6. 접근 제어자

#0.강의/1.자바로드맵/2.자바-기본

접근 제어자 이해1

자바는 `public`, `private` 같은 접근 제어자(access modifier)를 제공한다. 접근 제어자를 사용하면 해당 클래스 외부에서 특정 필드나 메서드에 접근하는 것을 허용하거나 제한할 수 있다.
이런 접근 제어자가 왜 필요할까? 예제를 통해 접근 제어자가 필요한 이유를 알아보자.

여러분은 스피커에 들어가는 소프트웨어를 개발하는 개발자다.
스피커의 음량은 절대로 100을 넘으면 안된다는 요구사항이 있다. (**100을 넘어가면 스피커의 부품들이 고장난다.**)

스피커 객체를 만들어보자.
스피커는 음량을 높이고, 내리고, 현재 음량을 확인할 수 있는 단순한 기능을 제공한다.
요구사항 대로 스피커의 음량은 100까지만 증가할 수 있다. 절대 100을 넘어가면 안된다.

Speaker

```
package access;

public class Speaker {

    int volume;

    Speaker(int volume) {
        this.volume = volume;
    }
```

```
    void volumeUp() {
        if (volume >= 100) {
            System.out.println("음량을 증가할 수 없습니다. 최대 음량입니다.");
        } else {
            volume += 10;
            System.out.println("음량을 10 증가합니다.");
        }
    }

    void volumeDown() {
        volume -= 10;
        System.out.println("volumeDown 호출");
    }

    void showVolume() {
        System.out.println("현재 음량:" + volume);
    }
}
```

생성자를 통해 초기 음량 값을 지정할 수 있다.

`volumeUp()` 메서드를 보자. 음량을 한번에 10씩 증가한다. 단 음량이 100을 넘게되면 더는 음량을 증가하지 않는다.

SpeakerMain

```
package access;

public class SpeakerMain {
    public static void main(String[] args) {
        Speaker speaker = new Speaker(90);
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();
    }
}
```

실행 결과

```
현재 음량: 90
음량을 10 증가합니다.
현재 음량: 100
음량을 증가할 수 없습니다. 최대 음량입니다.
현재 음량: 100
```

초기 음량 값을 90으로 지정했다. 그리고 음량을 높이는 메서드를 여러번 호출했다.
기대한 대로 음량은 100을 넘지 않았다. 프로젝트는 성공적으로 끝났다.

오랜 시간이 흘러서 업그레이드 된 다음 버전의 스피커를 출시하게 되었다. 이때는 새로운 개발자가 급하게 기존 코드를 이어받아서 개발을 하게 되었다. 참고로 새로운 개발자는 기존 요구사항을 잘 몰랐다. 코드를 실행해보니 이상하게 음량이 100이상 올라가지 않았다. 소리를 더 올리면 좋겠다고 생각한 개발자는 다양한 방면으로 고민했다.
`Speaker` 클래스를 보니 `volume` 필드를 직접 사용할 수 있었다. `volume` 필드의 값을 200으로 설정하고 이 코드를 실행한 순간 스피커의 부품들에 과부하가 걸리면서 폭발했다.

