# مقدمه

کتاب "Effective Java" به شما کمک می‌کند تا از زبان جاوا و کتابخانه‌های مهم مانند java.lang, java.util, java.io, java.util.concurrent, و java.util.function به طور کاربردی استفاده کنید. این کتاب ۱۱ موضوع را در بیش از ۹۰ توصیه مورد بررسی قرار می‌دهد. خواندن این کتاب مانند خواندن یک کتاب الگوریتم یا الگوی طراحی است ، که نباید یک بار آن را از ابتدا تا انتها بخوانید. می‌توانید هر موضوع را به صورت مستقل مطالعه کرده و درک کنید.

موضوعات این کتاب به منظور افزایش عملکرد کد نوشته شده، نیستند؛ بلکه بیشتر برای نوشتن کدی واضح، صحیح، قابل استفاده، پایدار، انعطاف‌پذیر و قابل نگهداری می‌باشند. قطعا کورس ما نیز الهام گرفته از این کتاب است و به مسائلی که در آن بررسی شده‌اند، می‌پردازد.

# فصل یک :ساخت و تخریب object

## از Consider static factory methods به جای constructors استفاده کنید.

از static factory methods به جای constructors استفاده کنید (اینجا static factory methods ربطی به Factory method pattern نداره و صرفاً اسمشون شبیه همه).

یکی از مزایای استفاده از این روش این است که متد‌های static factory برخلاف constructors می‌توانند نام‌گذاری شوند و اگر بتوانیم یک نام مناسب برای آن‌ها انتخاب کنیم، استفاده از آن‌ها آسان‌تر خواهد بود.

به عنوان مثال:

در کلاس BigInteger در جاوا ۴، متد BigInteger. probablePrime به کلاس BigInteger اضافه شد و جایگزین constructor قبلی شد. حالا با یک نگاه به این متد، می‌توان به راحتی فهمید که این متد چه کاری انجام می‌دهد. عملاً غیرممکن است که کسی بتواند از سیگنیچر کانستراکتور متوجه نوع آبجکت تولیدی از آن شود.

یکی از مزایای دیگر استفاده از static factory methods این است که با هر فراخوانی، مجبور به ایجاد یک آبجکت جدید نیستند. این امکان را به ما می‌دهد که جلوی ایجاد ابجکت‌های تکراری و غیرضروری را بگیریم.

مزیت دیگر استفاده از static factory این است که این متد‌ها لزوماً همان object را برنمی‌گردانند و می‌توانند هر زیرنوعی از آن کلاس را برگردانند. این امکان به ما می‌دهد که یک instance از یک کلاس را داشته باشیم بدون این که بخواهیم آن را public کنیم.

یکی از دیگر مزایای استفاده از static factory methods این است که کلاس بازگردانده شده، object می‌تواند از فراخوانی به فراخوانی با توجه به پارامتر‌های ورودی متفاوت باشد. به عنوان مثال، در EnumSet اگر تعداد ورودی‌ها کمتر از ۶۴ تا باشد، RegularEnumSet را برمی‌گرداند و اگر بیشتر باشد، JumboEnumSet را برمی‌گرداند.

آخرین مزیت استفاده از static factory این است که کلاس ما می‌تواند زیر کلاس‌های را در static factory methods برگرداند که هنوز وجود ندارند. این امکان به ما می‌دهد که خروجی استاتیک فکتوری متد یک نوعی interface و کلاس‌هایی که این متد در آینده برمی‌گرداند، نوعی از آن interface باشند

.

## زمانی که با تعداد زیادی پارامتر ورودی مواجه میشود از bilder استفاده کنید به جای constructor

کانستراکتور ها و استاتیک فکتوری متد ها یه محدودیتی دارند و اون هم اینه که اونا خیلی مقیاس پذیر نیستند در زمان افزایش تعداد پارامتر های ورودی در نظر بگیرید یک کلاسی که بخواد حقایق نوتریشن یک محصول خوراکی رو مدل کنه این جدول موارد زیادی داره شاید بیشتر از 20 تا آیتم ولی همه محصولات کل این 20 تا رو ندارن و خوب مقدارشون صفر است برای مثال رو یه سری محصولات مثلا شیر میزان پروتئین ضروری است و باید درج بشه ولی مثلا برای غلات صبحانه اصن لزومی نداره این مقدار درج بشه چون 0 است

