

# کدنامه<sup>۱</sup>

برنامه‌سازی پیشرفته | بهار ۱۴۰۱ | دانشکده مهندسی کامپیوتر دانشگاه صنعتی شریف

THE  
ART  
OF  
CLEAN  
CODE



# هنر کد تمیز #۱

<< متین داغیانی

هنگامی که می‌خواهیم یک برنامه بنویسیم، احتمالا اولین سوالی که در ذهنمان ایجاد می‌شود، این است که «چه کار کنیم که این برنامه، کار کند؟». بعد از آن که جواب این سوال را در مدت نسبتا کوتاهی پیدا کردیم، ویرایشگر خود را باز می‌کنیم و بدون توقف محسوسی، به کد زدن می‌پردازیم؛ در نهایت، به برنامه‌ی نوشته شده نگاه می‌کنیم و از این که در نگاه اول بدون هیچ مشکلی در حال کار کردن است، به خودمان می‌بالیم!

اما این پایان ماجرا نیست. نوشتن برنامه‌ای که به درستی و بدون نقص کار می‌کند، تنها بخشی از فرآیند نوشتن یک کد خوب است. کمتر پیش می‌آید که یک برنامه‌ی بزرگ، بعد از متولد شدن، به حال خود رها شود؛ چرا که حتی بهترین برنامه‌هایی که ساخته شده‌اند هم به بازنگری، توسعه و رفع اشکالات (دیباگ شدن) نیازمندند. اینجاست که تمیز کد زدن، بیش از هر چیز دیگری، اهمیت پیدا می‌کند. تمیز کد زدن، هنری است که به ما کمک می‌کند تا بتوانیم از کدی که نوشته‌ایم به خوبی مراقبت کنیم و آن را توسعه دهیم، و حتی زمینه همکاری دیگر برنامه‌نویسان را در پروژه‌مان فراهم کنیم.

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 به‌روز کنید (در آینده به‌طور مفصل در مورد این مبحث صحبت خواهیم کرد).

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

< چرا تعریف شده است؟

< چه کار می‌کند؟

به مثال زیر دقت کنید:

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

در این مثال، اسم `d` هیچ گونه اطلاعاتی در مورد این که این متغیر برای چه تعریف شده است و نشان‌گر چه چیزی است (زمان سپری شده در چند روز) را به ما نمی‌دهد؛ به همین دلیل است که باید نام بهتری برای این انتخاب کنیم، مانند:

```
int elapsedTimeInDays;
```

## متغیرها، اسم‌اند!

سعی کنید در نام‌گذاری متغیرهایتان، از اسم‌ها استفاده کنید. استفاده از فعل یا صفت به تنهایی توصیه نمی‌شود. 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 بی‌معنا 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 می‌کند. البته نباید در این موضوع زیاده‌روی کنید؛ طول نام متغیر به شرطی که معنادار باشد و اطلاعات کافی درباره متغیر و کاربردش به ما بدهد، باید تا حد امکان کوتاه باشد.

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

## جدا کردن کلمات

در زبان‌های مختلف برنامه نویسی، از انواع مختلفی از قواعد برای جدا کردن واژه‌های تشکیل‌دهنده اسامی متغیرها استفاده می‌شود (به عنوان مثال، `separated_by_underscore`، `PascalCase` و...). در زبان جاوا برای تعریف متغیرها از روش `camelCase` (حرف نخست کلمه‌ی اول کوچک، حرف نخست سایر کلمات بزرگ) استفاده می‌کنیم، مثلا:

## چگونه تمیز کد بنویسیم؟

یادگیری هر مهارتی، 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 می‌خواهید یک شناگر حرفه‌ای شوید. برای این کار، تعداد زیادی مقاله و مجلات ورزشی مختلف را مطالعه می‌کنید و چندین ویدیوی آموزشی از انواع تکنیک‌های شنای حرفه‌ای را مشاهده می‌کنید؛ با همه‌ی این‌ها، احتمالا پس از اینکه برای اولین بار به داخل آب می‌پرید، شاید به زحمت بتوانید خود را در سطح آب نگه دارید (اگر غرق نشوید)! نوشتن کد تمیز هم چندان به این ماجرا بی‌شباهت نیست. برای آن که در این هنر خبره شوید، به خواندن اکتفا نکنید و هر آن چه که فرا می‌گیرید را در برنامه‌نویسی خود به کار بگیرید و آن‌ها را تمرین کنید تا رفته رفته به شگرد شما تبدیل شوند.

