



4장 클래스와 객체 1

박숙영

blue@sookmyung.ac.kr

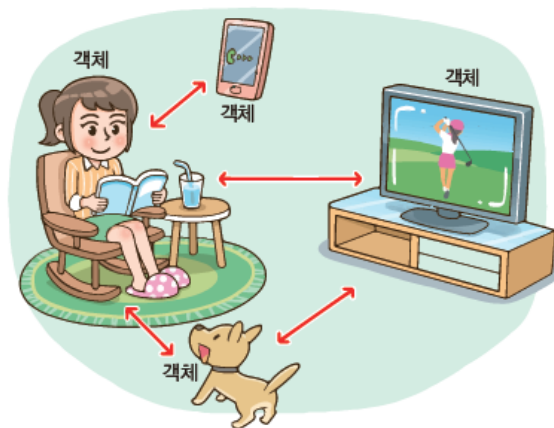
4장의 목표

1. 객체 지향과 절차 지향을 비교해서 설명할 수 있나요?
2. 특정한 객체를 찍어내는 클래스를 정의할 수 있나요?
3. 메소드 오버로딩을 사용하여 이름이 동일한 여러 개의 메소드를 정의할 수 있나요?
4. 생성자를 작성하여 객체를 초기화할 수 있나요?
5. 접근자와 생성자를 만들어서 접근을 제어할 수 있나요?



객체지향 프로그래밍

- 객체(object)를 사용하는 프로그래밍 방식을 객체 지향 프로그래밍(OOP: Object-Oriented Programming)이라고 한다.
- OOP는 실세계와 비슷하게 소프트웨어도 작성해보자는 방법론이다.

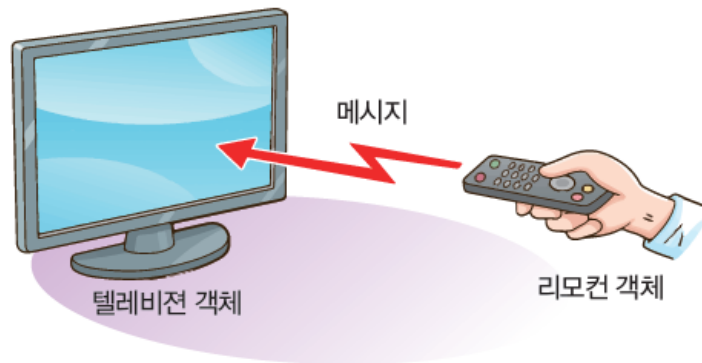


- 객체들은 객체 나름대로의 고유한 기능을 수행하면서 다른 객체들과 상호 작용한다.
- 객체들은 메시지(message)를 통하여 상호 작용하고 있다.

↙ 현실세계에 존재하는, 여사로 표현할 수 있는 모든 것

↙ 호출

ex) scanner.nextInt()



객체

- 객체는 **상태**와 **동작**을 가지고 있다. (또는 속성과 기능)
- 객체의 **상태(state)**는 객체의 속성이다.
- 객체의 **동작(behavior)**은 객체가 취할 수 있는 동작이다.



그림 4.1 텔레비전 객체의 예

절차지향 프로그래밍

- 절차 지향 프로그래밍은 **프로시저(procedure)**에 기반을 두고 있다.
- “절차”라는 용어는 “procedure”를 번역한 것이며, 프로시저는 **함수 또는 서브루틴**을 뜻한다.
- 절차 지향 프로그래밍에서 **프로그램은 함수(프로시저)들의 집합**으로 이루어진다.

```
int main(void) {  
    double radius = 10.0;  
    double area;  
    area = 3.14*radius*radius;  
    printf("면적=%f\n", area);  
  
    return 0;  
}
```



```
double getCircleArea(double r) {  
    double area;  
    area = 3.14*r*r;  
    return area;  
}
```

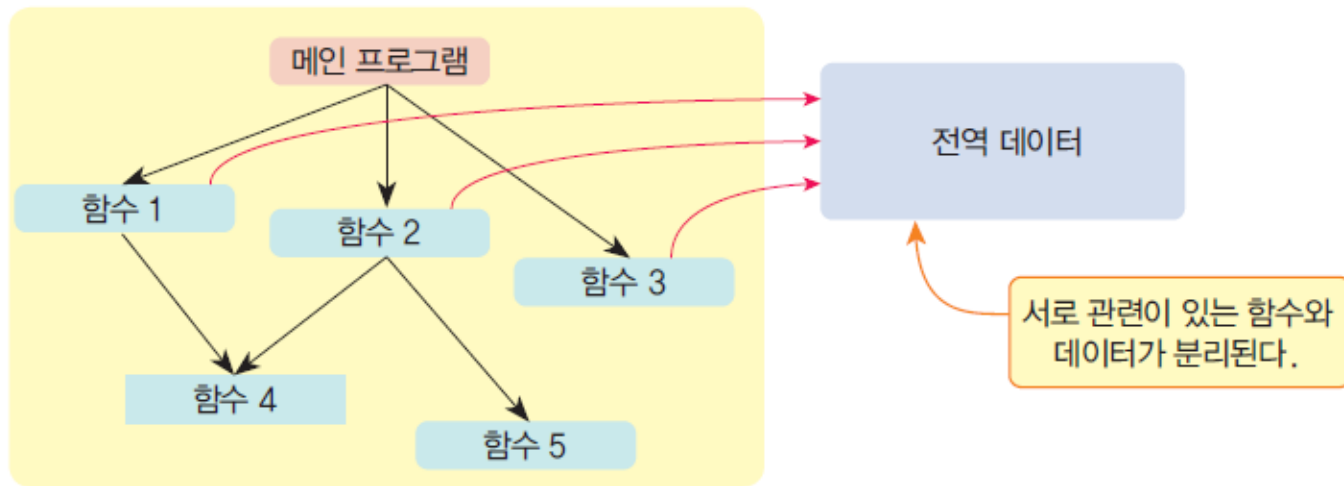
함수를 사용하여
소스를 작성한다.

```
int main(void) {  
    int radius = 10.0;  
    double area = getCircleArea(radius);  
    printf("면적=%f\n", area);  
    return 0;  
}
```

