دانشکده ریاضی و علوم کامپیوتر

برنامه‌سازی پیشرفته و کارگاه

Clean Code – Part 2

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

محمدحسین هاشمی، آرمان حسینی

زمستان ۱۴۰۳

فهرست

مقدمه

الان که این داک دست شماست، شش هفته از ترم گذشته و چیزهای زیادی با هم یاد گرفتیم. حتماً یادتونه که هفته‌ی اول یکی از داک‌های شما درباره‌ی کلین کد بود. توی اون داکیومنت ما درباره‌ی این حرف زدیم که کدهای ما جدا از کارکرد درست، باید تمیز باشن. یعنی هر کس تو هر زمانی بتونه کد ما رو بفهمه، تغییر بده یا از اون نگهداری کنه. درباره‌ی لزوم نام‌گذاری درست حرف زدیم و گفتیم که اسامی اجزا باید شناسنامه‌ی اون جزء باشن و البته دقت این کار، بنا به اهمیت و طول عمر هر عنصر از کد ما متفاوته. درباره‌ی قواعد نام‌گذاری (Naming Convention) در جاوا حرف زدیم و توضیح دادیم که هر برنامه‌نویس جاوایی، باید از این Convention پیروی کنه. بعد از نام‌گذاری هم از چیزهای مهمی حرف زدیم؛ مثل اینکه کدی که کامنت زیاد و غیرضروری داره کد تمیزی نیست، اینکه از redundancy یا تکرار کد باید جلوگیری کرد و چیزهای دیگه. و در کل توی پارت اول به چندتا از مهم‌ترین و پایدارترین قاعده‌های کلین‌کد پرداختیم که **باید** رعایت کنید.

حالا که چند هفته گذشته، فکر می‌کنم همه‌ی شما برنامه‌نویس‌های خیلی بهتری شدید. این رو هم می‌دونم که اصولی که درباره‌ی کلین کد یاد گرفتید رو توی کدهاتون رعایت می‌کنید و خب تمرین همیشه بهترین چیزه. خلاصه احتمالاً الان خیلی‌هاتون لزوم کد تمیز و خوانا رو بیشتر از قبل دیده و درک کرده باشید. پس توی پارت دوم، می‌خوایم درباره‌ی چند تا تاپیک پیشرفته‌تر حرف بزنیم. چیزهایی که شاید به اندازه‌ی نکات قبلی ملموس نباشن و درک‌شون زمان زیادی ببره. اما مهمن، دید خوبی از برنامه‌نویسی درست و تمیز بهتون می‌دن و شاید یک محرک باشن برای اینکه توی مسیر «خوب کد زدن» پیش برید؛ مخصوصاً برای کسایی که قصد ورود به بازارکار و استخدام شدن دارن!

چه چیزهایی باید متغیر باشن؟

پارت دوم رو با یه نکته‌ی کوتاه و جزئی شروع می‌کنیم. تا حالا به این موضوع فکر کردید که چه چیزهایی باید متغیر بشن؟ خب این سوال رو می‌شه از نگاه‌های زیادی جواب داد. مثلا خیلی از برنامه‌نویس‌ها اگه یک معادله یا فرمول طولانی و پیچیده داشته باشن، اول بخش‌هایی از اون رو حساب می‌کنن و توی متغیر می‌ریزن و بعد اون متغیرها رو توی فرمول‌شون استفاده می‌کنن تا خوانایی کدشون از بین نره. مثال زیر رو ببینید:

```java
//without variable
if (b*b - 4*a*c > 0){
    //statement
}

// with variable
double quadraticFormula = b*b - 4*a*c;
if (quadraticFormula > 0){
    //statement
}
```

شاید در نهایت خیلی از دیدگاه‌ها درباره‌ی متغیر بودن یا نبودن چیزها، سلیقه‌ای و subjective باشن.
اما از زاویه‌ای می‌خوایم حرف بزنیم که مهمه و تا میزان زیادی دور از نگاه شخصی!

