3. 객체 지향 프로그래밍

#1.인강/0.자바/2.자바-기본

절차 지향 프로그래밍1 - 시작

절차 지향 프로그래밍 vs 객체 지향 프로그래밍

프로그래밍 방식은 크게 절차 지향 프로그래밍과 객체 지향 프로그래밍으로 나눌 수 있다.

절차 지향 프로그래밍

- 절차 지향 프로그래밍은 이름 그대로 절차를 지향한다. 쉽게 이야기해서 실행 순서를 중요하게 생각하는 방식이다.
- 절차 지향 프로그래밍은 프로그램의 흐름을 순차적으로 따르며 처리하는 방식이다. 즉, "어떻게"를 중심으로 프로그래밍 한다.

객체 지향 프로그래밍

- 객체 지향 프로그래밍은 이름 그대로 객체를 지향한다. 쉽게 이야기해서 객체를 중요하게 생각하는 방식이다.
- 객체 지향 프로그래밍은 실제 세계의 사물이나 사건을 객체로 보고, 이러한 객체들 간의 상호작용을 중심으로 프로그래밍하는 방식이다. 즉, "무엇을" 중심으로 프로그래밍 한다.

둘의 중요한 차이

- 절차 지향은 데이터와 해당 데이터에 대한 처리 방식이 분리되어 있다. 반면 객체 지향에서는 데이터와 그 데이터에 대한 행동(메서드)이 하나의 '객체' 안에 함께 포함되어 있다.

우리는 지금까지 클래스와 객체를 사용해서 관련 데이터를 묶어서 사용하는 방법을 학습했다.
그럼 앞서 배운 것 처럼 단순히 객체를 사용하기만 하면 객체 지향 프로그래밍이라 할 수 있을까?
사실 **지금까지 우리가 작성한 모든 프로그램은 절차 지향 프로그램**이다.
그렇다면 무엇이 객체 지향 프로그래밍이란 말인가?

절차 지향에서 객체 지향으로 점진적으로 코드를 변경해보면서 객체 지향 프로그래밍을 이해해보자.

문제: 음악 플레이어 만들기

음악 플레이어를 만들어보자.

요구 사항:

1. 음악 플레이어를 켜고 끌 수 있어야 한다.
2. 음악 프레이어의 볼륨을 증가, 감소할 수 있어야 한다.
3. 음악 플레이어의 상태를 확인할 수 있어야 한다.

예시 출력:

```
음악 플레이어를 시작합니다
음악 플레이어 볼륨:1
음악 플레이어 볼륨:2
음악 플레이어 볼륨:1
음악 플레이어 상태 확인
음악 플레이어 ON, 볼륨:1
음악 플레이어를 종료합니다
```

절차 지향 음악 플레이어1

```java
package oop1;

public class MusicPlayerMain1 {

    public static void main(String[] args) {
        int volume = 0;
        boolean isOn = false;

        //음악 플레이어 켜기
        isOn = true;
        System.out.println("음악 플레이어를 시작합니다");

        //볼륨 증가
        volume++;
        System.out.println("음악 플레이어 볼륨:" + volume);

        //볼륨 증가
```

```
        volume++;
        System.out.println("음악 플레이어 볼륨:" + volume);

        //볼륨 감소
        volume--;
        System.out.println("음악 플레이어 볼륨:" + volume);

        //음악 플레이어 상태
        System.out.println("음악 플레이어 상태 확인");
        if (isOn) {
            System.out.println("음악 플레이어 ON, 볼륨:" + volume);
        } else {
            System.out.println("음악 플레이어 OFF");
        }

        //음악 플레이어 끄기
        isOn = false;
        System.out.println("음악 플레이어를 종료합니다");
    }

}
```

실행 결과

```
음악 플레이어를 시작합니다
음악 플레이어 볼륨:1
음악 플레이어 볼륨:2
음악 플레이어 볼륨:1
음악 플레이어 상태 확인
음악 플레이어 ON, 볼륨:1
음악 플레이어를 종료합니다
```

순서대로 프로그램이 작동하도록 단순하게 작성했다. 이 코드를 점진적으로 변경해보자.