با توجه به تعداد نوع های زیادی که از محصولات خوراکی وجود دارد چه نوع کانستراکتور یا استاتیک فکتوری متدی میتونیم استفاده کنیم برای این مسئله؟

یکی از پترن های سنتی مورد استفاده برای حل این مسئله استفاده از الگوی telescoping constructor هست به این شکل که برای هر محصول یک کانستراکتور جدید باید بسازیم از پارامتر های ضروری اون محصول این روش کار میکنه ولی مشکل اینه که نوشتنش و خواندنش برای کلاینتی که از اون استفاده میکنه واقعا سخته ، یک عیب دیگه هم که داره اگر استفاده کننده یه اشتباهی کنه تو پاس دادن پارامتر ها اگر شانس بیاره نوع پارامتر ها یکی نباشه و کامپایل ارور بده اگر نه میتونه باعث باگ بشه

پترن دومی که وجود داره در زمانه مواجه شدن با کانستراکتوری که وجود داره اینه که از الگو JavaBeans استفاده کنیم

اینجوری که کانستراکتور بدونه سیگنیچر داشته باشیم و از متد های get set استفاده کنیم این پترن معایب پترن قبلی رو نداره خیلی اسون هم هست استفاده ازش اما خوب متاسفانه معایب خودش رو هم داره مثلا وقتی شما با یک کانستراکتور یک آبجکت رو میسازیم اطمینان دارید که آبجکت ناسازگاری رو نساختید

یکی از معایب دیگر این الگو این است که با این الگو نمیشه آبجکت immutable ساخت ،

خوشبختانه راه حل سومی هم هست برای مواجهه با مشکل کانستراکتور با آرگومان های زیاد آپشنال و اون هم استفاده از پترن بیلدر هست

به این شکل که کاربر نهایی استفاده کننده کانستراکتور یا استاتیک فکتوری رو با ورودی های اجباری رو فراخوانی میکنه و یک بیلدر آبجکت دریافت میکنه و میتونه با فراخانی متد های ست بیلدر و دریافت مجدد آبجکت بیلدر و در نهایت فراخانی متد بیلد آبجکت مورد نظرش رو بسازه و به طور ساده ای هم میتونه اون آبجکت immutable باشه

یکی دیگه از مزایای این پترن اینه که این پترن مناسبه برای کلاس های hierarchies کلاس های ابسترکت میتوانند بیلدر ابسترکت خودشون رو داشته باشند

ذکر مثال کتاب

بیلدر ها خیلی فلکسیبل هستن میتوانند بارها بارها در سناریو ها مختلف فراخانی شودند

این پترن همچنین معایبی هم دارد مثلا حتما برای ساخت اون آبجکت اول بیلدر رو بسازی

در حالی که به نظر ساخت بیلدر یک کلاس کم هزینه است اما در عمل در بعضی از سناریو های کلان میتواند مشکل زا باشد

همچنین نوشتنش هم ممکنه یه جاهایی سخت تر باشه از پترن کانستراکتور تلسکوپی بنابراین باید جایی استفاده بشه که واقعا تعداد زیادی پارامتر وجود داشته باشه اما از این نکته غافل نشید که میتونه تعداد پارامترها تون در آینده افزایش داشته باشد

در نهایت الگو بیلدر انتخاب خوبی است برای آن دسته از کلاس های که تعداد پارامتر هاشون از تعداد انگشت شمار بیشتره مخصوصا که تعدادی از اون پارامتر ها آپشنال باشه همچین بیلدر از پترن جاوا بین امن تره

یک کلاس را به گونه‌ای پیاده‌سازی کنید که فقط یک نمونه از آن کلاس می‌سازد. دو روش معمول برای ایجاد Singleton وجود دارد: یکی با استفاده از یک فیلد عمومی و دیگری با استفاده از یک متد factory استاتیک. هر دو روش از یک سری مزایا و معایب برخوردارند. اولین روش از یک فیلد عمومی استاتیک و غیرقابل تغییر استفاده می‌کند که همیشه به همان نمونه اشاره می‌کند، در حالی که روش دوم از یک متد کارخانه‌ای برای بازگرداندن نمونه استفاده می‌کند. همچنین از روشی به نام Enum نیز برای پیاده‌سازی Singleton استفاده می‌شود که از دیدگاه سادگی و ایمنی بیشتری برخوردار است. این الگوها باید با دقت و بر اساس نیازهای خاص برای ایجاد Singleton در نرم‌افزارها انتخاب شوند.