تعدد و تکرار یک مقدار

```java
public class Main {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);

        //Some Swing Stuff

        JPanel menuPanel = new JPanel();
        menuPanel.setSize(300, 400);

        //Some Swing Stuff

        JPanel buyPanel = new JPanel();
        menuPanel.setSize(300, 400);

        //Some Swing Stuff

        JPanel loginPanel = new JPanel();
        menuPanel.setSize(300, 400);

        //Some Swing Stuff

        JPanel signupPanel = new JPanel();
        menuPanel.setSize(300, 400);

    }
}
```

مثال بالا، یک مثال آشناست که وسط صدها خط کد گرافیکی‌تون، چندین و چند پنل تعریف می‌کنید و قراره سایز این پنل‌ها یکسان باشن. فکر می‌کنید تکرار ده‌باره یا بیست‌باره یا پنجاه‌باره‌ی این ابعاد کار خوبیه؟ فکر کنم باهام موافقید که نیست. پس به جاش چیکار کنیم؟ کد زیر رو ببینید:

```
public class Main {
    public static void main(String[] args) {
        int pagesHeight = 400;
        int pagesWidth = 300;

        JFrame frame = new JFrame();
        frame.setSize(pagesWidth, pagesHeight);

        //Some Swing Stuff

        JPanel menuPanel = new JPanel();
        menuPanel.setSize(pagesWidth, pagesHeight);

        //Some Swing Stuff

        JPanel buyPanel = new JPanel();
        menuPanel.setSize(pagesWidth, pagesHeight);

        //Some Swing Stuff

        JPanel loginPanel = new JPanel();
        menuPanel.setSize(pagesWidth, pagesHeight);

        //Some Swing Stuff

        JPanel signupPanel = new JPanel();
        menuPanel.setSize(pagesWidth, pagesHeight);

    }
}
```

البته می‌شد از کلاس Dimension هم استفاده کرد:

```
Dimension pagesDimension = new Dimension(300, 400);

JFrame frame = new JFrame();
frame.setSize(pagesDimension);
```

همونطور که تو این مثال دیدید، منظور از تعدد یک مقدار، اینه که تکرار زیاد اون مقدار، **به یک قصد یکسان** باشه. یعنی مثلاً ممکنه عدد 2 توی یک کد بارها به مقاصد مختلفی استفاده شه اما ما طبیعتاً منظورمون این نیست!

اگه یه مقدار (عدد، رشته، یا هر چیزی) رو فقط یک بار نیاز داشته باشید طبیعتاً جایی ذخیره‌ش نمی‌کنید. مثلا برای چاپ Hello World! هیچکس کد زیر رو نمی‌نویسه.

```
String string = "Hello world!";
System.out.println(string);
```

از طرفی تقریبا هر وقتی که شما باید از یک مقدار چند جا استفاده کنید، اون مقدار رو توی یک متغیر می‌ریزید. خیلی وقت‌ها اما این موضوع از دست ما در میره و یک مقدار رو بارها و بارها توی کد استفاده می‌کنیم بدون اینکه از یک متغیر استفاده کنیم. مثالش رو هم دیدید؛ خیلی از ما ممکنه توی این مثال به این نکته دقت نکنیم یا اهمیتی بهش ندیم.

استفاده از متغیر برای مقادیر پرتکرار چرا مهمه؟

اول از همه بحث خوانایی کد مطرحه. مقداری که پرتکرار باشه، احتمال زیاد توی کد ما مهمه. پس مهمه که کارکرد اون مقدار برای هر کسی قابل‌فهم باشه. اگه این مقدار توی یک متغیر باشه و اسم خوبی برای اون متغیر بذاریم، می‌تونیم کارایی اون رو مستقل از باقی اجزای کد بفهمیم.

نکته‌ی بعدی، درباره‌ی قابل‌تغییر بودن کده. اگه این مقدار نیاز به تغییر داشته باشه، توی یک پروژه‌ی بزرگ پیدا کردن و تغییر دفعاتی که این مقدار تکرار شده سخت و زمان‌بره و حتی ممکنه یک اشتباه کارکرد کد رو مختل کنه. اگه این مقدار توی یک متغیر ذخیره شده باشه، کافیه فقط جایی رو تغییر بدیم که این متغیر رو مقداردهی کردیم!