절차 지향 프로그래밍2 - 데이터 묶음

앞서 작성한 코드에 클래스를 도입하자. `MusicPlayerData`라는 클래스를 만들고, 음악 플레이어에 사용되는 데이터들을 여기에 묶어서 멤버 변수로 사용하자.

절차 지향 음악 플레이어2 - 데이터 묶음

```
package oop1;

public class MusicPlayerData {
    int volume = 0;
    boolean isOn = false;
}
```

음악 플레이어에 사용되는 `volume`, `isOn` 속성을 `MusicPlayerData`의 멤버 변수에 포함했다.

```
package oop1;

/**
 * 음악 플레이어와 관련된 데이터 묶기
 */
public class MusicPlayerMain2 {

    public static void main(String[] args) {

        MusicPlayerData data = new MusicPlayerData();

        //음악 플레이어 켜기
        data.isOn = true;
        System.out.println("음악 플레이어를 시작합니다");

        //볼륨 증가
        data.volume++;
        System.out.println("음악 플레이어 볼륨:" + data.volume);

        //볼륨 증가
        data.volume++;
        System.out.println("음악 플레이어 볼륨:" + data.volume);

        //볼륨 감소
        data.volume--;
        System.out.println("음악 플레이어 볼륨:" + data.volume);

        //음악 플레이어 상태
        System.out.println("음악 플레이어 상태 확인");
        if (data.isOn) {
            System.out.println("음악 플레이어 ON, 볼륨:" + data.volume);
        } else {
```

```
                System.out.println("음악 플레이어 OFF");
            }

            //음악 플레이어 끄기
            data.isOn = false;
            System.out.println("음악 플레이어를 종료합니다");
        }

    }
```

음악 플레이어와 관련된 데이터는 `MusicPlayerData` 클래스에 존재한다. 이제 이 클래스를 사용하도록 기존 로직을 변경했다. 이후에 프로그램 로직이 더 복잡해져서 다양한 변수들이 추가되더라도 음악 플레이어와 관련된 변수들은 `MusicPlayerData data` 객체에 속해있으므로 쉽게 구분할 수 있다.

절차 지향 프로그래밍3 - 메서드 추출

코드를 보면 다음과 같이 중복되는 부분들이 있다.

```
//볼륨 증가
data.volume++;
System.out.println("음악 플레이어 볼륨:" + data.volume);

//볼륨 증가
data.volume++;
System.out.println("음악 플레이어 볼륨:" + data.volume);
```

그리고 각각의 기능들은 이후에 재사용 될 가능성이 높다.

- 음악 플레이어 켜기, 끄기
- 볼륨 증가, 감소
- 음악 플레이어 상태 출력

메서드를 사용해서 각각의 기능을 구분해보자.

절차 지향 음악 플레이어3 - 메서드 추출

```
package oop1;

/**
 * 메서드 추출
```

```
 */
public class MusicPlayerMain3 {

    public static void main(String[] args) {
        MusicPlayerData data = new MusicPlayerData();
        //음악 플레이어 켜기
        on(data);
        //볼륨 증가
        volumeUp(data);
        //볼륨 증가
        volumeUp(data);
        //볼륨 감소
        volumeDown(data);
        //음악 플레이어 상태
        showStatus(data);
        //음악 플레이어 끄기
        off(data);
    }

    static void on(MusicPlayerData data) {
        data.isOn = true;
        System.out.println("음악 플레이어를 시작합니다");
    }

    static void off(MusicPlayerData data) {
        data.isOn = false;
        System.out.println("음악 플레이어를 종료합니다");
    }

    static void volumeUp(MusicPlayerData data) {
        data.volume++;
        System.out.println("음악 플레이어 볼륨:" + data.volume);
    }

    static void volumeDown(MusicPlayerData data) {
        data.volume--;
        System.out.println("음악 플레이어 볼륨:" + data.volume);
    }

    static void showStatus(MusicPlayerData data) {
        System.out.println("음악 플레이어 상태 확인");
        if (data.isOn) {
            System.out.println("음악 플레이어 ON, 볼륨:" + data.volume);
```

```
            } else {
                System.out.println("음악 플레이어 OFF");
            }
        }

    }
```

각각의 기능을 메서드로 만든 덕분에 각각의 기능이 모듈화 되었다. 덕분에 다음과 같은 장점이 생겼다.