→ 데이터와 함수가 따로

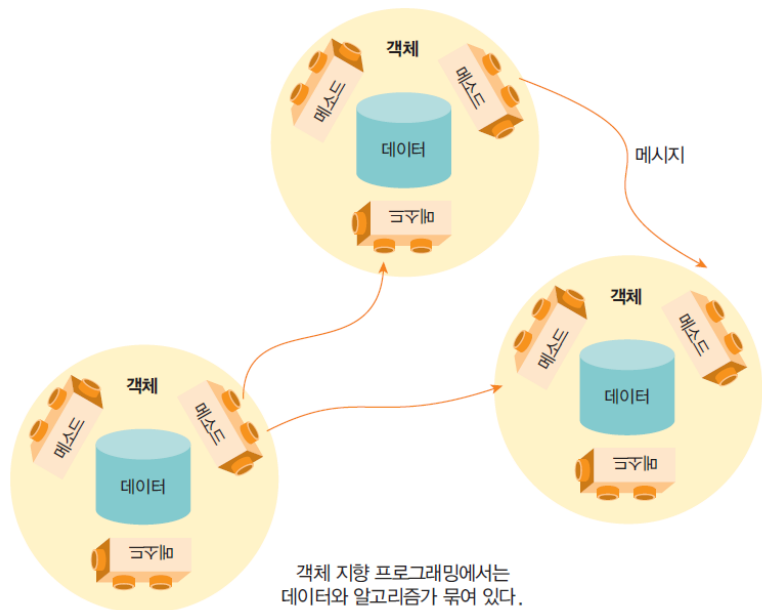
절차지향의 문제점

- 서로 관련 있는 함수와 데이터가 분리된다.
- (예) 앞장의 코드에서 원의 반지름을 나타내는 radius와 이 데이터를 사용하는 함수 `getCircleArea()`는 서로 분리되어 있다.



객체 지향 프로그래밍

- 객체 지향 프로그래밍(object-oriented programming)은 데이터와 함수를 하나의 덩어리로 묶어서 생각하는 방법이다.
- 데이터와 함수를 하나의 덩어리(객체)로 묶는 것을 캡슐화(encapsulation)라고 부른다.



객체 지향 프로그래밍의 간단한 예

- 앞의 코드에서 원의 반지름과 면적을 계산하는 함수를 하나로 묶으면 다음과 같다.

```
public class Circle {  
    double radius;           // 데이터  
    String color;            // 데이터  
    double getArea() { return 3.14*radius*radius; } // 함수  
}
```


객체지향의 특징

중요

- 객체지향 언어는 캡슐화, 상속 및 다형성으로 정의된다

① 캡슐화 → 정보 은닉, 관련 데이터 + 알고리즘 묶음

② 상속 → 재사용성 ↑

③ 다형성 → 동일한 이름의 동작이 객체 타입에 따라 다르게 동작

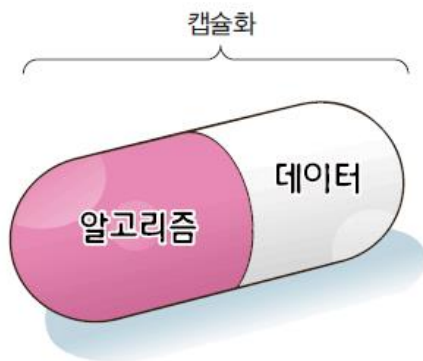
④ 추상화 → 중요 정보 남기고 덜 중요한 것만 빼먹음



OOP의 4개의 기둥

캡슐화(encapsulation)

- 관련된 데이터와 알고리즘을 하나의 묶음으로 정리
- 캡슐화의 목적
 - 서로 관련된 데이터와 알고리즘을 하나로 묶는 것
 - 객체를 캡슐로 싸서 객체의 내부를 보호하는 것
 - 즉 객체의 실제 구현 내용을 외부에 감추는 것



정보은닉

- 정보 은닉(information hiding)
 - 객체의 외부에서는 객체의 내부 데이터를 볼 수 없게 한다는 의미
- 객체 안의 데이터와 알고리즘은 외부에서 변경하지 못하게 막고,
- 공개된 인터페이스를 통해서만 객체에 접근하도록 하는 개념

사용자가 책을 마음대로
꺼내고 꽂을 수 있기 때문에
서가 엉망이 될 수 있다.



(a) 절차 지향 방법

사서를 통하여
책을 꺼내면
서가는 안전하다.



(b) 객체 지향 방법

그림 4.4 어떤 방법이 내부 데이터를 안전하게 보관할까?

소프트웨어 캡슐

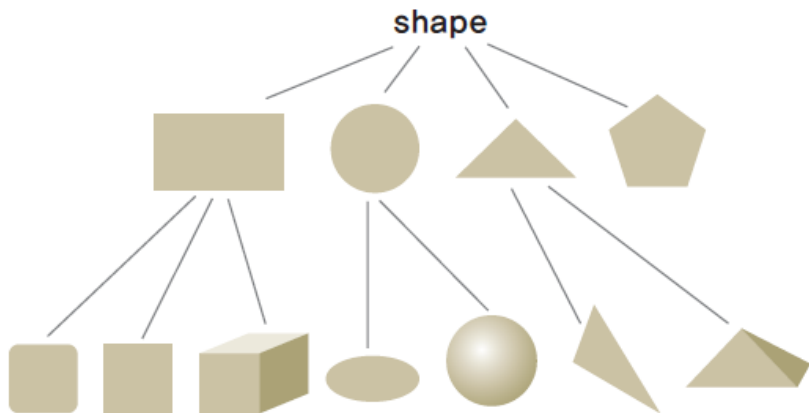
- 소프트웨어 캡슐 내부가 어떻게 구성되어 있으며 어떻게 돌아가는지 전혀 신경 쓸 필요가 없다. 그냥 사용법만 익혀서 사용하면 된다



상속

- 자동차 회사에서 새로운 모델을 개발할 때, 기존의 모델을 수정하는 방법도 많이 선택한다(페이스리프트).
- 우리는 이미 작성된 클래스(부모 클래스)를 이어받아서 새로운 클래스(자식 클래스)를 생성할 수 있다. **자식 클래스는 부모 클래스의 속성과 동작을 물려받는다.**

재사용성↑



상속은 기존에 만들어진 코드를 이어받아서 보다 쉽게 코드를 작성하는 기법입니다.