## مورد 4: غیرقابلیت‌بودن را با سازنده خصوصی اعمال کنید

کلاس‌هایی است که تنها شامل متدها و فیلدهای استاتیک می‌باشند، مانند java.lang.Math یا java.util.Arrays. این کلاس‌های کاربردی برای ساخت نمونه مورد استفاده قرار نمی‌گیرند، اما بدون وجود سازنده‌های صریح، یک سازنده پیش‌فرض عمومی فراهم می‌شود که ممکن است باعث ساخت نمونه‌های ناخواسته شود. برای جلوگیری از این موضوع، می‌توان یک سازنده خصوصی اضافه کرد که باعث می‌شود کلاس غیرقابل ساخت شود. این سازنده خصوصی باعث می‌شود که کلاس از خارج از خودش غیرقابل دسترس باشد و هر گونه فراخوانی ناخواسته داخلی را جلوگیری می‌کند. علاوه بر این، این اقدام جلوی ساخت زیرکلاس‌ها را می‌گیرد زیرا همه‌ی زیرکلاس‌ها باید سازنده‌ی کلاس پدر را فراخوانی کنند و با حذف سازنده قابل دسترسی کلاس پدر، این امکان را می‌گیرد. این روش ساده از ساخت سازنده خصوصی موجب جلوگیری از ساخت نمونه‌های ناخواسته و زیرکلاس‌ها می‌شود تا کلاس ابزار فقط به عنوان یک ابزار کاربرد داشته باشد.

dependency injection به hardwiring resources ترجیح دهید

در بسیاری از کلاس‌ها، یک یا چند منبع زیرین مورد نیاز است. به عنوان مثال، یک بررسی کننده املایی به یک لغتنامه وابسته است. برای پیاده‌سازی این کلاس‌ها، استفاده از کلاس‌های کمکی استاتیک (Item 4) یا سینگلتون‌ها (Item 3) رایج است. اما هر دو روش مناسب نیستند زیرا فرض می‌کنند که تنها یک لغتنامه مناسب وجود دارد. در عمل، هر زبانی لغتنامه خودش را دارد و برای واژگان ویژه از لغتنامه‌های خاص استفاده می‌شود. برای آزمون نیز ممکن است لازم باشد از یک لغتنامه خاص استفاده کنیم.

روشی که بهتر است، قابلیت پشتیبانی از چندین نمونه از یک کلاس است که هرکدام از آن‌ها از منبع مورد نظر مشتری استفاده کنند. یک الگوی ساده برای این کار، تزریق وابستگی است: به عبارت دیگر، لغتنامه به عنوان یک وابستگی برای بررسی کننده املایی در هنگام ایجاد نمونه، به آن تزریق می‌شود.

این الگوی تزریق وابستگی به اندازه‌ای ساده است که بسیاری از برنامه‌نویسان سال‌ها استفاده می‌کنند بدون اینکه بدانند این الگو یک نام خاص دارد. این الگو با تعداد دلخواهی از منابع و گراف‌های وابستگی کار می‌کند و این امکان را فراهم می‌کند که شی‌های وابسته را به اشتراک گذاریم.

استفاده از تزریق وابستگی می‌تواند انعطاف‌پذیری و قابلیت آزمون را بهبود بخشد، اما ممکن است در پروژه‌های بزرگ که حاوی هزاران وابستگی هستند، باعث شلوغی شود. برای کاهش این شلوغی، می‌توان از چارچوب‌های تزریق وابستگی مانند Dagger، Guice یا Spring استفاده کرد.

