

Clean Code

Chapter 6: Objects And Data Structures

Author: Hamed Damirchi

hameddamirchi32@gmail.com

github.com/hamed98

linkedin.com/in/hamed-damirchi-ba4085178/

مقدمه

یه دلیل واسه این که متغیر ها رو `private` کنیم وجود داره: این که هیچ جای دیگه ای بهش وابسته نباشه و فقط از داخل خود اون کلاس قابل تغییر باشه.

اما میخوایم بدونیم چرا برنامه نویس `getter` و `setter` تعریف میکنن؟ این طوری دوباره متغیر از بیرون قابل تغییر میشه که.

Data Abstraction

این دو قطعه کد رو ببینید، در اصل هر دو مختصات یه نقطه رو نگهداری میکنن.

Listing 6-1

Concrete Point

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

Listing 6-2

Abstract Point

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

ادامه – Data Abstraction

اما نکته زیبایی که تو دومی هست اینه که همیشه فهمید پیاده سازی، یه نقطه تو مختصات اقلیدسیه یا قطبی. ممکنه هیچ کدوم نباشن، اما interface به طرز صحیح یه ساختار داده رو فراهم میکنه.

تو دومی access policy داریم. به این معنا که میتونیم متغیر ها رو تکی بخونیم، اما موقع ست کردنش نمیتونیم مثلاً تنها X رو ست کنیم، باید طول و عرض رو با هم ست کنیم.

اما تو اولی فقط محدود به مختصات اقلیدسی هستیم، همچنین پیاده سازی ها مخفی نشدن. باید تا جایی که میشه پیاده سازی رو مخفی کنیم.

استفاده از abstraction این رو بهمون میده. در واقع باعث میشه کاربر بتونه ذات دیتا رو تغییر بده، بدون اینکه درگیر جزئیات پیاده سازی بشه.

یه مثال دیگه. این دو تا قطعه کد رو ببینید:

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

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

اولی مقدار سوخت رو به صورت **concrete** میده (یعنی دقیقا معلومه چیه! و از انتزاع استفاده نشده). اما دومی **abstract** تره. چون به صورت درصد میده. اولی صرفا **getter** عه.

در **abstract** کاربر هیچ ایده ای نداره که فرم دیتا چه شکلیه.

ترجیح با دومیه. چون جزئیات کمتری رو داره (تو پیاده سازیش محدود نیستی، دستت واسه تغییر خیلی بازه)

*نکته: کور کورانه اضافه کردن **getter** و **setter** واسه **Abstract** کردن اشتباهه!

پادتن‌ها و Object Data

object ها، دیتای خودشان رو پشت abstraction پنهان میکنن و یه سری تابع هایی که رو اون داده های کار میکنن رو public میکنن (در معرض دید میزارن).

اما data structure ها فقط دیتا رو نشون میدن و تابع ندارن.

جلوتر دو تا مثال میاریم که تفاوت object و data structure مشخص بشه.

```
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;
        }
    }
}
```

قطعه کد اول:
ادامه تو اسلاید بعدی

ادامه صفحه قبل

```
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();  
}  
}
```

این از قطعه کد اول. بریم بررسیش کنیم.

خب! این پیاده سازی ایده آل نیست. هر چند به ظاهر از OOP استفاده کرده، اما در اصل procedural عه. مثلا اگه بعدا بخوایم علاوه بر مساحت این شکل ها، محیط رو هم حساب کنیم چی؟ باید دوباره یه تابع بنویسیم و حالت های مختلف آبجکت shape رو بررسی کنیم و ...

همچنین اگر بعدا یه شکل دیگه اضافه بشه چی؟ باید دوباره این کلاس geometry رو تغییر بدیم.

همچنین این ریسک هست که پیاده سازی مثلا مساحت تو کلاس geometry برای آبجکت جدید فراموش بشه! و کامپایلر هیچ خطایی نمیده در اون صورت.

یه روش دیگه: این که توابع هر کلاس، تو خود اون کلاس باشن. بریم ببینیم.

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

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

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

    public double area() {
        return height * width;
    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

تفاوت procedural code و object oriented (مثال: قطعه کد اول و دوم):

procedural code اضافه کردن تابع جدید رو راحت تر میکنه چون نیازی به تغییر ساختار نیست. (مثلا تو مثال اول، اگه لازم باشه محیط هم اضافه بشه، احتیاجی به تغییر ساختار shape ها نیست و فقط این تابع تو geometry اضافه میشه). از طرف دیگه، در ساختار **object oriented**، اضافه کردن کلاس های جدید راحتتر چون احتیاجی به تغییر توابع و کلاس های قبلی نیست (چون توابعش تو خودش پیاده سازی شده).