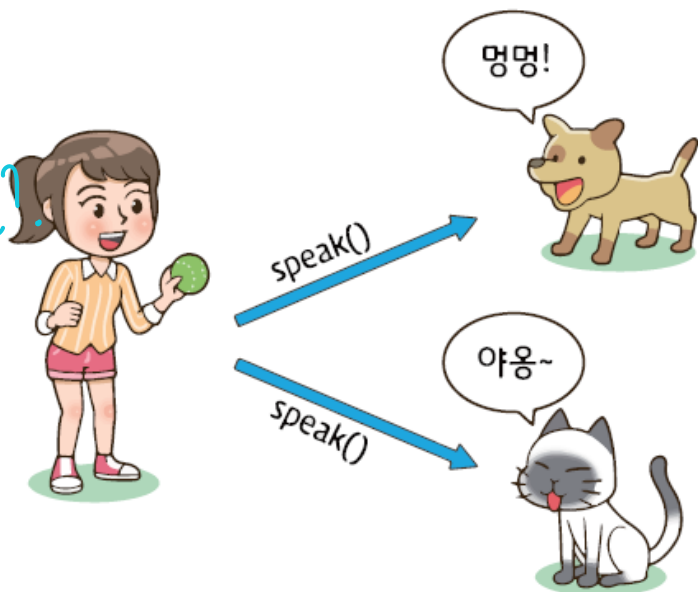


다형성

- 동일한 이름의 동작이라고 하여도 객체의 실제 타입에 따라서 동작의 내용이 달라질 수 있다는 것을 의미

다형성

어떻게 구현?



다형성은 객체의 동작이 실제 타입에 따라서 달라지는 것을 말합니다. "speak"라는 메시지를 받은 객체들이 모두 다르게 소리를 내는 것이 바로 다형성입니다.



추상화(abstraction)

- 불필요한 정보는 숨기고 중요한 정보만을 표현함으로써 프로그램을 간단히 만드는 기법

예) + : 더하기 연산의 추상화



실제 객체



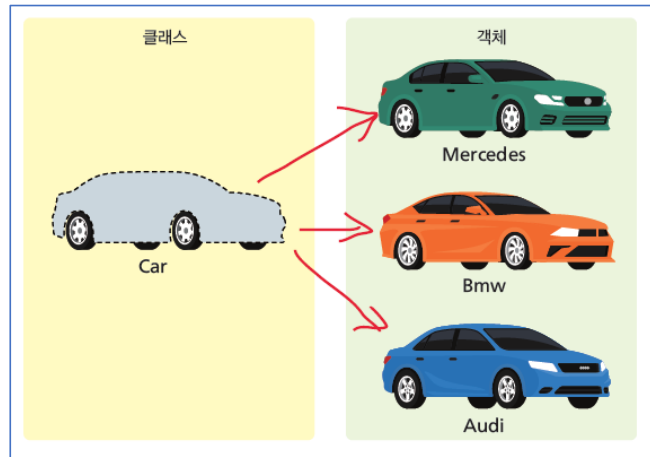
추상화된 객체

추상화는 필요한 것만을 남겨 놓는
것입니다. 추상화 과정이 없다면
사소한 것도 신경 써야 합니다.



클래스란?

- 객체에 대한 설계도를 클래스(class)라고 한다. 클래스란 특정한 종류의 객체들을 찍어내는 형틀(template) 또는 청사진(blueprint)이라고도 할 수 있다. 클래스로부터 만들어지는 각각의 객체를 그 클래스의 인스턴스(instance)라고 한다.



- 하나의 클래스로 여러 개의 인스턴스를 찍어내지만, 인스턴스마다 속성의 값은 다르다.

클래스의 작성

Syntax: 클래스 정의

```
class Account {  
    String type;  
    String name;  
    String acct-num;  
    int balance;  
  
    void withdraw (int m) {  
    }  
    void deposit (int m) {  
    }  
    // 총금  
    // 잔액조회  
}
```

클래스는 class라는 키워드로 시작한다.
이어서 클래스 이름을 적고, 중괄호 안에
필드와 메소드를 나열하면 된다.

public class 클래스이름 {

자료형 필드1;
자료형 필드2;
...

반환형 메소드1() { ... }
반환형 메소드2() { ... }
...

필드 → 상태

메소드 → 동작

예) 계좌(Account) → 객체

- 상태 (속성): 타입, 소유주(이름), 계좌번호, 잔액
- 동작 (기능): 출금, 입금, 통금, 조회

public class Circle {

public int radius;
public String color;

public double getArea() {
 return 3.14*radius*radius;
}

}

객체생성

- 클래스는 객체를 만들기 위한 설계도(틀)에 해당된다. 설계도를 가지고 어떤 작업을 할 수는 없다. 실제로 어떤 작업을 하려면 객체를 생성하여야 한다

```
public class CircleTest {  
    public static void main(String[] args) {  
  
        Circle obj;  
  
        obj = new Circle();  
        obj.radius = 100;  
        obj.color = "blue";  
  
        double area = obj.getArea();  
        System.out.println("원의 면적=" + area);  
    }  
}
```

원의 면적=31400.0

객체 생성 단계

- 참조 변수를 선언

```
Circle obj;
```

obj



- new 연산자를 사용하여 객체를 생성하고 객체의 참조값을 참조 변수에 저장

생성
`obj = new Circle();`

obj



→ 공간 할당됨! (실체화됐으므로)