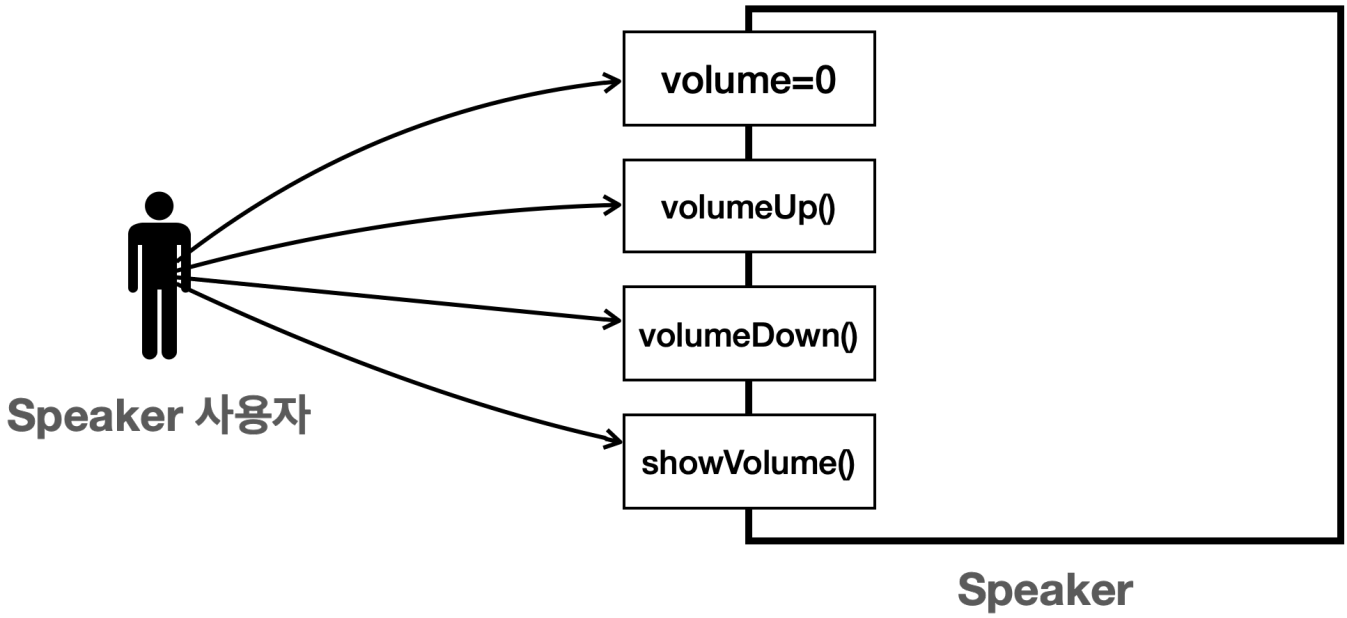
SpeakerMain - 필드 직접 접근 코드 추가

```java
package access;

public class SpeakerMain {
    public static void main(String[] args) {
        Speaker speaker = new Speaker(90);
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();

        //필드에 직접 접근
        System.out.println("volume 필드 직접 접근 수정");
        speaker.volume = 200;
        speaker.showVolume();
    }
}
```

실행 결과

```
현재 음량: 90
음량을 10 증가합니다.
현재 음량: 100
음량을 증가할 수 없습니다. 최대 음량입니다.
현재 음량: 100
volume 필드 직접 접근 수정
현재 음량: 200
```

volume 필드



`Speaker` 객체를 사용하는 사용자는 `Speaker`의 `volume` 필드와 메서드에 모두 접근할 수 있다.
앞서 `volumeUp()`과 같은 메서드를 만들어서 음량이 100을 넘지 못하도록 기능을 개발했지만 소용이 없다. 왜냐하면 `Speaker`를 사용하는 입장에서는 `volume` 필드에 직접 접근해서 원하는 값을 설정할 수 있기 때문이다.

이런 문제를 근본적으로 해결하기 위해서는 `volume` 필드의 외부 접근을 막을 수 있는 방법이 필요하다.

접근 제어자 이해2

이 문제를 근본적으로 해결하는 방법은 `volume` 필드를 `Speaker` 클래스 외부에서는 접근하지 못하게 막는 것이다.

