

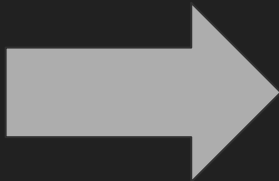
Clean Code

6. 객체와 자료 구조

자료 추상화

- 클래스에서 구현을 외부로 노출하지 마라.
 - 변수를 `private`으로 선언해도 `get`, `set`을 제공한다면 구현을 외부로 노출하는 것이다.

```
public class Point {  
    public double x;  
    public double y;  
}
```




```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

오른쪽도 결국 `get`, `set`을 제공하는건데 더 좋은건가..?


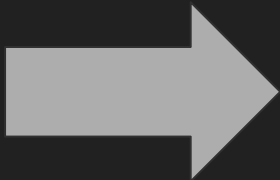
`get`, `set`이 없다면 데이터의 일부분만 수정하거나 가져오려면 어떻게 해야하는거지..?

자료 추상화

- 클래스에서 구현을 외부로 노출하지 마라.
 - 자료를 세세하게 공개하지 말고 추상적인 개념으로 표현하자.



```
public interface Vehicle {  
    double getFuelTankCapacityInGallons();  
    double getGallonsOfGasoline();  
}
```



```
public interface Vehicle {  
    double getPercentFuelRemaining();  
}
```

왼쪽보다 오른쪽이 추상화 단계가 더 높다.

추가적인 기능이 생길때마다 함수를 새로 만들어 사용하면 외부에 자료를 공개하지도 않고 코드의 반복성도 줄일 수 있을 것이다.

자료/객체 비대칭

- 자료구조는 자료를 그대로 공개한다.
 - 새 함수를 추가하기 용이하지만 새 자료형을 추가하기 힘들다.
- 객체는 추상화 뒤로 자료를 숨기고 자료를 다루는 함수만 공개한다.
 - 새 자료형을 추가하기 용이하지만 새 함수를 추가하기 힘들다.

새 함수 추가 :

< Geometry에 새로 추가하면 됨

모든 클래스에 직접 추가 과정 필요 >

새 자료형 추가 :

< 모든 함수에 새 자료형 추가 필요

클래스 하나를 새로 만들면 됨 >

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public double area(Object shape) {
        if (shape instanceof Square) {
            ...
        }
        else if (shape instanceof Circle) {
            ...
        }
    }
}
```

```
public class Square implements Shape {
    private Point topLeft;
    private double side;

    public double area() {
        return side*side
    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;

    public double area() {
        return 3.14 * radius * radius
    }
}
```

디미터 법칙 - 기차 충돌

- 모듈은 자신이 조작하는 객체의 내부 사정을 몰라야 된다는 법칙
 - 객체는 내부를 숨겨야 하며 자료구조는 내부를 노출한다.
 - 객체는 비공개 변수와 공개 함수를 포함하게, 자료구조는 무조건 함수 없이 공개 변수만 포함하게 만들면 쉬움

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

ctxt, Options, ScratchDir이 객체라면: 내부 구조를 숨겨야 하므로 디미터 법칙 위반

자료구조라면: 내부 구조를 노출하므로 디미터 법칙이 적용되지 않음 <= 따라서 자료구조다.

위의 경우는 get 함수를 사용했기 때문에 혼란을 일으킨다. 아래가 적절하다.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

디미터 법칙 - 구조체 감추기

```
ctxt.getAbsolutePathOfScratchDirectoryOption(); // 공개해야하는 메서드가 많아짐  
ctxt.getScratchDirectoryOption().getAbsolutePath(); // 앞의 함수가 자료구조를 반환해야함  
  
BufferedOutputStream bos = ctxt.createScratchFileStream(name)
```

만약 ctxt, Options, ScratchDir이 객체였다면:

원본 코드에서 임시 폴더의 절대 경로가 요구되는 이유는 임시 파일을 생성하기 위함이었음

= ctxt 객체에서 임시 파일을 생성하라고 시키면 적절함

디미터 법칙 - 잡종 구조

- 객체와 자료구조가 섞인 형태의 구조가 나오기도 함
 - 중요한 기능을 수행하는 함수와 공개 변수, get/set 함수가 존재
 - 공개 변수와 비공개 변수가 공존하기 때문에 혼란스럽다.
 - 새로운 함수, 새로운 자료구조 모두 추가하기 어렵다.

자료 전달 객체

- 자료 구조체의 전형적인 형태는 공개 변수만 있고 함수가 없는 클래스이며 이를 자료 전달 객체(DTO)라 한다.
 - 데이터베이스에 저장된 가공되지 않은 정보를 애플리케이션 코드에서 사용할 객체로 변환하는 구조체이다.

```
export class UserRequestDto extends PickType(User, [  
  'id',  
  'name',  
  'password',  
] as const) {}
```

데이터베이스에서 전달받은 User 객체에서 필요한 부분만 뽑아서 객체로 사용하는 DTO

자료 전달 객체 - 활성 레코드

- 공개 변수가 있거나 `get/set`, `save/find` 함수가 있는 DTO의 특수형태
- 데이터베이스 테이블이나 다른 저장소에서 에서 자료를 직접 변환한 결과로 나오는 DTO
- 자료구조로 취급해야하며 비즈니스 규칙이나 내부 자료를 숨기는 객체는 따로 생성해야한다.

예시) Rails의 ActiveRecord에는
자료형 외에도 ORM이 들어있다.

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```


결론

- 새로운 자료 타입을 추가하는 유연성이 필요하면 객체가 적합하다.
- 새로운 동작을 추가하는 유연성이 필요하면 자료구조가 적합하다.
- 뛰어난 개발자는 편향된 시선 없이 최적의 형태를 선택해야한다.