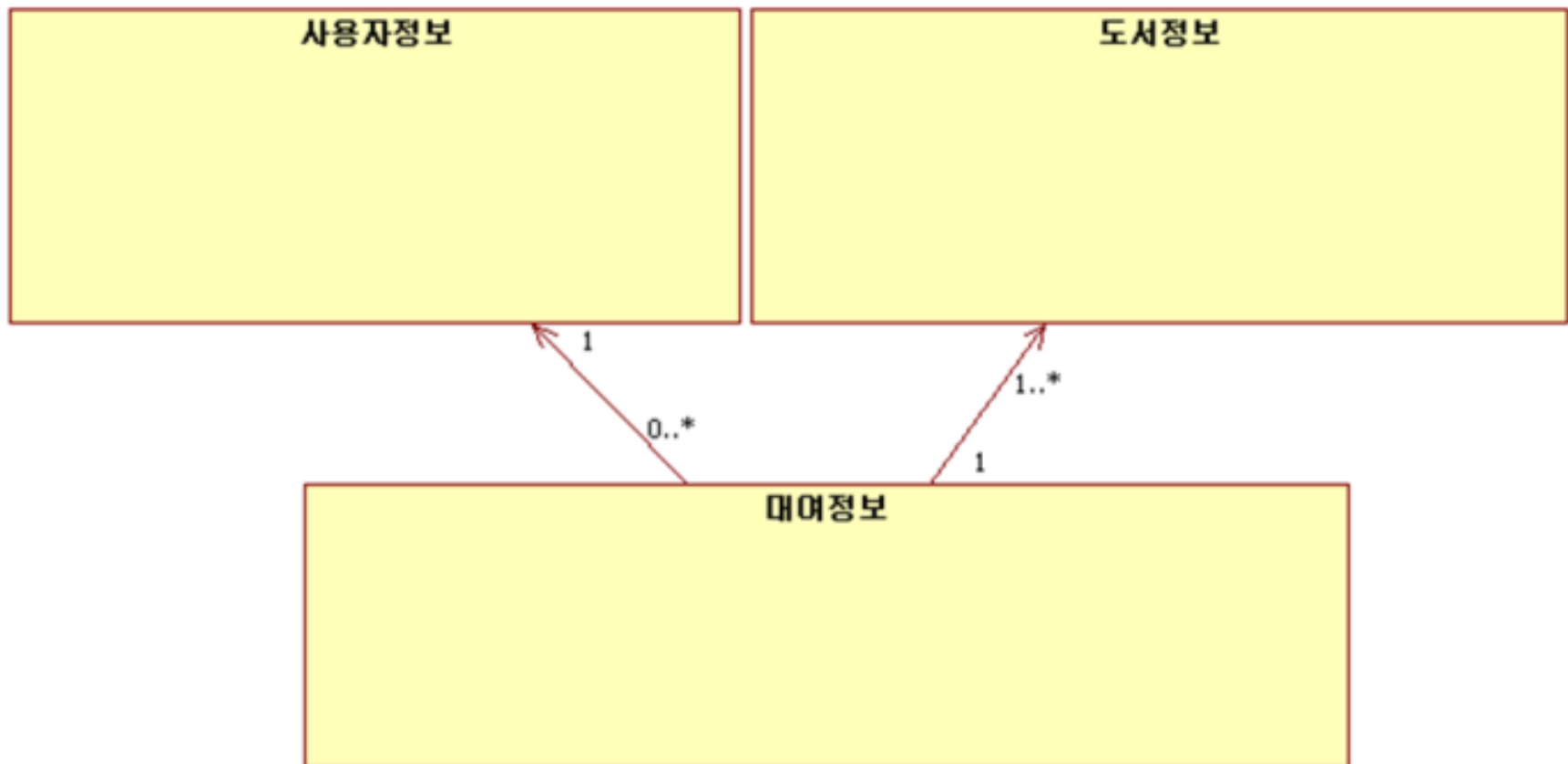
본 도서대여 시스템은 사용자정보와 도서정보를 기반으로 대여정보에 의해 대여 할 수 있다. 도서를 반납 시 대여기간이 초과한 경우 연체처리를 한다. 본 시스템에서는 사용자 정보를 사용자등록, 사용자수정, 사용자삭제, 사용자조회 할 수 있다. 도서정보는 도서등록, 도서수정, 도서삭제, 도서조회 할 수 있다. 대여 시 반드시 사용자 조회와 도서 조회를 해야 한다.