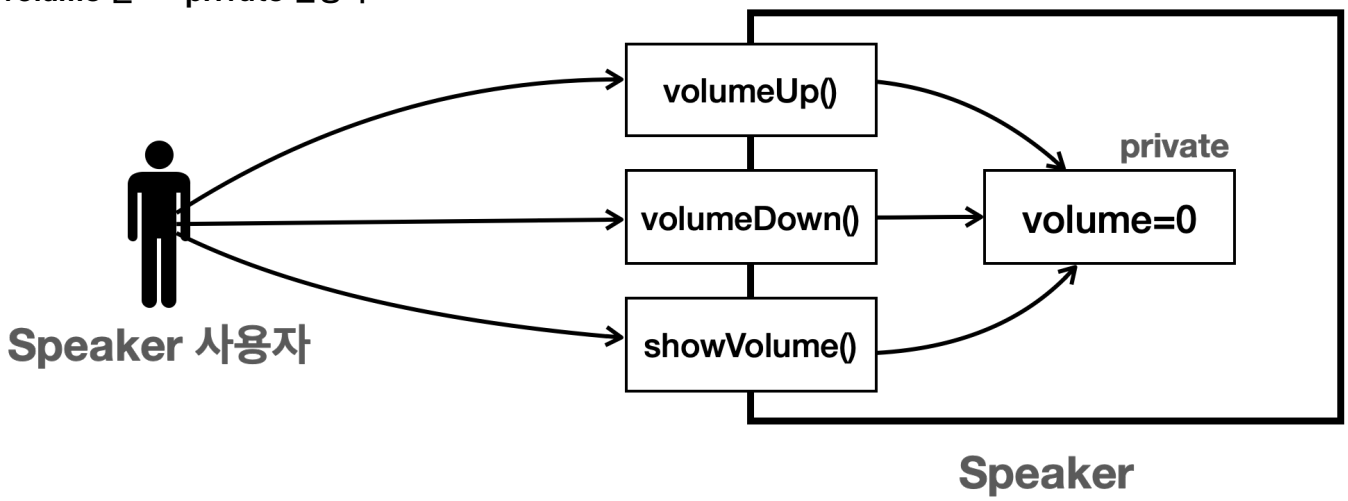
Speaker - volume 접근 제어자를 private으로 수정

```
package access;
```

```
public class Speaker {

    private int volume; //private 사용
    ...
}
```

`private` 접근 제어자는 모든 외부 호출을 막는다. 따라서 `private`이 붙은 경우 해당 클래스 내부에서만 호출할 수 있다.

volume 필드 - private 변경 후



그림을 보면 `volume` 필드를 `private`을 사용해서 `Speaker` 내부에 숨겼다.
외부에서 `volume` 필드에 직접 접근할 수 없게 막은 것이다. `volume` 필드는 이제 `Speaker` 내부에서만 접근할 수 있다.

이제 `SpeakerMain` 코드를 다시 실행해보자.

```
//필드에 직접 접근
System.out.println("volume 필드 직접 접근 수정");
speaker.volume = 200; //private 접근 오류
```

IDE에서 `speaker.volume = 200` 부분에 오류가 발생하는 것을 확인할 수 있다. 실행해보면 다음과 같은 컴파일 오류가 발생한다.

컴파일 오류 메시지

```
volume has private access in access.Speaker
```

`volume` 필드는 `private`으로 설정되어 있기 때문에 외부에서 접근할 수 없다는 오류이다.

volume 필드 직접 접근 - 주석 처리

```
//필드에 직접 접근
System.out.println("volume 필드 직접 접근 수정");
//speaker.volume = 200; //private 접근 오류
speaker.showVolume();
```

이제 `Speaker` 외부에서 `volume` 필드에 직접 접근하는 것은 불가능하다. 이 경우 자바 컴파일러가 컴파일 오류를 발생시킨다.
프로그램을 실행하기 위해서 `volume` 필드에 직접 접근하는 코드를 주석 처리하자.

만약 `Speaker` 클래스를 개발하는 개발자가 처음부터 `private`을 사용해서 `volume` 필드의 외부 접근을 막아두었다면 어떠했을까? 새로운 개발자도 `volume` 필드에 직접 접근하지 않고, `volumeUp()`과 같은 메서드를 통해서 접근했을 것이다. 결과적으로 `Speaker`가 폭발하는 문제는 발생하지 않았을 것이다.

> **참고:** 좋은 프로그램은 무한한 자유도가 주어지는 프로그램이 아니라 적절한 제약을 제공하는 프로그램이다.

접근 제어자 종류

자바는 4가지 종류의 접근 제어자를 제공한다.