Nested Code

یکی از مهارت‌های مهم در راستای نوشتن یه کد خوانا و تغییرپذیر، Never Nester بودنه! به چه کدی می‌گیم Nested؟ تشخیصش سخت نیست؛ به کدی که بیش از حد بلوک تو در تو داشته باشه. مثلا کد زیر رو ببینید:

```
public static int add(int a, int b){
    return a + b;
}
```

عمق این متد یه بلوکه و این متد Nested نیست.

```
public static int primeSum(int start, int end){
    if (end >= start){
        int sum = 0;
        for (int i = start; i <= end; i++) {
            int numberOfDivisor = 0;
            for (int j = 2; j <= Math.sqrt(i); j++) {
                if (i % j == 0)
                    numberOfDivisor++;
            }
            if (numberOfDivisor == 0)
                sum += i;
```

```
        }
        return sum;
    }
    else
        return 0;
}
```

حالا به این کد نگاه بندازید. این کد یه متد ساده‌ست که مجموع اعداد اول توی یک بازه رو برمی‌گردونه. فکر می‌کنید می‌تونیم این تابع رو با بلوک‌های تودرتوی کمتر بنویسیم؟

شاید با خودتون بگید که این کد برای درست کارکردن به همین تعداد از بلوک‌های تودرتو نیاز داره، و مثلاً برای کم کردنش می‌تونیم if مربوط به بررسی بزرگتر بودن end رو پاک کنیم. طبیعتاً این درست نیست. همونطور که قبلاً هم گفتیم برای رعایت کلین کد نباید به کارایی کد ضربه زد.

راه‌های زیادی هست که بتونیم یک کد Nested رو جمع‌وجورتر یا denest کنیم. با هم دو رویکرد کلی برای انجام این کار رو بررسی می‌کنیم که معمولاً اوضاع کد رو بهتر می‌کنن!

دو روش برای کاهش Nest بودن کدها!

استخراج بخشی از کد (Extraction)

وقتی یک کدی Nested باشه، در واقع به این معنیه که چند فرآیند به صورت مرحله به مرحله دارن انجام می‌شن. مثلا یک if statement داریم که در صورت درست بودنش یک حلقه اجرا می‌شه برای انجام یک کاری، و بخشی از خود اون کار با یه حلقه‌ی دیگه انجام می‌شه و به همین ترتیب!

حالا اگه بتونیم بعضی از این فرآیندهای درونی رو از بین این بلوک‌ها استخراج کنیم، کدمون کمتر Nest خواهد بود! یعنی چی؟ کد زیر رو ببینید:

```
public static boolean isPrime(int x){
    int numberOfDivisor = 0;
    for (int j = 2; j <= Math.sqrt(x); j++) {
        if (x % j == 0)
            numberOfDivisor++;
    }
    if (numberOfDivisor == 0)
        return true;
    return false;
}

public static int primeSum(int start, int end){
    if (end >= start){
        int sum = 0;
        for (int i = start; i <= end; i++) {
            if (isPrime(i))
```

```
                sum += i;
        }
        return sum;
    }
    return 0;
}
```

این کد یک بازنویسی از مثال صفحه‌ی قبله. این بار اومدیم اول بودن عدد رو توی یک متد مجزا بررسی کردیم و اون رو توی متد اصلی‌مون (primeSum) کال می‌کنیم. یعنی در واقع فرآیند فهمیدن اول بودن یا نبودن عدد رو از متد استخراج کردیم و توی یه متد جدید نوشتیم. اگه این دو کد رو مقایسه کنید می‌بینید که الان کدمون یک بلوک تو در تو کمتر داره.