- **중복 제거:** 로직 중복이 제거되었다. 같은 로직이 필요하면 해당 메서드를 여러번 호출하면 된다.
- **변경 영향 범위:** 기능을 수정할 때 해당 메서드 내부만 변경하면 된다.
- **메서드 이름 추가:** 메서드 이름을 통해 코드를 더 쉽게 이해할 수 있다.

> **모듈화**: 쉽게 이야기해서 레고 블럭을 생각하면 된다. 필요한 블럭을 가져다 꼽아서 사용할 수 있다. 여기서는 음악 플레이어의 기능이 필요하면 해당 기능을 메서드 호출 만으로 손쉽게 사용할 수 있다. 이제 음악 플레이어와 관련된 메서드를 조립해서 프로그램을 작성할 수 있다.

절차 지향 프로그래밍의 한계

지금까지 클래스를 사용해서 관련된 데이터를 하나로 묶고, 또 메서드를 사용해서 각각의 기능을 모듈화했다. 덕분에 상당히 깔끔하고 읽기 좋고, 유지보수 하기 좋은 코드를 작성할 수 있었다. 하지만 여기서 더 개선할 수 는 없을까?

우리가 작성한 코드의 한계는 바로 데이터와 기능이 분리되어 있다는 점이다. 음악 플레이어의 데이터는 `MusicPlayerData`에 있는데, 그 데이터를 사용하는 기능은 `MusicPlayerMain3`에 있는 각각의 메서드에 분리되어 있다. 그래서 음악 플레이어와 관련된 데이터는 `MusicPlayerData`를 사용해야 하고, 음악 플레이어와 관련된 기능은 `MusicPlayerMain3`의 각 메서드를 사용해야 한다.

데이터와 그 데이터를 사용하는 기능은 매우 밀접하게 연관되어 있다. 각각의 메서드를 보면 대부분 `MusicPlayerData`의 데이터를 사용한다. 따라서 이후에 관련 데이터가 변경되면 `MusicPlayerMain3` 부분의 메서드들도 함께 변경해야 한다. 그리고 이렇게 데이터와 기능이 분리되어 있으면 유지보수 관점에서도 관리 포인트가 2곳으로 늘어난다.

객체 지향 프로그래밍이 나오기 전까지는 지금과 같이 데이터와 기능이 분리되어 있었다. 따라서 지금과 같은 코드가 최선이었다. 하지만 객체 지향 프로그래밍이 나오면서 데이터와 기능을 온전히 하나로 묶어서 사용할 수 있게 되었다.

데이터와 기능을 하나로 온전히 묶는다는 것이 어떤 의미인지 이해하기 위해 간단한 예제를 만들어보자.

클래스와 메서드

클래스는 데이터인 멤버 변수 뿐 아니라 기능 역할을 하는 메서드도 포함할 수 있다.

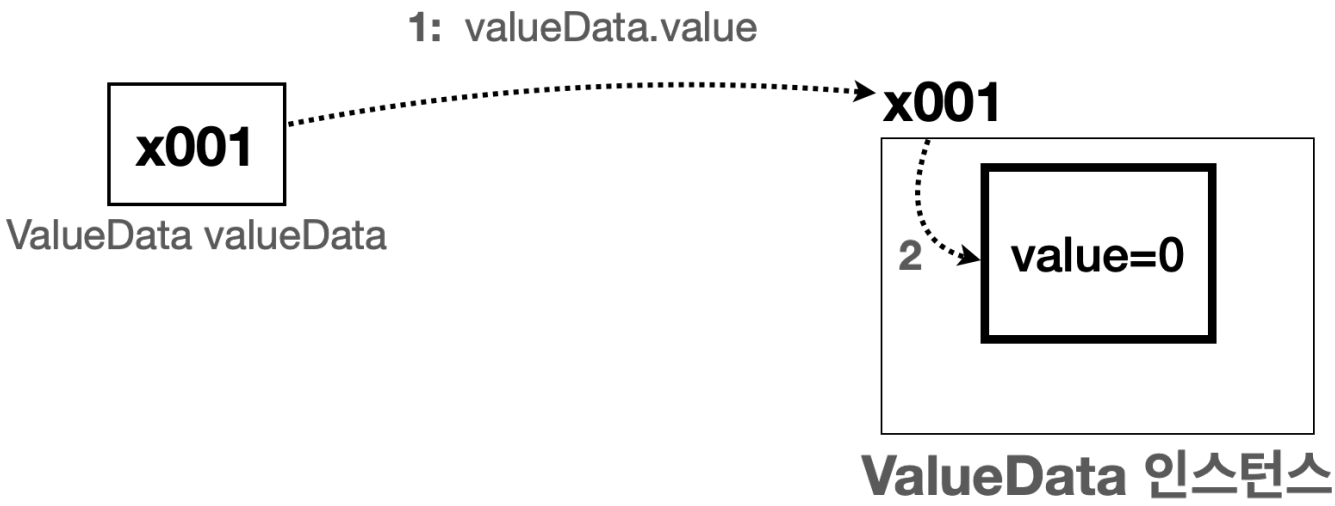
먼저 멤버 변수만 존재하는 클래스로 간단한 코드를 작성해보자.

```
package oop1;

public class ValueData {
    int value;
}
```

```
package oop1;

public class ValueDataMain {

    public static void main(String[] args) {
        ValueData valueData = new ValueData();
        add(valueData);
        add(valueData);
        add(valueData);
        System.out.println("최종 숫자=" + valueData.value);
    }

    static void add(ValueData valueData) {
        valueData.value++;
        System.out.println("숫자 증가 value=" + valueData.value);
    }
}
```