## متغیرهای تمیز

متغیرها، همه جا هستند و ما همیشه در حال تعریف و نام‌گذاری متغیرهای گوناگون هستیم. از آن جایی که این کار را بسیار زیاد انجام می‌دهیم، بهتر است آن را با شیوه درست انجام دهیم. در ادامه به برخی نکات ساده و مفید برای خلق متغیرهای خوش‌نام می‌پردازیم.

## اسامی پرمعنا

انتخاب یک اسم خوب برای یک متغیر، کاری زمان‌بر و گاهی حوصله‌سربر است، اما زمان بیشتری را برای شما در آینده‌ای نه چندان دور (که به اشکال‌زدایی و به‌روز کردن کد خود می‌پردازید)

```

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

```

به نظر همه چیز مرتب است! اما با کمی دقت، می‌توان این‌کد را با یک اقدام ساده کوتاه‌تر کرد، بی‌آن‌که به خوانایی آن صدمه‌ای وارد شود: «اگر عبارتی شرطی دارید که تنها شامل یک خط دستور است، آکولادها را حذف کرده و دستور را در همان خط شرط و با یک فاصله بعد از اتمام پرانتز شرط بنویسید، مشروط بر آن‌که خط از ۱۵۰ کاراکتر طولانی‌تر نشود». بدین ترتیب، قطعه‌کد بالا را به‌روز می‌کنیم:

```

if (health < 5) isHurt = true;
else isHurt = false;
if (isHurt) playerStatus = "You're hurt!";
else playerStatus = "You're cool dude";

```

همان‌گونه که می‌بینید، نسخه به‌روز شده، به زبان انگلیسی نزدیک‌تر است. البته، در این مثال خاص، می‌توان حتی باز هم کد را ساده‌تر کرد، ولی باید دقت کنیم که این ساده‌سازی، سبب کاهش خوانایی کد نشود. در این مثال به خصوص، ساده‌سازی به شکل زیر (جای‌گزینی شرط با عبارت boolean)، توصیه می‌شود:

```

isHurt = health < 5;
if (isHurt) playerStatus = "You're hurt!";
else playerStatus = "You're cool dude";

```

**نکته مهم:** ساده‌سازی شرط با استفاده از عملگر سه‌گانه شرطی ؟ و : را تنها زمانی به کار ببرید که صورت عبارت شرطی و مقدار آن، بسیار ساده، سراسر است و به سرعت و آسانی، قابل فهم باشد.

### توابع (متدها)

تابع راهکار هوشمندانه‌ای است که بیش از آن‌چه به نظر می‌آید می‌تواند در طراحی یک برنامه‌ی خوب به شما کمک کند. با استفاده از توابع متعدد در برنامه‌ی خود، می‌توانید با کدهای طولانی چندصدخطی خداحافظی کرده و به‌طور چشمگیری برنامه خود را بهینه کنید. در ادامه با اصولی آشنا می‌شویم که به ما کمک می‌کنند تا بتوانیم بهتر از این ابزار استفاده کنیم و کیفیت کدهای خود را بالاتر ببریم.

### متدهای خوش‌نام

فرض کنید در یک پروژه بزرگ (مانند پروژه AP!)، 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 نقطه‌ای است که انتخاب اسامی مناسب و اصولی می‌تواند کار شما را بسیار آسان‌تر کند. برای نام‌گذاری متدها:

✦ از افعال امری یا پرسشی استفاده کنید.

✦ تابع را توصیف کنید، طوری که بتوان با خواندن آن متوجه شد که این تابع چه کار می‌کند و چرا تعریف شده است.

```

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

```

### عبارات شرطی و حلقه‌ها

عبارات شرطی از پرستفاده‌ترین جملات در مکالمات عادی و روزانه ما هستند. شاید به همین دلیل است که سر و کله آن‌ها در اکثر زبان‌های برنامه‌نویسی پیدا می‌شود. به علاوه، حلقه‌ها نیز از پررنگ‌ترین ویژگی‌های هر زبان محسوب می‌شوند تا امکان مدیریت عملیات پی‌درپی و متوالی را در اختیار توسعه‌دهنده‌ها قرار دهند؛ البته، این ابزارهای مفید می‌توانند بعضاً بسیار گیج‌کننده نیز باشند، مانند زمانی که با تعداد بسیاری پرانتز، آکولاد، حلقه‌های تو در تو و... دست و پنجه نرم می‌کنید. در ادامه، قصد داریم تا با اصولی آشنا شویم که می‌توانند ما را از این سردرگمی‌ها نجات دهند.