متمم این تعریف:

اضافه کردن **data structure** جدید در **procedural code** ها سخته، چون باید همه ی توابع تغییر کنه. از طرفی اضافه کردن تابع جدید در **OO** سخته چون باید همه کلاس ها تغییر کنن (تابع جدید رو پیاده سازی کنن)

اگر برنامه شما جوریه که بیشتر لازم میشه که **data type** اضافه کنید تا تابع، روش **OO** مناسب تره، اما اگر جوریه که بیشتر لازم میشه تابع اضافه بشه به آبجکت های فعلی، بهتره از روش **procedural** استفاده بشه.

قانون دیمیتِر (Law of demeter)

میگه "فقط با دوستان خودت (نه دوست دوست) صحبت کن!"

به طور دقیق تر، هر تابع f درون کلاس C فقط میتونه توابع این ها رو صدا کنه:

- تابع های درون C

- تابعی های شیء ای که توسط f ساخته میشه

- شیء ای که به f پاس داده میشه

- شیء ای که متغیر درونی C عه. (instance variable)

همچنین نباید خروجی یک تابعی که صدا زده رو صدا کنه. مثلا این خط، این قانون رو نقض میکنه (از سورس آپاچی برداشته شده!)

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

قانون دیمیتِر – train wreck

به این نوع کد (اسلاید قبل) میگویند train wreck. این نوع کال کردن ها باعث بی نظمی بیشتر میشه. بهتره اگه مجبوریم اون کد رو بنویسیم، به این شکل بنویسیمش:

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

قانون دیمیتِر میگه یه ماژول نباید اطلاعات زیادی راجع به بقیه ماژول ها بدونه.

اما این جا این نقض شده. چون این ماژول میدونه که ctxt دارای options عه و اون هم دارای scratch directory عه و اون هم دارای یه Absolute path عه.

قانون دیمیتِر – train wreck

اگر `ctx, options, scratchDir` شیء باشن، نباید ساختار درونی خودشون رو فاش کنن و در غیر این صورت، نقض آشکار قانون دیمیتِر. اما اگه صرفاً `data structure` باشن، فاش کردن ساختار جزء طبیعتشونه و نقض قانون دیمیتِر محسوب نمیشه.

مسئله سر تابع های `accessor` عه. اگه کد به این شکل نوشته میشد مشکلی نبود:

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

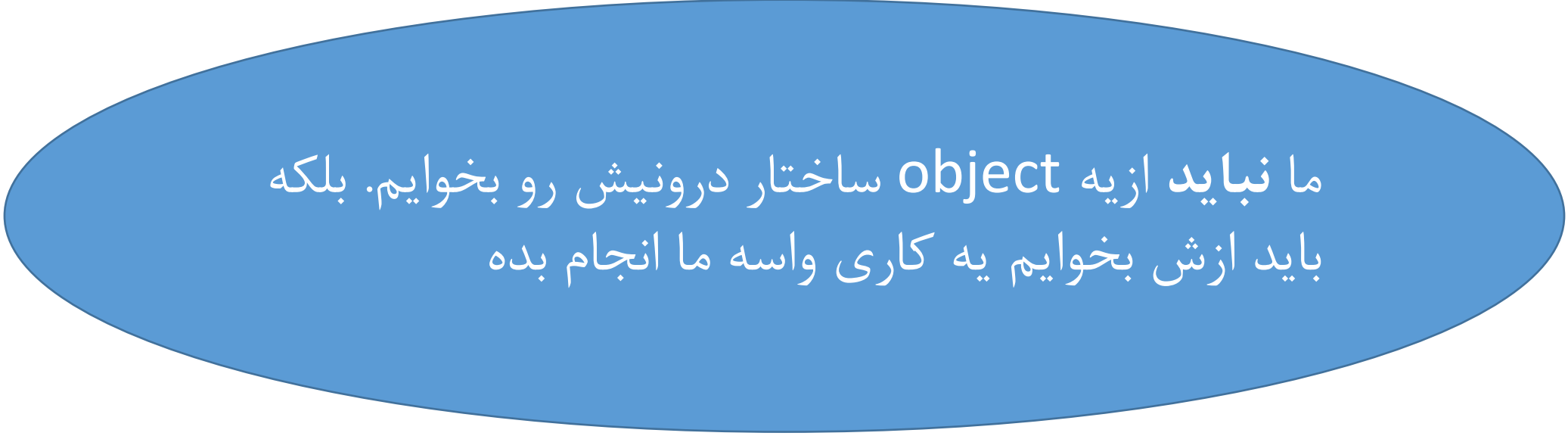
قانون دیمیتِر – Hybrids

این چیزی که تو اسلاید قبل گفته شد، گاهی باعث میشه بعضیا برن به سمت استفاده هیبریدی. یعنی نصف object و نصف data structure. به این صورت که تو کلاس یه سری تابع میزارن، متغیر های public و accessor های پابلیک استفاده میکنن، متغیر های private رو public میکنن.