접근 제어자의 종류

- `private`: 모든 외부 호출을 막는다.
- `default` (package-private): 같은 패키지안에서 호출은 허용한다.
- `protected`: 같은 패키지안에서 호출은 허용한다. 패키지가 달라도 상속 관계의 호출은 허용한다.
- `public`: 모든 외부 호출을 허용한다.

순서대로 `private`이 가장 많이 차단하고, `public`이 가장 많이 허용한다.
`private -> default -> protected -> public`

참고로 `protected`는 상속 관계에서 자세히 설명한다.

package-private

접근 제어자를 명시하지 않으면 같은 패키지 안에서 호출을 허용하는 `default` 접근 제어자가 적용된다.

`default`라는 용어는 해당 접근 제어자가 기본값으로 사용되기 때문에 붙여진 이름이지만, 실제로는 `package-private`이 더 정확한 표현이다. 왜냐하면 해당 접근 제어자를 사용하는 멤버는 동일한 패키지 내의 다른 클래스에서만 접근이 가능하기 때문이다. 참고로 두 용어를 함께 사용한다.

접근 제어자 사용 위치

접근 제어자는 필드와 메서드, 생성자에 사용된다.

추가로 클래스 레벨에도 일부 접근 제어자를 사용할 수 있다. 이 부분은 뒤에서 따로 설명한다.

접근 제어자 예시

```
public class Speaker { //클래스 레벨

    private int volume; //필드

    public Speaker(int volume) {} //생성자

    public void volumeUp() {} //메서드
    public void volumeDown() {}
    public void showVolume() {}
}
```

접근 제어자의 핵심은 속성과 기능을 외부로부터 숨기는 것이다.

- `private`은 나의 클래스 안으로 속성과 기능을 숨길 때 사용, 외부 클래스에서 해당 기능을 호출할 수 없다.
- `default`는 나의 패키지 안으로 속성과 기능을 숨길 때 사용, 외부 패키지에서 해당 기능을 호출할 수 없다.
- `protected`는 상속 관계로 속성과 기능을 숨길 때 사용, 상속 관계가 아닌 곳에서 해당 기능을 호출할 수 없다.
- `public`은 기능을 숨기지 않고 어디서든 호출할 수 있게 공개한다.

접근 제어자 사용 - 필드, 메서드

다양한 상황에 따른 접근 제어자를 확인해보자.

주의! 지금부터는 패키지 위치가 매우 중요하다. 패키지 위치에 주의하자

필드, 메서드 레벨의 접근 제어자

AccessData

```
package access.a;

public class AccessData {

    public int publicField;
    int defaultField;
    private int privateField;

    public void publicMethod() {
        System.out.println("publicMethod 호출 "+ publicField);
    }

    void defaultMethod() {
        System.out.println("defaultMethod 호출 " + defaultField);
    }

    private void privateMethod() {
        System.out.println("privateMethod 호출 " + privateField);
    }

    public void innerAccess() {
        System.out.println("내부 호출");
        publicField = 100;
        defaultField = 200;
        privateField = 300;
        publicMethod();
        defaultMethod();
        privateMethod();
    }
}
```

- 패키지 위치는 `package access.a`이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- 순서대로 `public`, `default`, `private`을 필드와 메서드에 사용했다.
- 마지막에 `innerAccess()`가 있는데, 이 메서드는 내부 호출을 보여준다. 내부 호출은 자기 자신에게 접근하는 것이다. 따라서 `private`을 포함한 모든 곳에 접근할 수 있다.

이제 외부에서 이 클래스에 접근해보자.

AccessInnerMain

```
package access.a;

public class AccessInnerMain {
    public static void main(String[] args) {
        AccessData data = new AccessData();
        //public 호출 가능
        data.publicField = 1;
        data.publicMethod();

        //같은 패키지 default 호출 가능
        data.defaultField = 2;
        data.defaultMethod();

        //private 호출 불가
        //data.privateField = 3;
        //data.privateMethod();

        data.innerAccess();
    }
}
```

- 패키지 위치는 `package access.a` 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- `public` 은 모든 접근을 허용하기 때문에 필드, 메서드 모두 접근 가능하다.
- `default` 는 같은 패키지에서 접근할 수 있다. `AccessInnerMain` 은 `AccessData` 와 같은 패키지이다. 따라서 `default` 접근 제어자에 접근할 수 있다.
- `private` 은 `AccessData` 내부에서만 접근할 수 있다. 따라서 호출 불가다.
- `AccessData.innerAccess()` 메서드는 `public` 이다. 따라서 외부에서 호출할 수 있다. `innerAccess()` 메서드는 외부에서 호출되었지만 `innerAccess()` 메서드는 `AccessData` 에 포함되어 있다. 이 메서드는 자신의 `private` 필드와 메서드에 모두 접근할 수 있다.

