چکیده ای از هنر کد تمیز در زبان جاوا

بیایید از ابتدا شروع کنیم...

هنگامی که می خواهیم یک برنامه بنویسیم، احتمالا اولین سوال که در ذهنمان ایجاد می شود این است که : «چه *کار کنم که این برنامه کار کند؟*». بعد از آن که جواب این سوال را در مدت نسبتا کوتاهی پیدا کردیم، ویرایشگر خود را باز می کنیم و بدون توقف محسوسی به کد زدن می پردازیم. و درنهایت به برنامه ای که نوشته ایم نگاه می کنیم و از این که در نگاه اول بدون هیچ مشکلی دارد کار می کند به خودمان می بالیم!

اما این پایان ماجرا نیست. نوشتن برنامه ای که به درستی و بدون نقص کار می کند تنها بخشی از فرآیند نوشتن یک *کد* خوب است. کمتر پیش می آید که یک برنامه بزرگ بعد از متولد شدن ،به حال خود رها شود، چرا که حتی بهترین برنامه هایی که ساخته شده اند هم به *بازنگری، توسعه* و *رفع اشکالات (دیباگ کردن)* نیازمندند.

اینجاست که *تمیز کد زدن* بیش از هر چیز دیگری اهمیت پیدا می کند. هنری که به ما کمک می کند تا بتوانیم از کدی که نوشته ایم به خوبی مراقبت کنیم و آن را توسعه دهیم، و حتی زمینه همکاری دیگر برنامه نویسان را در پروژه مان فراهم کنیم.

چگونه تمیز کد بزنیم؟

یادگیری مهارت از دو بخش تشکیل شده است: دانش و کار. شما باید اصول، الگوها، تمرین‌ها و شیوه‌های اکتشافی که یک هنرمند میداند را یاد بگیرید و آن دانش را با انگشتان، چشم‌ها، و تمام وجودتان حس کنید و سخت تمرین کنید!
فرض کنید می خواهید یک شناگر حرفه ای شوید. برای این کار تعداد زیادی مقاله و مجلات ورزشی مختلف را مطالعه می کنید و چندین ویدئوی آموزشی از انواع تکنیک های شنای حرفه ای را مشاهده می کنید. با همه این ها احتمالا بعد از اینکه برای اولین بار به داخل آب می پرید شاید به زحمت بتوانید خود را در سطح آب نگه دارید (اگر غرق نشوید) !

نوشتن کد تمیز هم چندان به این ماجرا بی شباهت نیست. برای آن که در این هنر خبره شوید، به خواندن اکتفا نکنید و هر آن چه که فرا می گیرید را در برنامه نویسی خود به کار گیرید و آن ها را تمرین کنید تا رفته رفته به شگرد شما تبدیل شوند.

متغیرهای تمیز

متغیرها همه جا هستند و ما مدام در حال تعریف و نام گذاری متغیرها هستیم. از آن جایی که این کار را بسیار زیاد انجام می دهیم، بهتر است آن را با شیوه درست انجام دهیم. در ادامه به برخی نکات ساده و مفید برای خلق متغیرهای خوش نام می پردازیم.

اسامی پرمعنا

انتخاب یک اسم خوب برای یک متغیر کار زمان بر و گاهی حوصله سربر است. اما زمان بیشتری را برای شما در آینده ای نه چندان دور (که به اشکال زدایی و به روز کردن کد خود می پردازید) ذخیره خواهد کرد. بنابراین در انتخاب اسامی دقت و وسواس به خرج دهید

و اگر بعدا نام بهتری به ذهنتان رسید بلافاصله کد خود را به روز کنید (در آینده به طور مفصل در مورد این مبحث صحبت خواهیم کرد).

روی نامگذاری متغیرها فکر کنید. سعی کنید از اسامی یا گروههای اسمی استفاده کنید که بتوانند به این سوالات پا سخ دهند:

- چرا تعریف شده است؟
- چه کار می کند؟

به مثال زیر دقت کنید:

```
int d; // elapsed time in days
```

در این مثال، اسم d هیچ گونه اطلاعاتی در مورد اینکه این متغیر برای چه تعریف شده است و چه چیزی را نشان می دهد (زمان سپری شده در چند روز) به ما نمی دهد. به همین دلیل است که باید نام بهتری را برای آن انتخاب کنیم. مانند:

```
int elapsedTimeInDays;
```

متغیرها اسم اند!