radius	
color	
getArea() { ... }	

- 객체의 필드와 메소드를 사용

```
obj.radius = 100;  
obj.color = "blue";
```

obj

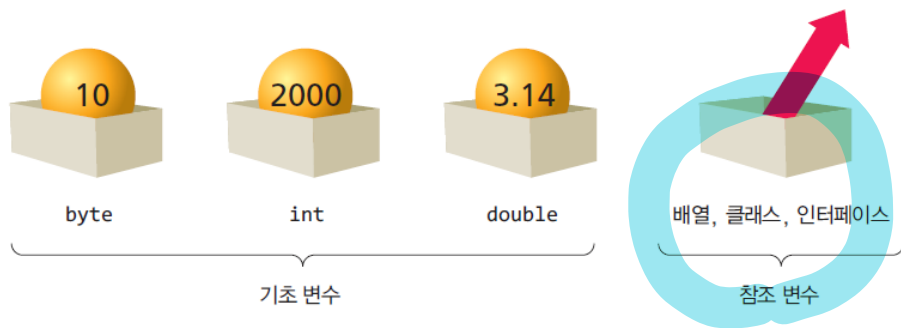


```
double area = obj.getArea();
```

radius	100
color	"blue"
getArea() { ... }	

참조 변수

- 참조 변수는 객체를 참조할 때 사용되는 변수로서 객체의 참조값이 저장
- 참조값은 일반적으로 객체의 주소



프로그래밍시 혼동을 많이 일으키는 부분은, 다음과 같이 선언하면 객체가 생성된다고 생각하는 것이다.

```
Circle obj;
```

위의 문장은 단순히 객체를 참조하는 변수 obj를 선언해놓은 것이다. 객체의 이름만 정해놓은 것이라고 생각해도 좋다. 아직 객체는 생성되지 않았다. 참조 변수만 생성된 것이다. 자바에서 모든 객체는 new 연산자를 이용해야만 비로소 생성된다. 하지만 C++ 언어에서는 위의 문장으로 실제 객체가 생성된다. 자바랑 혼동하지 말자.

경고



예제: 데스크 램프 클래스

- 집에서 사용하는 데스크 램프를 클래스로 작성하여 보면 다음과 같다.

DeskLamp
-isOn : bool
+turnOn() +turnOff()



DeskLamp.java

```
public class DeskLamp {  
    // 인스턴스 변수 정의  
    private boolean isOn; // 켜짐이나 꺼짐과 같은 램프의 상태  
  
    // 메소드 정의  
    public void turnOn() { isOn = true; }  
    public void turnOff() { isOn = false; }  
    public String toString() { // 객체를 출력하면 이 메소드가 호출된다.  
        return "현재 상태는 " + (isOn == true ? "켜짐" : "꺼짐");  
    }  
}
```

클래스 안에서만
보아야 함
→

예제: 데스크 램프 클래스

DeskLampTest.java

```
public class DeskLampTest {  
    public static void main(String[] args) {  
        // 역시 객체를 생성하려면 new 예약어를 사용한다.  
        DeskLamp myLamp = new DeskLamp();  
  
        // 객체의 메소드를 호출하려면 도트 연산자인 .을 사용한다.  
        myLamp.turnOn();  
        System.out.println(myLamp);  
        myLamp.turnOff();  
        System.out.println(myLamp);  
    }  
}
```

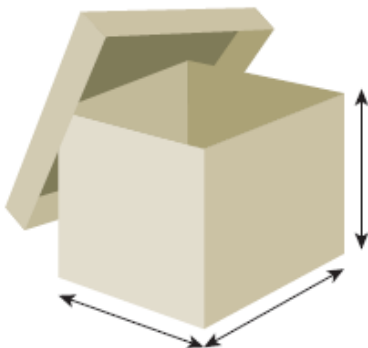
*myLamp.toString 메소드 호출과 같은 결과
→ 생각해도 사용가능하도록!*

현재 상태는 켜짐
현재 상태는 꺼짐

예제:

상자를 나타내는 Box 클래스를 작성하여 보자. Box 클래스는 가로 길이, 세로 길이, 높이를 나타내는 필드를 가지고, 상자의 부피를 계산하는 메소드를 가진다.

Box 클래스를 정의하고 Box 객체를 하나 생성한다. Box 객체 안의 가로 길이, 세로 길이, 높이를 20, 20, 30으로 설정하여 보자. 상자의 부피를 출력하여 본다.



상자의 가로, 세로, 높이는 20, 20, 30입니다.

상자의 부피는 12000.0입니다.

예제:

```
class Box {
    int width;
    int length;
    int height;
    double getVolume() {
        return (double) width*height*length;
    }
}

public class BoxTest {
    public static void main(String[] args) {
        Box b;
        b = new Box();
        b.width = 20;
        b.length = 20;
        b.height = 30;
        System.out.println("상자의 가로, 세로, 높이는 " + b.width + ", " + b.length + ", " + b.height + "입니다.");
        System.out.println("상자의 부피는 " + b.getVolume() + "입니다.");
    }
}
```


예제: Television 클래스 작성하고 객체 생성해보기

```
public class Television {  
    int channel;    // 채널 번호  
    int volume;    // 볼륨  
    boolean onOff; // 전원 상태
```

```
    public static void main(String[] args) {
```

```
        중요! * Television myTv = new Television();
```

```
        myTv.channel = 7;
```

```
        myTv.volume = 10;
```

```
        myTv.onOff = true;
```

```
        Television yourTv = new Television();
```

```
        yourTv.channel = 9;
```

```
        yourTv.volume = 12;
```

```
        yourTv.onOff = true;
```

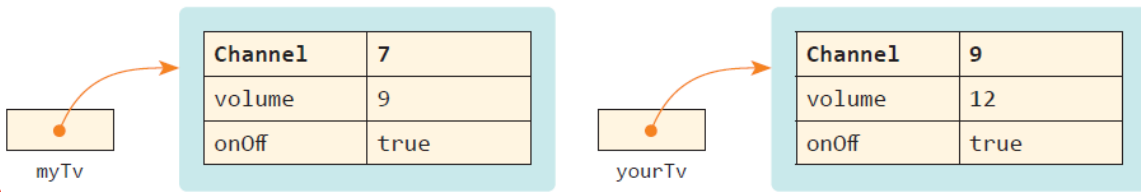
```
        System.out.println("나의 텔레비전의 채널은 " + myTv.channel + "이고 볼륨은 " + myTv.volume + "입니다.");
```

```
        System.out.println("너의 텔레비전의 채널은 " + yourTv.channel + "이고 볼륨은 " + yourTv.volume + "입니다.");
```

```
    }
```

```
}
```

아래 그림에서 보듯이 myTv의 데이터는 yourTv의 데이터와 완전히 분리되어 있다.



특성을 갖는 클래스에게
main method도 주어져서
driver class로도 사용하게 함

바깥에 선언된 것은
예를 들어 자바에서는
new를 사용하지 않음!
중요! new로 공간을
할당해야 함

나의 텔레비전의 채널은 7이고 볼륨은 10입니다.
너의 텔레비전의 채널은 9이고 볼륨은 12입니다.