실행 결과

```
숫자 증가 value=1
숫자 증가 value=2
숫자 증가 value=3
최종 숫자=3
```



`ValueData`라는 인스턴스를 생성하고 외부에서 `ValueData.value`에 접근해 숫자를 하나씩 증가시키는 단순한 코드이다. 코드를 보면 데이터인 `value`와 `value`의 값을 증가시키는 기능인 `add()` 메서드가 서로 분리되어 있다.

자바 같은 객체 지향 언어는 클래스 내부에 속성(데이터)과 기능(메서드)을 함께 포함할 수 있다. 클래스 내부에 멤버 변수 뿐만 아니라 메서드도 함께 포함할 수 있다는 뜻이다.

이번에는 숫자를 증가시키는 기능도 클래스에 함께 포함해서 새로운 클래스를 정의해보자.

```
package oop1;

public class ValueObject {

    int value;

    void add() {
        value++;
        System.out.println("숫자 증가 value=" + value);
    }
}
```

이 클래스에는 데이터인 `value`와 해당 데이터를 사용하는 기능인 `add()` 메서드를 함께 정의했다.
이제 이 클래스가 어떻게 사용되는지 확인해보자.

참고: 여기서 만드는 add() 메서드에는 static 키워드를 사용하지 않는다.
메서드는 객체를 생성해야 호출할 수 있다. 그런데 `static`이 붙으면 객체를 생성하지 않고도 메서드를 호출할 수 있다.
`static`에 대한 자세한 내용은 뒤에서 설명한다.

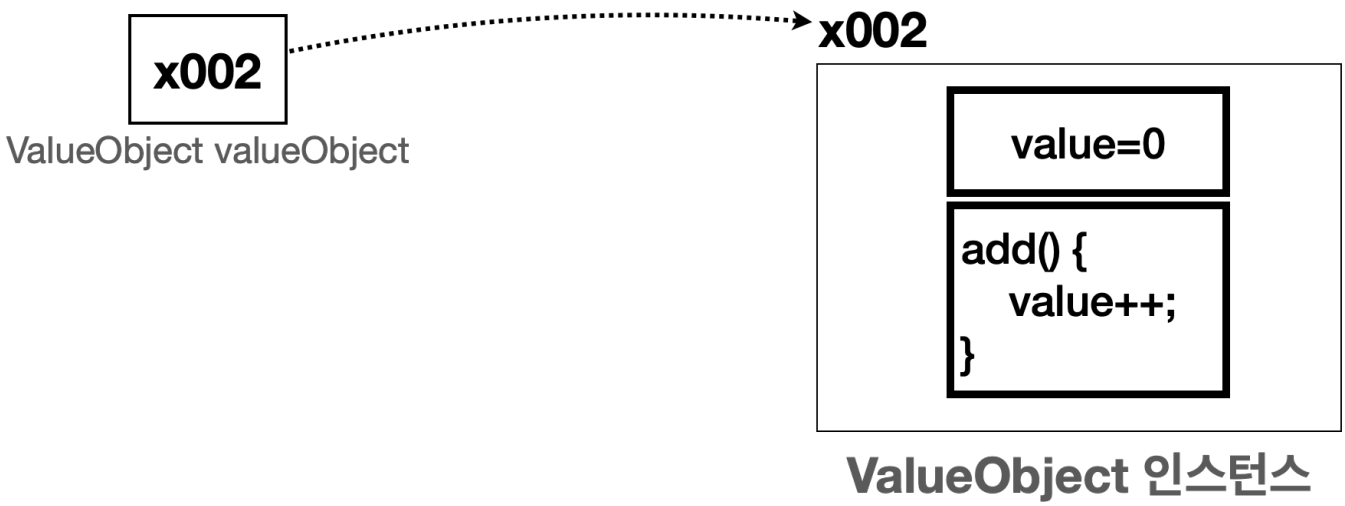
```java
package oop1;

public class ValueObjectMain {

    public static void main(String[] args) {
        ValueObject valueObject = new ValueObject();
        valueObject.add();
        valueObject.add();
        valueObject.add();
        System.out.println("최종 숫자=" + valueObject.value);
    }
}
```