به طور خلاصه، بهتر است از سینگلتون یا کلاس‌های کمکی استاتیک برای پیاده‌سازی کلاس‌هایی که وابسته به یک یا چند منبع زیرین هستند، استفاده نکنیم و به جای آن، منابع یا فابریک‌هایی برای ایجاد آن‌ها به عنوان پارامتر به سازنده (یا فابریک استاتیک یا سازنده) بدهیم. این روش، به عنوان تزریق وابستگی شناخته می‌شود و انعطاف‌پذیری، قابل استفاده مجدد بودن و قابلیت آزمون یک کلاس را به طور قابل ملاحظه‌ای ارتقا می‌دهد.

## از ایجاد اشیاء غیر ضروری خودداری کنید

یک موضوع مرسومی هست که یک ابجکت رو چند بار استفاده کنیم برای فاگشنالیتی های مختلف ، این کار میتواند شسته رفته تر و سریع تر باشد نسبت به ساختن ابجکت جدید ، یک ابجکت میتواند بارها و بار ها استفاده بشه اگر که immutable باشه ، همچنین نا گفته نماند که میشود از ابجکت های mutable هم چند بار استفاده کرد اگر جایی تغییر نمیکنند در صورت لزوم

شما همچنین میتواند از ساخت ابجکت جدید با استفاده از پترن های مثل static factory methods جلوگیری کنید مانند انجام چنین کاری استفاده از ( Boolean.valueOf(String به جای (Boolean(String

بعضی ابجکت ها هم ساختشون هزینه بر هست ، که اون ها هم کاندید های خوبی هستند برای برای چند بار استفاده شدند مانند

static boolean isRomanNumeral(String s) {

return s.matches("^(?=.)M\*(C[MD]|D?C{0,3})"

+ "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

}

این کد رو میشه بهبود داد به شکل زیر

public class RomanNumerals {

private static final Pattern ROMAN = Pattern.compile(

"^(?=.)M\*(C[MD]|D?C{0,3})"

+ "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

static boolean isRomanNumeral(String s) {

return ROMAN.matcher(s).matches();

}

}

در ماشین من شاهد افزایش ۶ برابری پرفومنس بودیم

هم طول نوشتن کد سریع تر شد

هم شسته رفته تر و هم پرفومنس بیشتر

عمل باکسینگ هم نوعی ابجکت سازی هست و خوب هزینه بر هست بهتر هست در سناریو های که امکاش هست از پریمیتیو ها استفاده بشه بجای کلاسشون

## ارجاعات ابجکت های منسوخ را حذف کنید

اگر شما به زبان های که memory management اونا دست خودتون بود مثل سی یا سی پلاس پلاس کار میکردید و بعد به زبانی مثل زبان جاوا سوییچ میکردید متوجه میشدید که memory mangment خودکار داند چقدر کار باهاشون راحت تره

اما همیشه این زبان ها نمیتونند به شکل کامل جلوگیری کنند از memory leak به مثال پایین نگاه کنید

public class Stack {

private Object[] elements;

private int size = 0;

private static final int DEFAULT\_INITIAL\_CAPACITY = 16;

public Stack() {

elements = new Object[DEFAULT\_INITIAL\_CAPACITY];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

/\*\*

\* Ensure space for at least one more element, roughly

\* doubling the capacity each time the array needs to grow.

\*/

private void ensureCapacity() {

if (elements.length == size)

elements = Arrays.copyOf(elements, 2 \* size + 1);

}

}

در این مثال در تابع pop ما نشت حافظه داریم چرا ؟

چون در زمان پاپ کردن مقداربازگشتی همواره در مموری میمونه این در حالی هست که هیچ وقت هم در این سناریو ما امکان نداره دوباره بهش روجوع بشه وخوب باید تابع pop ما به شکل زیر اصلاح بشه

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

elements[size] = null; // Eliminate obsolete reference

return result;

}

شایان ذکر است هر جای لازم نیست چنین کار هایی کنید در سنارسو های خاص مثل جایی که کلاس ما خودش مموری منیجمنت رو دست گرفته

**یکی دیگه از جا های که ما دچهار مموری لیک میشیم ، کش ها هستند**

ما میاییم ابجکت ها که داریم رو در کش ها ذخیره میکنیم برای دسترسی سریع تر بهشون ولی گاهی اوقات این کش ها مقداری زیادی در خودشون ذخیره میکنند که خیلی وقت هست که ما سراغشون نرفتیم

