به نام خداوند بخشنده‌ی مهربان

شماره‌ی ۴ ۲۸ اسفند ۹۹

کد نامه

ویژه‌ی دانش‌جویان برنامه‌سازی پیشرفته نیم‌سال دوم ۱۴۰۰-۱۳۹۹ دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف

در این شماره از کدنامه، می‌خوانید:

هنر کد تمیز در جاوا (قسمت سوم)

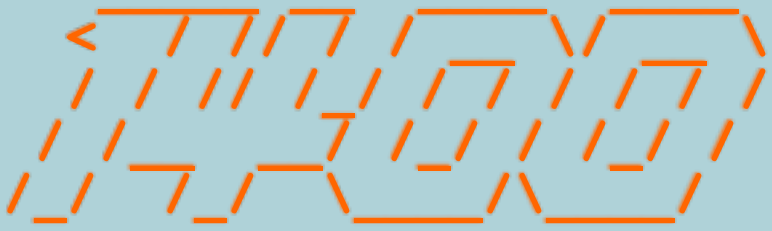
آیا می‌دانستید؟

یکی از مهم‌ترین عوامل تولید باگ و اشکال در توسعه‌ی نرم‌افزار، وجود **کدهای تکراری و یا بسیار مشابه** است؟

هنر کد تمیز در جاوا (قسمت سوم)

تلاش کنید همواره مرتب، منظم و تمیز کد بنویسید

در شماره‌های اول و دوم کدنامه، با برخی از اصول مقدماتی کدنویسی تمیز آشنا شده‌اید؛ حال، با توجه به آموختن شی‌گرایی، لازم است بدانید چه‌طور می‌توان اصول کدنویسی تمیز را در پروژه‌هایی شامل چندین کلاس، فایل و متد، اجرایی کرد و از برخی از عوامل ایجاد باگ، هم‌چون کدهای تکراری، جلوگیری به عمل آورد. رعایت این قوانین، هم‌چون قوانین پیشین، همواره مهم و ضروری است. با کدنامه همراه باشید.



سال نو مبارک! 🎉🎉

هنر کد تمیز در جاوا (قسمت سوم)

متین داغیانی

پروژه‌های تمیز در جاوا

تا این‌جا، با مفاهیم اصلی برنامه نویسی شی‌گرا در زبان جاوا آشنا شده‌ایم. در این قسمت، قصد داریم تا با مرور این مفاهیم و البته معرفی برخی نکات جدید، ارتباطات میان آن‌ها را کشف کنیم و ببینیم که چه‌طور می‌توانیم با قرار دادن مناسب اجزا در کنار یک‌دیگر، پروژه‌های تمیزتری ایجاد کنیم.

مروری بر قواعد نام‌گذاری

پکیج (Package)

پکیج‌ها به ما این امکان را می‌دهند تا فایل‌هایی (کلاس‌ها، واسط‌ها و...) که به یک‌دیگر شبیه هستند و یا با یک‌دیگر ارتباط دارند را در یک محل، نگه‌داری و سازمان‌دهی کنیم. در نام‌گذاری پکیج‌ها در جاوا، تماما از **حروف کوچک** استفاده می‌کنیم. اگر نام پکیج از چند واژه تشکیل شده باشد، **نیازی نیست** از کاراکتر خاصی مثل - یا _ برای جدا سازی آن‌ها استفاده کنیم. به مثال‌های زیر توجه کنید:

```
com.example.deepspace // Correct

com.example.deep_space // Improper

com.example.deepSpace // Improper
```

کلاس (Class) و واسط (Interface)

برای نام‌گذاری کلاس‌ها، از سبک **PascalCase** استفاده می‌کنیم؛ یعنی کلمات بدون هیچ فاصله یا جداکننده‌ای به صورت پشت سر هم ظاهر می‌شوند، هر واژه با حرف بزرگ آغاز شده و در ادامه‌ی آن، از حروف کوچک الفبای انگلیسی استفاده می‌شود. نام کلاس، در اکثر مواقع، **یک اسم یا یک گروه اسمی** است، مانند Character یا ImmutableList. **نام واسط** نیز معمولا یک اسم یا گروه اسمی است، اما گاهی یک صفت یا گروه وصفی است، مانند Readable.