실행 결과

```
숫자 증가 value=1
숫자 증가 value=2
숫자 증가 value=3
최종 숫자=3
```

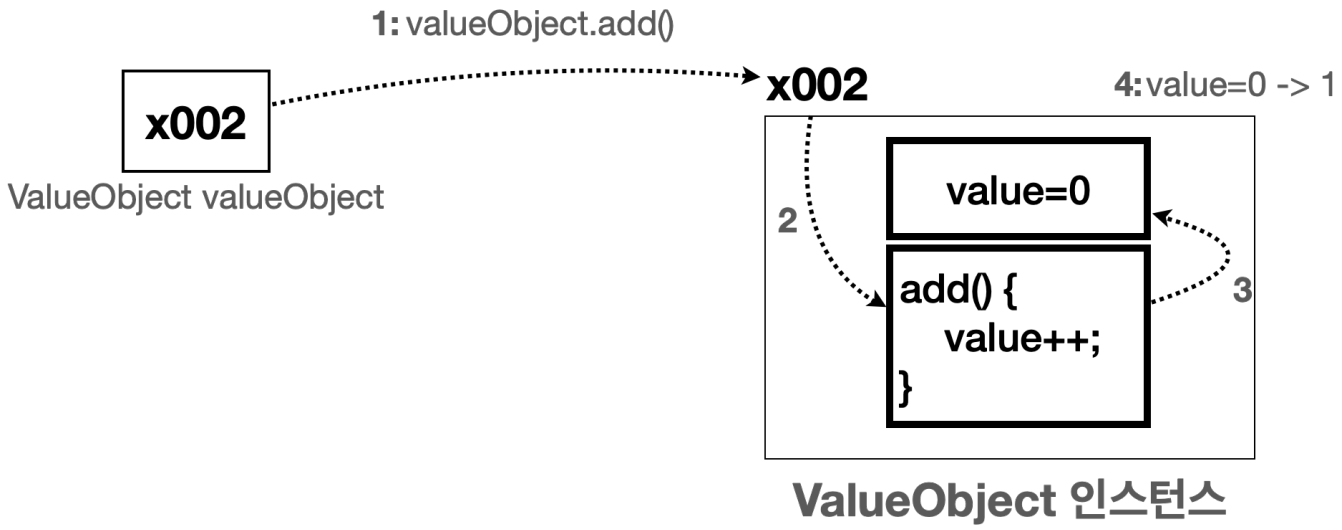
인스턴스 생성

```java
ValueObject valueObject = new ValueObject();
```



`valueObject`라는 객체를 생성했다. 이 객체는 멤버 변수 뿐만 아니라 내부에 기능을 수행하는 `add()` 메서드도 함께 존재한다.