메소드 오버로딩(Method Overloadnig) ⇒ 중복정의

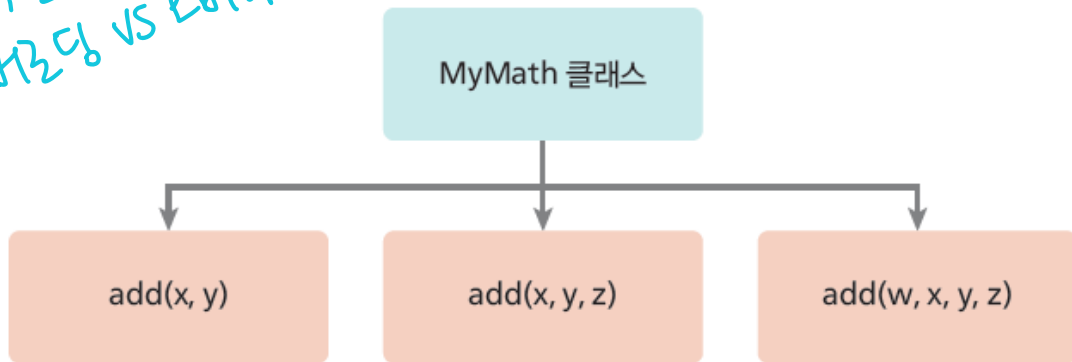
이름이 동일한 여러 개의 메소드 작성 가능

- 단, 메소드 이름은 동일하면서 매개변수의 수, 타입, 순서 가 달라야 한다. → parameter라 구분

기능이 동일하다!



기본
오버로딩 VS 오버라이딩



메소드 오버로딩이란 이름이 같은 메소드를 여러 개 정의하는 것입니다. 다만 각각의 메소드가 가지고 있는 매개 변수는 달라야 합니다.



예제

```
public class MyMath {  
  
    int add(int x, int y)          { return x+y;      }  
    int add(int x, int y, int z)   { return x+y+z;    }  
    int add(int x, int y, int z, int w) { return x+y+z+w; }  
  
    public static void main(String[] args) {  
        MyMath obj;  
        obj = new MyMath(); → 매개변수가 없는 생성자 메소드  
        System.out.print(obj.add(10, 20)+" ");  
        System.out.print(obj.add(10, 20, 30)+" ");  
        System.out.print(obj.add(10, 20, 30, 40)+" ");  
    }  
}
```

30 60 100

생성자(constructor)

- 객체가 생성될 때 객체를 초기화하는 특수한 메소드



- * 클래스 이름과 동일하면서 반환값이 없는 메소드

- 중복 정의 가능

생성자

```
class Pizza {
```

```
    int size;
```

```
    String type;
```

공통의

```
    public Pizza() {
```

```
        size = 12;
```

```
        type = "슈퍼슈프림";
```

```
    }
```

```
    public Pizza(int s, String t) {
```

```
        size = s;
```

```
        type = t;
```

```
    }
```

```
}
```

```
public class PizzaTest {
```

```
    public static void main(String[] args) {
```

```
        Pizza obj1 = new Pizza();
```

```
        System.out.println("(" + obj1.type + " , " + obj1.size + ",)");
```

```
        Pizza obj2 = new Pizza(24, "포테이토");
```

```
        System.out.println("(" + obj2.type + " , " + obj2.size + ",)");
```

```
    }
```

```
}
```

반환값이 없으면서 클래스 이름과 동일?
⇒ 생성자!

매개변수 없음

obj1



size	12
type	"슈퍼슈프림"
Pizza() { ... }	

```
Pizza obj1 = new Pizza();
```

(슈퍼슈프림 , 12,)
(포테이토 , 24,)

예제

- 앞의 Television 클래스에 생성자를 추가하여서 업그레이드 하여 보자. 생성자는 객체가 생성될 때, channel, volume, onOff 필드를 초기화한다.

```
class Television {  
    private int channel; // 채널 번호  
    private int volume; // 볼륨  
    private boolean onOff; // 전원 상태  
  
    Television(int c, int v, boolean o) {  
        channel = c;  
        volume = v;  
        onOff = o;  
    }  
    void print() {  
        System.out.println("채널은 " + channel + "이고 볼륨은 " + volume + "입니다.");  
    }  
}
```

예제

```
public class TelevisionTest {  
    public static void main(String[] args) {  
        Television myTv = new Television(7, 10, true);  
        myTv.print();  
  
        Television yourTv = new Television(11, 20, true);  
        yourTv.print();  
    }  
}
```

채널은 7이고 볼륨은 10입니다.
채널은 11이고 볼륨은 20입니다.

기본 생성자(default constructor)

- 매개 변수가 없는 생성자
- 만약 개발자가 생성자를 하나도 정의하지 않으면 자바 컴파일러는 기본 생성자를 자동으로 만든다. *→ 많이 봤음! ex) Box() { }*

```
class Box {  
    int width, height, depth;  
}  
  
public class BoxTest {  
    public static void main(String[] args) {  
        Box b = new Box();  
        System.out.println("상자의 크기: (" + b.width + "," + b.height + "," + b.depth + ")");  
    }  
}
```

상자의 크기: (0,0,0)

기본 생성자가 추가되지 않는 경우

- 프로그래머가 생성자를 한 개 이상 정의했을 경우

```
public class Box {  
    int width, height, depth;  
    public Box(int w, int h, int d) {  
        width=w;        height=h;        depth=d;  
    }  
  
    public static void main(String[] args) {  
        Box b = new Box(); // 오류 발생!!  
        System.out.println("상자의 크기: (" + b.width + "," + b.height + "," + b.depth + ")");  
    }  
}
```

this 참조 변수



this vs this()

- this는 현재 객체 자신을 가리키는 참조 변수
- this는 컴파일러에서 자동으로 생성
- 흔히 생성자에서 매개 변수 이름과 필드 이름이 동일한 경우에 혼동을 막기 위해서 사용

```
01 public class Circle {  
02     int radius;  
03  
04     public Circle(int radius) {  
05         (this).radius = radius;  
06     }  
07     double getArea() {  
08         return 3.14*radius*radius;  
09     }  
10 }
```

먼저 가까운 Scope에서 찾음
field의 변수라고 구분하기 위함

this.radius는 필드이고
radius는 매개 변수라는
것을 알 수 있네요.

this()

- this()는 다른 생성자를 의미 → 동인코딩에서 내의 다른 생성자의 기능을 사용하기 위함
(생성자 안에서 다른 생성자 호출)

```
01 public class Circle {  
02     int radius;  
03  
04     public Circle(int radius) {  
05         this.radius = radius;  
06     }  
07  
08     public Circle() {  
09         this(0);  
10     }  
11  
12     double getArea() {  
13         return 3.14 * radius * radius;  
14     }  
15 }
```

Circle(0)을 호출한다.