سعی کنید در نامگذاری متغیرهایتان از *اسمها* استفاده کنید. استفاده از فعل یا صفت به تنهایی توصیه نمی شوند. همچنین اسامی مخفف یا خاص را در نامگذاری به کار نبرید.

```
int best_number;    //Correct
int best;           //Improper
int the_sample_result;  //Correct
int smplerslt;          //Improper
```

از نامهای طولانی نترسید

طولانی بودن نام متغیر بهتر از بی معنا بودن آن است. در یک برنامه شاید صدها یا هزاران متغیر وجود داشته باشند و اگر تمام آنها را با حروف
الفبا یا ترکیب آنها با اعداد تعریف کرده باشید، شاید هرگز نتوانید بفهمید که این متغیر چیست و چه اطلاعاتی را در خود ذخیره می-کند.

البته نباید در این موضوع زیاده روی کنید. طول نام متغیر به شرطی که معنادار باشد و اطلاعات کافی درباره خود با ما بدهد باید تا حد امکان کوتاه باشد.

```
int s = a + b; //Improper
int sumOfTwoVariables = a + b; //Correct
```

جدا کردن کلمات

در زبان های مختلف برنامه نویسی انواع مختلفی از قواعد برای جداکردن واژه های تشکیل دهنده نام متغیرها وجود دارد. (PascalCase، sepereted_by _underscore و ...)

در زبان جاوا برای تعریف متغیر ها از روش camelCase استفاده می کنیم. مثلا:

```
int client_message_code; //Incorrect
int ClientMessageCode; // Incorrect
int clientMessageCode; //    Correct
```

عبارات شرطی و حلقه ها

عبارت های شرطی از پراستفاده ترین جملات در مکالمات عادی و روزانه ما هستند. شاید به همین دلیل است که سر و کله آنها در اکثر زبان های برنامه نویسی پیدا می شود.

به علاوه، حلقه ها نیز از پررنگ ترین ویژگی های هر زبان محسوب می شوند تا امکان مدیریت عملیات پی درپی و متوالی را در اختیار توسعه دهنده ها قرار دهند .

البته این ابزارهای مفید میتوانند بعضا بسیار گیج کننده نیز باشند! زمانی که با تعداد بسیاری پرانتز، آکولاد، حلقه های تو در تو و ... دست و پنجه نرم می کنید.

در ادامه قصد داریم تا با اصولی آشنا شویم که می توانند ما را از این سردرگمی ها نجات دهند.

از شر آکولادها خلاص شو!

همان طور که می دانید، آکولاد ها بخش جدایی ناپذیر بسیاری از زبان های برنامه نویسی از جمله جاوا هستند. می توانید ردپای آن ها را همه جا پیدا کنید: شرط ها، حلقه ها، توابع، کلاس ها و ... .

در نتیجه گریز از آن ها به طور کامل تقریبا ناممکن است، اما راه هایی وجود دارند که می توانند به ما در استفاده بهینه تر از آن ها کمک کنند.

به قطعه کد زیر دقت کنید. فرض کنید سه متغیر تعریف شده اند و ما می خواهیم از آن ها استفاده کنیم تا وضعیت سلامتی یک بازیکن بازی های اکشن را به او گزارش دهیم.

```
```java
boolean isHurt;
String playerStatus;
int health = 100;
```
```

برای این کار از عبارات شرطی زیر استفاده می کنیم:

```
if (health < 5){
    isHurt = true;
}else{
    isHurt = false;
}

// Rest of your game's the code goes here

if(isHurt){
    playerStatus = "You're badly hurt!";
}else{
    playerStatus = "You're cool dude";
}
```

به نظر همه چیز مرتب است! اما با کمی دقت می توان مشاهده کرد که می توان به یک اقدام ساده این قطعه کد را ساده تر و خوانا تر کرد.

اگر عبارت شرطی دارید که تنها شامل یک *خط دستور* است، آکولاد ها را حذف کنید. بدین ترتیب قطعه کد بالا را به روز می کنیم:

```
if (health < 5)
    isHurt = true;
else
    isHurt = false;

//The rest of your game's code goes here

if(isHurt)
    playerStatus = "You're badly hurt!";
else
    playerStatus = "You're cool dude";
```

البته می توان شرط و دستور را در یک خط نیز قرار داد :

```
if (health < 5) isHurt = true;
else isHurt = false;
```

و یا حتی باز هم آن را ساده تر کرد:

```
isHurt = health < 5; // "health < 5" is a boolean statement
```