실행 결과

```
publicMethod 호출 1
defaultMethod 호출 2
내부 호출
publicMethod 호출 100
defaultMethod 호출 200
privateMethod 호출 300
```

AccessOuterMain

```
package access.b;

import access.a.AccessData;

public class AccessOuterMain {
    public static void main(String[] args) {
        AccessData data = new AccessData();
        //public 호출 가능
        data.publicField = 1;
        data.publicMethod();

        //다른 패키지 default 호출 불가
        //data.defaultField = 2;
        //data.defaultMethod();

        //private 호출 불가
        //data.privateField = 3;
        //data.privateMethod();

        data.innerAccess();
    }
}
```

- 패키지 위치는 `package access.b` 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- `public` 은 모든 접근을 허용하기 때문에 필드, 메서드 모두 접근할 수 있다.
- `default` 는 같은 패키지에서 접근할 수 있다. `access.b.AccessOuterMain` 은 `access.a.AccessData` 와 다른 패키지이다. 따라서 `default` 접근 제어자에 접근할 수 없다.
- `private` 은 `AccessData` 내부에서만 접근할 수 있다. 따라서 호출 불가다.
- `AccessData.innerAccess()` 메서드는 `public` 이다. 따라서 외부에서 호출할 수 있다. `innerAccess()` 메서드는 외부에서 호출되었지만 해당 메서드 안에서는 자신의 `private` 필드와 메서드에 접근할 수 있다.

실행 결과

```
publicMethod 호출 1
내부 호출
publicMethod 호출 100
defaultMethod 호출 200
```

```
privateMethod 호출 300
```

참고로 생성자도 접근 제어자 관점에서 메서드와 같다.

접근 제어자 사용 - 클래스 레벨

클래스 레벨의 접근 제어자 규칙

- 클래스 레벨의 접근 제어자는 `public`, `default`만 사용할 수 있다.
  - `private`, `protected`는 사용할 수 없다.
- `public` 클래스는 반드시 파일명과 이름이 같아야 한다.
  - 하나의 자바 파일에 `public` 클래스는 하나만 등장할 수 있다.
  - 하나의 자바 파일에 `default` 접근 제어자를 사용하는 클래스는 무한정 만들 수 있다.

`PublicClass.java` **파일**

```
package access.a;

public class PublicClass {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        DefaultClass1 class1 = new DefaultClass1();
        DefaultClass2 class2 = new DefaultClass2();
    }
}

class DefaultClass1 {
}

class DefaultClass2 {
}
```

- 패키지 위치는 `package access.a`이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- `PublicClass`라는 이름의 클래스를 만들었다. 이 클래스는 `public` 접근 제어자다. 따라서 파일명과 이 클래스의 이름이 반드시 같아야 한다. 이 클래스는 `public`이기 때문에 외부에서 접근할 수 있다.
- `DefaultClass1`, `DefaultClass2`는 `default` 접근 제어자다. 이 클래스는 `default`이기 때문에 같은

패키지 내부에서만 접근할 수 있다.

- `PublicClass`의 `main()`을 보면 각각의 클래스를 사용하는 예를 보여준다.
  - `PublicClass`는 `public` 접근 제어다. 따라서 어디서든 사용할 수 있다. `DefaultClass1`, `DefaultClass2`와는 같은 패키지에 있으므로 사용할 수 있다.