بهتره برای این چنین سناریو هایی مکانیزم های بچینیم که این مشکل پیش نیاد

یکی از راه حل ها اینه که از WeakHashMap استفاده کنیم به جای HashMap این موجودیت خودش اینطوری عمل میکنه که اگر از کی یک رفرنس دیگر وجود نداشته باشه gcمیتونه حذفش کنه

دیگر راه حل اینه که در یک جاب یا ترد جداگانه و در بازه های زمانی بیایم و کشمون رو تمیز کنیم

## ارجاعات اشیاء منسوخ را حذف کنید

استفاده از فاینالایزرها (finalizers) و کلینرها (cleaners) عموماً غیرپیش‌بینی‌شده، خطرناک و در بسیاری از موارد غیرضروری است. این استفاده‌ها ممکنه باعث رفتار ناپایدار، کارایی پایین و مشکلات قابلیت حمل‌ونقل شود.

استفاده از فاینالایزرها در جاوا از نسخه 9 به بعد منسوخ شده‌اند، اما هنوز هم در کتابخانه‌های جاوا استفاده می‌شوند. جایگزین فاینالایزرها در جاوا نیز کلینرها هستند. کلینرها کمتر خطرناک از فاینالایزرها هستند، اما همچنان غیرپیش‌بینی‌شده، کند و در بسیاری از موارد غیرضروری هستن

یک نقطه ضعف فاینالایزرها و کلینرها این است که هیچ تضمینی وجود ندارد که زمان اجرای آن‌ها به‌سرعت صورت بگیرد. این بدان معناست که هرگز نباید درون یک فاینالایزر یا کلینر اقدامات زمانی انجام دهید. به عنوان مثال، به این امید که یک فاینالایزر یا کلینر فایل‌ها را ببندد، که می‌تواند باعث باقی‌ماندن فایل‌های باز شده (open file descriptors) به‌دلیل تأخیر سیستم در اجرای فاینالایزرها یا کلینرها شود و منجر به خرابی برنامه شود

همچنین، اجرای به‌موقع فاینالایزرها و کلینرها در اصل به الگوریتم مبادله‌زباله بستگی دارد که بسیار متفاوت در هر پیاده‌سازی است. رفتار یک برنامه که به به‌موقع بودن اجرای فاینالایزر یا کلینر وابسته باشد همچنین ممکن است متفاوت باشد. این مسئله می‌تواند منجر به مشکلات ناپیش‌بینی در محیط‌های مختلف شود

## سعی کنید از try-with-resources استفاده کنید تا try-finally

بعضی از لایبرری های جاوا مثل java.sql.Connection معمولا نیاز دارند که متد کلوزشون رو کال کنیم و خوب try-finally راه خیلی خوبی به نظر میرسه

static String firstLineOfFile(String path) throws IOException {

BufferedReader br = new BufferedReader(new FileReader(path));

try {

return br.readLine();

} finally {

br.close();

}

}

ولی کجا کثیف کاری پیش میاد ؟

وقتی دوتا resurse داریم که بخواییم ببندیمشون

static void copy(String src, String dst) throws IOException {

InputStream in = new FileInputStream(src);

try {

OutputStream out = new FileOutputStream(dst);

try {

byte[] buf = new byte[BUFFER\_SIZE];

int n;

while ((n = in.read(buf)) >= 0)

out.write(buf, 0, n);

} finally {

out.close();

}

} finally {

in.close();

}

}

از جاوا ۷ به بعد این امکان به جاوا اضافه شد که از try-with-rsources استفاده کنیم ، فقط باید منابع ما اینترفیس AutoCloseable را ایمپلیمنت کرده باشند

در ان صورت میشود کد بالا رو به این شکل نوشت

static void copy(String src, String dst) throws IOException {

try (InputStream in = new FileInputStream(src);

OutputStream out = new FileOutputStream(dst)) {

byte[] buf = new byte[BUFFER\_SIZE];

int n;

while ((n = in.read(buf)) >= 0)

out.write(buf, 0, n);

}

}