البته توی متدسازی و استخراج، زیاده‌روی هم خوب نیست! مثلا تو همین نمونه می‌شه باز هم متد ساخت تا جمع رو هم انجام بده. ولی خب اینطوری شلوغی کد از بین نمی‌ره و صرفا از شکلی به شکل دیگه تغییر می‌کنه، پس چندان کار خوبی نیست.

جابجایی شرط مثبت با منفی! (Inversion)

خیلی از مواقع دلیل وجود بلوک‌های تودرتو اینه که ما شروطی رو برای اجرا شدن بخش اصلی متد در نظر داریم. یعنی مثلا می‌گیم اگه شرط فلان برقرار بود، حالا تمام این کارها رو انجام بده! حتی ممکنه بیش از یک شرط وجود داشته باشه و خب به ازای هر شرط، طبیعتاً کد ما یه درجه بیشتر Nest می‌شه.

یکی از راه‌های مرسوم توی این متدها، اولویت دادن شرط منفی به شرط مثبته. یعنی به جای اینکه بگیم اگه شرط x برقرار بود این کار رو بکن، بگیم اگه شرط x برقرار نبود هیچ کاری نکن! یکبار دیگه به مثالی که قبل‌تر زدیم نگاه کنید. فکر می‌کنید این جابجایی تو اون کد امکان‌پذیره؟ اگه آره، به چه شکل؟ اول خودتون بهش فکر کنید و حتی دست به کد بشید، و بعد کد زیر رو ببینید.

```
public static int primeSum(int start, int end){
    if (end < start)
        return 0;
    int sum = 0;
    for (int i = start; i <= end; i++) {
        if (isPrime(i))
            sum += i;
    }
    return sum;
}
```

چیکار کردیم؟ به جای اینکه بگیم اگه end بزرگ‌ترمساوی بود کارت رو انجام بده، گفتیم که اگه end کوچیک‌تر بود هیچ کاری نکن (می‌دونید که دستور return هر وقت خونده بشه، اجرای متد به پایان

می‌رسه. یعنی اگه وارد بلوک if بشیم دیگه باقی متد خونده نمی‌شه و در واقع عملی انجام نمی‌گیره).
اگه شرط if درست نباشه، باقی خطوط متد اجرا می‌شن که بخش اصلی متد هستن.

چرا Nested Code خوب نیست؟

حالا که هر دو روش رو روی کد اولمون پیاده کردیم، بیاین یک بار دیگه قبل و بعد این فرآیند رو کنار هم ببینیم:

```
//Nested version
public static int primeSum(int start, int end){
    if (end >= start){
        int sum = 0;
        for (int i = start; i <= end; i++) {
            int numberOfDivisor = 0;
            for (int j = 2; j <= Math.sqrt(i); j++) {
                if (i % j == 0)
                    numberOfDivisor++;
            }
            if (numberOfDivisor == 0)
                sum += i;
        }
        return sum;
    }
    else
        return 0;
}
//After denesting
public static boolean isPrime(int x){
    int numberOfDivisor = 0;
    for (int j = 2; j <= Math.sqrt(x); j++) {
        if (x % j == 0)
            numberOfDivisor++;
    }
    if (numberOfDivisor == 0)
        return true;
    return false;
}
public static int primeSum(int start, int end){
    if (end < start)
        return 0;
    int sum = 0;
    for (int i = start; i <= end; i++) {
        if (isPrime(i))
            sum += i;
    }
    return sum;
}
```

می‌بینید که متدمون حالا خیلی کمتر از قبل بلوک‌های تودرتو داره. فکر می‌کنید کد دوم چه مزیتی نسبت به کد اول داره؟

درباره‌ی مزایای کدی که بلوک‌های تودرتوی کمتری داره، می‌شه زیاد صحبت کرد. چند نکته رو خیلی خلاصه مرور می‌کنم تا اگه برتری کد Denest شده به کد Nested رو حس نمی‌کنید، براتون بهتر جا بیفته:

کدی که Nested باشه خوانایی کمی داره. هر دو کد صفحه‌ی قبل رو ببینید. تو کد دوم الان خیلی سریع‌تر می‌فهمیم که در صورت کوچیک‌تر بودن end، اجرای متد باید متوقف باشه در حالی که توی کد اول تا زمانی که به else برسیم صرفا می‌دونیم که چه کارهایی در صورت بزرگ‌تر بودن end رخ می‌دن! از طرفی توی کد دوم باید لوپ دوم رو تحلیل کنیم تا متوجه شیم که دقیقا چه اعدادی در حال جمع شدن هستن و این موضوع توی مثال‌های پیچیده‌تر خوانش کد رو سخت می‌کنه. در حالی که توی کد دوم با دیدن کلمه‌ی isPrime، به‌سرعت می‌شه فهمید که اعداد مدنظر، اعداد اول هستن.

پیدا کردن و رفع باگ‌های کد Denested راحت‌تره. درک این موضوع سخت نیست؛ قبلاً هم توضیح دادیم که دیباگ کردن کدی که به هر طریقی خوانایی نداره، سخته. فارغ از این موضوع، برای پیدا کردن باگ کدی که لایه‌های تودرتوی زیادی داره باید عملکرد لایه‌ها رو وابسته به هم بررسی کنیم. در حالی که برای دیباگ کردن همیشه بررسی بخش‌های مختلف فرآیند به شکل مستقل کار بهتریه. از طرفی توی کد Nested وقتی باگ رو پیدا کنید و به‌فرض توی رفع اون دقیق نباشید، مشکل جدیدتون باز هم تمام لایه‌هارو درگیر می‌کنه. ولی اگر کدتون رو Nested نزده باشید، مشکل جدیدتون کاری به بخش‌های دیگه نداره.

تغییر دادن کدی که Nested نیست، ساده‌تره. یعنی چی؟ مثلا فرض کنید که بخوایم این کد رو تغییر بدیم و مجموع اعدادی رو توی بازه‌ی [start, end] بدست بیاریم که توی صد جمله‌ی اول دنباله‌ی فیبوناچی هستن اگه متد این موضوع رو بنویسید، تغییرات توی متد primeSum در حد کال کردن متد جدیده. اما اگه کد Nested باشه، باید تغییرات خیلی بیشتری توی متد بدین که ممکنه کارکرد باقی بخش‌های متد رو هم تحت‌الشعاع قرار بده.

متدی که Nested باشه، استفاده‌ی کمتری داره. قابل‌درکه. توی متدی که Nest شده باشه، کارکرد هر لایه از اون وابسته به لایه‌های دیگه‌ست. یعنی به‌نوعی کارهای مختلفی با هم ترکیب شدن و این متد جایی استفاده می‌شه که همه‌ی این کارها نیاز باشن. اما اگه همین کد رو Denest کنیم و به

متدهای کوچیک‌تر تقسیم کنیم، این کارهای جزئی هم می‌تونن مستقل از هم استفاده بشن. مثلا تو مثالی که دیدیم، متد primeSum فقط برای یک قصد خاص استفاده می‌شه و کاربرد متنوعی نداره. حالا اگه بخوایم در ادامه‌ی همین کد، یک متد دیگه بنویسیم که با اعداد اول کار داشته باشه، متد primeSum به‌دردمون نمی‌خوره اما توی نسخه‌ی دوم می‌تونیم از isPrime استفاده کنیم.

جمع‌بندی

بی‌راه نیست اگه بگیم که if ها و حلقه‌ها از عناصر پرتکرار کدهای ما هستن و خیلی پیش میاد که به تعداد زیاد درون همدیگه باشن. ما زیاد کد Nested می‌زنیم و با اینکه معمولاً اعصاب خودمون رو هم خرد می‌کنه، تلاشی برای بهبودش نمی‌کنیم!

