Chapter 6. 객체와 자료 구조(Objects and Data Structures)

아래 두 클래스를 비교해 보자

```
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);
}
```

- 전자는 자료구조를 명백하게 표현한다.
- 후자는 무슨 자료구조를 가지고 있는지 알 수가 없다.
- 후자는 메서드가 접근 정책을 강제한다.
    - 읽을때는 각 값을 개별적으로
    - 쓸때는 두 값을 한번에 설정해야 한다.

또 다른 예제

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

```
}
```

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

- 자료를 세세하게 공개하기 보다는 추상적인 개념으로 표현하는 편이 낫다.
- 변수 사이에 함수라는 계층을 넣는다고 구현이 감춰지지 않는다. *추상화해라*
    - 단순한 get/set메서드.

자료/객체 비대칭

객체

- 구체적인 데이터(자료)를 숨기고 데이터를 다루는 함수만을 제공

자료구조

- 구체적인 데이터를 그대로 공개하고 별다른 함수 제공하지 않음

두 개념은 정반대이다.

예제 1. 자료구조 지향형

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

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}
```

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

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws
NoSuchShapeException {
    if (shape instanceof Square) {
      Square s = (Square) shape;
      return s.side * s.side;
    } else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle) shape;
      return r.height * r.width;
    } else if (shape instanceof Circle) {
      Circle c = (Circle) shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

수정이 발생한다면..

1. Geometry클래스에 perimeter()함수를 추가할 때: 도형클래스에 영향 없음
2. 새로운 도형을 추가할 때: Geometry의 모든 함수를 변경해야

예제 2. 객체 지향형

```
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;
  public double area() { /* 인터페이스 구현, 다형성 */
```

```
      return PI * radius * radius;
    }
  }
```

수정이 발생한다면..

1. 새로운 도형을 추가할 때: 기존코드에 아무 영향이 없음.
2. 새로운 함수를 추가할 때: 기존 도형클래스를 모두 고쳐야함.

자료구조 지향과 객체 지향형에는 상호 보완적인 특질이 있다.

- 절차적 코드는 기존 자료 구조를 변경하지 않으면서 새 함수를 추가하기 쉽다.
- 객체 지향 코드는 기존 함수를 변경하지 않으면서 새 클래스를 추가하기 쉽다.
- *객체 지향 코드에서 어려운 변경은 절차적 코드에서 쉽다. 반대도 마찬가지*

성숙한 프로그래머는 모든 것이 객체라는 생각이 미신임을 잘 안다. 상황에 적절한 방식을 쓰자

디미터 법칙

클래스 C의 메소드 f는 다음의 메소드만 호출해야 한다.

- 클래스 C의 메서드
- f가 새성한 객체의 메서드
- f의 인수로 넘어온 객체의 메서드
- C 인스턴스 변수에 저장된 객체의 메서드

위에서 허용된 메소드가 반환하는 객체의 메소드는 호출하면 안된다.

가령 아래와 같은 코드는

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

디미터의 법칙을 위반하므로 아래와 같이 변경한다.

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir =
scratchDir.getAbsolutePath();
```

- 이 코드를 포함하는 함수는 세가지 객체의 세가지 함수의 정보를 알고 있다. 너무 많다.
- ctxt, opts, scratchDir은 정말 객체인가?

만약 객체가 아닌 자료구조였다면

아래와 같이 표현 가능하고,

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

만약 정말 객체였다면

코드를 뒤져보니,

```
String outFile = outputDir + "/" +
className.replace('.', '/') + ".class";
FileOutputStream fout = new
FileOutputStream(outFile);
BufferedOutputStream bos = new
BufferedOutputStream(fout);
```

절대 경로가 필요한 이유는 임시 파일생성을 위해서였다.

애초에 이런 구조를 이끌어내는 구조를 개선하자.

```
BufferedOutputStream bos =
ctxt.createScratchFileStream(classFileName);
```

잡종 구조

- 복잡한 함수가 포함된 자료구조
- get/set 메서드로 내부 자료구조를 그대로 노출한 클래스

양쪽 진영의 단점만 모이고 말았다.

자료전달 객체(DTO)

- Data Transfer Object
- 공개변수만 있고 함수가 없는 클래스
- DB나 소켓에서 데이터 파싱시에 사용

Active Record (pattern)

```
class Post < ApplicationRecord
  belongs_to :bulletin, optional: true
  has_many :comments, dependent: :destroy
  mount_uploader :picture, PictureUploader
end

class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true
end
```

- 보통 DB의 테이블 구조가 반영된 클래스
- 객체를 통해 테이블 데이터 조작
- *Active Record는 자료구조이니 비즈니스 로직을 넣지 말자.*

요약

- 자료구조와 객체를 확실히 구분해서 쓰자.
- 성숙한 프로그래머는 모든 것이 객체라는 생각이 미신임을 잘 안다. 상황에 적절한 방식을 쓰자.

더보기

*Builder 패턴은 Demeter의 법칙을 위반할까?

```
public User getUser() {
  return new
    User.UserBuilder("Jhon", "Doe")
    .age(30)
    .phone("1234567")
    .address("Fake address 1234")
    .build();
}
```

*user.filter().map().reduce();

*Class Decomposition

```
// Data Structure Oriented. Scala Ver.
trait Expr {
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}

class Number(n: Int) extends Expr {
  def numValue: Int = n
  def leftOp: Expr = throw new Error(”Number.leftOp”)
  def rightOp: Expr = throw new
Error(”Number.rightOp”)
}
```

```
  class Sum(e1: Expr, e2: Expr) extends Expr {
    def numValue: Int = throw new Error("Sum.numValue")
    def leftOp: Expr = e1
    def rightOp: Expr = e2
  }

  def eval(e: Expr): Int =
    if (e.isInstanceOf[Number])
      e.asInstanceOf[Number].numValue
    else if (e.isInstanceOf[Sum])
      eval(e.asInstanceOf[Sum].leftOp) +
      eval(e.asInstanceOf[Sum].rightOp)
    else throw new Error("Unknown expression " + e)
```

```
// Object-Oriented Decomposition
trait Expr {
  def eval: Int
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

- 만약 새로운 메서드가 추가된다면, 모든 클래스에 새 메스드가 추가되어야
- 첫번째 버전과 대칭.
- a * b + a * c -> a * (b + c)
    - 컨텍스트에 의존하는 구조는 한 객체 내에서 처리가 어렵다.

결국 첫번째 스타일과 두번재 스타일 모두 상황에 따라 필요하다.

```
// Scala Pattern Matching
trait Expr
```

```
case class Number (n: Int) extends Expr
case class Sum(el: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
  case Number(n) => n // if e instanceof Number
  case Sum(el, e2) => eval(el) + eval(e2)
}
```

- 자료구조 스타일을 훨씬 깔끔하게 해결 가능