PublicClassInnerMain

```
package access.a;

public class PublicClassInnerMain {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        DefaultClass1 class1 = new DefaultClass1();
        DefaultClass2 class2 = new DefaultClass2();
    }
}
```

- 패키지 위치는 `package access.a`이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- `PublicClass`는 `public` 클래스이다. 따라서 외부에서 접근할 수 있다.
- `PublicClassInnerMain`와 `DefaultClass1`, `DefaultClass2`는 같은 패키지이다. 따라서 접근할 수 있다.

```
package access.b;

//import access.a.DefaultClass1;
import access.a.PublicClass;

public class PublicClassOuterMain {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();

        //다른 패키지 접근 불가
        //DefaultClass1 class1 = new DefaultClass1();
        //DefaultClass2 class2 = new DefaultClass2();
    }
}
```

- 패키지 위치는 `package access.b`이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- `PublicClass`는 `public` 클래스이다. 따라서 외부에서 접근할 수 있다.

- `PublicClassOuterMain`와 `DefaultClass1`, `DefaultClass2`는 다른 패키지이다. 따라서 접근할 수 없다.

캡슐화

캡슐화(Encapsulation)는 객체 지향 프로그래밍의 중요한 개념 중 하나다. 캡슐화는 데이터와 해당 데이터를 처리하는 메서드를 하나로 묶어서 외부에서의 접근을 제한하는 것을 말한다. 캡슐화를 통해 데이터의 직접적인 변경을 방지하거나 제한할 수 있다.
캡슐화는 쉽게 이야기해서 속성과 기능을 하나로 묶고, 외부에 꼭 필요한 기능만 노출하고 나머지는 모두 내부로 숨기는 것이다.

이전에 객체 지향 프로그래밍을 설명하면서 캡슐화에 대해 알아보았다. 이때는 데이터와 데이터를 처리하는 메서드를 하나로 모으는 것에 초점을 맞추었다. 여기서 한발짝 더 나아가 캡슐화를 안전하게 완성할 수 있게 해주는 장치가 바로 접근 제어자다.

그럼 어떤 것을 숨기고 어떤 것을 노출해야 할까?

1. 데이터를 숨겨라
객체에는 속성(데이터)과 기능(메서드)이 있다. 캡슐화에서 가장 필수로 숨겨야 하는 것은 속성(데이터)이다.
`Speaker`의 `volume`을 떠올려보자. 객체 내부의 데이터를 외부에서 함부로 접근하게 두면, 클래스 안에서 데이터를 다루는 모든 로직을 무시하고 데이터를 변경할 수 있다. 결국 모든 안전망을 다 빠져나가게 된다. 따라서 캡슐화가 깨진다.

우리가 자동차를 운전할 때 자동차 부품을 다 열어서 그 안에 있는 속도계를 직접 조절하지 않는다. 단지 자동차가 제공하는 엑셀 기능을 사용해서 엑셀을 밟으면 자동차가 나머지는 다 알아서 하는 것이다.

우리가 일상에서 생각할 수 있는 음악 플레이어를 떠올려보자. 음악 플레이어를 사용할 때 그 내부에 들어있는 전원부나, 볼륨 상태의 데이터를 직접 수정할 일이 있을까? 우리는 그냥 음악 플레이어의 켜고, 끄고, 볼륨을 조절하는 버튼을 누를 뿐이다. 그 내부에 있는 전원부나, 볼륨의 상태 데이터를 직접 수정하지 않는다. 전원 버튼을 눌렀을 때 실제 전원을 받아서 전원을 켜는 것은 음악 플레이어의 일이다. 볼륨을 높였을 때 내부에 있는 볼륨 장치들을 움직이고 볼륨 수치를 조절하는 것도 음악 플레이어가 스스로 해야하는 일이다. 쉽게 이야기해서 우리는 음악 플레이어가 제공하는 기능을 통해서 음악 플레이어를 사용하는 것이다. 복잡하게 음악 플레이어의 내부를 까서 그 내부 데이터까지 우리가 직접 사용하는 것은 아니다.

객체의 데이터는 객체가 제공하는 기능인 메서드를 통해서 접근해야 한다.