حالا که با هم دو روش برای کم کردن تعداد این بلوک‌ها و تودرتویی کد یاد گرفتیم، از این به بعد بیشتر حواستون باشه و سعی کنید خوانایی کدهاتون رو با عمق کم بلوک‌ها بیشتر کنید. اگر هم می‌خواید مثال بزرگتری ببینید تا این مفهوم براتون جا بیفته؛ ویدیوی کوتاه زیر رو ببینید:

Why You Shouldn't Nest Your Code? – CodeAesthetic

کد مقیاس‌پذیر (Scalable Code)

تا الان صدبار توی داک‌های کلین‌کد از ما شنیدید که یک کد باید قابل‌تغییر باشه. و طبیعتاً اگه کد ما طوری باشه که این تغییرات راحت انجام بشن، اتفاق ارزشمندیه. اینجا می‌خوایم کمی بیشتر در این باره حرف بزنیم؛ تحت عنوان کدهای مقیاس‌پذیر.

مقیاس‌پذیری یعنی چی؟ یعنی یک کد رو بتونیم به‌راحتی بزرگ‌تر یا حتی کوچیک‌تر کنیم. به عبارتی اگه خواستیم شرایط یک عملیات (اعم از خروجی‌ها و ورودی‌ها و ...) رو تغییر بزرگی بدیم، نیاز به عوض کردن منطق و پیکره‌ی کدمون نداشته باشیم. یک مثال ساده ببینید:

```
//First approach
int a = 1, b = 2, c = 3;
System.out.println(a + " " + b + " " + c);

//Second approach
int start = 1, end = 3;
for (int i = start; i <= end; i++) {
    System.out.print(i + " ");
}
```

هر دو قطعه‌کدی که می‌بینید کار یکسانی رو انجام می‌دن؛ چاپ اعداد 1 تا 3 کنار هم. فکر می‌کنید کدوم بهتره؟ خب واقعیت اینه که کد دوم پیچیده‌تر به‌نظر می‌آد و شاید با خودتون بگید که چه نیازی به پیچوندن کد برای چاپ 3 عدد هست. اما در کمال ناباوری کد دوم کد باارزش‌تر و بهتره، چون مقیاس‌پذیره؛ یعنی مثلاً اگه بهتون بگن که حالا اعداد 18 تا 100000 رو کنار هم چاپ کنید، با کد دوم این کار به راحتی براتون ممکنه؛ در حالی که کد اول رو باید بریزید دور!

فکر کنم تقریباً دستتون اومد که کد مقیاس‌پذیر یا Scalable Code چیه. این کانسپت می‌تونه دید خیلی خوبی به شما درباره‌ی کد تمیز بده. شاید اگه توی این مفهوم دیپ بشید نگاهتون به حل مسائل ساده‌تر بشه و این مهارت، شاید بیشتر از هر چیزی به شمایی که الان یا در آینده توی بازار کار قرار می‌گیرید، کمک کنه.

ویدیوی یوتوب و تمرین

این توضیحات من رو یک مقدمه بدونید و برای تکمیل آموزش این بخش، ویدیوی بی‌نظیر زیر رو ببینید که با حل یک مثال Scalable Code رو توضیح می‌ده:

FizzBuzz: One Simple Interview Question – Tom Scott

حواستون باشه که این ویدیو مثل ویدیوهایی که قبلاً براتون گذاشتیم، صرفاً منبعی برای مطالعه‌ی بیشتر نیست و به طور مستقیم بخشی از آموزش این مبحث برای شماست. برای تمرین‌تون هم بهش نیاز دارید. **پس حتماً این ویدیو رو ببینید** و بعدش برید سراغ تمرین کلین کد 2 که توی کوئرا براتون باز شده.

چه چیزهایی یاد گرفتیم؟

- فهمیدیم که اگه یه مقدار مهم داریم که توی کد پرتکراره، بهتره که اون رو توی یه متغیر نگه داریم.
- درباره‌ی این حرف زدیم که Nested Code چیه و چرا کد خوبی نیست.
- دو روش یاد گرفتیم که معمولاً می‌تونیم با اون‌ها Nest بودن کدمون رو کمتر کنیم.
- با هم یاد گرفتیم که Scalable Code یعنی چی، چرا خوبه که کدمون Scalable باشه و با کمک ویدیوی یوتوب معرفی‌شده، اون رو بیشتر درک کردیم.