### شرط‌های مرکب

از جمله مواردی که می‌تواند به شدت کدمان را ناخوانا کند، شرط‌های مرکب یا چندخطی است. 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 || con-
    dition7))) {

    // commands
}

```

### از شر آکولاد، خلاص شوید!

همان‌طور که می‌دانید، آکولادها بخش جدایی‌ناپذیر بسیاری از زبان‌های برنامه‌نویسی - از جمله جاوا - هستند، به طوری که می‌توانید ردپای آن‌ها را همه جا پیدا کنید، مانند شرط‌ها، حلقه‌ها، توابع و بسیاری از مکان‌های دیگر؛ در نتیجه، گریز از آن‌ها به‌طور کامل، تقریباً ناممکن است؛ اما راه‌هایی وجود دارند که می‌توانند به ما در استفاده‌ی بهینه‌تر از آن‌ها، کمک کنند. به قطعه‌کد زیر دقت کنید. فرض کنید سه متغیر زیر تعریف شده‌اند و ما می‌خواهیم از آن‌ها استفاده کنیم تا وضعیت سلامتی یک بازیکن را در یک بازی اکشن، به او گزارش دهیم:

```

boolean isHurt;
String playerStatus;
int health = 100;

```

برای این کار، از عبارات شرطی زیر استفاده می‌کنیم:

```

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

```

```
public static void main(String[] args) {
    System.out.println(getPrimesCount());
}
```

```
public String name() // wrong
public String getCustomerName() // Correct
```

یا به عنوان مثالی دیگر،

```
public boolean adult() { // Improper
    return age >= 18;
}
public boolean isAdult() { // Correct
    return age >= 18;
}
```

### فاصله‌گذاری اجتماعی!

فاصله‌گذاری صحیح، مفهومی بیش‌تر از رعایت چند قانون سخت‌گیرانه است. فاصله‌ها به کمک ما می‌آیند تا بتوانیم برنامه‌ای که نوشته‌ایم را بهتر درک کنیم و با کم‌ترین زحمت و در سریع‌ترین زمان، اطلاعات قابل قبولی از چیدمان اجزای مختلف برنامه به دست آوریم. فاصله‌گذاری مناسب و اصولی در یک کتاب، می‌تواند یک متن خسته‌کننده را به یک نوشته‌ی خوانا تبدیل کند، به طوری که بتوان با یک نگاه سریع، از کارکردش سر در آورد. به طور مشابه، برنامه‌ها نیاز دارند تا با کمی «تمیزکاری»، آن‌ها را سازمان‌مند کرده و از آشفتگی سابق نجات دهیم. در ادامه خواهیم دید که به چه روش‌هایی، کدمان را مرتب کنیم تا ساختار منظم‌تر و پایاتری پیدا کند.

### خطوط نه چندان بلند

طولانی بودن بیش از اندازه خطوط باعث می‌شود خوانایی کد به شدت کاهش یابد. طول هر خط از کد اجرایی برنامه، لازم است حداکثر به اندازه‌ی ۱۲۰ تا ۱۵۰ کاراکتر در نظر گرفته شود.

### کاراکتر Tab

در اکثر برنامه‌های ویرایش متن، امکان تغییر طول کاراکتر Tab بر حسب تعداد فاصله (space) وجود دارد. این مقدار معمولاً ۴ یا ۸ فاصله در نظر گرفته می‌شود. توجه کنید که در تمام کدتان از یک فاصله مشخص استفاده شده باشد تا تراز آن در تمامی خطوط، یکسان و مشخص باشد.

### تورفتگی‌ها (Indentation)

اکثر برنامه‌هایی که با آن‌ها سر و کار داریم، دارای یک ساختار سلسله‌مراتبی هستند: فایل‌ها، کلاس‌ها، متدهای درون کلاس‌ها، بلاک‌های درون متدها و در حالت کلی، بلاک‌های درون بلاک‌های دیگر. برای اینکه این محدوده‌ها را به بهترین شکل از یک‌دیگر تمیز دهیم، باید متناسب با هر کدام، از فاصله‌گذاری مناسب استفاده کنیم. عبارات موجود در سطح فایل، مانند اکثر تعاریف کلاس، به هیچ وجه فرورفته نیستند. متدهای درون یک کلاس در یک سطح فرورفتگی به سمت راست کلاس قرار دارند. بلاک‌های به کاررفته در پیاده‌سازی متد نیز در یک سطح فرورفتگی به سمت راست متد قرار می‌گیرند؛ و در حالت کلی، هر بلاک داخلی باید نسبت به بلاک بیرونی، یک سطح تورفتگی به سمت راست داشته باشد. به عنوان مثال:

```
public class Test {
    private static int number;
    public static void main() {
        innerBlock() {
            operations;
            innerInnerBlock() {
                operations;
            }
        }
    }
}
```

### کوتاه و مختصر

یکی از دلایلی که از توابع استفاده می‌کنیم، تقسیم کد اصلی و منطق آن به قسمت‌های کوچک‌تر و در عین حال کارا است تا از نامفهوم شدن آن جلوگیری کنیم، اما توابع بزرگ و طولانی به همان نسبت می‌توانند دردسرساز باشند؛ بنابراین بهتر است تا حد امکان سعی کنیم تا توابع را در تعداد خطوط کمتری پیاده‌سازی 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، یکی از اصول کدنویسی تمیز در جاوا، بیان می‌دارد که طول یک تابع تمیز در جاوا، بدون حساب سرآیند و آکولاد باز و بسته‌ی تابع، حداکثر ۲۰ تا ۳۰ خط است.

### توابع مسئولیت‌پذیر

«یک تابع، باید تنها یک کار انجام دهد و آن را به بهترین شکل ممکن به سرانجام برساند». شاید بتوان گفت این جمله مهم‌ترین نکته‌ای است که در پیاده‌سازی توابع باید به آن توجه کنیم. به این مثال توجه کنید:

فرض کنید می‌خواهید برنامه‌ای بنویسید که تعداد اعداد اول سه رقمی را چاپ کند. برای این کار لازم است تا مراحل زیر را طی کنیم:

- < در یک حلقه بر روی اعداد ۱۰۰ تا ۹۹۹ پیمایش کنیم.
- < به ازای هر عدد، به کمک یک حلقه اول و یا مرکب بودن عدد را تشخیص داده و در صورت نیاز، به شمارنده یک واحد اضافه کنیم.
- < در نهایت، تعداد را چاپ کنیم.

طبق توضیحات داده شده، به جای آن‌که تمام این مراحل را در یک تابع پیاده‌سازی کنیم، باید برای هر کدام از مراحل ۱ و ۲، یک متد مجزا تعریف کرده و در نهایت جواب را در متد main چاپ کنیم:

```
public class ThreeDigitsPrimeNumbers {
    public static boolean isPrime(int n){
        // Checking...
    }

    public static int getPrimesCount() {
        int counter = 0;
        for (int i = 100; i < 1000; ++i) {
            if (isPrime(i)) counter++;
        }
        return counter;
    }
}
```

## زیاده‌روی نکنیم

کامنت‌ها می‌توانند در بسیاری از اوقات به ما کمک کنند، اما استفاده بیش از اندازه از آن‌ها، نتیجه‌ی عکس دارد. نباید این نکته را فراموش کنیم که این کامنت‌ها هستند که در کنار کد ما قرار می‌گیرند و نه برعکس! هم‌چنین، در نظر داشته باشید که استفاده از کامنت برای شفاف‌سازی و ارائه‌ی توضیحات بیشتر، همواره آخرین راه حل می‌باشد؛ به عبارت دیگر، نباید برای یک کد کثیف کامنت گذاشت. بهتر است فکر دیگری به حال آن بکنید! در بسیاری از موقعیت‌ها، توجه به نام‌گذاری‌ها، فاصله‌ها، طراحی و پیاده‌سازی اجزا و مواردی از این دست، می‌تواند بسیار بیش‌تر از یک کامنت طولانی و مبهم، به تمیزی کد ما کمک کند.

## کامنت‌های TODO

گاهی اوقات، منطقی است که یادداشت‌های «TODO» را در قالب کامنت‌های TODO // انجام دهید. اصطلاح TODO به کارهایی گفته می‌شود که به نظر برنامه‌نویس، باید انجام شود، اما به دلایلی فعلاً نمی‌تواند انجام دهد. این TODO، ممکن است یک یادآوری برای حذف یک ویژگی منسوخ شده یا یک درخواست برای شخص دیگری جهت بررسی یک مشکل باشد، یا حتی ممکن است درخواستی از نویسندگان دیگر کد باشد که به نام بهتری فکر کنند یا یک یادآوری برای ایجاد تغییری وابسته به قسمتی دیگر از برنامه باشد که هنوز پیاده‌سازی نشده است. TODO هرچه باشد، بهانه‌ای برای قراردادن کد بد در برنامه نیست. امروزه، اکثر IDE های خوب (از جمله IntelliJ)، ویژگی‌های خاصی را برای یافتن همه‌ی کامنت‌های TODO ارائه می‌دهند، بنابراین احتمالاً این کامنت‌ها به آسانی گم نمی‌شوند. اگر هم نمی‌خواهید کد شما با TODO پر شود، مرتباً از آن‌ها را اسکن کرده و مواردی را که می‌توانید، حذف کنید. توجه شود که در کد نهایی تحویل داده شده، TODO و یا تکه‌کد کامنت شده، موجود نباشد.