در حالی که این کار یه اشتباه بزرگه و باعث میشه هم اضافه کردن تابع جدید و هم اضافه کردن ساختار جدید، سخت تر بشه. و این نوع کد بیانگر یه طراحی بی نظمه که نویسنده به فکر این نبوده که یه سری متغیر ها رو محفوظ نگهداره.

قانون دیمیتِر – پنهان کردن ساختار

خیلی مهم:



ما نباید از `object` ساختار درونیش رو بخوایم. بلکه باید ازش بخوایم یه کاری واسه ما انجام بده

در واقع ما از Data structure ها میتونیم ساختار درونیش رو بخوایم

قانون دیمیتِر – پنهان کردن ساختار

شاید واسه رفع مشکل یکی از این دو کار رو بخوایم انجام بدیم:

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

or

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

اما مشکلی که هست اینه که اولی منجر به "انفجار توابع"! میشه و دومی هم مستلزم اینه که `getScratchDirectoryOption()` ، `data structure` برگردونه، نه `object` (چون در غیر این صورت قانون نقض میشه دیگه!)

قانون دیمیتِر – پنهان کردن ساختار

این اسلاید مهم!

اگره `ctxt` آبجکته، نباید ازش ساختار درونیش رو بخوایم، بلکه باید ازش بخوایم یه کاری رو واسه ما انجام بده. این که ما `absolute path` رو میخوایم، از ساختار درونیش چیزی میخوایم.

سوال اینه: چرا اصلا باید `absolute path` رو بخوایم؟

تو همین ماژول (کد واقعی آپاچی!) یه کم که بریم پایین تر میبینیم از خروجی `outputDir` این جا استفاده شده:

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

قانون دیمیتِر – پنهان کردن ساختار

اولا این کد خیلی جالب نیست! چون سطوح مختلفی از انتراع در این کد دیده میشه و خوب نیست این (فصل تابع بحث شد)

دوما، ما outputDir رو میخوایم که scratchFile رو درست کنیم.

چطوره این کار رو بسپاریم به خود ctxt ؟

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

این کار منطقی تره و باعث میشه که ctxt ساختار درونی خودش رو مخفی نگه داره و تابع ما هم توابعی که یک سطح ازش دور هستن رو صدا نمیکنه و بنابراین قانون دیمیتِر نقض نمیشه.

Data Transfer Objects

DTO ها کلاس هایی هستند که دارای متغیر های **public** هستند و تابع هم ندارند.

DTO ها واسطه انتقال داده هستند و در انتقال داده بین برنامه و پایگاه داده ها یا مثلا **parse** کردن پیام های سوکت کاربرد های زیادی دارن.

یه فرم رایج، فرم **bean** هست. تو این فرم متغیر ها همه **private** هستن و همشون **getter** و **setter** دارن. به اعتقاد عمو باب، این فرم خیلی مزیت خاصی نداره.

Active Records

Active record ها نوع خاصی از DTO هستند. با این فرق که دارای یه سری توابع navigational هستند. مثل save و find.

این active record ها معمولا مستقیم map میشن روی دیتابیس.

متاسفانه گاهی بعضی دولوپر ها بخش هایی از منطق برنامه رو این جا مینویسن که کار اشتباهیه. این بخش فقط باید کار های مرتبط با پایگاه داده رو انجام بده

نتیجه گیری

آبجکت ها رفتار ها رو expose میکنند، این باعث میشه اضافه کردن یه ساختار جدید بدون تغییر رفتار بقیه، راحت تر بشه

از طرفی Data structure ها ساختار رو expose میکنند واین باعث میشه اضافه کردن تابع های جدید(رفتار های جدید) راحت تر بشه، اما اضافه کردن یه ساختار جدید سخت تر میشه.

در بعضی سیستم ها ما میخوایم انعطاف پذیری روی اضافه کردن تابع داشته باشیم، در بعضی ها میخوایم ور اضافه کردن ساختار انعطاف داشته باشیم.

توسعه دهندگان خوب این ها رو میشناسن و با توجه به نیاز، یکی از این دو روش رو انتخاب میکنند.