인스턴스의 메서드 호출



인스턴스의 메서드를 호출하는 방법은 멤버 변수를 사용하는 방법과 동일하다. `.`(dot)을 찍어서 객체 접근한 다음에 원하는 메서드를 호출하면 된다.

```
valueObject.add(); //1
x002.add(); //2: x002 ValueObject 인스턴스에 있는 add() 메서드를 호출한다.
```

3: `add()` 메서드를 호출하면 메서드 내부에서 `value++`을 호출하게 된다. 이때 `value`에 접근해야 하는데, 기본으로 본인 인스턴스에 있는 멤버 변수에 접근한다. 본인 인스턴스가 `x002` 참조값을 사용하므로 자기 자신인 `x002.value`에 접근하게 된다.

4: `++` 연산으로 `value`의 값을 하나 증가시킨다.

정리

- 클래스는 속성(데이터, 멤버 변수)과 기능(메서드)을 정의할 수 있다.
- 객체는 자신의 메서드를 통해 자신의 멤버 변수에 접근할 수 있다.
  - 객체의 메서드 내부에서 접근하는 멤버 변수는 객체 자신의 멤버 변수이다.

객체 지향 프로그래밍

지금까지 개발한 음악 플레이어는 데이터와 기능이 분리되어 있었다. 이제 데이터와 기능을 하나로 묶어서 음악 플레이어라는 개념을 온전히 하나의 클래스에 담아보자. 프로그램을 작성하는 절차도 중요하지만 지금은 음악 플레이어라는 개념을 객체로 온전히 만드는 것이 더 중요하다. 음악 플레이어라는 객체를 지향해보자!

그러기 위해서는 프로그램의 실행 순서 보다는 음악 플레이어 클래스를 만드는 것 자체에 집중해야 한다. 음악 플레이어가 어떤 속성(데이터)을 가지고 어떤 기능(메서드)을 제공하는지 이 부분에 초점을 맞추어야 한다.

지금부터 우리는 음악 플레이어를 개발하는 개발자가 될 것이다. 이것을 어떻게 사용할지는 분리해서 생각하자. 쉽게 이야기해서 음악 플레이어를 만들어서 제공하는 개발자와 음악 플레이어를 사용하는 개발자가 분리되어 있다고 생각하면

된다.

음악 플레이어

- 속성: `volume`, `isOn`
- 기능: `on()`, `off()`, `volumeUp()`, `volumeDown()`, `showStatus()`

이것을 가지고 음악 플레이어를 만들어보자.

객체 지향 음악 플레이어

```
package oop1;

public class MusicPlayer {

    int volume = 0;
    boolean isOn = false;

    void on() {
        isOn = true;
        System.out.println("음악 플레이어를 시작합니다");
    }

    void off() {
        isOn = false;
        System.out.println("음악 플레이어를 종료합니다");
    }

    void volumeUp() {
        volume++;
        System.out.println("음악 플레이어 볼륨:" + volume);
    }

    void volumeDown() {
        volume--;
        System.out.println("음악 플레이어 볼륨:" + volume);
    }

    void showStatus() {
        System.out.println("음악 플레이어 상태 확인");
        if (isOn) {
            System.out.println("음악 플레이어 ON, 볼륨:" + volume);
        } else {
```

```
            System.out.println("음악 플레이어 OFF");
        }
    }

}
```

`MusicPlayer` 클래스에 음악 플레이어에 필요한 속성과 기능을 모두 정의했다. 이제 음악 플레이어가 필요한 곳에서 이 클래스만 있으면 온전한 음악 플레이어를 생성해서 사용할 수 있다. 음악 플레이어를 사용하는데 필요한 모든 속성과 기능이 하나의 클래스에 포함되어 있다!

```
package oop1;

/**
 * 객체 지향
 */
public class MusicPlayerMain4 {

    public static void main(String[] args) {
        MusicPlayer player = new MusicPlayer();
        //음악 플레이어 켜기
        player.on();
        //볼륨 증가
        player.volumeUp();
        //볼륨 증가
        player.volumeUp();
        //볼륨 감소
        player.volumeDown();
        //음악 플레이어 상태
        player.showStatus();
        //음악 플레이어 끄기
        player.off();
    }

}
```

실행 결과

```
음악 플레이어를 시작합니다
음악 플레이어 볼륨:1
음악 플레이어 볼륨:2
음악 플레이어 볼륨:1
음악 플레이어 상태 확인
```

```
음악 플레이어 ON, 볼륨:1
음악 플레이어를 종료합니다
```

`MusicPlayer`를 사용하는 코드를 보자. `MusicPlayer` 객체를 생성하고 필요한 기능(메서드)을 호출하기만 하면 된다. 필요한 모든 것은 `MusicPlayer` 안에 들어있다!