- 속성 : 사용자ID, 사용자암호, 성명, 도서ID, 도서명, 출판사, 대여ID, 대여일, 반납일
- 메소드 : 대여, 반납, 연체, 사용자등록, 사용자수정, 사용자 삭제, 사용자 조회, 도서등록, 도서수정, 도서삭제, 도서조회

캡슐화 설계

캡슐화를 고려하여 클래스다이어그램을 설계한다.

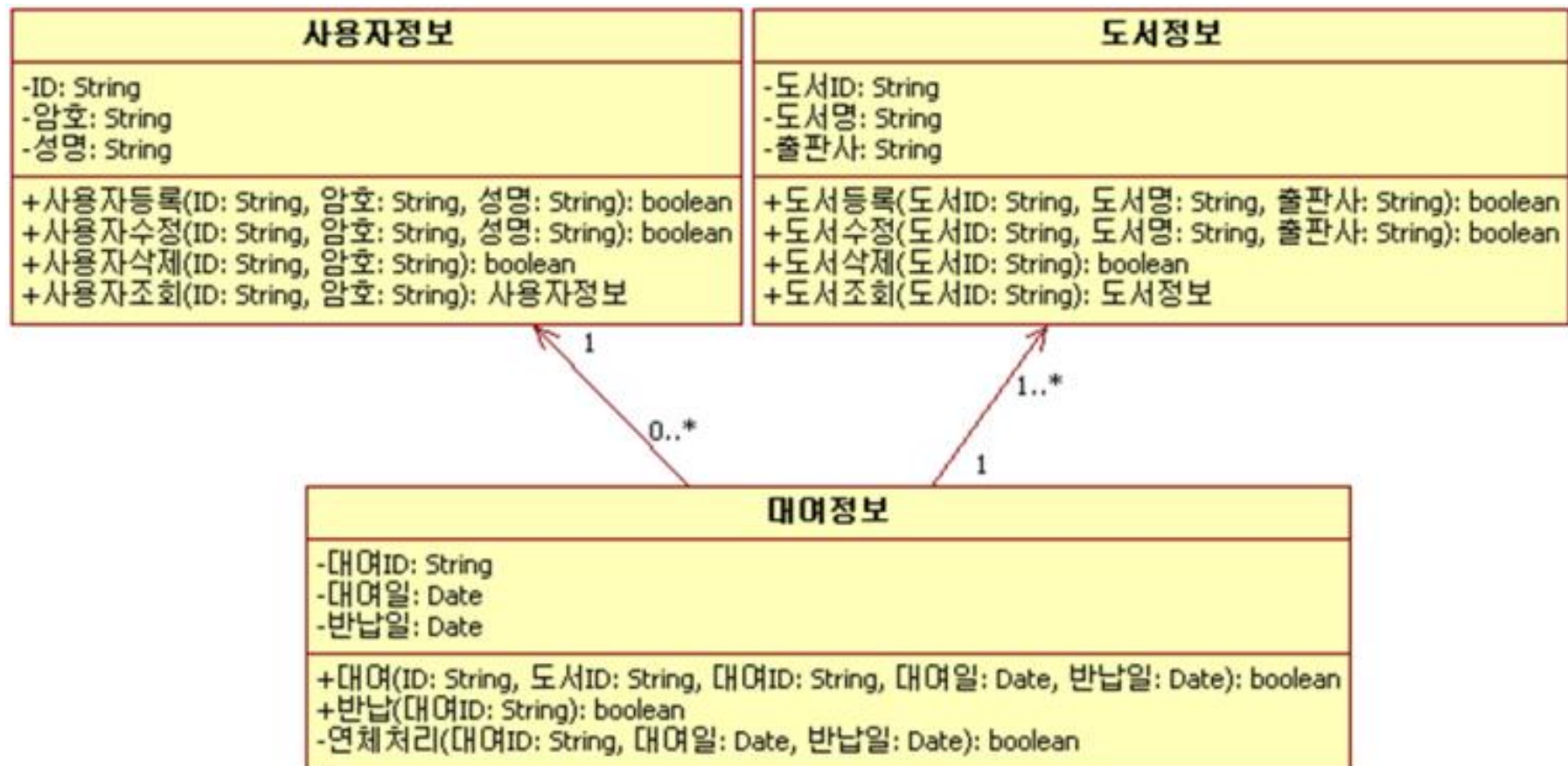
① 도서대여시스템의 요구사항



캡슐화 설계

캡슐화를 고려하여 클래스다이어그램을 설계한다.