접근제어(access control) ☆ 3월

- 클래스의 멤버에 접근하는 것을 제어하는 것
- public이나 private의 접근 지정자를 멤버 앞에 붙여서 접근을 제한하게 된다.

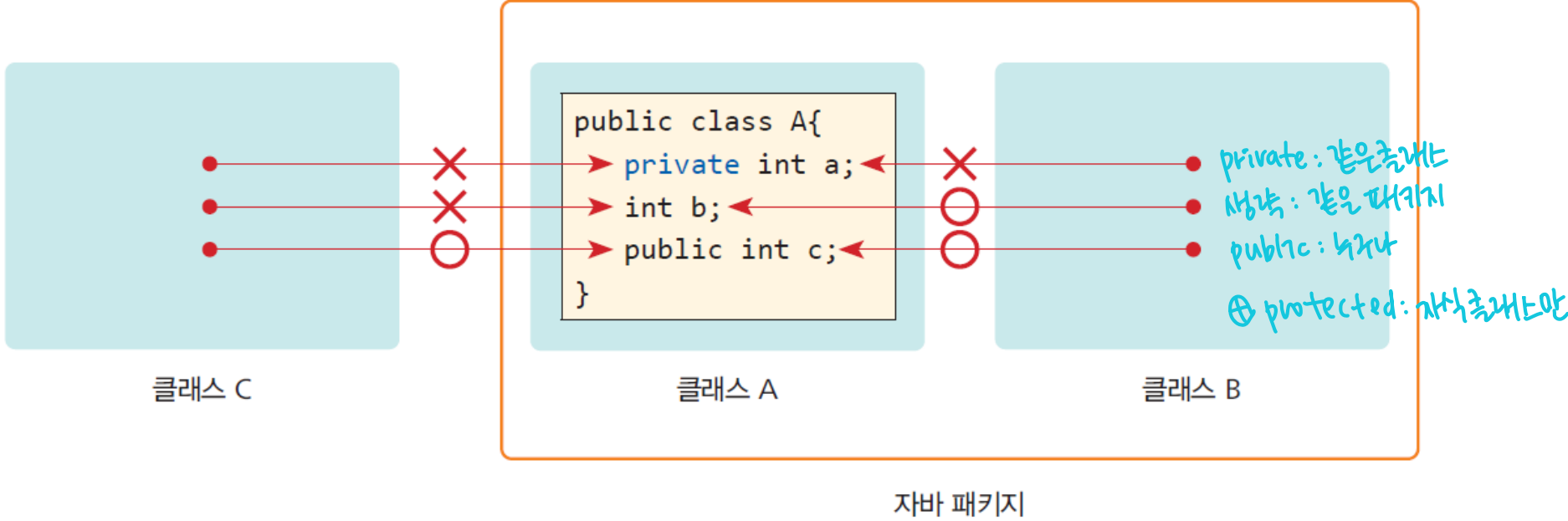
■ 접근 제어지정자

- ☆
 - private
 - 클래스 안에서만 접근할 수 있다.
 - public
 - 누구나 접근할 수 있다.
 - protected
 - 자식 클래스만 접근할 수 있다. 아직 학습하지 않음
 - 접근 지정자가 없는 경우 (생략)
 - default라고 불리며 동일한 패키지 안에서만 접근이 가능하다.
적지는 않음

예제

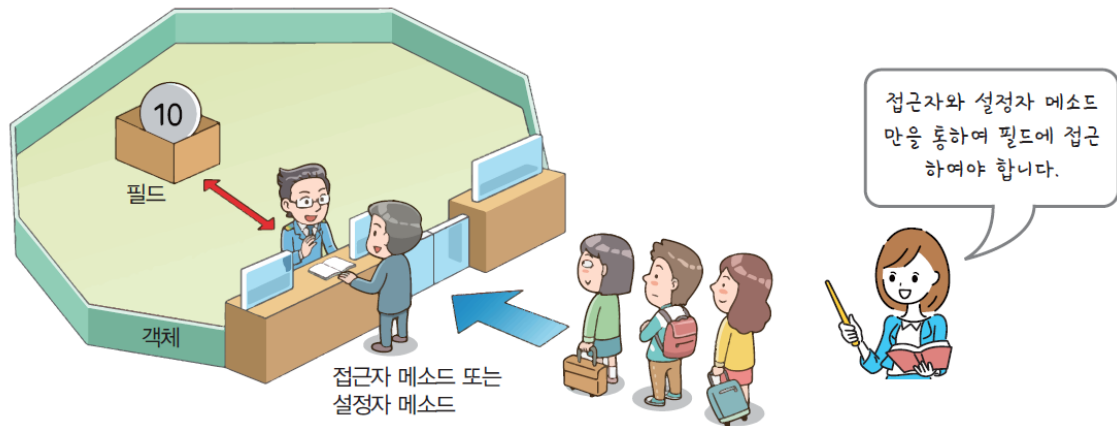
```
class A {  
    private int a;    // 전용  
    int b;            // 디폴트  
    public int c;     // 공용  
}  
  
public class Test {  
    public static void main(String args[]) {  
  
        A obj = new A(); // 객체 생성  
  
        obj.a = 10;           // 전용 멤버는 다른 클래스에서는 접근 안 됨  
        obj.b = 20;           // 디폴트 멤버는 접근할 수 있음  
        obj.c = 30;           // 공용 멤버는 접근할 수 있음  
    }  
}
```

접근 제어



접근자와 설정자

- 필드값을 반환하는 접근자(getters) 메소드
- 필드값을 설정하는 설정자(setters) 메소드
- 이러한 메소드는 대개 get이나 set이 메소드 이름 앞에 붙여진다.
 - 예를 들면 getBalance()는 접근자이고 setBalance()는 설정자이다.