متد (Method)

نام متدها به صورت **camelCase** نوشته می‌شود؛ به عبارت دیگر، کلمات بدون هیچ فاصله یا جداکننده‌ای به صورت پشت سر هم ظاهر می‌شوند، هر واژه **به جز اولین کلمه** با حرف بزرگ آغاز شده و در ادامه‌ی آن از حروف کوچک استفاده می‌شود. نام متد، معمولا یک **فعل یا گروه فعلی** است، مانند sendMessage یا stop.

ثابت (Constant)

منظور از ثابت‌ها در این مقاله، متغیرهایی هستند که به صورت static و final تعریف شده‌اند. این متغیرها با هدف تغییرناپذیری (immutability) تعریف می شوند؛ به عبارت دیگر، تنها یک بار مقداردهی شده و نمی‌توان مقدار آن‌ها را تغییر داد. ثابت‌ها، شامل موارد زیر هستند (برای آشنایی بیش‌تر، می‌توانید آن‌ها را در گوگل جست‌وجو کنید):

- Primitive types
- Strings
- Immutable types
- Immutable containers of immutable types

توجه داشته باشید که ثابت‌ها باید هنگام تعریف، مقداردهی اولیه شوند.

نام ثابت به شکل **CONSTANT_CASE** نوشته می‌شود؛ همه‌ی حروف بزرگ بوده و هر کلمه از دیگری با کاراکتر _ جدا می‌شود:

```
static final int NUMBER = 5;
static final SomeMutable[] MY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }
```

ساختار پروژه

به طور کلی، ساختار یکتا و منحصربه‌فردی برای پروژه‌های جاوا وجود ندارد؛ با این حال، به یکی از این ساختارهای پرکاربرد که در ابزار مدیریت پروژه Maven (که به زودی با آن آشنا می‌شوید) استفاده می‌شود، اشاره می‌کنیم که پس از آشنایی، از این ساختار، پیروی کنید. بهتر است فایل‌های کد جاوا را در `src/main/java` و فایل‌های منبع (text، image، xml و...) را در `src/main/resources` قرار دهید.

ساختار فایل‌های سورس کد (جاوا)

هر فایل سورس، از قسمت‌های «گزاره‌ی پکیج»، «import» و کلاس اصلی (که هم‌نام با اسم فایل است) تشکیل شده است. برای جداسازی هر یک از این اجزا از دیگری، از یک خط خالی استفاده می‌شود.

سازمان‌دهی اجزای کلاس

مهم‌ترین نکته‌ای که در چیدمان اجزای یک کلاس باید به آن توجه کرد، داشتن یک سیر و ارتباط منطقی و طبیعی است، به گونه‌ای که بتوانید به راحتی نحوه‌ی عملکرد یک فایل را برای فرد دیگری توضیح دهید. به عنوان مثال، توابعی که دیرتر به کدتان اضافه می‌کنید را لزوما در انتهای فایل قرار ندهید، بلکه بسته به نوع و عملکرد، آن را در مناسب‌ترین محل، تعریف و پیاده‌سازی کنید. برای یک‌دست و منظم کردن فایل سورس‌کد، می‌توانید از **الگوی زیر** استفاده کنید:

```
public class CleanCode {

    // static fields

    // instance fields

    // constructor(s)

    // static methods

    // Getters and setters

    // methods from implemented interfaces

    // instance methods

    // equals(), toString(), ...

}
```

توجه: پکیج‌بندی درست و قرار دادن کلاس‌هایی که وظیفه‌ی مشابهی دارند، در یک پکیج، سبب تمیزی ساختار پروژه خواهد شد.