① 도서대여시스템의 요구사항





캡슐화 설계

캡슐화를 고려하여 클래스다이어그램을 설계한다.

② 학사관리시스템의 요구사항

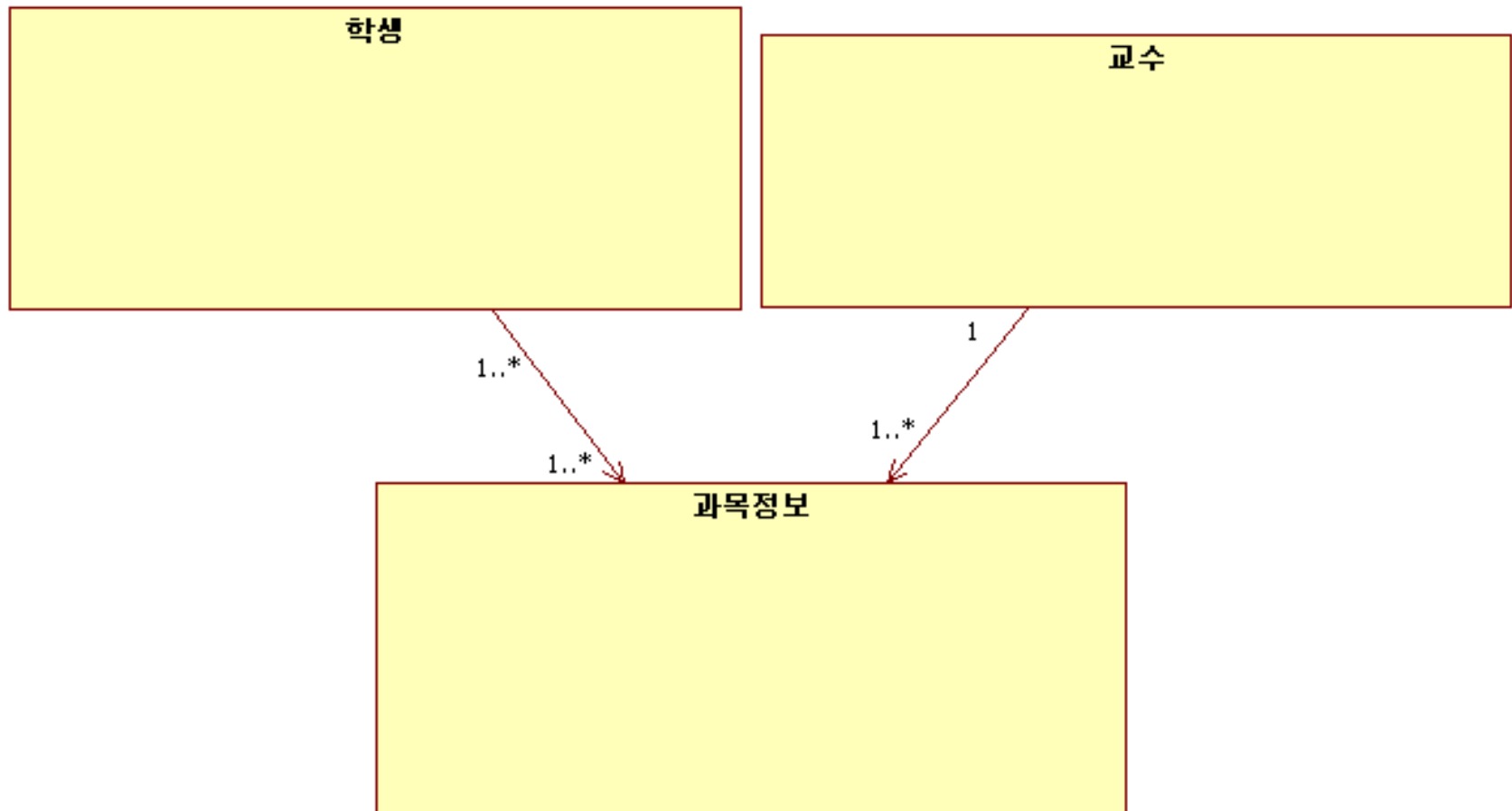
본 학사관리 시스템을 통해 학생은 수강신청, 성적조회를 할 수 있으며, 교수는 수강과목등록, 성적입력을 할 수 있다. 학생과 교수에게 제공되는 기능은 과목 정보를 통해 가능하다.