- `MusicPlayer`를 사용하는 입장에서는 `MusicPlayer`의 데이터인 `volume`, `isOn` 같은 데이터는 전혀 사용하지 않는다.
- `MusicPlayer`를 사용하는 입장에서는 이제 `MusicPlayer` 내부에 어떤 속성(데이터)이 있는지 전혀 몰라도 된다. `MusicPlayer`를 사용하는 입장에서는 단순하게 `MusicPlayer`가 제공하는 기능 중에 필요한 기능을 호출해서 사용하기만 하면 된다.

캡슐화

`MusicPlayer`를 보면 음악 플레이어를 구성하기 위한 속성과 기능이 마치 하나의 캡슐에 쌓여있는 것 같다. 이렇게 속성과 기능을 하나로 묶어서 필요한 기능을 메서드를 통해 외부에 제공하는 것을 캡슐화라 한다.

객체 지향 프로그래밍 덕분에 음악 플레이어 객체를 사용하는 입장에서 진짜 음악 플레이어를 만들고 사용하는 것 처럼 친숙하게 느껴진다. 그래서 코드가 더 읽기 쉬운 것은 물론이고, 속성과 기능이 한 곳에 있기 때문에 변경도 더 쉬워진다. 예를 들어서 `MusicPlayer` 내부 코드가 변하는 경우에 다른 코드는 변경하지 않아도 된다. `MusicPlayer`의 `volume`이라는 필드 이름이 다른 이름으로 변한다고 할 때 `MusicPlayer` 내부만 변경하면 된다. 또 음악 플레이어가 내부에서 출력하는 메시지를 변경할 때도 `MusicPlayer` 내부만 변경하면 된다. 이 경우 `MusicPlayer`를 사용하는 개발자는 코드를 전혀 변경하지 않아도 된다. 물론 외부에서 호출하는 `MusicPlayer`의 메서드 이름을 변경한다면 `MusicPlayer`를 사용하는 곳의 코드도 변경해야 한다.

문제와 풀이

문제1 - 절차 지향 직사각형 프로그램을 객체 지향으로 변경하기

다음은 직사각형의 넓이(Area), 둘레 길이(Perimeter), 정사각형 여부(square)를 구하는 프로그램이다.

- 절차 지향 프로그래밍 방식으로 되어 있는 코드를 객체 지향 프로그래밍 방식으로 변경해라.
- `Rectangle` 클래스를 만들어라.
- `RectangleOopMain`에 해당 클래스를 사용하는 `main()` 코드를 만들어라.

절차 지향 코드

```java
package oop.ex;

public class RectangleProceduralMain {
    public static void main(String[] args) {
        int width = 5;
        int height = 8;
        int area = calculateArea(width, height);
        System.out.println("넓이: " + area);

        int perimeter = calculatePerimeter(width, height);
        System.out.println("둘레 길이: " + perimeter);

        boolean square = isSquare(width, height);
        System.out.println("정사각형 여부: " + square);
    }

    static int calculateArea(int width, int height) {
        return width * height;
    }

    static int calculatePerimeter(int width, int height) {
        return 2 * (width + height);
    }

    static boolean isSquare(int width, int height) {
        return width == height;
    }

}
```

실행 결과

```
넓이: 40
둘레 길이: 26
정사각형 여부: false
```

정답

```java
package oop.ex;

public class Rectangle {
    int width;
    int height;
```

```
    int calculateArea() {
        return width * height;
    }

    int calculatePerimeter() {
        return 2 * (width + height);
    }

    boolean isSquare() {
        return this.width == this.height;
    }

}
```

```
package oop.ex;

public class RectangleOopMain {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        rectangle.width = 5;
        rectangle.height = 8;

        int area = rectangle.calculateArea();
        System.out.println("넓이: " + area);

        int perimeter = rectangle.calculatePerimeter();
        System.out.println("둘레 길이: " + perimeter);

        boolean square = rectangle.isSquare();
        System.out.println("정사각형 여부: " + square);
    }
}
```

문제2 - 객체 지향 계좌

은행 계좌를 객체로 설계해야 한다.