جداسازی اجزا

استفاده از خطوط خالی برای جداسازی اجزای مختلف برنامه، راهکار هوشمندانه‌ای برای نشان دادن ارتباط آن‌ها با یک‌دیگر است. در تعریف توابع سعی کنید به کمک خطوط خالی ارتباط توابع را مشخص کنید. اگر گروهی از توابع هستند که در عملیات مشترکی مورد استفاده قرار می‌گیرند (مثلا تمام توابعی که مربوط به پیاده‌سازی بخش شبکه هستند)، بهتر است آن‌ها را پشت سر هم تعریف کرده و بین‌شان ۲ خط خالی قرار دهید. حال برای مجزا کردن این توابع از گروهی دیگر، کافی است بین آن‌ها از تعداد خطوط خالی بیش‌تری استفاده کنید.

کد تکراری، ممنوع!

به گفته‌ی Robert Martin، مهندس نرم‌افزار مشهور، کد تکراری یا با کارکرد بسیار مشابه، ریشه‌ی بسیاری مشکلات در طراحی نرم‌افزار است. کدهای شامل قطعه‌های تکراری یا تقریبا یکسان، نشانه‌هایی از بی‌برنامگی، اهمیت ندادن و غیر حرفه‌ای بودن برنامه‌نویس است. وظیفه‌ی اول و آخر (!) هر توسعه‌دهنده‌ی نرم‌افزار، آن است که تکه‌کدهای تکراری را در برنامه‌ی خود، **از بین ببرد**. IntelliJ شامل ابزارهایی هوشمند است که بسیاری از قطعات تکراری را در کد پیدا می‌کنند. به طور کلی، کد تکراری باید به یک کلاس یا متد دیگر منتقل شود (به کمک ابزارهایی هم‌چون Extract Method) تا در صورت نیاز به تغییر کد، تنها نیاز به تغییر یک‌باره داشته باشیم و مجبور نشویم تمامی توابع را برای کدهای احتمالی تکراری، بررسی کنیم.

هر کلاس، فقط یک مسئولیت

اصل SRP در مهندسی نرم‌افزار (که مخفف Single Responsibility Principle است) بیان می‌دارد که هر کلاس در کد، باید **فقط یک** وظیفه‌ی مشخص داشته باشد و چند کار متفاوت و بی‌ربط را با هم انجام ندهد. مثلا، در یک اپ فروشگاهی، نباید یک کلاس، هم وظیفه‌ی مدیریت کالاها و هم وظیفه‌ی مدیریت کاربران را به عهده داشته باشد؛ اگر کلاسی این‌گونه طراحی شده است، لازم است آن را به دو کلاس مجزا تجزیه کرد.

خصوصی‌سازی تا حد امکان

فیلدهای کلاس، به جز فیلدهایی که استاتیک بوده و یا برای ساخت الگوهایی هم‌چون Signleton مورد نیاز هستند، نباید به صورت public تعریف شوند (مگر آن‌که در نمودار UML طراحی شده، چنین چیزی قید شده باشد) و **باید private باشند**. لازم است برای دسترسی به این فیلدها، تنها از طریق متدهای getter و setter اقدام شود.

هم‌چنین، اگر یکی از فیلدهای کلاس از نوع آرایه، Set و یا HashMap است، لازم است متدهایی جدا برای حذف، افزودن یا تغییر اعضای آن داده‌ساختارها تعریف شود و تغییر نباید روی خروجی getter انجام شود. استفاده از setter ها در این موقعیت، کمک می‌کنند که کنترل بهتری روی معتبر بودن داده‌هایی که قرار است وارد آرایه شوند، داشته باشیم.

هرچه نزدیک‌تر، بهتر

سعی کنید متغیرهای محلی را درست قبل از اولین استفاده از آن‌ها تعریف کنید. با تعریف یک‌باره‌ی تمامی این نوع متغیرها در ابتدای یک بلاک، ممکن است در ادامه، نوع آن‌ها یا علت تعریفشان را فراموش کنید و نیاز باشد تا مکررا به ابتدای بلاک بازگردید. رعایت دقیق این نکته و نکات مشابه آن، سبب می‌شود کدتان یک رفتار و جریان طبیعی را دنبال کند.