2. 기능을 숨겨라

객체의 기능 중에서 외부에서 사용하지 않고 내부에서만 사용하는 기능들이 있다. 이런 기능도 모두 감추는 것이 좋다.
우리가 자동차를 운전하기 위해 자동차가 제공하는 복잡한 엔진 조절 기능, 배기 기능까지 우리가 알 필요는 없다. 우리는 단지 엑셀과 핸들 정도의 기능만 알면 된다.
만약 사용자에게 이런 기능까지 모두 알려준다면, 사용자가 자동차에 대해 너무 많은 것을 알아야 한다.
사용자 입장에서 꼭 필요한 기능만 외부에 노출하자. 나머지 기능은 모두 내부로 숨기자

정리하면 데이터는 모두 숨기고, 기능은 꼭 필요한 기능만 노출하는 것이 좋은 캡슐화이다.

이번에는 잘 캡슐화된 예제를 하나 만들어보자.

BankAccount

```
package access;

public class BankAccount {

    private int balance;

    public BankAccount() {
        balance = 0;
    }

    // public 메서드: deposit
    public void deposit(int amount) {
        if (isAmountValid(amount)) {
            balance += amount;
        } else {
            System.out.println("유효하지 않은 금액입니다.");
        }
    }

    // public 메서드: withdraw
    public void withdraw(int amount) {
        if (isAmountValid(amount) && balance - amount >= 0) {
            balance -= amount;
        } else {
            System.out.println("유효하지 않은 금액이거나 잔액이 부족합니다.");
        }
```

```
    }

    // public 메서드: getBalance
    public int getBalance() {
        return balance;
    }

    // private 메서드: isAmountValid
    private boolean isAmountValid(int amount) {
        // 금액이 0보다 커야함
        return amount > 0;
    }
}
```

```
package access;

public class BankAccountMain {

    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.deposit(10000);
        account.withdraw(3000);
        System.out.println("balance = " + account.getBalance());
    }
}
```

은행 계좌 기능을 다룬다. 다음과 같은 기능을 가지고 있다.

private

- `balance` : 데이터 필드는 외부에 직접 노출하지 않는다. `BankAccount`가 제공하는 메서드를 통해서만 접근할 수 있다.
- `isAmountValid()` : 입력 금액을 검증하는 기능은 내부에서만 필요한 기능이다. 따라서 `private`을 사용했다.

public

- `deposit()` : 입금
- `withdraw`(): 출금
- `getBalance()` : 잔고

`BankAccount`를 사용하는 입장에서는 단 3가지 메서드만 알면 된다. 나머지 복잡한 내용은 모두 `BankAccount` 내부에 숨어있다.

만약 `isAmountValid()`를 외부에 노출하면 어떻게 될까? `BankAccount`를 사용하는 개발자 입장에서는 사용할 수 있는 메서드가 하나 더 늘었다. 여러분이 `BankAccount`를 사용하는 개발자라면 어떤 생각을 할까? 아마도 입금과 출금 전에 본인이 먼저 `isAmountValid()`를 사용해서 검증을 해야 하나? 라고 의문을 가질 것이다.

만약 `balance` 필드를 외부에 노출하면 어떻게 될까? `BankAccount`를 사용하는 개발자 입장에서는 이 필드를 직접 사용해도 된다고 생각할 수 있다. 왜냐하면 외부에 공개하는 것은 그것을 외부에서 사용해도 된다는 뜻이기 때문이다. 결국 모든 검증과 캡슐화가 깨지고 잔고를 무한정 늘리고 출금하는 심각한 문제가 발생할 수 있다.

접근 제어자와 캡슐화를 통해 데이터를 안전하게 보호하는 것은 물론이고, `BankAccount`를 사용하는 개발자 입장에서 해당 기능을 사용하는 복잡도도 낮출 수 있다.

문제와 풀이

문제 - 최대 카운터와 캡슐화

`MaxCounter` 클래스를 만드세요.
이 클래스는 최대값을 지정하고 최대값 까지만 값이 증가하는 기능을 제공합니다.