شرط های مرکب

از جمله مواردی که می تواند به شدت کدمان را ناخوانا کند، شرط های مرکب یا چندخطی ا ست. برای آنکه بتوانیم بهتر آنها را تحلیل و بررسی کنیم، بهتر است هر شرط را در یک خط بنویسیم (تمام عملگرها باید یا در ابتدا و یا در انتهای خطوط آورده شوند) :

```
/* valid style */
if (condition1 ||
(condition2 && condition3) ||
condition4 ||
(condition5 && (condition6 || condition7)))
{
    //Command 1
    //Command 2
}
```

توابع (متدها)

تابع راهکار هوشمندانه ای است که بیش از آنچه به نظر میاید می تواند در طراحی یک برنامه خوب به شما کمک کند. با استفاده از توابع متعدد در برنامه خود می توانید با کدهای طولانی چندصدخطی خداحافظی کرده و به طور چشمگیری برنامه خود را بهینه سازید. در ادامه با اصولی آشنا می شویم که به ما کمک می کنند تا بتوانیم بهتر از این ابزار استفاده کنیم و کیفیت کدهای خود را بالاتر ببریم.

متدهای خوش نام

فرض کنید در یک پروژه بزرگ بیش از هزاران متد مختلف با عملکردهای متنوع در یک کلاس تعریف شده باشند. شکی نیست که یافتن یک متد خاص در میان آنها یا فهمیدن منطق و علت تعریف آن به بزرگترین معمای زندگیتان تبدیل خواهد شد!
اینجا همان نقطه ای است که انتخاب اسامی مناسب و اصولی می تواند کار شما را بسیار آسان تر کند.
برای نامگذاری متدها:

- از افعال امری یا پرسشی استفاده کنید.
- تابع راتوصیف کنید، طوری که بتوان با خواندن آن متوجه شد که این تابع **چه کار** می کند و **چرا** تعریف شده است.

```
public class Customer {
    private String customerName;
    private Integer age;
    public String getCustomerName()
    //I'll call this method to "get customer's name."
    {
        return this.customerName;
    }
```

```
    public boolean isAdult()
    // I'll call this method to know if he/she "is adult".
    {
        return this.age >= 18;
     }
}
```

کوتاه و مختصر

یکی از دلایلی که از توابع استفاده میکنیم، تقسیم کد اصلی و منطق آن به قسمتهای کوچک تر و در عین حال کارا است تا از نامفهوم شدن آن جلوگیری کنیم. اما توابع بزرگ و طولانی به همان نسبت می توانند دردسرساز باشند. بنابراین بهتر است تا حد امکان سعی کنیم تا توابع را در تعداد خطوط کمتری پیاده سازی کنیم.
این که تعداد خطوط یک تابع استاندارد حداکثر چقدر باید باشد به عوامل متفاوتی بستگی دارد و میتواند بسته به پروژه، عملکرد تابع، زبان برنامه نویسی و ... این میزان متغیر باشد. با این وجود به طور متوسط می توان گفت که طول یک تابع تمیز حداکثر **20 تا 30 خط** است.

توا بع مسئولیت پذیر

> یک تابع باید تنها **یک** کار انجام دهد ، و آن را به **بهترین** شکل ممکن به سرانجام برساند.

شاید بتوان گفت این جمله مهمترین نکته ای است که در پیاده سازی توابع باید به آن توجه کنیم. به این مثال توجه کنید.
فرض کنید می خواهید برنامه ای بنویسید که تعداد اعداد اول سه رقمی را چاپ کند. برای این کار لازم است تا مراحل زیر را طی کنیم:

۱. در یک حلقه بر روی اعداد 100 تا 999 پیمایش کنیم.
۲. به ازای هر عدد، به کمک یک حلقه اول/مرکب بودن عدد را تشخیص داده و در صورت نیاز به شمارنده یک واحد اضافه کنیم.
۳. در نهایت باید تعداد را چاپ کنیم.

طبق توضیحات داده شده، به جای آنکه تمام این مراحل را در یک تابع پیاده سازی کنیم، باید برای هر کدام از مراحل 1 و 2، یک متد مجزا تعریف کرده و در نهایت جواب را در متد main چاپ کنیم:

```
public class ThreeDigitsPrimeNumbers {
    public boolean isPrime(int n){
        // Checks if the number is prime or not
     }

    public int getPrimesCount(){
    int counter = 0;
    for (int i = 100; i < 1000; ++i) {
```

```
          if (isPrime(i))
              counter ++;
          }
          return counter;
    }
    public static void main(String[] args) {
       System.out.println(new ThreeDigitsPrimeNumbers()
                                                  .getPrimesCount());
    }
}
```