- 속성 : 학번, 암호, 성명, 학생학과, 학년, 교수ID, 교수암호, 교수성명, 교수학과, 과목ID, 과목점수, 학점
- 메소드 : 수강신청, 성적조회, 수강과목등록, 성적입력

캡슐화 설계

캡슐화를 고려하여 클래스다이어그램을 설계한다.

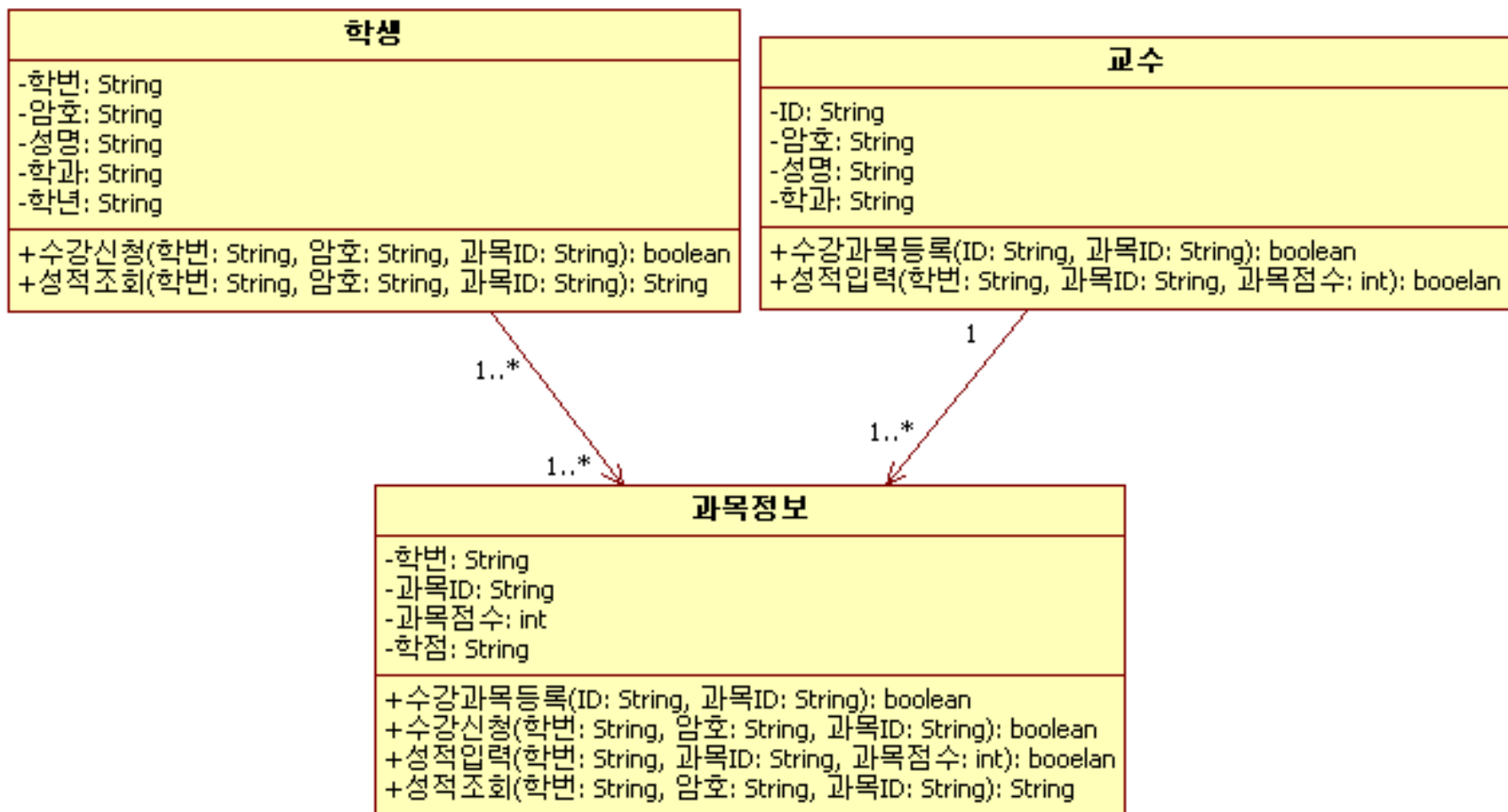
② 학사관리시스템의 요구사항



캡슐화 설계

캡슐화를 고려하여 클래스다이어그램을 설계한다.

② 학사관리시스템의 요구사항





Thank You