- `int count`: 내부에서 사용하는 숫자 입니다. 초기값은 0입니다.
- `int max`: 최대값 입니다. 생성자를 통해 입력합니다.
- `increment()` 숫자를 하나 증가합니다.
- `getCount()` 지금까지 증가한 값을 반환합니다.

요구사항

- **접근 제어자를 사용해서 데이터를 캡슐화 하세요.**
- **해당 클래스는 다른 패키지에서도 사용할 수 있어야 합니다.**

```
package access.ex;

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

```
        MaxCounter counter = new MaxCounter(3);
        counter.increment();
        counter.increment();
        counter.increment();
        counter.increment();
        int count = counter.getCount();
        System.out.println(count);
    }
}
```

실행 결과

```
최대값을 초과할 수 없습니다.
3
```

정답

```
package access.ex;

public class MaxCounter {
    private int count = 0;
    private int max;

    public MaxCounter(int max) {
        this.max = max;
    }

    public void increment() {
        if (count >= max) {
            System.out.println("최대값을 초과할 수 없습니다.");
            return;
        }
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

문제 - 쇼핑 카트

`ShoppingCartMain` 코드가 작동하도록 `Item`, `ShoppingCart` 클래스를 완성해라.

요구사항

- **접근 제어자를 사용해서 데이터를 캡슐화 하세요.**
- **해당 클래스는 다른 패키지에서도 사용할 수 있어야 합니다.**
- 장바구니에는 상품의 종류를 최대 10가지만 담을 수 있다.
  - 상품의 종류 10개 초과 등록시: "장바구니가 가득 찼습니다." 출력

```
package access.ex;

public class ShoppingCartMain {

    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        Item item1 = new Item("마늘", 2000, 2);
        Item item2 = new Item("상추", 3000, 4);

        cart.addItem(item1);
        cart.addItem(item2);

        cart.displayItems();
    }
}
```

실행 결과

```
장바구니 상품 출력
상품명:마늘, 합계:4000
상품명:상추, 합계:12000
전체 가격 합:16000
```

Item 클래스

```
package access.ex;

public class Item {
    private String name;
    private int price;
    private int quantity;

    //TODO 나머지 코드를 완성해라.
}
```

ShoppingCart 클래스

```
package access.ex;

public class ShoppingCart {
    private Item[] items = new Item[10];
    private int itemCount;

    //TODO 나머지 코드를 완성해라.
}
```

정답

```
package access.ex;

public class Item {
    private String name;
    private int price;
    private int quantity;

    public Item(String name, int price, int quantity) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    public String getName() {
        return name;
    }
```

```
    public int getTotalPrice() {
        return price * quantity;
    }
}
```

각각의 `Item`의 가격과 수량을 곱하면 각 상품별 합계를 구할 수 있다. `price`와 `quantity`를 외부에 반환한 다음에 외부에서 곱해서 상품별 합계를 구해도 되지만, `getTotalPrice()` 메서드를 제공하면 외부에서는 단순히 이 메서드를 호출하면 된다. 이 메서드의 핵심은 자신이 가진 데이터를 사용한다는 점이다.

```
package access.ex;

public class ShoppingCart {
    private Item[] items = new Item[10];
    private int itemCount;

    public void addItem(Item item) {
        if (itemCount >= items.length) {
            System.out.println("장바구니가 가득 찼습니다.");
            return;
        }

        items[itemCount] = item;
        itemCount++;
    }

    public void displayItems() {
        System.out.println("장바구니 상품 출력");
        for (int i = 0; i < itemCount; i++) {
            Item item = items[i];
            System.out.println("상품명:" + item.getName() + ", 합계:" +
item.getTotalPrice());
        }
        System.out.println("전체 가격 합:" + calculateTotalPrice());
    }

    private int calculateTotalPrice() {
        int totalPrice = 0;
        for (int i = 0; i < itemCount; i++) {
            Item item = items[i];
```

```
            totalPrice += item.getTotalPrice();
        }
        return totalPrice;
    }

}
```

- `calculateTotalPrice()` : 이 메서드 내부에서만 사용되므로 `private` 접근 제어자를 사용한다.

정리