- `Account` 클래스를 만들어라.
  - `int balance` 잔액
  - `deposit(int amount)` : 입금 메서드
    - 입금시 잔액이 증가한다.

    - `withdraw(int amount)` : 출금 메서드
        - 출금시 잔액이 감소한다.
        - 만약 잔액이 부족하면 **잔액 부족**을 출력해야 한다.
- `AccountMain` 클래스를 만들고 `main()` 메서드를 통해 프로그램을 시작해라.
    - 계좌에 10000원을 입금해라.
    - 계좌에서 9000원을 출금해라.
    - 계좌에서 2000원을 출금 시도해라. ➜ `잔액 부족` 출력을 확인해라.
    - 잔고를 출력해라. `잔고: 1000`

실행 결과

```
잔액 부족
잔고: 1000
```

정답

```
package oop.ex;

class Account {
    int balance; // 잔액

    void deposit(int amount) {
        balance += amount;
    }

    void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("잔액 부족");
        }
    }
}
```

```
package oop.ex;

public class AccountMain {

    public static void main(String[] args) {
        Account account = new Account();
        account.deposit(10000);
        account.withdraw(9000);
```

```
        account.withdraw(2000);
        System.out.println("잔고: " + account.balance);
    }
}
```

정리

객체 지향 프로그래밍 vs 절차 지향 프로그래밍

객체 지향 프로그래밍과 절차 지향 프로그래밍은 서로 대치되는 개념이 아니다. 객체 지향이라도 프로그램의 작동 순서는 중요하다. 다만 어디에 더 초점을 맞추는가에 둘의 차이가 있다. 객체 지향의 경우 객체의 설계와 관계를 중시한다. 반면 절차 지향의 경우 데이터와 기능이 분리되어 있고, 프로그램이 어떻게 작동하는지 그 순서에 초점을 맞춘다.

절차 지향 프로그래밍

- 절차 지향 프로그래밍은 이름 그대로 절차를 지향한다. 쉽게 이야기해서 실행 순서를 중요하게 생각하는 방식이다.
- 절차 지향 프로그래밍은 프로그램의 흐름을 순차적으로 따르며 처리하는 방식이다. 즉, "어떻게"를 중심으로 프로그래밍 한다.

객체 지향 프로그래밍

- 객체 지향 프로그래밍은 이름 그대로 객체를 지향한다. 쉽게 이야기해서 객체를 중요하게 생각하는 방식이다.
- 객체 지향 프로그래밍은 실제 세계의 사물이나 사건을 객체로 보고, 이러한 객체들 간의 상호작용을 중심으로 프로그래밍하는 방식이다. 즉, "무엇을" 중심으로 프로그래밍 한다.

둘의 중요한 차이

- 절차 지향은 데이터와 해당 데이터에 대한 처리 방식이 분리되어 있다. 반면 객체 지향에서는 데이터와 그 데이터에 대한 행동(메서드)이 하나의 '객체' 안에 함께 포함되어 있다.

객체란?

세상의 모든 사물을 단순하게 추상화해보면 속성(데이터)과 기능 딱 2가지로 설명할 수 있다.

자동차

- 속성: 차량 색상, 현재 속도
- 기능: 엑셀, 브레이크, 문 열기, 문 닫기

동물

- 속성: 색상, 키, 온도

- 기능: 먹는다. 걷는다.

게임 케릭터

- 속성: 레벨, 경험치, 소유한 아이템들
- 기능: 이동, 공격, 아이템 획득

객체 지향 프로그래밍은 모든 사물을 속성과 기능을 가진 객체로 생각하는 것이다. 객체에는 속성과 기능만 존재한다.
이렇게 단순화하면 세상에 있는 객체들을 컴퓨터 프로그램으로 쉽게 설계할 수 있다.
이런 장점들 덕분에 지금은 객체 지향 프로그래밍이 가장 많이 사용된다.
참고로 실세계와 객체가 항상 1:1로 매칭되는 것은 아니다.

객체 지향의 특징은 속성과 기능을 하나로 묶는 것 뿐만 아니라 캡슐화, 상속, 다형성, 추상화, 메시지 전달 같은 다양한 특징들이 있다. 앞으로 이런 특징들을 하나씩 알아가보자.