잡근자와 설정자 예

```
class Account {  
    private int regNumber;  
    private String name;  
    private int balance;  
  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public int getBalance() { return balance; }  
    public void setBalance(int balance) { this.balance = balance; }  
}  
  
public class AccountTest {  
    public static void main(String[] args) {  
        Account obj = new Account();  
        obj.setName("Tom");  
        obj.setBalance(100000);  
        System.out.println("이름은 " + obj.getName() + " 통장 잔고는 " + obj.getBalance() + "입니다.");  
    }  
}
```

→ 직접 접근 불가

이름은 Tom 통장 잔고는 100000입니다.

접근자와 설정자를 사용하는 이유

- 설정자에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있다.
- 필요할 때마다 필드값을 동적으로 계산하여 반환할 수 있다.
- 접근자만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있다.

```
public void setAge(int age)
{
    if( age < 0 )
        this.age = 0;
    else
        this.age = age;
}
```

Lab: 안전한 배열 만들기

- 만약 인덱스가 배열의 크기를 벗어나게 되면 실행 오류가 발생한다. 따라서 실행 오류를 발생하지 않는 안전한 배열을 작성하여 보자.

<<SafeArray>>

-a[]: int

+length: int

+get(index: int):int

+put(index: int, value: int)

```
class SafeArray {  
    private int a[];  
    public int length;  
  
    public SafeArray(int size) {  
        a = new int[size];  
        length = size;  
    }  
    public int get(int index) {  
        if (index >= 0 && index < length) {  
            return a[index];  
        }  
        return -1;  
    }  
    public void put(int index, int value) {  
        if (index >= 0 && index < length)  
            a[index] = value;  
        else  
            System.out.println("잘못된 인덱스 " + index);  
    }  
}
```

Lab: 안전한 배열 만들기

```
public class SafeArrayTest {  
    public static void main(String args[]) {  
        SafeArray array = new SafeArray(3);  
  
        for (int i = 0; i < (array.length + 1); i++) {  
            array.put(i, i * 10);  
        }  
    }  
}
```

잘못된 인덱스 3

무엇을 클래스로 만들어야 할까?

- TV, 자동차와 같이 실제 생활에서 사용되는 객체들은, 클래스로 작성하기가 비교적 쉽다. 하지만 프로그램 안에서는 실제 생활에는 사용되지 않는 추상적인 객체들도 많이 존재한다.
- 요구 사항이 문서화되면 클래스 식별 과정을 시작할 수 있다. 요구 사항에서 클래스를 식별하는 한 가지 방법은 모든 “명사”를 표시하는 것이다.

은행은 정기 예금 계좌와 보통 예금 계좌를 제공한다. 고객들은 자신의 계좌에 돈을 입금할 수 있으며 계좌에서 돈을 인출할 수 있다. 그리고 각 계좌는 기간에 따라 이자를 지급한다. 계좌마다 이자는 달라진다.



메소드 결정

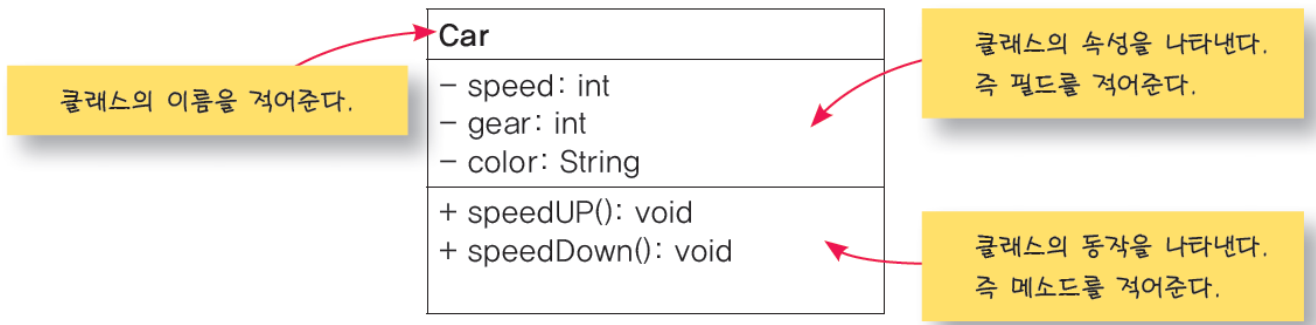
- 요구 사항 문서에서 “동사”에 해당되는 부분이 메소드가 되는 경우가 많다. 앞의 요구 사항 문서에서 “고객들은 자신의 계좌에 돈을 입금할 수 있으며 계좌에서 돈을 인출할 수 있다.”라는 문장이 있다. 여기서 “입금한다”, “인출한다”와 같은 동사는 deposit(), withdraw() 메소드로 매핑시킬 수 있다.

은행은 정기 예금 계좌와 보통 예금 계좌를 제공한다. 고객들은 자신의 계좌에 돈을 입금할 수 있으며 계좌에서 돈을 인출할 수 있다. 그리고 각 계좌는 기간에 따라 이자를 지급한다. 계좌마다 이자는 달라진다.



UML

- UML(Unified Modeling Language): UML은 클래스만을 그리는 도구는 아니고 객체지향설계 시에 사용되는 일반적인 모델링 언어라고 할 수 있다.
- UML을 사용하면 소프트웨어를 본격적으로 작성하기 전에 구현하고자 하는 시스템을 시각화하여 검토할 수 있다.



가시성 표시자







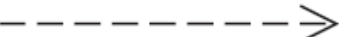
- 필드나 메소드의 이름 앞에는 가시성 표시자(visibility indicator)가 올 수 있다.
- +는 public을, -는 private을 의미한다.



+	Public
-	Private
#	Protected
/	Derived
~	Package

클래스 간의 관계

표 4-1 UML에서 사용되는 화살표의 종류

관계	화살표
일반화(generalization), 상속(inheritance)	
구현(realization)	
구성관계(composition)	
집합관계(aggregation)	
유향 연관(direct association)	
양방향 연관(bidirectional association)	
의존(dependency)	

의존 관계

- 의존(dependency)이란 하나의 클래스가 다른 클래스를 사용하는 관계이다.