```
public void verifyUser (User user) {  
    // TODO: Add code to verify the user  
}
```

## IntelliJ shortcuts

میان‌بر Ctrl+Alt+L (در اوبونتو، Ctrl+Alt+Win+L و در مک، Option+Command+L): این میان‌بر، بسیاری از اصول کلین‌کد را مستقیماً در کد شما اعمال می‌کند؛ گرچه تمامی آن‌ها را به درستی پوشش نمی‌دهد و لازم است پس از اعمال آن، مجدداً کد خود را به طور کامل بررسی کنید.

میان‌بر Shift+F6 (در مک‌های مجهز به تاج‌بار، Shift+Fn+F6): با فراخوانی این میان‌بر، می‌توانید کلیدواژه‌ای که انتخاب کرده‌اید (اسم متد و یا متغیر) را با یک نام بهتر، به راحتی در کل کد تعویض کنید، بدون این‌که نگران مشکلات احتمالی Replace All باشید.

میان‌بر Ctrl+Alt+M (در مک، Option+Command+M): این میان‌بر، می‌تواند قطعه کدی که انتخاب کرده‌اید را به یک متد دیگر انتقال دهد تا متد اول، طولانی، شلوغ و «کثیف» نشود.

```
}  
public static void anotherMethod(){  
    operations;  
}  
}
```

## کاراکترهای متشخص

هر برنامه‌ای، پر از دستورات و عملیات ریاضی و کاراکترهای غیر حرفی است که موارد زیر به افزایش خوانایی آن‌ها کمک می‌کند:

همواره قبل از باز کردن پرانتز یک فاصله قرار دهید (میان پرانتزها و یا عبارت داخلی آن، بهتر است هیچ فاصله‌ای وجود نداشته باشد):

```
if(a==2); // Improper  
if (a == 2); // Correct
```

بهتر است همواره قبل و بعد از عملگرهای ریاضی یا دودویی از یک فاصله استفاده کنیم (البته به جز عملگرهای ++ یا ==):

```
a = b + c; // Correct  
c = a > b; // Correct  
b = a && c; // Correct  
a=b+c; // Improper  
a ++; // Improper
```

بعد از کاراکتر نقطه‌ویرگول (سمی‌کالن) در حلقه‌ها، یک فاصله می‌گذاریم، اما قبل از آن معمولاً هیچ فاصله‌ای نیست:

```
for(int i = 0 ; i < 10;i++); // Improper  
for (int i = 0; i < 10; i++); // Correct
```

## کامنت‌گذاری

چرا کامنت؟

کامنت‌ها، راهنماهایی هستند که کمک می‌کنند تا وقتی با یک برنامه بیگانه روبرو می‌شویم، بدون درگیر شدن بیش از اندازه با جزئیات و نحوه پیاده‌سازی، با کلیتی از عملکرد برنامه، نحوه استفاده از توابع، چیدمان اصلی فایل‌ها و مواردی از این دست، آشنا شویم. هم‌چنین کامنت‌ها نقشی حیاتی در کتابخانه‌های مختلف بازی می‌کنند، چرا که غالباً می‌توانند توضیحات جامعی از نحوه استفاده از امکانات و توابع آن کتابخانه را در اختیار کاربر قرار دهند.

## یک کامنت خوب

کامنت‌ها، باید به ما بگویند که «چه اتفاقی دارد می‌افتد»، «چگونه انجام می‌شود»، «هر کدام از پارامترها یا آرگومان‌های مورد استفاده، چه معنایی دارند» و «احیاناً چه محدودیت‌ها یا ایراداتی ممکن است وجود داشته باشند». این موارد، به ویژه در خصوص متدها، بسیار مهم و کاربردی هستند. توجه داشته باشید که کامنت‌های طولانی (بیش از ۸۰ الی ۱۲۰ کاراکتر) را در چند خط بنویسید و از نوشتن همه‌ی آن‌ها در یک خط، خودداری کنید.



کوتاه و مختصر بودن در تمامی قسمت‌های کد مهم و پر اهمیت است، اما در main بسیار مهم‌تر! در برنامه‌نویسی، تابع main برنامه معمولاً اولین قسمتی است که هنگام بررسی کد به آن توجه می‌شود، یک main شلوغ که تمامی قسمت‌های برنامه در آن نوشته شده باشد یکی از کثیف‌ترین کارهایی است که یک برنامه‌نویس می‌تواند انجام دهد. برای برنامه‌نویسی حرفه‌ای، main شما باید کوتاه و پر مفهوم باشد و تقریباً به صورت کامل از فراخوانی توابعی که در زیر برنامه‌های مختلف کدتان نوشته‌اید تشکیل شده باشد.

**ده محک کدنویسی تمیز!**

هرگاه خواستید کد خود را از منظر «تمیزی»، ارزیابی کنید، می‌توانید فاکتورهای اصلی ذکر شده در زیر را در کد خود بررسی کنید:

- ✦ از نام‌های تک‌حرفی، مخفف یا نامفهوم و بی‌معنی برای متغیرها و یا متدها استفاده نشده باشد.
- ✦ اسامی متغیرها و توابع، camelCase باشند.
- ✦ طول هر تابع، حداکثر ۲۰ تا ۳۰ خط باشد.
- ✦ هر خط برنامه، حداکثر به طول ۱۲۰ تا ۱۵۰ کاراکتر باشد؛ شرط‌های طولانی به چند خط شکسته شده باشند (سرآیند متد، مشکلی ندارد).
- ✦ برای نام متدها (به جز main) از افعال امری و یا پرسشی استفاده شده باشد.
- ✦ از Ctrl+Alt+L در IntelliJ استفاده شده باشد، به گونه‌ای که کد پس از اعمال این میان‌بر، دیگر تغییری نکند.
- ✦ فاصله‌گذاری طرفین اپراتورها رعایت شده باشد.
- ✦ تورفتگی‌های کد، مناسب و درست باشند.
- ✦ هیچ کامنت TODO ای در کد نباشد و همگی قبل از ارسال پاک شده باشند.
- ✦ هیچ تکه‌کدی در برنامه، کامنت نشده باشد.

“

It is not  
enough for code  
to work

-Robert C.Martin

”

## کدنامه

شماره: ۱

تاریخ انتشار: ۲۰ اسفند ۱۴۰۰

نویسندگان: متین داغیانی

طراحی: سجاد سلطانیان و حسام الدین سلیمانی

دستیاران آموزشی دروس مبانی برنامه‌سازی و برنامه‌سازی پیشرفته، طی نیمسال‌های گذشته، مطالب متعدد درسی را در قالب فایل‌های گوناگون منتشر می‌کردند. از نیم‌سال گذشته، برآن شدیم تا با تشکیل «کدنامه»، تمامی مطالب آموزشی را در این قالب تدوین کنیم تا هم به سرعت بتوان آنها را مطالعه کرد و هم بتوانیم به این بهانه، محتوای کاربردی‌تری را در اختیار دانشجویان قرار دهیم. توجه شود که «کدنامه»، به هیچ عنوان، یک نشریه نیست و زمان عرضه مشخصی نیز ندارد، بلکه تنها قالبی جدید برای عرضه همان مطالب و محتوای آموزشی است و در آن تنها به مباحث درسی مخصوص درس برنامه‌سازی پیشرفته نیم‌سال جاری پرداخته می‌شود.

مطالعه مطالب «کدنامه»، برای انجام بهتر پروژه و تمرین‌های درس برنامه‌سازی پیشرفته، اکیدا توصیه می‌شود.