CarTest.java

```
01 public class CarTest {  
02     public static void main(String[] args) {  
03         Car myCar = new Car();  
04         myCar.speedUp();  
05     }  
06 }
```

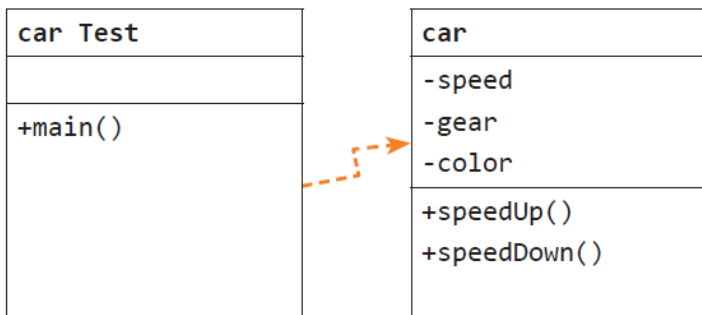
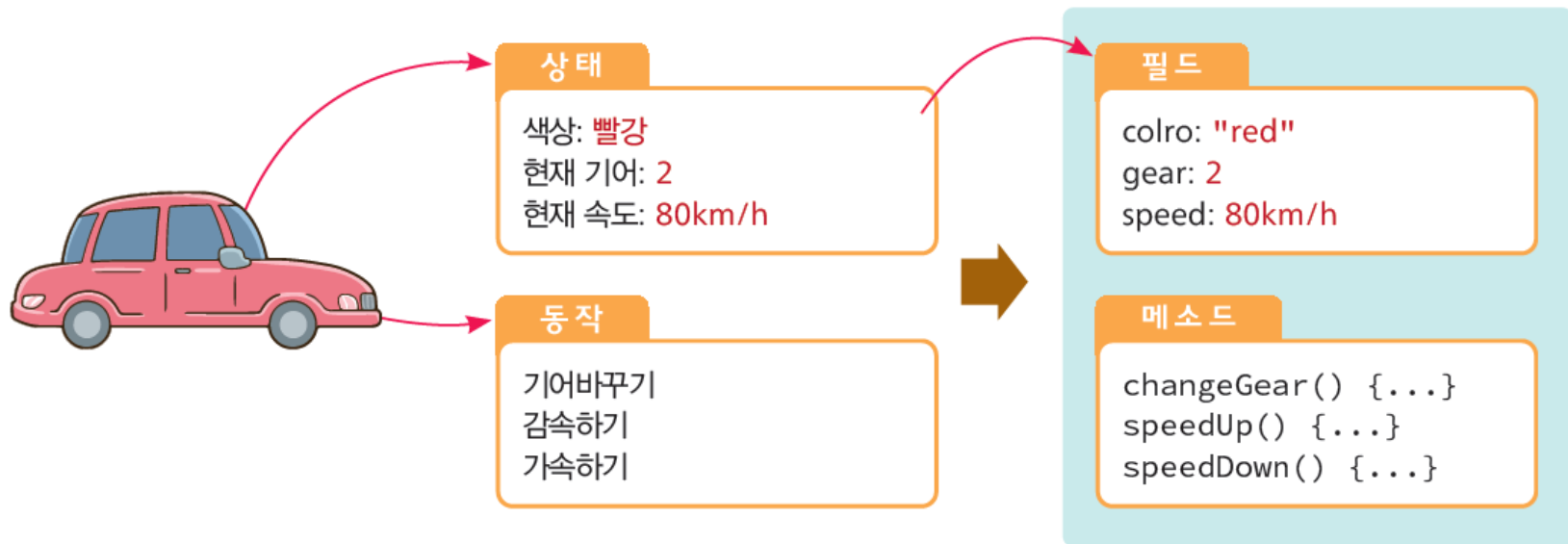


그림 4.6 Car 예제의 UML

Lab: 자동차 클래스 작성

- 자동차를 나타내는 클래스를 정의하여 보자. 예를 들어, 자동차 객체의 경우 속성은 색상, 현재 속도, 현재 기어 등이다. 자동차의 동작은 기어 변속하기, 가속하기, 감속하기 등이 있다.



Sol: 자동차 클래스 작성

```
class Car {  
    String color; // 색상  
    int speed; // 속도  
    int gear; // 기어  
    @Override  
    public String toString() {  
        return "Car [color=" + color + ", speed=" + speed + ", gear=" + gear + "];"  
    }  
    void changeGear(int g) {        gear = g;        }  
    void speedUp() {        speed = speed + 10;        }  
    void speedDown() {        speed = speed - 10;        }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.changeGear(1);  
        myCar.speedUp();  
        System.out.println(myCar);  
    }  
}
```

Car [color=null, speed=10, gear=1]

Lab: 은행 계좌 클래스 작성

- 은행 계좌를 클래스로 정의하여 보자. 먼저 잔액은 변수 `balance`로 표시한다. 예금 입금은 `deposit()` 메소드로, 예금 인출은 `withdraw()` 메소드로 나타내면 된다.

BankAccount
-owner : string
-accountNumber : int
-balance : int
+deposit()
+withdraw()



Soi: 은행 계좌 클래스 작성

```
class BankAccount { // 은행 계좌
    int accountNumber; // 계좌 번호
    String owner; // 예금주
    int balance; // 잔액을 표시하는 변수

    void deposit(int amount) { // 저금        balance += amount;    }
    void withdraw(int amount) { // 인출        balance -= amount;    }
    public String toString(){
        return "현재 잔액은 " + balance + "입니다.";
    }
}
```

```
public class BankAccountTest {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount();
        myAccount.deposit(10000);
        System.out.println(myAccount);
        myAccount.withdraw(8000);
        System.out.println(myAccount);
    }
}
```

현재 잔액은 10000입니다.

현재 잔액은 2000입니다.

Mini Project: 주사위 클래스

- 주사위를 Dice 클래스로 모델링한다. Dice 클래스는 주사위면(face)을 필드로 가지고 있고 roll(), getValue(), setValue() 등의 메소드를 가지고 있다. 생성자에서는 주사위면을 0으로 초기화한다.

주사위1= 5 주사위2= 5

주사위1= 3 주사위2= 4

주사위1= 1 주사위2= 1

(1, 1)이 나오는데 걸린 횟수= 3